[Intro 3](#_Toc130999632)

[Prime operazioni 4](#_Toc130999633)

[Valore e tipi 5](#_Toc130999634)

[Commenti 5](#_Toc130999635)

[Classi 6](#_Toc130999636)

[Variabili 6](#_Toc130999637)

[Stringhe 7](#_Toc130999638)

[Liste 8](#_Toc130999639)

[Dizionari 10](#_Toc130999640)

[Tuple 11](#_Toc130999641)

[Espressioni e istruzioni 11](#_Toc130999642)

[Funzioni 12](#_Toc130999643)

[Espressioni booleane 13](#_Toc130999644)

[Cicli 14](#_Toc130999645)

[Input da tastiera 15](#_Toc130999646)

[Files 15](#_Toc130999647)

[Modulo math 16](#_Toc130999648)

[Modulo os 16](#_Toc130999649)

[Libreria Numpy 17](#_Toc130999650)

[Modulo random 20](#_Toc130999651)

[Libreria Pandas 21](#_Toc130999652)

[Libreria Matplotlib 24](#_Toc130999653)

[Modulo pyplot 24](#_Toc130999654)

[Modulo seaborn 27](#_Toc130999655)

[Accesso dati in modo alternativo 28](#_Toc130999656)

[Modulo tweepy 29](#_Toc130999657)

[Modulo glob 29](#_Toc130999658)

[Modulo Scikit-Learn 30](#_Toc130999659)

[Metriche e valutazione performances modelli con scikit-learn 31](#_Toc130999660)

[Regressione con scikit-learn 32](#_Toc130999661)

[Riduzione di dimensionalità con scikit-learn 37](#_Toc130999662)

[Classificazione con scikit-learn 38](#_Toc130999663)

[Clustering con scikit-learn 39](#_Toc130999664)

# Intro

Python è un linguaggio di programmazione orientato agli oggetti, creato nel 1991, molto utilizzato per web development, software development, matematica e system scripting. Lavora su diverse piattaforme (Windows, Mac, Linux…), ha una sintassi semplice ed intuitiva simile all’inglese, che permette la creazione di programmi con poche linee rispetto a linguaggi più complessi.

Molti PC hanno Python già installato: per controllare la propria versione di Python (col cmd.exe) (o cercare nello start):

C:\Users\Your Name>python --version

Ci sono due versioni di Python, chiamate Python 2 e Python 3. Sono molto simili, pertanto imparandone una non è difficile passare poi all’altra. Di fatto, ai primi livelli di apprendimento le differenze tra le due versioni sono poche. Questo libro fa riferimento alla più recente versione Python 3, ma troverete anche alcune annotazioni su Python 2.

Immagine che contiene lettera

Descrizione generata automaticamenteL’**interprete** di Python è un programma che legge ed esegue il codice Python. A seconda del vostro ambiente di lavoro, lo potete avviare facendo click su un’icona, oppure digitando in una riga di comando. In alcune installazioni di Python, è compreso anche un ambiente di sviluppo di base chiamato IDLE. All’avvio, dovreste vedere un output simile a questo:

Le prime tre righe contengono informazioni sull’interprete e il sistema operativo in cui viene eseguito, per cui nel vostro caso concreto potrebbero essere diverse. Ma occhio al numero di versione, che in questo esempio è : comincia con 3, il che significa che state usando Python 3. Se cominciasse con 2, vorrebbe dire che state usando (avete indovinato!) Python 2.

L’ultima riga è un **prompt**, che comunica che l’interprete è pronto a ricevere il codice che inserirete. Se scrivete una riga di codice e poi premete Invio, l’interprete elabora immediatamente il risultato. Ora siete pronti per iniziare.

Si può usare Python in **modalità interattiva**, detta anche “a riga di comando”, che vuol dire interagire direttamente con l’interprete. La modalità interattiva è un buon modo per iniziare e fare esperimenti, ma se si deve lavorare con più di qualche riga di codice, può diventare in breve tempo un impiccio.

In alternativa alla riga di comando, si può scrivere e salvare un programma in un file di testo semplice, chiamato **script**, ed usare poi l’interprete in **modalità script** per eseguirlo. Per convenzione, i file contenenti programmi Python hanno nomi che terminano con l’estensione *.py* . Poiché Python consente entrambe queste modalità, potete provare dei pezzi di codice in modalità interattiva prima di inserirli in uno script.

Uno script di solito contiene una serie di istruzioni. Se ci sono più istruzioni, i risultati compaiono uno alla volta, man mano che le istruzioni vengono eseguite. In modalità script, un’espressione, di per sé, non ha effetti visibili.

Un **modulo** è un file che contiene una raccolta di funzioni correlate (come math, pandas, numpy…). Prima di poter usare le funzioni contenute in un modulo, lo dobbiamo importare con un’**istruzione di importazione**:

import math

Questa istruzione crea un **oggetto modulo** chiamato math. Se visualizzate l’oggetto modulo, ottenete alcune informazioni a riguardo:

math

L’oggetto modulo contiene le funzioni e le variabili definite all’interno del modulo stesso. Per accedere a una funzione del modulo, dovete specificare, nell’ordine, il nome del modulo e il nome della funzione, separati da un punto. Questo formato è chiamato **notazione a punto** o dot notation.

math.log10(20)

Si può anche dare un alias ad un modulo:

import pandas as pd

# Prime operazioni

Questo è un esempio di **istruzione di stampa**, che a dispetto del nome non stampa nulla su carta: visualizza solo un risultato sullo schermo. In questo caso il risultato sono le parole:

print(‘hello world’)

Python dispone di **operatori**, che sono simboli speciali che rappresentano i calcoli fondamentali, come l’addizione e la moltiplicazione. Gli operatori + - \* / e \*\* eseguono nell’ordine addizione, sottrazione, moltiplicazione, divisione ed elevamento a potenza:

4 + 3   
4 - 3  
4\*3  
4/3  
4\*\*3

Python utilizza la notazione anglosassone, per cui i separatori delle migliaia sono le virgole, mentre il punto è usato per separare le cifre decimali.

Altre operazioni semplici sono la **divisione intera**, che divide due numeri e arrotonda il risultato all’intero inferiore (simbolo //); e l’**operatore modulo** (utile per controllare se un numero è divisibile per un altro senza resto, per numeri primi), che restituisce il resto dell’operazione di divisione tra due numeri interi (simbolo %) :

ore = minuti // 60 # restituisce risultato intero

resto = minuti % 60

Quando un’espressione contiene più operatori, la successione con cui viene eseguito il calcolo dipende dall’**ordine delle operazioni**. Per quelle matematiche, Python segue le stesse regole di precedenza comunemente usate in matematica. L’acronimo **PEMDAS** è un modo utile per ricordare le regole:  
- **P**arentesi: hanno il livello di precedenza più elevato e possono essere usate per forzare la valutazione di un’espressione secondo qualsiasi ordine si desideri, dato che le espressioni tra parentesi sono valutate per prime;  
- **E**levamento a potenza: ha la priorità successiva  
- **M**oltiplicazione e **D**ivisione: hanno priorità superiore a somme e differenze   
- **A**ddizione e **S**ottrazione: ultime in graduatoria  
- Da sinistra a destra: gli operatori con la stessa priorità vengono valutati da sinistra verso destra (eccetto la potenza)

In linea generale non è possibile effettuare operazioni matematiche sulle stringhe, anche se la stringa sembra un numero; ma ci sono due eccezioni: + e \*. L’operatore + esegue il **concatenamento**, cioè unisce le stringhe collegandole ai due estremi. Anche l’operatore \* funziona sulle stringhe: ne esegue la **ripetizione**.

La funzione int() prende un dato valore e lo converte, se possibile, in un numero intero; può anche convertire valori in virgola mobile in interi, ma non arrotonda bensì tronca la parte decimale.

La funzione float() converte interi e stringhe in numeri a virgola mobile (come int ma con decimali).

Infine, str() converte l’argomento in una stringa.

# Valore e tipi

Un **valore** è uno degli elementi di base che un programma è in grado di elaborare, come ad esempio una lettera oppure un numero. Per sapere a quale tipo appartiene un dato valore, basta chiederlo all’interprete:

type(2)

Nei responsi, la parola “class” (classe) viene usata nel senso di categoria; un tipo è una categoria di valori. Alle stringhe corrisponde il tipo *str* (se racchiusi tra due apici ‘ ‘), agli interi il tipo *int*, ai numeri con parte decimale il tipo *dec*.

# Commenti

Al crescere delle sue dimensioni e della sua complessità, un programma diventa anche sempre più difficile da leggere. Per questo motivo, è buona abitudine aggiungere ai vostri programmi delle annotazioni che spiegano in linguaggio naturale ciò che il programma sta facendo. Queste annotazioni si chiamano **commenti**, contrassegnati dal simbolo #.

Il commento si può trovare su una riga a sé stante, ma anche in coda a una riga dopo parte di codice (tutto ciò che viene scritto dopo il simbolo e fino alla fine della riga, viene trascurato e non influisce in alcun modo sull’esecuzione del programma).

# Classi

Si possono usare i vari tipi predefiniti in Python (lista, tupla, dizionario, variabile, stringa…), ma se ne possono anche creare di nuovi. Un tipo personalizzato, definito dal programmatore, è chiamato anche **classe**. Una definizione di classe ha questa sintassi:

class Nomeclasse:  
 ‘’’ descrizione della classe ‘’’  
 z = 5.0  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age

L’intestazione indica che la nuova classe si chiama . Il corpo è una stringa di documentazione che spiega cosa fa la classe. Al suo interno si possono poi definire metodi e variabili, come fatto con la variabile *z*. Inoltre, tutte le classi hanno al loro interno una funzione \_\_init\_\_(), che viene sempre eseguita quando la classe viene inizializzata: questa si può definire all’interno di una classe creata, per imporre condizioni al momento della creazione. È prassi che i parametri di \_\_init\_\_() abbiano gli stessi nomi degli attributi. I parametri sono opzionali, quindi se chiamate una classse senza argomenti, ottenete i valori di default. Se fornite un argomento, esso va a sovrascrivere i valori default.

Un altro metodo speciale è \_\_str\_\_(), che ha lo scopo di restituire una rappresentazione di un oggetto in forma di stringa. \_\_init\_\_() rende più facile istanziare un oggetto, mentre \_\_str\_\_() è solitamente utile per il debugging.

A questo punto, si possono creare oggetti della classe creata, richiamando il nome della classe come se fosse una funzione con ():

Nuovo = Nomeclasse()

La creazione di un nuovo oggetto è detta **istanziazione**, e l’oggetto è un’**istanza** della classe. Quando stampate un’istanza, Python informa a quale classe appartiene e in quale posizione di memoria è collocata. Ogni oggetto è un’istanza di una qualche classe, per cui i termini “oggetto” ed “istanza” sono equivalenti.

Potete assegnare dei valori ad un’istanza usando la notazione a punto:

Nuovo.x = 1.0  
Nuovo.y = 4.0

In questo caso stiamo assegnando dei valori a degli elementi di un oggetto, ai quali è stato attribuito un nome (x e y). Questi elementi sono detti **attributi**. La variabile *Nuovo* fa riferimento ad un oggetto *Nomeclasse* che contiene due attributi *x* e *y*, ed ogni attributo fa riferimento ad un numero in virgola mobile. Potete cambiare lo stato di un oggetto con un’assegnazione ad uno dei suoi attributi.

# Variabili

Una delle caratteristiche più potenti di un linguaggio di programmazione è la capacità di elaborare delle variabili. Una variabile è un nome che fa riferimento ad un valore. Generalmente, i programmatori chiamano le variabili con dei nomi significativi, in modo da documentare a cosa servono. **Un’istruzione di assegnazione** serve a creare una nuova variabile, specificandone il nome, e ad assegnarle un valore:

messaggio = ‘ciao’

Le stringhe nelle istruzioni di stampa sono racchiuse tra apici ( ) oppure virgolette ( ). Virgolette e apici sono equivalenti; la maggioranza degli utenti usa gli apici, eccetto nei casi in cui nel testo da stampare sono contenuti degli apici (che possono essere usati anche come apostrofi o accenti). In questi casi, frequenti con l’italiano, bisogna usare le virgolette.

Immagine che contiene tavolo

Descrizione generata automaticamenteI nomi possono essere lunghi a piacere e possono contenere sia lettere che numeri, ma non possono iniziare con un numero. È possibile usare anche le lettere maiuscole, ma per i nomi di variabile è convenzione utilizzare solo lettere minuscole. In ogni caso, tenete conto che, per l’interprete, maiuscole e minuscole sono diverse, pertanto *spam*, *SPAM* e *Spam* sono variabili diverse. Il trattino basso o underscore, , può far parte di un nome: è usato spesso in nomi di variabile composti da più parole. Se assegnate un nome non valido alla variabile, otterrete un errore di sintassi. Inoltre le **parole chiave riservate** di Python non si possono usare come nomi di variabile (se usate da errore).

Ricordarsi sempre che il lato sinistro di un’istruzione di assegnazione deve essere un nome di variabile.

Una variabile può anche essere riassegnata ad un nuovo valore:

x = 5 # ora è uguale a 5  
x = 7 # ora è uguale a 7

Una variabile può anche avere un aggiornamento (incremento se aumenta, decremento se diminuisce):

x = x + 1

# Stringhe

Una stringa è una sequenza di caratteri. Potete accedere ai singoli caratteri usando gli operatori parentesi quadre:

stringa = ‘mela’  
stringa[1] # e

L’espressione all’interno delle parentesi quadre è chiamato **indice**. L’indice è un numero intero che indica (di qui il nome) il carattere della sequenza che desiderate estrarre. L’indice parte dallo 0 (prima posizione) e va verso sinistra. Con un indice di valore negativo, si conta a ritroso dalla fine della stringa.

Si può verificare la lunghezza di una stringa nel seguente modo:

len(stringa) # 6

Per selezionare una porzione di una scritta, si può fare nel seguente modo (se specificati entrambi, è un range, altrimenti se :n dall’inizio fino all’n-1 carattere, se n: dall’n-1 fino alla fine):

stringa[1:2] #el  
stringa[:2] #me  
stringa[1:] #la

Le stringhe sono **immutabili**, in altre parole, non è consentito cambiare una stringa esistente.

Tra i vari metodi (nessuno di questi sostituisce la stringa originale, ma si possono usare per una nuova stringa) che si possono usare per le stringhe, ci sono:  
- *.upper()* *.lower() : per rendere la stringa maiuscola/minuscola.*  
- *.find(‘abc’,1,2) :* per trovare uno o più caratteri in una stringa; se con più caratteri indica il primo carattere che si trova; il primo numero indica il carattere da cui iniziare a cercare, il secondo il carattere a cui fermarsi; restituisce -1 se non trova il carattere.  
- *in* : se un carattere è presente o no in una stringa (*True* se vero, *False* se non è vero): ‘el’ in stringa   
- *.capitalize()* per rendere l’iniziale maiuscola.  
- *.count(‘a’)* per contare il numero di caratteri presenti in una stringa.  
- *.index(‘abc’)* per ottenere la posizione della stringa.  
- *.isalnum()* per vedere se è composta completamente da caratteri alfanumerici.  
- *.isascii()* per vedere se è composta completamente da caratteri ascii.  
- *.isdecimal()* per vedere se è composta completamente da caratteri decimali.  
- *.isalpha()* per vedere se è composta completamente da caratteri dell’alfabeto.  
- *.islower()* per vedere se è composta completamente da caratteri minuscoli.  
- *.isspace()* per vedere se è composta completamente da caratteri spazio.  
- *.isupper()* per vedere se è composta completamente da caratteri maiuscoli.  
- *.join(tupla)* per rendere le stringhe contenute in una tupla come una sola stringa.  
- *.replace(‘abc’, ‘def’)* per sostituire *abc* con *def*.  
- *.partition(‘abc’)* per dividere la stringa quando incontra *abc*.  
- *.rfind(‘abc’,1,2) :* come *.find* ma per l’ultima che trova.  
- *.rindex(‘abc’)* come *.index* ma per l’ultima che trova.  
- *.swapcase()* per rendere le maiuscole minuscole e viceversa.   
- *.title()* per rendere l’iniziale di ogni parola maiuscola.

Si possono anche confrontare le parole, con == per vedere se coincidono, o con >/< per metterle in ordine alfabetico (attenzione che, in caso di maiuscole e minuscole, Python riporta sempre prima tutte le maiuscole).

Per rendere una lista di parole come unica stringa, si fa così:

delimita = ‘ ‘ # per inserire gli spazi tra le varie parole  
s = delimita.join(lista)

# 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 (anche diversi all’interno della stessa lista, così come una lista dentro un’altra lista, detta **lista nidificata**). I valori che fanno parte della lista sono chiamati **elementi**.

Ci sono vari modi per creare una nuova lista; quello più semplice consiste nel racchiudere i suoi elementi tra parentesi quadrate [] :

lista1 = [‘abc’, 1, ‘13’, 2.4, [3,’ciao’]]

La sintassi per accedere agli elementi di una lista è la stessa usata per i caratteri di una stringa: le parentesi quadre, con un’espressione tra parentesi che specifica l’indice dell’elemento (primo elemento sempre il numero 0; con un indice di valore negativo, si conta a ritroso dalla fine della lista). Ma a differenza delle stringhe, le liste sono mutabili. Quando l’operatore parentesi quadre compare sul lato sinistro di un’assegnazione, identifica l’elemento della lista che sarà riassegnato:

lista1[2] # ‘13’  
lista1[2] = ‘14’ # ‘14’

Per avere la lunghezza di una lista, si fa con len(lista).

Si possono fare alcune operazioni tipiche, come concatenazione di liste (col + tra due liste), e ripetizione di una lista più volte (con l’operatore \*). Come per le stringhe, si può eseguire lo slicing (attenzione ad assegnare valori, che le liste sono modificabili a differenza delle stringhe):

lista1[1:2] # [1, ‘13’]  
lista1 [:2] # [‘abc’, 1, ‘13’]  
lista1[1:] #[‘13’, 2.4, [3,’ciao’]]

Tra i vari metodi che si possono usare per le liste, ci sono:  
- *.append(…)* : per aggiungere uno o + elementi a fine lista*.*- *.clear()* : per rimuovere tutti gli elementi da una lista*.*- *.copy(…)* : per copiare una lista*.*- *.count()* : per contare il numero di elementi in una lista*.*- *.extend(…)* : per aggiungere una lista alla fine di un’altra lista*.*- *.index(…)* : per avere l’indice di un particolare elemento in una lista*.*- *.insert(4, …)* : per aggiungere uno o + elementi in una particolare posizione della lista*.*  
- *.pop(…)* : per rimuovere un elemento dalla lista (usando l’indice) (senza indice, rimuove l’ultimo elemento) (anche per ottenere un elemento in particolare da estrarre da una lista)*.*- *.remove(…)* : per rimuovere un elemento dalla lista (usando un valore che si vuole togliere)*.*- *.reverse()* : per invertire l’ordine degli elementi in una lista*.*- *.sort(…)* : per ordinare gli elementi di una lista*.*

Per cancellare un elemento di una lista, si può usare anche del lista1[3] .

Per convertire una stringa in una lista di valori (uno per ogni lettera/spazio), si può fare così:

list(stringa)

Per convertire una stringa formata da più parole in una lista di valori (uno per ogni parola), si può fare così (se … non viene definito, lo considera come lo spazio, ma si può modificare a piacere in base all’utilizzo):

stringa.split(…)

Attenzione che due liste, anche se contengono gli stessi elementi, se si confrontano sono considerate diverse da Python (in questo caso si dice che le due liste sono **equivalenti**, perché contengono gli stessi elementi, ma non **identiche**, perché non sono lo stesso oggetto. Se due oggetti sono identici, sono anche equivalenti, ma se sono equivalenti non sono necessariamente identici).

Posso riassegnare una lista ad un altro valore (chiamato **alias**) nel seguente modo:

a = b

In questo caso, se l’oggetto munito di alias è mutabile, i cambiamenti provocati da un alias si riflettono anche sull’altro. In genere è più sicuro evitare gli alias quando si sta lavorando con oggetti mutabili. Per gli oggetti immutabili come le stringhe, gli alias non sono un problema.

# Dizionari

Un **dizionario** è simile ad una lista, ma è più generico. Infatti, mentre in una lista gli indici devono essere numeri interi, 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 e un valore è detta **coppia chiave-valore** o anche **elemento**. In linguaggio matematico, un dizionario rappresenta una relazione di corrispondenza, o **mappatura**, da una chiave a un valore, e si può dire pertanto che ogni chiave “mappa in” un valore. Una chiave è unica, non ci possono essere due chiavi uguali in un dizionario (dev’essere univoca, permettere di identificare unicamente la coppia); invece ci possono essere molti valori uguali, in chiavi diverse.

La funzione *dict* crea un nuovo dizionario privo di elementi:

dizio = dict()

Le parentesi graffe {} rappresentano un dizionario vuoto. Per aggiungere elementi al dizionario, usate le parentesi quadre:

dizio[‘one’] = ‘uno’ # chiave ‘one’, valore ‘uno’ 🡪 {‘one’ : ‘uno’}

oppure anche in questo modo:

dizio = {‘one’ : ‘uno’, ‘two’ : ‘due’, ‘three’: ‘tre’}

Attenzione che l’ordine delle coppie chiave-valore non è necessariamente lo stesso: se scrivete lo stesso esempio nel vostro computer, potreste ottenere un altro risultato ancora. In genere, l’ordine degli elementi di un dizionario è imprevedibile. Ma questo non è un problema, perché gli elementi di un dizionario non sono indicizzati con degli indici numerici. Infatti, per cercare un valore si usano invece le chiavi:

dizio[‘two’] # per elemento di chiave ‘two’ 🡪 ritorna il valore ‘due’

La funzione *len(dizio)* è applicabile ai dizionari, e restituisce il numero di coppie chiave-valore.

Anche l’operatore *in* funziona con i dizionari: informa se qualcosa compare come chiave nel dizionario (non dice il relativo valore però; e non funziona con i valori):

‘one’ in dizio

Tra i vari metodi che si possono usare per i dizionari, ci sono:  
- *.values(chiave)* : per vedere se un valore è contenuto in un dizionario*.*  
- *.get(chiave, valore)* : per ottenere il valore relativo a chiave (valore è facoltativo, il default è None, restituisce il valore definito se la chiave non è presente nel dizionario)*.*  
- *.clear()* : per rimuovere tutti gli elementi da un dizionario*.*  
- *.fromkeys(chiavi, valore)* : per creare un dizionario, dando le chiavi (il valore è opzionale, default è None)*.*  
- *.items()* : per ottenere una lista contenente tuple con le coppie chiave-valore presenti nel dizionario*.*  
- *.pop(chiave)* : per eliminare una coppia data la chiave*.*- *.popitem()* : per eliminare l’ultima coppia inserita nel dizionario*.*- *.setdefault(chiave, value)* : per ottenere il valore relativo alla chiave (il value è opzionale, default è None, funziona solo se la chiave non esiste e nel caso lo impone come nuova chiave)*.*- *.update(*{‘one’ : ‘uno’}*)* : per modificare una coppia in un dizionario*.*

# Tuple

Una tupla è una sequenza di valori. I valori possono essere di qualsiasi tipo, sono indicizzati tramite numeri interi, e in questo somigliano moltissimo alle liste. La differenza fondamentale è che le tuple sono immutabili (al contrario delle liste).

Una tupla può essere creata in diversi modi (con o senza parentesi tonde, ma per convenzione si mettono); una tupla con un solo elemento deve avere la virgola dopo di esso (per distinguerla da una variabile/stringa). Si può anche creare con la funzione tuple() per creare una tupla (se ha come argomento una lista/stringa/tupla, il risultato è una tupla con singoli elementi come gli elementi/lettere).

tupla1 = (1, 4.3, ‘13’, ‘ciao’)  
tupla1 = 1, 4.3, ‘13’, ‘ciao’  
tupla2 = 1,   
tupla3 = tuple()

La maggior parte degli operatori delle liste funzionano anche con le tuple. L’operatore parentesi quadre indicizza un elemento della tupla; allo stesso modo funzionano tutte le operazioni di slicing:

tupla1[0] # 1  
tupla1 [1:2] # (4.3, ‘13’)  
tupla1 [:2] # (1, 4.3, ‘13’)  
tupla1 [1:] #(4.3, ‘13’, ’ciao’)

Tuttavia, se si prova a modificare un elemento in una tupla, si ottiene errore. Se si vuole sostituire qualcosa, si può sempre “giocare” con lo slicing:

tupla1 = ‘A’ + tupla1[1:]

Funzionano anche gli operatori di confronto (=/>/<): confrontando due tuple, si parte dalla prima copia e quando trova un maggiore/minore definisce il >/<.

Le tuple si possono usare con la funzione *zip(…)* per creare coppie di elementi partendo da liste (poi il risultato può essere inserito in liste o dizionari, a seconda dell’esigenza):

s = ‘abc’  
t = [1,2,3]  
z = zip(s,t) # (‘a’,1) (‘b’,1) (‘c’,1) (‘a’,2) (‘b’,2) (‘c’,2) (‘a’,3) (‘b’,3) (‘c’,3)

# Espressioni e istruzioni

Un’**espressione** è una combinazione di valori, variabili e operatori. Un valore è considerato già di per sé un’espressione, come pure una variabile. Quando scrivete un’espressione al prompt dei comandi (come richiamare una variabile, un’operazione tra variabili…), l’interprete la **valuta**, cioè trova il valore dell’espressione.

Un’**istruzione** è una porzione di codice che l’interprete Python può eseguire e che ha un qualche effetto, come creare una variabile o mostrare un valore. Quando scrivete un’istruzione, l’interprete la **esegue**, cioè fa quello che l’istruzione dice di fare (come stampare una variabile, assegnare un valore ad una variabile…). In linea generale, le istruzioni, a differenza delle espressioni, non contengono valori.

# Funzioni

Una **funzione** è una serie di istruzioni che esegue un calcolo, alla quale viene assegnato un nome. Per definire una funzione, dovete specificarne il nome e scrivere la serie di istruzioni. In un secondo tempo, potete “chiamare” la funzione mediante il nome che le avete assegnato. Il **nome** di una funzione è la parola che precede la parentesi. L’espressione tra parentesi è chiamata **argomento** della funzione, e il risultato che produce è il tipo di valore dell’argomento che abbiamo inserito. Si usa dire che una funzione “prende” o “riceve” un argomento e, una volta eseguita l’elaborazione, “ritorna” o “restituisce” un risultato. Il risultato è detto **valore di ritorno**.

Una delle caratteristiche più utili dei linguaggi di programmazione è la loro capacità di prendere dei piccoli mattoni e **comporli** tra loro. Per esempio, l’argomento di una funzione può essere un qualunque tipo di espressione, operazioni aritmetiche incluse, e anche chiamate di funzione.

Ci sono sia funzioni predefinite o “built-in”, che sono parte integrante di Python, ma è anche possibile crearne di nuove. Una **definizione di funzione** specifica il nome di una nuova funzione e la serie di istruzioni che viene eseguita quando la funzione viene chiamata.

def saluto():  
 print(‘ciao’)

*def* è una parola chiave riservata che indica la definizione di una nuova funzione. Il nome della funzione è *saluto*. Le regole per i nomi delle funzioni sono le stesse dei nomi delle variabili: lettere, numeri e underscore ( ) sono permessi, ma il primo carattere non può essere un numero. Non si possono usare parole riservate, e bisogna evitare di avere una funzione e una variabile con lo stesso nome.

Le parentesi vuote dopo il nome indicano che la funzione non accetta alcun argomento. La prima riga della definizione di funzione è chiamata **intestazione**; il resto è detto **corpo**. L’intestazione deve terminare con i due punti, e il corpo deve essere obbligatoriamente indentato, cioè deve avere un rientro rispetto all’intestazione. Per convenzione, l’indentazione è sempre di quattro spazi. Il corpo può contenere un qualsiasi numero di istruzioni.

La definizione di una funzione crea un **oggetto funzione** che è di tipo *function*. La sintassi per chiamare la nuova funzione è la stessa che abbiamo visto per le funzioni predefinite. Una volta definita una funzione, si può anche utilizzarla all’interno di un’altra funzione. Ovviamente, una funzione deve essere definita prima di poterla usare: la definizione della funzione deve sempre precedere la sua chiamata.

All’interno della funzione, gli argomenti che le vengono passati sono assegnati ad altrettante variabili chiamate **parametri**. Questa funzione assegna l’argomento ricevuto ad un parametro chiamato *ciao*. Quando la funzione viene chiamata, stampa il valore del parametro (qualunque esso sia) due volte.

def stampa2volte(ciao):  
 print(ciao)  
 print(ciao)

Quando create una variabile in una funzione, essa è **locale**, cioè esiste solo all’interno della funzione.

Una **stringa di documentazione**, o docstring, è una stringa posta all’inizio di una funzione che ne illustra l’interfaccia. Per convenzione, la docstring è racchiusa tra triple virgolette, che le consentono di essere divisibile su più righe (stringa a righe multiple). È breve, ma contiene le informazioni essenziali di cui qualcuno potrebbe aver bisogno per usare la funzione. Spiega in modo conciso cosa fa la funzione (senza entrare nei dettagli di come lo fa). Spiega che effetti ha ciascun parametro sul comportamento della funzione e di che tipo devono essere i parametri stessi (se non è ovvio).

Le funzioni che abbiamo scritto finora sono “vuote”. Detto in modo semplicistico, non hanno valore di ritorno; ma a voler essere precisi, il loro valore di ritorno è *None*. L’istruzione *return* restituisce un risultato ottenuto dalle operazioni fatte nella funzione (non appena viene eseguita un’istruzione , la funzione termina senza eseguire ulteriori istruzioni).

def calcolo(x,y):  
 z = x \* y  
 return z

# Espressioni booleane

Un’**espressione booleana** è un’espressione che può essere o vera o falsa. I risultati di un’operazione booleana restituiscono *True* (vero) se sono uguali, *False* (falso) altrimenti. L’operatore *==* permette di confrontare valori tra di loro:

5 == 5

*True* e *False* sono valori booleani (*class ‘bool’* con la funzione *type*), non semplici stringhe.

Gli **operatori di confronto** esistenti sono i seguenti:  
- diverso (!=)  
- uguale (==)  
- maggiore (>)  
- minore (<)  
- maggiore uguale (>=)  
- minore uguale (<=)

Gli **operatori logici** invece sono:  
- and: *True* se vere tutte le espressioni   
- or: *True* se vera solo una delle espressioni  
- not: nega un’espressione booleana

Invece, l’*any* prende una sequenza di valori booleani e restituisce *True* se almeno uno dei valori è *True*. Invece l’*all* restituisce *True* se ogni elemento di una sequenza è *True*.

# Cicli

Capiterà spesso di dover controllare se si verificano determinate condizioni, e di variare di conseguenza il comportamento del programma. Le **istruzioni condizionali** servono proprio a questo. La forma più semplice di istruzione condizionale è l’istruzione *if* (“se” in inglese):

if x > 0:  
 print(‘x è positivo’)

Non c’è limite al numero di istruzioni che possono essere scritte nel corpo, ma deve sempre essercene almeno una.  
Talvolta può servire che il corpo sia privo di istruzioni (di solito quando c’è del codice ancora da scrivere); in questo caso potete usare l’istruzione , che serve solo da segnaposto temporaneo e nulla più.

Se vogliamo eseguire un’operazione in caso la condizione dell’if non sia valida, inseriamo un *else*:

if x > 0:  
 print(‘x è positivo’)  
else  
 print(‘x non è positivo’)

Se vogliamo eseguire un’operazione in caso di diverse condizioni, inseriamo degli *elif* (abbreviazione per else if) e un *else* (non obbligatorio, ma se c’è è uno solo ed è per ultimo):

if x > 0:  
 print(‘x è positivo’)  
elif x < 0:  
 print(‘x è negativo’)  
else  
 print(‘x è uguale a 0’)

Le condizioni vengono controllate nell’ordine dall’alto al basso: se la prima è falsa, viene controllata la seconda e così via. Non appena una condizione risulta vera, viene eseguito il ramo corrispondente e l’istruzione termina. Anche se risultassero vere altre condizioni successive, viene eseguita sempre e soltanto la prima che risulta vera.

Si può anche inserire un’istruzione condizionale nel corpo di un’altra istruzione condizionale. Anche se l’indentazione delle istruzioni aiuta ad evidenziare la struttura, le **condizioni nidificate** diventano rapidamente difficili da leggere, quindi è meglio usarle con moderazione.

Un altro tipo di ciclo, quello legato all’**iterazione** (come viene chiamata la ripetizione nella programmazione), è il ciclo *while*. Fino a quando la condizione del *while* è verificata, il ciclo continua ad eseguire le istruzioni al suo interno.

x = 0  
while x < 10:  
 print x  
 x = x +1

Il corpo del ciclo deve cambiare il valore di una o più variabili in modo che la condizione prima o poi diventi falsa e il ciclo abbia termine. Altrimenti, il ciclo verrebbe ripetuto continuamente, dando luogo ad un **ciclo infinito**.

Se si vuole stabilire il momento in cui è necessario terminare un ciclo solo durante il flusso di esecuzione, si può usare l’istruzione *break* per interrompere il ciclo e saltarne fuori.

x = 0  
while x < 10:  
 print x  
 if x == 6:  
 break  
 x = x +1

# Input da tastiera

In Python esiste una funzione predefinita chiamata che sospende il programma ed attende che l’utente scriva qualcosa. Quando l’utente preme il tasto *Invio/Start*, il programma riprende e restituisce quello che l’utente ha inserito, come stringa:

testo = input(‘come ti chiami?\n’)

Restituisce una domanda, e poi assegna al valore *testo* ciò che viene inserito dall’utente. (\n manda a capo la stringa)

# Files

Un file di testo è un una sequenza di caratteri salvata su un dispositivo permanente come un disco fisso, una memoria flash o un CD-ROM. La funzione predefinita *open* richiede come parametro il nome di un file e restituisce un **oggetto file** che potete utilizzare per questo scopo (il secondo parametro definisce se il file è in sola lettura ‘r’, se ci si può scrivere sopra ‘w’) (il predefinito, se non specificato, è in modalità lettura ‘r’) (Se il file esiste già, l’apertura in modalità scrittura lo ripulisce dai vecchi dati e riparte da zero, quindi fate attenzione! Se non esiste, ne viene creato uno nuovo.).

file1 = open(‘words.txt’, ‘w’)

L’oggetto file comprende alcuni metodi, tra cui:  
- .*readline()* : legge i caratteri da un file finché non giunge ad un ritorno a capo, e restituisce il risultato sotto forma di stringa.   
- .*readlines()* : legge tutti i caratteri di un file.  
- .*read()* : restituisce il contenuto di un file.  
- .*clore()* : per chiudere il file.  
- *.tell()* : per avere la posizione del file.  
- *.write(…)* : per scrivere una riga in un file.  
- *.writelines(…)* : per scrivere una serie di righe in un file.  
- *.seek(…)* : per cambiare la posizione di un file.  
- *.writelines()* : per scrivere una serie di righe in un file.  
- *.writelines()* : per scrivere una serie di righe in un file.  
- *.writelines()* : per scrivere una serie di righe in un file.  
- *.writelines()* : per scrivere una serie di righe in un file.

# Modulo math

Python è provvisto di un modulo matematico che comprende buona parte delle funzioni matematiche d’uso frequente.

Per importarlo

import math

Alcune funzioni sono:  
- math.log10(n) : per il logaritmo in base 10.   
- math.log(n) : per il logaritmo naturale in base *e*.   
- math.log(n[, base]) : per il logaritmo in base *base*.  
- math.sin(n) math.cos(n) math.tan(n) math.asin(n) math.acos(n) math.atan(n): per funzioni trigonometriche.  
- math.dist(p,q) : ritorna la distanza euclidea tra due punti.   
- math.degrees(n): converte l’angolo in gradi da radianti.  
- math.radians(n): converte l’angolo in radianti da gradi.  
- math.pi : per il valore di *pi greco*, accurato a 15 cifre decimali.   
- math.e : per il valore di *e*, accurato a 15 cifre decimali.  
- math.tau : per il valore di *tau*, accurato a 15 cifre decimali.  
- math.inf : per il valore di + infinito.  
- math.nan : per il valore di *NaN*.  
- math.e : per il valore di *e*, accurato a 15 cifre decimali.  
- math.sqrt(n) : per radice quadrata.   
- math.cbrt(n) : per radice cubica.  
- math.exp(n) : per en.  
- math.exp2(n) : per 2n.  
- math.pow(n,m) : per nm.   
- math.factorial(n) : per il fattoriale di un numero.  
- math.ceil(n) : per il più piccolo intero maggiore o uguale a n.   
- math.floor(n) : per il più grande intero minore o uguale a n.  
- math.comb(n,k) : per combinazione senza ripetizione, ovvero scelta di k cose tra n senza ripetizione né ordine.   
- math.isfinite(n) : ritorna *True* se non è infinito o NaN, altrimenti *False*.  
- math.isinf(n) : ritorna *True* se è infinito positivo o negativo, altrimenti *False*.  
- math.isnan(n) : ritorna *True* se NaN (Not a Number), altrimenti *False*.  
- math.perm(n,k) : per permutazione senza ripetizione, ovvero scelta di k cose tra n senza ripetizione e con ordine.   
- math.gamma(n) : ritorna la funzione gamma ad *n*.

# Modulo os

Python è provvisto di un modulo os, o operating system, che permette di interagire col sistema operativo.

Per importarlo

import os

Alcune funzioni sono:  
- *os.chdir(…)* per cambiare la directory di lavoro corrente (working directory).  
- *os.getcwd()* per ottenere la directory di lavoro corrente (working directory).  
- *os.remove(path)* per rimuovere un file ad un determinato path.  
- *os.rename(path1, path2)* per rinominare un file/directory.  
- *os.mkdir(path)* per creare una directory ad un determinato path.  
- *os.listdir(path)* per creare una lista con i nomi di tutti i files in una determinata directory.  
- *os.rmdir(path)* per rimuovere una cartella (directory) ad un determinato path.  
- *os.path.exists(name)* per verificare se un file in una cartella esiste o no.  
- *os.path.getsize(name)* per ottenere la dimensione di un file.  
- *os.path.isdir(name)* per verificare se è una directory.  
- *os.path.isfile(name)* per verificare se è un file.  
- *os.path.getsize(name)* per ottenere la dimensione di un file.

# Libreria Numpy

NumPy is a Python library, used for working with arrays, and it is short for "Numerical Python". It also has functions for working in domain of linear algebra, fourier transform, and matrices.

In Python we have lists that serve the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is up to 50x faster than traditional Python lists. NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. Also it is optimized to work with latest CPU architectures.

Per installarla:

pip install numpy

Per importarla (di solito con l’alias *np*):

import numpy as np

In caso sia necessario sapere la versione di numpy:

print(np.\_\_version\_\_)

Si può creare un array nel seguente modo (viene definito come *numpy.ndarray* *type*) (possono essere passati dentro la parentesi diversi tipi di oggetti, come tuple, liste, dizionari…) (gli array di numpy sono formati da elementi dello stesso tipo, non funzionano come liste o dizionari che possono avere oggetti di tipologie diverse):

arr = np.array(1, 2, 3, 4, 5)

Le dimensioni di un array possono essere molto diverse:   
- 0, o array scalari;  
- 1, o array unidimensionali;  
- array multidimensionali.

arr = np.array(42) # 0 dimensioni  
arr = np.array([1, 2, 3, 4, 5]) # 1 dimensione   
arr = np.array([[1, 2, 3], [4, 5, 6]]) # 2 dimensioni  
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]]) # 3 dimensioni

Si può anche imporre la dimensione di un array nel seguente modo:

arr = np.array([1, 2, 3, 4, 5], ndmin = 5) # 5 dimensione

Per richiamare un elemento in un array monodimensionale, la procedura è la solita (partendo da 0 come al solito) (anche con gli array si può utilizzare il negative indexing, con numeri negativi contando a ritroso):

arr[0] # 1

Per un array multidimensionale, la situazione cambia (e così via per più dimensioni):

arr = np.array([[1, 2, 3], [4, 5, 6]]) # 2 dimensioni  
arr[1, 2] # 3° elemento 2° riga, quindi 6

Lo slicing funziona come al solito (ma sempre occhio alle dimensioni dell’array) (il terzo punto è lo step dello slicing, dice ogni quanti prendere un valore, default 1) (sempre possibile considerare i negativi):

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
arr[1, 1:3:2] # 7,9

Si può prestabilire il tipo di un array al momento della sua creazione:

arr = np.array([1, 2, 3, 4], dtype='S')  
arr.dtype

I tipi possibili sono:  
- integer (i)  
- boolean (b)  
- unsigned integer (u)  
- float (f)  
- complex float (c)  
- timedelta (m)  
- datetime (M)  
- object (O)  
- string (S)  
- unicode string (U)  
- fixed chunk of memory for other type (v, sta per void )

Allo stesso modo si può cambiare il tipo di un array:

newarr = arr.astype('i')

Tra le varie funzioni presenti:  
- *.ndim()* per verificare la dimensione di un array.  
- *.copy()* per creare una coppia di un array (cambiamenti ad uno dei due elementi non portano a cambiamenti all’altro) (“possiedono” i dati).  
- *.view()* per avere una vista dell’array (cambiamenti ad uno dei due elementi portano a cambiamenti all’altro) (non “possiedono” i dati).  
- *.base()* per verificare se un array “possiede” o no i dati (*None* se li possiede, diverso da *None* se non li possiede).  
- *.shape* per verificare la conformazione di un array (ritorna una tupla con numeri di elementi in ogni layer).  
- *.dtype* per verificare il tipo di un array.  
- *.astype('i')* per modificare il tipo di un array.  
- *.reshape(2, 3, 2)* per modificare la conformazione di un array (ad esempio da 1D a 3D) (il numero totale di elementi dev’essere uguale al prodotto di tutti i numeri nella parentesi del reshape: es. 12 = 2\*3\*2) (si può avere una dimensione non specificata, a cui diamo il valore -1: Numpy calcola da solo la dimensione nel caso) (si può anche convertire un array multidimensionale in un array 1D mettendo solamente -1).  
- *np.nditer(arr, ::2)* per iterare gli elementi di un array (::2 facoltativo, per prendere un elemento ogni 2).  
- *np.ndenumerate(arr)* per iterare gli elementi di un array aggiungendo anche l’indice (::2 facoltativo, per prendere un elemento ogni 2).  
- *np.concatenate((a,b), axis = 1)* per fare una join tra due array lungo l’asse 1 (se non specificato mette in asse 0).  
- *np.stack((a,b), axis = 1)* per fare una join tra due array lungo l’asse 1 (rispetto a *concatenate,* crea un nuovo asse).  
- *np.hstack((a,b))* per fare una join tra due array lungo le righe.  
- *np.vstack((a,b))* per fare una join tra due array lungo le colonne.  
- *np.dstack((a,b))* per fare una join tra due array lungo la profondità.  
- *np.array\_split(a, 3, axis=1)* per splittare un array in 3 parti (si può specificare l’asse su cui eseguire lo split).  
- *np.hsplit(a, 3)* per splittare un array in 3 parti lungo le righe.  
- *np.where(a == 3)* per cercare un elemento in un array, ed ottenere l’indice (ritorna tutti gli indici in cui è presente il valore).  
- *np.searchsorted(a, [3,4], side='right')* per trovare dove un elemento in un array andrebbe inserito per mantenere gli elementi in ordine (primo indice, da sinistra a destra, in cui il valore ricercato non è più grande del successivo) (si può scegliere se partire da destra o sinistra, default sinistra) (si possono mettere anche più valori insieme).  
- *np.sort(a)* per ordinare gli elementi in un array (numeri, parole, booleani…).  
- *np.add(x,y)* fa la stessa cosa dell’*append* per le liste, le mette in successione.  
- *np.subtract(x,y)* per sottrarre gli elementi in y dagli elementi in x.  
- *np.multiply(x,y)* per moltiplicare gli elementi in x con gli elementi di y.  
- *np.divide(x,y)* per dividere gli elementi in x con gli elementi in y.  
- *np.mod(x,y)* per ottenere il resto degli elementi in x agli elementi in y.  
- *np.remainder(x,y)* per ottenere il resto degli elementi in x agli elementi in y.  
- *np.divmod(x,y)* per ottenere il risultato della divisione e il resto per gli elementi in x rispetto agli elementi in y.  
- *np.absolute(x)* per ottenere il modulo degli elementi di x.  
- *np.abs(x)* per ottenere il modulo degli elementi di x.  
- *np.trunc(x)* per rimuovere i decimali e ottenere il float più vicino per gli elementi di x.  
- *np.fix(x)* per rimuovere i decimali e ottenere il float più vicino per gli elementi di x.  
- *np.round(x, 3)* per arrotondare gli elementi di x, in base al numero di decimali specificato.  
- *np.floor(x)* per arrotondare gli elementi di x all’intero minore.  
- *np.ceil(x)* per arrotondare gli elementi di x all’intero maggiore.  
- *np.log2(x)* per calcolare il logaritmo in base 2 degli elementi dell’array.  
- *np.log10(x)* per calcolare il logaritmo in base 10 degli elementi dell’array.  
- *np.log(x)* per calcolare il logaritmo in base e (o logaritmo naturale) degli elementi dell’array.  
- *np.exp(x)* per calcolare l’esponenziale dell’array (earray).  
- *np.exp2(x)* per calcolare il 2 all’array (2array).  
- *np.power(3,x)* per calcolare la potenza dell’array (3array).  
- *np.sum([x,y], axis=1)* per sommare tutti gli elementi dell’array in un solo risultato (si può specificare l’asse).  
- *np.cumsum(x)* per fare la somma cumulata degli elementi dell’array.  
- *np.prod([x,y], axis=1)* per moltiplicare tutti gli elementi di uno o più array in un unico risultato (si può specificare l’asse).  
- *np.cumprod(x)* per fare il prodotto cumulato degli elementi dell’array.  
- *np.sum(x, n=1)* per sottrarre tutti gli elementi dell’array (n = 1 in default, altrimenti si possono sottrarre più volte se specificato).  
- *np.lcm(x,y)* per trovare il minimo comune multiplo di due array.  
- *np.gcd(x,y)* per trovare il massimo comune denominatore di due array.  
- *np.pi* per il pi greco.  
- *np.sin(x)* per trovare il seno di un array.  
- *np.deg2rad(x)* per trovare i gradi partendo dai radianti di un array.  
- *np.rad2deg(x)* per trovare i radianti partendo dai gradi di un array.  
- *np.arcsin(x)* per trovare il valore degli angoli partendo da valori di seno/coseno/tangente di un array.  
- *np.hypot(x,y)* per trovare l’ipotenusa di un triangolo rettangolo partendo da due cateti.  
- *np.unique(x)* per trovare i valori unici in un array.   
- *np.union1d(x,y)* per trovare l’unione di due set di array.  
- *np.intersect1d(x,y)* per trovare l’intersezione di due set di array.  
- *np.setdiff1d(x,y)* per trovare le differenze di due set di array (valori del primo non presenti nel secondo).  
- *np.setxor1d(x,y)* per trovare valori presenti o nel primo o nel secondo array, ma non in entrambi.  
- *np.zeros(10)* per creare un array formato da 10 0.  
- *np.ones(10)* per creare un array formato da 10 1.  
- *np.full((3,5), 3.14)* per creare un array 3x5 riempito tutto da valore 3.14.  
- *np.arange(0,20,2)* per creare un array con una sequenza da 0 a 20 ogni due numeri.  
- *np.linspace(0,2,5)* per creare un array con una sequenza che divide il range 0-2 in esattamente (5-1) spazi.  
- *np.eye(3)* per creare un array 3x3 come matrice identità (diagonale pari a 1, il resto pari a 0).  
- *np.union1d(x,y)* per trovare l’unione di due set di array.  
- *np.min(x)* per trovare il minimo di un array.  
- *np.max(x)* per trovare il massimo di un array.  
- *np.mean(x)* per trovare la media di un array.  
- *np.std(x)* per trovare la deviazione standard di un array.  
- *np.var(x)* per trovare la varianza di un array.  
- *np.argmin(x)* per trovare l’indice del minimo di un array.  
- *np.argmax(x)* per trovare l’indice del massimo di un array.  
- *np.median(x)* per trovare la mediana di un array.  
- *np.percentile(x, 75)* per trovare il percentile (qua 75, da specificare) di un array.

# Modulo random

Python è provvisto di un modulo random (dalla libreria Numpy) che crea elementi random. Se non installato, prima va installato Numpy.

Per importarlo

import random

Alcune funzioni sono:  
- *random.seed(a)* per scegliere un seed random (a per imporlo).  
- *random.random()* per un float random compreso tra 0 e 1.   
- *random.randrange(start, stop, step)* per un numero random in un range (si può definire start, default 0, stop, e step, l’incremento default 1).   
- *random.randint(start, stop)* per un intero random compreso tra due valori da specificare obbligatoriamente.   
- *random.choice(…)* per scegliere un elemento random in un oggetto (lista, dizionario, tupla…).   
- *random.choices(…, k)* per scegliere k elementi random in un oggetto (lista, dizionario, tupla…).  
- *random.shuffle(…)* per mischiare un oggetto (lista, dizionario, tupla…).   
- *random.sample(…, k)* per un campione di k elementi presi random da un oggetto (lista, dizionario, tupla…).  
- *random.uniform(a,b)* per un float random compreso tra due numeri dati  
- *random.betavariate()* per un random float tra 0 e 1 basato sulla distribuzione Beta.  
- *random.expovariate()* per un random float basato sulla distribuzione Esponenziale.  
- *random.gammavariate()* per un random float basato sulla distribuzione Gamma.  
- *random.gauss()* per un random float basato sulla distribuzione Gaussiana.  
- *random.lognormvariate()* per un random float basato sulla distribuzione log-normale.  
- *random.normalvariate()* per un random float basato sulla distribuzione normale.  
- *random.vonmisesvariate()* per un random float basato sulla distribuzione di von Mises.  
- *random.paretovariate()* per un random float basato sulla distribuzione di Pareto.  
- *random.weibullvariate()* per un random float basato sulla distribuzione di Weibull.

# Libreria Pandas

Pandas is a Python library used for working with data sets: it has functions for analyzing, cleaning, exploring, and manipulating data.

Per installarla:

pip install pandas

Per importarla (di solito con l’alias pd):

import *pandas* as pd

In caso sia necessario sapere la versione di pandas:

print(pd.\_\_version\_\_)

Si possono creare oggetti semplice, come le serie (simili ad una colonna in una tabella, o ad un array monodimensionale) (si possono anche specificare gli indici da usare, se non inseriti di default mette il solito ordine da 0 in avanti) (si possono creare a partire da diversi oggetti, come liste, dizionari…):

a = [1, 7, 2]  
myvar = pd.Series(a, index = ["x", "y", "z"])  
  
calories = {"day1": 420, "day2": 380, "day3": 390}  
myvar = pd.Series(calories)

Un insieme di serie (o colonne) può essere definito come dataframe, ovvero una sorta di tabella contenente dati; si tratta di una struttura dati bidimensionale (come un array a due dimensioni) (si possono definire gli indici delle righe, o osservazioni, con index dentro la parentesi)

):

data = {"calories": [420, 380, 390], "duration": [50, 40, 45]}  
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])

Per individuare una o più righe, ci sono modi diversi:

df.loc[[0,1]] # il risultato è un insieme di serie  
df.loc[0, 3] # il risultato è un dataframe  
df.loc["day2"] # in base all’indice della riga  
df.iloc[[0,1]] # il risultato è un insieme di serie  
df[0:3] # le prime 3 righe, dalla 0 alla 2  
df[:5] # per le prime 5 righe  
df[5:] # per le ultime 5 righe  
df[::2] # per una riga ogni 2  
df[::-1] # per le righe in ordine contrario  
df[‘id1’:’id4’] # dal paziente id1 al paziente id4

La differenza principale tra *.loc* e *.iloc* è che *.loc* guarda il nome della riga (ad esempio, la 4° riga può avere come indice ‘2’ o ‘ciao’), mentre *.iloc* guarda la posizione della riga (quindi giusto col numero).

Per individuare una colonna:

df.area # per colonne che si chiamano con una stringa, e che non sono nomi in conflitto (class, and…)  
df[‘area’]  
df[[‘nomi’, ‘cognomi’]]  
df[df.density > 100] # con condizione  
df[df[‘density’] > 100] # con condizione

Per selezionare delle celle in base a condizione:

data.loc[data[‘density’] > 100, ['pop', 'density']]  
df.loc[dates[0], "A"]  
data.iloc[0, 2]  
df.at[dates[0], "A"]  
df.iat[1, 1]  
df[df["E"].isin(["two", "four"])] # righe dove E ha valore two o four

Per modificare una cella:

df.loc[7, 'Duration'] = 45 # modifica la riga 7, colonna ‘Duration’

Per importare un file csv come Data Frame:

df = pd.read\_csv('data.csv')

Per importare un file di Excel come Data Frame:

pd.read\_excel("foo.xlsx", "Sheet1", index\_col=**None**, na\_values=["NA"])

Per importare un file JSON (stesso formato di un dizionario di Python) come Data Frame:

df = pd.read\_json('data.json')

data = {"Duration":{"0":60, "1":60, "2":60, "3":45, "4":45, "5":60},  
"Pulse":{"0":110, "1":117, "2":103, "3":109, "4":117, "5":102},  
"Maxpulse":{"0":130, "1":145, "2":135, "3":175, "4":148, "5":127},  
"Calories":{"0":409, "1":479, "2":340, "3":282, "4":406, "5":300}}  
df = pd.DataFrame(data)

In generale, si possono specificare molti parametri:

df = pd.DataFrame(data, index = …, columns = …, dtype = …, )

Per esportare un dataframe come file csv:

df.to\_csv("foo.csv")

Per esportare un dataframe come file di Excel:

df.to\_excel("foo.xlsx", sheet\_name="Sheet1")

Per modificare il numero di righe mostrate quando si richiede di stampare un dataframe pandas:

pd.options.display.max\_rows = 9999  
print(df)

Alcune funzioni sono:  
- *df.fillna(x, inplace = True)*: per sostituire i valori mancanti nelle varie righe con ciò che si vuole (x in questo caso) (inplace = *True* per modificare il Data Frame originale) (si può mettere una variabile con valori di interesse come x, per imputare media, mediana, moda…).  
*- df["Calories"].fillna(130, inplace = True)* : per sostituire i valori mancanti in una particolare colonna (‘Calories’) nelle varie righe con ciò che si vuole (130 in questo caso) (inplace = *True* per modificare il Data Frame originale) (si può mettere una variabile con valori di interesse, tipo media, mediana, moda…).  
- *df.*dropna*(inplace = True)*: per togliere le righe con valori mancanti (inplace = *True* per modificare il Data Frame originale).  
*- df.head(10)*: per osservare le prime 10 righe (default 5 se non specificato).  
*- df.tail(20)*: per osservare le ultime 20 righe (default 5 se non specificato).  
- *df.index()*: per sapere il range e il tipo degli indici di un dataframe.   
- *df.columns()*: per sapere i nomi delle colonne di un dataframe.  
- *df.rename(columns = {'Literacy %':'Literacy percentage'}, inplace=True)* per rinominare una o più colonne.   
- *df.values()*: per ottenere le celle di un dataframe.  
*- df.info()*: per informazioni sul Data Frame (n° righe e colonne, nome e tipo delle colonne, quantità di valori nulli per colonna, memoria usata dal Data Frame).  
- *df.dtypes* per capire i tipi delle colonne di un dataframe  
- *df.describe()* per avere statistiche descrittive (conto, media, deviazione standard, minimo, massimo, 1° 2° e 3° quartile, tipo della colonna) di tutte le colonne numeriche di un dataframe.  
- *df[‘anno’,’guadagni’].describe()* per avere statistiche descrittive (conto, media, deviazione standard, minimo, massimo, 1° 2° e 3° quartile, tipo della colonna) di alcune particolari colonne numeriche di un dataframe.  
- *df.sort\_values(‘nome’, ascending=False)* per ordinare le righe di un dataframe in base ad una colonna.  
- *df.sort\_index(axis=1, ascending=False)* per ordinare le righe in base all’indice del dataframe.   
­- *pd.to\_datetime(df['Date'])* per convertire una non data in formato data.  
- *df.duplicated()* mostra gli indici delle righe doppie.  
- *df.drop\_duplicates(inplace = True)* per eliminare le righe doppie (inplace = *True* per modificare il Data Frame originale).  
- *df.corr()* per la matrice di correlazione fra colonne (solo per colonne numeriche).   
*- df.isnull(10)*: per valore booleano se ci sono valori mancanti.  
*- df.notnull(10)*: per valore booleano se non ci sono valori mancanti.  
- *pd.merge(df1, df2, on='Employee\_id')* per fare un join di due dataframe in base ad una colonna.  
- *df.groupby('Employee\_name').sum()* per raggruppare le righe di un dataframe in base ad una colonna (il .sum() facoltativo, si può aggiungere una funzione del genere per fare un calcolo ulteriore sulla colonna, tipo size()…).  
- *pd.concat([df1, df2])* per concatenare due dataframe con le stesse colonne  
- *df.T* per trasporre i dati  
- *df.mean()* per calcolare la media delle colonne numeriche di un dataframe.  
- *df["grade"].cat.set\_categories(["bad", "medium", "good"])* per creare una variabile categorica con 3 categorie.  
- *df["grade"].cat.rename\_categories(["bad", "medium", "good"])* per rinominare una variabile categorica con 3 categorie.  
- *df["grade"].cat.set\_categories(["bad", "medium", "good"])* per creare una variabile categorica con 3 categorie.  
- *df.index.is\_unique* per trovare se gli indici delle righe sono tutti unici o no  
- *df.index.duplicated()* per trovare se ogni indice delle righe è unico o no  
- *df.loc[~df.index.duplicated(), :]* per trovare ed escludere le righe doppie  
- *df.columns.is\_unique* per trovare se i nomi delle colonne sono tutti unici o no  
- *df.astype({'col1': 'int32'})* per modificare il tipo di una o più colonne (tipi possibili: ‘category’, ‘int64’, ‘float64’, ‘object’, ‘float’, ‘bool’, ‘Datetime64’…)

# Libreria Matplotlib

Matplotlib is a low level graph plotting library in python that serves as a visualization utility.

Per installarla:

pip install matplotlib

Per importarla:

import matplotlib

In caso sia necessario sapere la versione di pandas:

print(matplotlib.\_\_version\_\_)

# Modulo pyplot

Il modulo pyplot è un modulo appartenente alla libreria matplotlib.

Per importarlo

import matplotlib.pyplot as plt

Alcune funzioni generali:  
- *plt.show()* per mostrare il grafico creato con le righe di codice precedente; sempre alla fine di tutto il codice del grafico.  
- *plt.xlabel("variabile x")* per l’intestazione della variabile alle ascisse.  
- *plt.ylabel("variabile y")* per l’intestazione della variabile alle ordinate.  
- *plt.xlim(-1, 11)* per limiti dell’asse x (se si vuole un asse nel verso contrario, si può mettere l’ordine al contrario).   
- *plt.ylim(-1.5, 1.5)* per limiti dell’asse y (se si vuole un asse nel verso contrario, si può mettere l’ordine al contrario).   
- plt.axis([-1, 11, -1.5, 1.5]) per specificare i limiti di entrambi gli assi contemporaneamente (se invece di valori si specifica 'equal', i due assi sono proporzionati come unità).  
- *plt.title("titolo grafico", loc = 'left')* per il titolo del grafico (la posizione al centro in default).  
- *plt.grid(axis = 'x', color = 'green', linestyle = '--', linewidth = 0.5)* per una griglia aggiunta al grafico (si può specificare l’asse o no, e modificare le solite impostazioni grafiche).   
- *plt.subplot(1, 2, 1)* per visualizzare più grafici nella stessa schermata (il primo valore è il numero di righe, il secondo il numero di colonne, il terzo è per l’ordine dei grafici) (va messo appena prima del plt.plot() del singolo grafico che interessa)  
- *plt.suptitle("titolo totale")* titolo generale in caso di subplot (dopo i plt.plot(), appena prima del plt.show() ).   
- *plt.legend(title = "titolo legenda", loc='upper left', frameon=False, center', ncol=2, ['first', 'second'])* per aggiungere la legenda ad un grafico che la richiede (titolo facoltativo) (loc per la posizione) (frameon per bordo) (ncol per disporlo su più colonne) (tra le parentesi quadre per i nomi da scegliere, se van cambiati da quanto già specificato)

Per modificare tutte le scritte (titolo, assi, …):

font1 = {'family':'serif','color':'blue','size':20}  
plt.title("Sports Watch Data", fontdict = font1)

Per **grafico x/y semplice** (dati due array, ognuno con coordinate di x e y di un tot di punti, plot unisce i vari punti in una spezzata) ('o' per non fare la spezzata, ma mettere solo i punti nel grafico) (se non si specifica una delle due coordinate dei vari punti, da dei valori default 0,1,2… in base alla lunghezza dell’altra coordinata) (per stili diversi della linea, ls = '-' o 'solid' linea solida default, ':' o 'dotted' linea a punti, '--' o 'dashed' linea spezzata, '-.' o 'dashdot' linea con punti e spezzata, ‘’ o ‘ ’ o ‘None’ per niente linea) (per colori diversi della riga, specificato con color/c, 'r' rosso, 'g' verde, 'b' blu, 'c' ciano, 'm' magenta, 'y' gialla, 'k' nera, 'w' bianca, o nomi completi con stringa completa) (per dimensione punti ms = …) (per colore bordi dei punti mec = '…' con colori “soliti”) (per colore interno dei punti mfc = '…' con colori “soliti”) (per larghezza linea, linewidth) (label per cambiare il nome definito del contenuto del grafico):

xpoints = np.array([1, 8, 4, 7])  
ypoints = np.array([3, 10, 5, 9])  
plt.plot(xpoints, ypoints, 'o', ls = ‘-’, ms = 20, mec = 'r', mfc = 'b', linewidth = '20.5', label=’nome’)  
plt.show()

Si può aggiungere un marcatore per i vari punti, per indicarli oltre alla spezzata :  
- 'o' un cerchio  
- '\*' una stella  
- '.' un punto   
- ',' un pixel   
- 'x' / 'X' una X  
- 'v' / '^' / '<' / '>' un triangolo  
- 'p' un pentagono  
- 'P' / '+' un +  
- 's' un quadrato  
- 'D' / 'd' un diamante  
- '1' / '2' / '3' / '4' Un tri  
- 'H' / 'h' un esagono  
- '|' una linea verticale  
- '\_' una linea orizzontale

plt.plot(xpoints, ypoints, marker = 'o')  
plt.show()

Per mettere più linee in un grafico:

plt.plot(x1,y1)  
plt.plot(x2,y2)  
# oppure  
plt.plot(x1,y1,x2,y2)  
plt.show()

Per **scatterplot**

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])  
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])  
plt.scatter(x, y, color = ‘green’)  
plt.show()

Per altre modifiche grafiche (colore, dimensione punti, trasparenza con alpha, dimensioni punti) (per la colormap la standard è viridis; spesso si usa grayscale\_cmap(cmap) ovvero la scala di grigi):

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])  
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])  
colors = np.array([0, 10, 20, 30, 40, 45, 50, 55, 60, 70, 80, 90, 100])  
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])  
plt.scatter(x, y, c=colors, cmap='viridis', s=sizes, alpha=0.5)  
plt.colorbar()  
plt.show()

Per **barplot** (con variabile numerica e variabile categorica sulle x) (con impostazioni su colore singolo, larghezza barre, )

x = np.array(["A", "B", "C", "D"])  
y = np.array([3, 8, 1, 10])  
plt.bar(x,y, color = ‘red’, width = 0.1)  
plt.show()

Se si vuole fare barplot orizzontale (height per larghezza barre invece di width):

plt.barh(x, y)

Per **istogramma** (mostra la distribuzione di frequenza di una variabile) (bins per il numero di barre) (alpha per la trasparenza) (edgecolor per il colore dei bordi delle barre)

x = np.random.normal(170, 10, 250)  
plt.hist(x, bins=30, alpha=0.5, edgecolor = ‘red’, color = ‘green’)  
plt.show()

Si possono anche mettere più istogrammi nello stesso grafico

plt.hist(x1)  
plt.hist(x2)  
plt.hist(x3)

Per **pie chart** (grafici a torta, per vedere percentuali di variabili categoriali) (startangle per l’angolo di partenza della prima categoria, con 0 come default in alto) (explode per far uscire una o più categorie dal grafico, per dare risalto)

y = np.array([35, 25, 25, 15])  
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]  
myexplode = [0.2, 0, 0, 0]  
mycolors = ["black", "hotpink", "b", "#4CAF50"]  
plt.pie(y, labels = mylabels, startangle = 90, explode = myexplode, colors = mycolors)  
plt.show()

Per **density plot / contour plot** (grafico 3D che fa vedere la densità di due distribuzioni) (col numero specifico il numero di linee di densità che voglio disegnare) (cmap per la colormap)

x = np.linspace(0, 5, 50)  
y = np.linspace(0, 5, 40)  
X, Y = np.meshgrid(x, y)  
Z = f(X, Y)  
plt.contour(X, Y, Z, 20, cmap='RdGy')  
plt.colorbar()

# Modulo seaborn

Spesso per migliorare i grafici creati con matplotlib, si può combinare con Seaborn per ottenere migliori visualizzazioni: basta solo importare seaborn, mantenendo tutto il resto del codice identico, per vedere i risultati

import seaborn as sns

Tuttavia, si può anche usare il modulo per creare grafici interessanti.

Per una stima smooth di una distribuzione di densità, ovvero una **kernel density estimate** (rispetto a usare un istogramma), si può fare così con seaborn:

sns.kdeplot(x, shade=True)

Si può combinare una kde (kernel density estimate) col corrispettivo istogramma nel seguente modo:

sns.distplot(data['x'])

Per una **distribuzione joint** di due distribuzioni di densità di due variabili, ci sono diversi modi:

sns.kdeplot(data); # 2 dimensional kde  
sns.jointplot("x", "y", data, kind='kde') # 2 dimensional kde con marginali smooth  
sns.jointplot("x", "y", data, kind='hex') # 2 dimensional kde con istogrammi marginali  
sns.jointplot("x", "y", data=tips, kind='reg'); # 2 dimensional joint distribution plot con rette di regressione e marginali

Per i **pair plots** di più distribuzioni insieme:

sns.pairplot(dataframe, hue='species', size=2.5);  
sns.PairGrid(data, vars=['age', 'split\_sec', 'final\_sec', 'split\_frac'], hue='gender', palette='RdBu\_r')

Per i grafici di funzioni di tipo **categorico**:

sns.factorplot("day", "total\_bill", "sex", data=tips, kind="box")

Per **barplot**:

sns.factorplot("year", data=planets, aspect=2, kind="count", color='steelblue', hue='method')

Per i **violin plot**:

sns.violinplot("gender", "split\_frac", data=data, palette=["lightblue", "lightpink"], hue="gender", split=True, inner="quartile");

Per scatter con regressione lineare:

sns.lmplot('final\_sec', 'split\_frac', col='gender', data=data, markers=".", scatter\_kws=dict(color='c'))

# Accesso dati in modo alternativo

Quando si usa Google Colab, può essere utile prenderli direttamente da Google Drive (ad esempio da cartelle condivise): in questo caso, si può fare nel seguente modo

from google.colab import drive  
drive.mount('/content/drive')

df = pd.read\_csv("/content/drive/MyDrive/df.csv")

with open("/content/drive/MyDrive/file.json", "r") as f:  
 file\_json = json.load(f)

Invece, per importare dati da github, si può fare nel seguente modo:

!git clone <https://github.com/andrelp99/data.csv>

# Modulo tweepy

Per installarla:

pip install tweepy #version 4.12.0

Per importarla:

import tweepy

In caso sia necessario sapere la versione di pandas:

print(tweepy.\_\_version\_\_)

Per installare la nuova versione di tweepy:

!pip install --upgrade tweepy

# Modulo glob

Il modulo glob è utilizzato per importare in modo ricorsivo una serie di file presenti in una particolare cartella.

Per importare il modulo

import glob

Per creare la lista di file importati (la parte che cambia di file in file è quella al posto dell’asterisco):

fileL = glob.glob('C:\\test\_radiom\\t\*\\\*.dcm')

# Modulo Scikit-Learn

There are several Python libraries that provide solid implementations of a range of machine learning algorithms. One of the best known is Scikit-Learn, a package that provides efficient versions of a large number of common algorithms. The benefit of his uniformity is that once you understand the basic use and syntax of Scikit-Learn for one type of model, switching to a new model or algorithm is very straightforward.

I passaggi da intraprendere sono i seguenti:  
1. Scegliere una classe di modelli da importare;   
2. Scegliere gli iperparametri del modello;  
3. Predisporre i dati in formato matrice di features;  
4. Fittare il modello ai dati col metodo *fit();*  
5. Applicare il modello a nuovi dati (per supervised learning, spesso col *predict()*; per unsupervised learning, spesso con *transform()* o *predict()*).

Per installare scikit-learn:

pip install scikit-learn

Si può usare scikit-learn per alcune operazioni di preprocessing:  
- standardizzazione (rendere tutte le variabili con varianza pari a 1 e media pari a 0):

from sklearn import preprocessing  
X\_train = np.array([[ 1., -1., 2.], [ 2., 0., 0.], [ 0., 1., -1.]])  
scaler = preprocessing.StandardScaler().fit(X\_train)  
X\_scaled = scaler.transform(X\_train)

X\_scaled.mean(axis=0)  
X\_scaled.std(axis=0)

- scalare i valori in un range specificato:

from sklearn import preprocessing  
X\_train = np.array([[ 1., -1., 2.], [ 2., 0., 0.], [ 0., 1., -1.]])  
min\_max\_scaler = preprocessing.MinMaxScaler()  
X\_train\_minmax = min\_max\_scaler.fit\_transform(X\_train)

X\_scaled.mean(axis=0)  
X\_scaled.std(axis=0)

- dividere un dataset in train e test:

**from** **sklearn.model\_selection** **import** train\_test\_split  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

- normalizzazione del dataset (3 tipi di normalizzazioni in base al parametro specificato tra parentesi: ‘l1’, ‘l2’, ‘max’):

from sklearn import preprocessing  
X = np.array([[ 1., -1., 2.], [ 2., 0., 0.], [ 0., 1., -1.]])  
X\_normalized = preprocessing.normalize(X, norm='l2')

- encoding di alcune variabili categoriche (esprimere il sesso come 0 o 1 invece di Male e Female…):

from sklearn import preprocessing  
enc = preprocessing.OrdinalEncoder()  
X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]  
enc.fit(X)  
enc.transform([['female', 'from US', 'uses Safari']])

- discretizzazione (o binning) di variabili quantitative (dividere in range di valori variabili quantitative):

from sklearn import preprocessing  
X = np.array([[ -3., 5., 15 ], [ 0., 6., 14 ], [ 6., 3., 11 ]])  
est = preprocessing.KBinsDiscretizer(n\_bins=[3, 2, 2], encode='ordinal').fit(X)  
est.transform(X)

- imputazione valori mancanti (con mean o altre statistiche possibili, come most\_frequent, oppure anche con valori scelti a priori):

**from** **sklearn.impute** **import** SimpleImputer  
imp = SimpleImputer(missing\_values=np.nan, strategy='mean')  
imp.fit([[1, 2], [np.nan, 3], [7, 6]])  
X = [[np.nan, 2], [6, np.nan], [7, 6]]  
print(imp.transform(X))

# Metriche e valutazione performances modelli con scikit-learn

In generale, per metriche di scoring relativamente alle previsioni dei modelli:

<https://scikit-learn.org/stable/modules/model_evaluation.html#the-scoring-parameter-defining-model-evaluation-rules>

Alcune di quelle più importanti per la classificazione sono:  
- **Precision**: rapporto tp / (tp + fp) (con tp true positive, e fp false positive), tra 0 e 1, si cerca valore vicino a 1:

from sklearn.metrics **import** precision\_score  
precision\_score(y\_true, y\_pred, average=**None**)

- **Recall**: rapporto tp / (tp + fn) (con tp true positive, e fn false negative), tra 0 e 1, si cerca valore vicino a 1:

from sklearn.metrics import recall\_score  
recall\_score(y\_true, y\_pred, average=**None**)

- **Accuracy**: precisione stime in classificazione multilabel

from sklearn.metrics **import** accuracy\_score  
accuracy\_score(y\_true, y\_pred, normalize = False)

- **Jaccard**: dimensione dell’intersezione fratto dimensione dell’unione di due label set.

**from** **sklearn.metrics** **import** jaccard\_score  
jaccard\_score(y\_true, y\_pred, average=**None**)

- **ROC**: calcola la ROC AUC, ovvero Area Under the Receiver Operating Characteristic Curve.

**from** **sklearn.metrics** **import** roc\_auc\_score  
roc\_auc\_score(y, clf.predict\_proba(X), multi\_class='ovr')

Alcune di quelle più importanti per il [clustering](#_Clustering_con_scikit-learn) (vedi quella parte nel documento) sono:  
- fowlkes\_mallows\_score:   
- rand\_score:   
- completeness\_score:   
- homogeneity\_score:   
- v\_measure\_score:

Alcune di quelle più importanti per la regressione sono:  
- Varianza spiegata: quantità di variabilità spiegata dalla regressione (massimo valore 1, quello ricercato).

**from** **sklearn.metrics** **import** explained\_variance\_score  
explained\_variance\_score(y\_true, y\_pred)

- R2: coefficiente di determinazione, capacità predittiva del modello, compreso tra 0 e 1 (valore ricercato).

**from** **sklearn.metrics** **import** r2\_score  
r2\_score(y\_true, y\_pred)

# Regressione con scikit-learn

Per **regressione lineare semplice** (definisce i coefficienti di un modello lineare per minimizzare la somma dei residui2 tra i target osservati nel dataset e i target predetti dall’approssimazione lineare):

**from** **sklearn** **import** linear\_model  
x = np.array([[0, 0], [1, 1], [2, 2]])  
y = np.array([0, 1, 2])  
reg = linear\_model.LinearRegression()  
reg.fit(x,y)  
reg.coef\_[0] # pendenza retta  
reg.intercept\_ # intercetta retta

Si possono anche cercare i Non negative Least Squared, ovvero con coefficienti tutti non negativi, utile in caso  di rappresentazione di alcuni fenomeni fisici o naturali che non possono assumere valori negativi. Per imporlo, basta mettere *True* nella parentesi di *LinearRegression().*

Per **Ridge regression** (compensa alcuni problemi degli OLS, imponendo una penalità per la dimensione dei coefficienti):

**from** **sklearn** **import** linear\_model  
x = np.array([[0, 0], [1, 1], [2, 2]])  
y = np.array([0, 1, 2])  
reg = linear\_model.Ridge(alpha = 0.5)  
reg.fit(x,y)  
reg.coef\_  
reg.intercept\_

Per il **Lasso** (modello lineare che permette la stima di coefficienti sparsi; si può preferire in contesti dove si preferiscono soluzioni con pochissimi o 0 coefficienti con valori non nulli) (il parametro alpha definisce il livello di sparsità dei coefficienti stimati):

**from** **sklearn** **import** linear\_model  
x = np.array([[0, 0], [1, 1], [2, 2]])  
y = np.array([0, 1, 2])  
reg = linear\_model.Lasso(alpha = 0.1)  
reg.fit(x,y)  
reg.coef\_  
reg.intercept\_

Per il modello **Elastic Net** (modello di regressione lineare utilizzato con normo-regolarizzazione dei coefficienti *\*l1\** e *\*l2\**; utile per modelli sparsi, che mantengono pesi non uguali a 0 - come il Lasso - mantenendo le proprietà di regolarizzazione del Ridge):

**from** **sklearn.linear\_model** **import** ElasticNet  
regr = ElasticNet(random\_state=0, alpha = 0.1, max\_iter = 1000)  
regr.fit(X, y)  
regr.coef\_  
regr.intercept\_  
regr.predict([[0, 0]])

Per i **modelli lineari generalizzati** (modelli di regressione lineare in cui la funzione di perdita è rimpiazzata dalla devianza di una distribuzione particolare, come Normale, Poisson, Gamma, inversa della Gaussiana...):

**from sklearn.linear\_model import TweedieRegressor**x = np.array([[0, 0], [1, 1], [2, 2]])  
y = np.array([0, 1, 2])  
**reg = TweedieRegressor(power=1, alpha=0.5, link='log')**  
reg.fit(x,y)  
reg.coef\_  
reg.intercept\_

La distribuzione dipende dal parametro di partenza:  
- = 0 distribuzione Normale  
- = 1 distribuzione di Poisson  
- = 2 distribuzione Gamma  
- = 3 inversa della distribuzione Gaussiana

Per la **regressione con le SVM** (Support Vector Machines) si chiama anche SVR, o Support Vector Regression. Ci sono tre tipi possibili, come per la classificazione delle SVM:  
- *SVR()*: considera solo il kernel lineare;  
- *LinearSVR()*: fornisce un'implementazione più rapida;  
- *NuSVR()*: formulazione diversa dalle precedenti.

X = [[0, 0], [2, 2]]  
y = [0.5, 2.5]  
from sklearn import svm  
regr = svm.SVR()  
regr.fit(X, y)  
regr.predict([[1, 1]])

Per la **regressione con Decision Tree**:

**from** **sklearn** **import** tree  
X = [[0, 0], [2, 2]]  
y = [0.5, 2.5]  
clf = tree.DecisionTreeRegressor()  
clf = clf.fit(X, y)  
clf.predict([[1, 1]])

Per la **regressione con Random Forest**:

from sklearn.ensemble import RandomForestRegressor

rng = np.random.RandomState(42)  
x = 10 \* rng.rand(200)  
def model(x, sigma=0.3):  
    fast\_oscillation = np.sin(5 \* x)  
    slow\_oscillation = np.sin(0.5 \* x)  
    noise = sigma \* rng.randn(len(x))  
    return slow\_oscillation + fast\_oscillation + noise

y = model(x)

forest = RandomForestRegressor(200)  
forest.fit(x[:, None], y)

Per la **regressione logistica** (chiamata anche logit regression, dove le probabilità che descrivono l’outcome sono modellate utilizzando una funzione logistica) (si definisce il *penalty*, con valori possibili *None*, *l1*, *l2* o *elasticnet*):

**from** **sklearn.linear\_model** **import** LogisticRegression  
clf = LogisticRegression(random\_state=0, max\_iter = 100, penalty = ‘l2’).fit(X, y)  
clf.predict(X[:2, :])  
clf.predict\_proba(X[:2, :])  
clf.score(X, y)  
clf.intercept\_  
clf.coef\_

Per l’**SGD** (Stochastic Gradient Descent, modello semplice ma molto efficace quando il numero dei campioni e delle features è molto elevato):

**from** **sklearn.linear\_model** **import** SGDRegressor  
**from** **sklearn.pipeline** **import** make\_pipeline # bisogna sempre scalare l’input  
reg = make\_pipeline(StandardScaler(), SGDRegressor(max\_iter=1000, tol=1e-3))  
reg.fit(X, y)

Per il **Perceptron** (modello semplice, lavora bene su dataset enormi, aggiorna il suo modello in base agli errori che esso compie in fase di apprendimento):

**from** **sklearn.linear\_model** **import** Perceptron  
clf = Perceptron(tol=1e-3, random\_state=0)  
clf.fit(X, y)  
clf.score(X, y)

Per il modello **KNearest Neighbors Regression** (utile quando ci sono più variabili continue che discrete; basato sui k vicini di ogni punto, con k specificato dall’utente) (*n\_neighbors* specifica il numero di vicini) (*algorithm* specifica l’algoritmo di ricerca usato) (*p* per la metrica usata, con 1 distanza di Manhattan, 2 distanza euclidea):

X = [[0], [1], [2], [3]]  
y = [0, 0, 1, 1]  
**from** **sklearn.neighbors** **import** KNeighborsRegressor  
neigh = KNeighborsRegressor(n\_neighbors=5, algorithm = ‘auto’, p = 2)  
neigh.fit(X, y)  
neigh.predict([[1.5]])

Per il modello **Radius Nearest Neighbors Regression** (utile quando ci sono più variabili continue che discrete; basato sul raggio *radius* in cui si considerano vicini i vari punti, con *radius* specificato dall’utente) (*radius* specifica il raggio in cui considerare i vicini) (*algorithm* specifica l’algoritmo di ricerca usato) (*p* per la metrica usata, con 1 distanza di Manhattan, 2 distanza euclidea):

X = [[0], [1], [2], [3]]  
y = [0, 0, 1, 1]  
**from** **sklearn.neighbors** **import** RadiusNeighborsRegressor  
neigh = RadiusNeighborsRegressor(radius=1.0, algorithm = ‘auto’, p = 2)  
neigh.fit(X, y)  
neigh.predict([[1.5]])

Per il regressore basato sul **Bagging** (classe di algoritmi che costruisce diversi stimatori di campioni random presi dal dataset di training, e poi forma una previsione finale aggregando queste singole stime; utili per ridurre la varianza di uno stimatore base; in generale si usano spesso per migliorare le stime di un singolo stimatore senza doverlo adattare ad un modello completamente diverso) (*n\_estimators* per definire il numero di stimatori nell’insieme) (*max\_samples* per definire il numero di campioni da estrarre per trovare ogni singola stima) (*bootstrap* con *True* per scegliere se le features sono estratte con ripetizione):

**from** **sklearn.ensemble** **import** BaggingRegressor  
**from** **sklearn.datasets** **import** make\_regression  
X, y = make\_regression(n\_samples=100, n\_features=4, n\_informative=2, n\_targets=1, random\_state=0, shuffle=**False**)  
regr = BaggingRegressor(estimator=SVR(), n\_estimators=10, random\_state=0, max\_samples = 1,bootstrap = True).fit(X, y)  
regr.predict([[0, 0, 0, 0]])

Per il modello **AdaBoost** (fitta una sequenza di modelli deboli su versioni modificate del dataset di training, e poi combina queste previsioni, pesandole in base alla precisione, per una previsione finale) (*n\_estimators* per il numero massimo di stimatori a cui il boosting viene stoppato) (*learning\_rate* per il peso applicato ad ogni regressore ad ogni iterazione boosting)

**from** **sklearn.ensemble** **import** AdaBoostRegressor  
**from** **sklearn.datasets** **import** make\_regression  
X, y = make\_regression(n\_features=4, n\_informative=2, random\_state=0, shuffle=**False**)  
regr = AdaBoostRegressor(random\_state=0, n\_estimators=50, learning\_rate = 1.0)  
regr.fit(X, y)  
regr.predict([[0, 0, 0, 0]])  
regr.score(X, y)

Per il modello **Gradient Boosting** (generalizzazione del boosting) (*n\_estimators* per il numero massimo di stimatori a cui il boosting viene stoppato) (*learning\_rate* per il peso applicato ad ogni regressore ad ogni iterazione boosting):

from sklearn.datasets import make\_regression  
from sklearn.ensemble import GradientBoostingRegressor  
X, y = make\_regression(random\_state=0)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)  
reg = GradientBoostingRegressor(random\_state=0, n\_estimators=50, learning\_rate = 1.0)  
reg.fit(X\_train, y\_train)  
reg.predict(X\_test[1:2])  
reg.score(X\_test, y\_test)

Per il modello **MLP** (Multi Layer Perceptron, usa l'errore quadratico come funzione di perdita)

from sklearn.neural\_network import MLPRegressor  
from sklearn.model\_selection import train\_test\_split  
X, y = make\_regression(n\_samples=200, random\_state=1)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=1)  
regr = MLPRegressor(random\_state=1, max\_iter=500).fit(X\_train, y\_train)  
regr.predict(X\_test[:2])  
regr.score(X\_test, y\_test)

# Riduzione di dimensionalità con scikit-learn

L’obiettivo della riduzione di dimensionalità è di ridurre il numero di variabili presenti un dataset perdendo il numero minore possibile di features rilevanti per lo studio in essere.

Per la riduzione di dimensionalità con la **PCA** (Analisi Componenti Principali, usata per decomporre un dataset multivariato in una serie di componenti ortogonali che spieghino la quantità massima possibile di varianza) (*n\_components* per predefinire il numero di componenti principali) (*tol* per la tolerance):

**from** **sklearn.decomposition** **import** PCA  
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])  
pca = PCA(n\_components=None, tol = 0)  
pca.fit(X)  
pca.explained\_variance\_ratio\_  
pca.singular\_values\_  
pca.components\_  
pca.explained\_variance\_

Per riduzione di dimensionalità con la **KernelPCA** (estensione della PCA che raggiunge la riduzione di dimensionalità non-lineare tramite l’uso di Kernel) (*max\_iter* per il numero massimo di iterazioni) (*n\_components* per predefinire il numero di componenti principali) (*tol* per la tolerance):

from sklearn.decomposition import KernelPCA  
transformer = KernelPCA(n\_components=None, kernel='linear', max\_iter = 100)  
X\_transformed = transformer.fit\_transform(X)  
X\_transformed.shape

Per la riduzione di dimensionalità tramire **FactorAnalysis** (modello probabilistico, simile alla PCA ma non si possono fare assunzioni sulle componenti ottenute) (*max\_iter* per il numero massimo di iterazioni) (*n\_components* per predefinire il numero di componenti principali) (*tol* per la tolerance):

from sklearn.decomposition import FactorAnalysis  
transformer = FactorAnalysis(n\_components=None, kernel='linear', max\_iter = 100)  
X\_transformed = transformer.fit\_transform(X)  
X\_transformed.shape

Per la riduzione di dimensionalità tramire **Non-Negative Matrix Factorization (NMF,** approccio alternativo alla decomposizione che assume che i dati e le componenti siano non negativi) (*n\_components* per predefinire il numero di componenti principali) (*tol* per la tolerance) (*max\_iter* per il numero massimo di iterazioni):

X = np.array([[1, 1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])  
from sklearn.decomposition import NMF  
model = NMF(n\_components=None, init='random', random\_state=0)  
W = model.fit\_transform(X)  
H = model.components\_

# Classificazione con scikit-learn

Per la **classificazione con le SVM** (Support Vector Machines),

# Clustering con scikit-learn

Clustering of unlabeled data can be performed with the module sklearn.cluster. Each clustering algorithm comes in two variants: a class, that implements the fit method to learn the clusters on train data, and a function, that, given train data, returns an array of integer labels corresponding to the different clusters. For the class, the labels over the training data can be found in the labels\_ attribute.

Per il clustering col **metodo delle k-medie** (clusterizza i dati cercando di separarli in n gruppi, specificati a priori, minimizzando l’inerzia o la somma al quadrato della distanza within; algoritmo che scala bene per grandi numeri di campioni) (*n\_clusters* per il numero di cluster scelto) (*n\_init* per il numero di volte che l’algoritmo lavora con centroidi differenti) (*max\_iter* per il massimo numero di iterazioni dell’algoritmo) (si possono definire anche altri aspetti, come il *verbose* e la *tol*):

**from** **sklearn.cluster** **import** KMeans  
X = np.array([[1, 2], [1, 4], [1, 0], [10, 2], [10, 4], [10, 0]])  
kmeans = KMeans(n\_clusters=2, random\_state=0, n\_init="auto", max\_iter = 300).fit(X)  
kmeans.labels\_  
kmeans.predict([[0, 0], [12, 3]])  
kmeans.cluster\_centers\_

Per il clustering con **l’Affinity Propagation** (crea cluster mandando messaggi tra coppie di campioni fino alla convergenza; il dataset finale può essere descritto con un piccolo numero di esemplari considerati come membri più significativi del campione) (*convergence\_iter* definisce dopo quante interazione senza cambiamenti di cluster c’è la convergenza) ():

**from** **sklearn.cluster** **import** AffinityPropagation  
X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])   
clustering = AffinityPropagation(random\_state=5, max\_iter = 200, convergence\_iter = 15).fit(X)  
clustering  
clustering.labels\_  
clustering.predict([[0, 0], [12, 3]])  
clustering.cluster\_centers\_

Per il clustering con il MeanShift (punta a scoprire cluster in campioni molto densi; algoritmo basato sui centroidi, che lavora sull’aggiornamento dei centroidi in modo da essere la media dei punti all’interno di una data regione) (si possono mettere i *seeds*) (*cluster\_all = True* per imporre che anche i punti lontani, che non entrano in kernel, vanno comunque associati) (*max\_iter* per il numero di iterazioni massimo):

**from** **sklearn.cluster** **import** MeanShift  
X = np.array([[1, 1], [2, 1], [1, 0], [4, 7], [3, 5], [3, 6]])  
clustering = MeanShift(bandwidth=2, cluster\_all = True, max\_iter = 300).fit(X)  
clustering  
clustering.labels\_  
clustering.predict([[0, 0], [5, 5]])  
clustering.cluster\_centers\_

Per il clustering con il SpectralClustering (per un low-dimensional embedding; molto efficiente se la matrice di affinità è sparsa; il numero di cluster va specificato a priori; lavora bene con un numero ridotto di clusters, non è consigliato per numeri elevati di clusters) (*n\_clusters* per il numero di clusters specificato a priori) (*n\_init* è il numero di volte che le k-medie vengono ripetute; alla fine verrà scelto il miglior output con *n\_init* runs consecutive) (*gamma* per il coefficiente del kernel) (niente *predict* o *cluster\_centers*):

**from** **sklearn.cluster** **import** SpectralClustering  
X = np.array([[1, 1], [2, 1], [1, 0], [4, 7], [3, 5], [3, 6]])  
clustering = SpectralClustering(n\_clusters=8, assign\_labels='discretize', random\_state=0, n\_init = 10, gamma = 1.0).fit(X)  
clustering  
clustering.labels\_

Per il clustering con il **Clustering gerarchico** (produce clusters annidati unendoli o separandoli in successione; questa gerarchia di clusters può essere rappresentata da un albero, o dendrogramma, con la radice come l’unico cluster contenente tutto, e le foglie come piccoli clusters con un solo campione; i clustering agglomerativi creano una gerarchia secondo metriche come la distanza di Ward, il massimo linkage, il linkage medio o il singolo linkage) (*n\_clusters* imposto come *None* se la *distance\_threshold* ha valore) (*metric* per il tipo di distanza usata, se “euclidean”, “l1”, “l2”, “manhattan”, “cosine”, or “precomputed”) (*linkage* per il metodo di separazione/unione dei clusters, ***‘ward’, ‘complete’, ‘average’, ‘single’***) (*distance\_threshold* per la soglia oltre al quale l’algoritmo si ferma):

from sklearn.cluster import AgglomerativeClustering  
X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])  
clustering = AgglomerativeClustering(**n\_clusters = 2, metric = ‘euclidean’, linkage = ‘ward’, distance\_threshold = None**).fit(X)  
clustering  
clustering.labels\_  
clustering.n\_clusters\_  
clustering.n\_leaves\_

Per il clustering con il **DBSCAN** (individua i clusters come aree ad alta densità rispetto ad aree a bassa densità di punti; i cluster possono essere di ogni forma, rispetto ad esempio alle k-medie e alle loro forme convesse; si basa sul concetto di core samples, ovvero campioni in aree ad alta densità) (*eps* definisce la distanza massima tra due campioni perché questi possano essere definiti vicini) (*min\_samples* definisce il numero minimo di campioni vicini ad un punto perché questo possa essere considerato core sample):

from sklearn.cluster import DBSCAN  
X = np.array([[1, 2], [2, 2], [2, 3], [8, 7], [8, 8], [25, 80]])  
clustering = DBSCAN(eps=0.5, min\_samples=5).fit(X)  
clustering  
clustering.labels\_

Per il clustering con l’algoritmo **OPTICS** (simile al DBSCAN, ma il valore di eps non è fisso ma compreso in un range di valori) (*min\_samples* definisce il numero minimo di campioni vicini ad un punto perché questo possa essere considerato core sample) (*max\_eps*definisce la distanza massima tra due campioni perché questi possano essere definiti vicini):

from sklearn.cluster import OPTICS  
X = np.array([[1, 2], [2, 5], [3, 6], [8, 7], [8, 8], [7, 3]])  
clustering = OPTICS(min\_samples=5).fit(X)  
clustering  
clustering.labels\_  
clustering.cluster\_hierarchy\_

Per il clustering con l’algoritmo BIRCH (costruisce un albero, comprimendo i dati in un set di nodi di Clustering Feature) (*n\_clusters* definisce il numero di cluster finali) (*threshold* definisce il raggio entro cui i subcluster vengono ottenuti):

**from** **sklearn.cluster** **import** Birch  
X = [[0, 1], [0.3, 1], [-0.3, 1], [0, -1], [0.3, -1], [-0.3, -1]]  
brc = Birch(n\_clusters=**None,** threshold = 0.5)  
brc.fit(X)  
brc.predict(X)  
brc.subcluster\_centers\_

Per valutare le performance dei vari metodi di clusterizzazione:  
- **Rand index**: misura la similarità tra due metodi

from sklearn import metrics  
labels\_true = [0, 0, 0, 1, 1, 1]  
labels\_pred = [0, 0, 1, 1, 2, 2]  
metrics.rand\_score(labels\_true, labels\_pred)

metrics.adjusted\_rand\_score(labels\_true, labels\_pred)

- **Mutual Information**: misura la concordanza di due assignement, ignorando le permutazioni

from sklearn import metrics  
labels\_true = [0, 0, 0, 1, 1, 1]  
labels\_pred = [0, 0, 1, 1, 2, 2]  
metrics.adjusted\_mutual\_info\_score(labels\_true, labels\_pred)

- **omogeneità**: ogni cluster comprende solo membri di una singola classe

from sklearn import metrics  
labels\_true = [0, 0, 0, 1, 1, 1]  
labels\_pred = [0, 0, 1, 1, 2, 2]  
metrics.homogeneity\_score(labels\_true, labels\_pred)

- **completezza**: tutti i membri di una classe sono assegnati allo stesso cluster

from sklearn import metrics  
labels\_true = [0, 0, 0, 1, 1, 1]  
labels\_pred = [0, 0, 1, 1, 2, 2]  
metrics.completeness\_score(labels\_true, labels\_pred)

- **V-measure**: media armonica tra omogeneità e completezza

from sklearn import metrics  
labels\_true = [0, 0, 0, 1, 1, 1]  
labels\_pred = [0, 0, 1, 1, 2, 2]  
metrics.v\_measure\_score(labels\_true, labels\_pred)

- **Fowlkes-Mallows index:** metrica per misurare la matrice di confusione

from sklearn import metrics  
labels\_true = [0, 0, 0, 1, 1, 1]  
labels\_pred = [0, 0, 1, 1, 2, 2]  
metrics.fowlkes\_mallows\_score(labels\_true, labels\_pred)

- **Coefficiente di Silhouette**: per valutare la matrice di confusione

from sklearn import metrics  
kmeans\_model = KMeans(n\_clusters=3, random\_state=1).fit(X)  
labels = kmeans\_model.labels\_  
metrics.silhouette\_score(X, labels, metric='euclidean')

- **Matrice di contingenze**: riporta le cardinalità della vera classe predetta correttamente

from sklearn.metrics.cluster import contingency\_matrix  
x = ["a", "a", "a", "b", "b", "b"]  
y = [0, 0, 1, 1, 2, 2]  
contingency\_matrix(x, y)