Mini-guida introduttiva a Python, numpy e matplotlib

Scarica il file .py

OBIETTIVO In questa scheda tratteremo gli aspetti fondamentali del linguaggio di programmazione Python che saranno utili per le schede successive.

Di seguito elenchiamo gli argomenti trattati.

- Cenni generali su Python.
- Installazione dell'ambiente di lavoro Thonny.
- Elementi del linguaggio Python: variabili e oggetti, stringhe, costrutto di selezione, moduli.
- Elementi di numpy: array, generazione di array, indicizzazione degli array, matematica con gli array.
- Elementi di matplotlib: grafici a linee, grafici a dispersione, grafici con barre di errore, elementi grafici.

INTRODUZIONE A PYTHON

Python nacque a opera dell'ormai celebre informatico olandese Guido Van Rossum, definito dalla comunità di Python come «benevolo dittatore a vita», poiché continua tuttora ad avere l'ultima parola sulle decisioni che riguardano il processo di sviluppo di Python. I primi vagiti del nuovo linguaggio furono nel periodo di Natale del 1989, come Van Rossum raccontò 7 anni dopo:

«Più di sei anni fa, nel dicembre 1989, stavo cercando un progetto di programmazione per "hobby" che mi avrebbe dovuto tenere occupato nella settimana vicina a Natale. Il mio ufficio... sarebbe stato chiuso, ma io avevo un computer, e non molto di più. Decisi di scrivere un interprete per un nuovo linguaggio di scripting a cui avrei pensato dopo: … sarebbe dovuto appartenere agli hacker di Unix. Scelsi Python come nome per il progetto, essendo leggermente irriverente (e sono un grande fan di Monty Python's Flying Circus)»

Nota *Monty Python's Flying Circus* è una serie televisiva britannica, prodotta in 4 stagioni per un totale di 45 episodi, trasmessa dalla BBC, dal 1969 al 1974. La serie andò oltre i confini di ciò che era considerato accettabile, tanto per lo stile quanto per il contenuto, e ha avuto una duratura influenza, non solo sulla commedia britannica, ma a livello internazionale.

Queste sono le principali caratteristiche di Python:

- un linguaggio semplice, intuitivo e potente quanto i suoi maggiori concorrenti;
- codice sorgente aperto, in modo che ognuno possa partecipare al suo sviluppo;
- un codice facilmente comprensibile, quasi come l'inglese parlato;
- ottimale per accelerare i tempi di sviluppo dei programmi.

Negli anni a seguire, lo sviluppo del linguaggio fu molto rapido e oggi Python è tra i linguaggi di programmazione più usati al mondo.

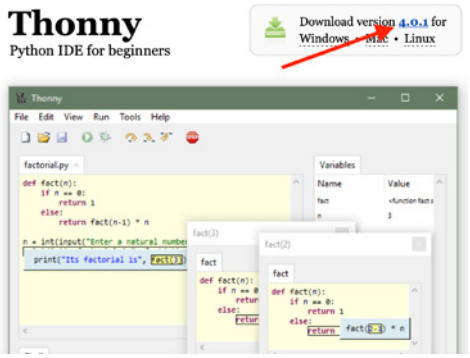
L'AMBIENTE DI SVILUPPO

Per iniziare a imparare Python con il nostro computer, dobbiamo installare l'interprete Python che ci permette di eseguire gli *script* che scriveremo mediante un editor di testo: anche questo dovrà essere installato se non ne possediamo uno. Per fortuna esistono ambienti in cui l'interprete e l'editor di testo sono integrati in un unico applicativo. Noi faremo uso di uno di questi: *Thonny IDE*.

Suggerimento IDE significa *Integrated Development Environment* ovvero ambiente di sviluppo integrato fondamentale per i programmatori nello sviluppo, debugging ed esecuzione del codice sorgente di un programma.

Thonny è un'applicazione gratuita ideata appositamente per l'apprendimento di Python e contiene un editor di testo evoluto per la scrittura di codice sorgente, oltre che l'interprete Python.

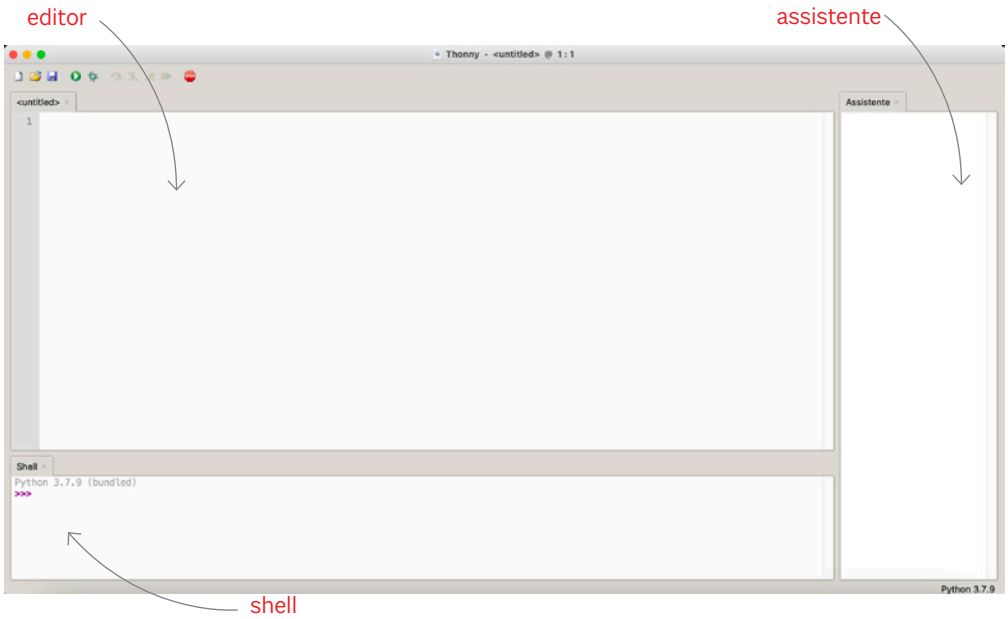
Nella home page di Thonny https://thonny.org clicchiamo sul link che rimanda alla versione corrente, come in figura sotto.



Nella pagina che segue clicchiamo sul link a *thonny-xxl-4.0.1.exe* che avvia il download di Thonny. Al momento la versione 4.0.1 è quella più recente, ma in futuro potremo scaricare anche versioni successive.

Prestiamo attenzione a scaricare la versione xxl, poiché soltanto questa contiene i moduli numpy e matplotlib necessari alle schede successive, altrimenti questi moduli vanno installati successivamente tramite il gestore di pacchetti presente negli strumenti di Thonny.

Al termine del download eseguiamo la procedura di installazione e, una volta terminata, possiamo avviare Thonny. Nella sua finestra principale sono visualizzati l'*editor di testo*, la *shell* e l'*assistente*.



Useremo l'editor per scrivere il codice dei nostri programmi, mentre nella shell vedremo comparire gli output in formato testuale generati dai programmi che eseguiremo. Infine, l'assistente ci verrà in soccorso quando nei nostri programmi saranno presenti errori che non sappiamo risolvere da soli.

ELEMENTI DEL LINGUAGGIO PYTHON

LE PAROLE CHIAVE E I MODULI

Il linguaggio Python ha soltanto 33 parole chiave native, dalle quali si può costruire qualunque programma con qualsiasi grado di complessità.

and	else	import	return
as	except	in	True
assert	exec	is	try
break	False	lambda	while
class	finally	None	with
continue	for	not	yield
def	from	or	
del	global	pass	
elif	if	raise	

Queste parole chiave sono esattamente come i mattoncini da costruzioni, tramite i quali è possibile assemblare qualunque costruzione complessa. Talvolta risulta comodo partire da piccole costruzioni già assemblate che facilitano la creazione di costruzioni più grandi: Python adotta la stessa filosofia mettendo a disposizione dell'utente i *moduli*.

I moduli sono librerie di *comandi*, *funzioni* e *classi* impacchettate insieme per fornire un set di funzionalità aggiuntive utilizzabili dal programmatore. Esistono due tipi di moduli:

- i moduli nativi, inclusi nell'installazione di Python che abbiamo sulla nostra macchina e sono anche detti built-in;
- i moduli esterni, creati dal programmatore stesso, oppure da terze parti.

Nelle attività useremo il modulo nativo turtle e i due moduli esterni numpy e matplotlib.

Ricorreremo spesso all'uso di funzioni definite all'interno dei moduli. Le funzioni sono sequenze di istruzioni, identificate da un nome univoco scelto da chi le ha programmate, che svolgono compiti precisi e definiti che discuteremo di volta in volta. Una funzione può avere:

- un *input* sotto forma di variabili passate in ingresso;
- un *output*, ovvero variabili ritornate in uscita al termine del loro compito;
- *effetti collaterali*, per esempio l'esecuzione di una stampa a video.

LE VARIABILI E LA FUNZIONE PRINT

Una variabile è uno spazio di memoria RAM contenente dati a cui è stato attribuito un identificatore, che nel codice sorgente corrisponde al nome della variabile. All'interno del codice le variabili devono essere prima di tutto inizializzate, ovvero dobbiamo assegnare loro dei valori.

Creiamo un nuovo programma (codice sotto) nel quale inizializziamo due varabili: temperatura e giorno. L'inizializzazione avviene tramite l'operatore di assegnamento = (righe 1 e 2). Stampiamo poi i loro valori mediante la funzione print (righe 3 e 4).

```
1 temperatura = 23.5
2 giorno = "Lunedì"
3 print(temperatura)
4 print(giorno)
```

Dopo aver premuto il tasto di esecuzione dello script corrente, il risultato che otteniamo è la stampa nella shell dei due valori che abbiamo assegnato a temperatura e giorno.

```
>>> %Run temperatura.py
  23.5
  Lunedì
>>>
```

Notiamo che l'inizializzazione di temperatura (riga 1) differisce dall'inizializzazione di giorno (riga 2) poiché in quest'ultima abbiamo fatto uso delle virgolette " ". Questo è dovuto al fatto che sono due variabili di tipo diverso: temperatura è di *tipo numerico*, mentre giorno è di *tipo stringa* poiché contiene una sequenza di caratteri "(alfabetici, numerici, ...)".

I nomi delle due variabili, `temperatura e giorno`, vengono passati alla funzione `print`, la quale ne stampa i contenuti.

La funzione `print` permette di stampare il contenuto delle variabili o, più in generale, anche il risultato di funzioni o operazioni.

Programmando in Python abbiamo ampia libertà nella scelta dei nomi delle variabili, tenendo a mente che:

- Python è case-sensitive, quindi le lettere maiuscole sono distinte da quelle minuscole (per esempio, `temperatura` è un nome diverso `da Temperatura`);
- nei nomi è possibile usare lettere, numeri o il carattere _, ma il nome non può mai iniziare con un numero;
- i nomi delle 33 parole chiave di Python sono riservati e non possono mai essere usati come nomi di variabili.

Consigliamo sempre di costruire i nomi delle variabili che siano esplicativi di ciò che la variabile dovrà contenere.

Nell'esempio appena visto abbiamo due variabili, una che contiene un numero, una che contiene una stringa. In genere il contenuto di una variabile può essere un numero intero o decimale, un numero complesso, una stringa, un valore booleano, oppure un *oggetto* come, per esempio, un array che vedremo in seguito. Ciascuno di questi è un tipo di oggetto, per cui, sulla base di quanto appena visto, possiamo affermare che `temperatura` e `giorno` sono variabili di tipo diverso.

Python è un linguaggio dotato di controllo dei tipi dinamico, per cui non è necessario dichiarare in anticipo il tipo di una variabile, visto che sarà l'interprete a definire i tipi delle variabili utilizzate nel programma. Python individua il tipo della variabile nel momento in cui a essa viene assegnato un valore (l'assegnamento corrisponde all'operatore =).

Possiamo concatenare i valori delle variabili con le stringhe mediante il costrutto delle *f-string*.

Le f-string sono dette anche stringhe formattate e sono definite apponendo una `f` davanti all'apice singolo o doppio che apre la stringa. Vediamo subito un esempio:

```
1 #risolvere l'equazione 10 * x = 5
2 a = 10
3 b = 5
4 print(f"La soluzione dell'equazione è {b/a}") #esempio di f-string
```

```
>>> %Run stringhe3.py
  La soluzione dell'equazione è 0.5
>>>
```

Alle righe 2 e 3 assegniamo i valori 10 e 5 rispettivamente ad `a` e `b` e poi stampiamo la stringa formattata contenente il valore di `b/a` inserito tra parentesi graffe. Notiamo che l'operazione `b/a` viene eseguita prima della stampa e il risultato della `print` corrisponde alla stampa di `La soluzione dell'equazione è 0.5`. Quindi, le parentesi graffe delimitano una variabile oppure una espressione Python che dovrà essere valutata prima di venire inserita nella stringa.

Osserviamo la riga 1 e la seconda metà della riga 4 del codice sopra: si tratta di commenti al codice. In Python è possibile inserire commenti all'interno del codice, sempre preceduti dal carattere cancelletto #. I commenti sono utili per annotare il codice e sono completamente ignorati durante l'esecuzione del programma.

All'interno di una stringa possiamo inserire i valori di più variabili e possiamo fare in modo che questi siano arrotondati a un numero di cifre decimali prefissato. Il codice seguente fornisce un esempio.

```
1 T = 293
2 r = 0.011767810
3 print(f"La sfera a temperatura {T:.2f} K ha un raggio pari a: {r:.4f} m")
```

Le righe 1 e 2 assegnano valori alle variabili `T` e `r`, poi la riga 3 stampa una f-string nella quale sono inseriti:

- il valore della variabile `T` arrotondato a 2 cifre decimali {`T:.2f`};
- il valore della variabile `r` arrotondato a 4 cifre decimali {`r:.4f`}.

L'esecuzione del programma stampa: `La sfera a temperatura 293.00 K ha un raggio pari a: 0.0118 m`. Notiamo che il valore di `r` è stato arrotondato.

IL COSTRUTTO DI SELEZIONE IF-ELSE

In qualche programma può essere utile modificare il flusso di esecuzione delle istruzioni in virtù di una condizione che si verifica all'interno del programma: per fare ciò esiste il *costrutto di selezione*. Esso ci permette di selezionare se un blocco di istruzioni debba essere eseguito o meno, a seconda di una condizione. Analizziamo il codice seguente.

```
1 if (a_A < 0):
2     print("La forza di attrito tra il corpo A e il piano è troppo elevata. ")
3     exit()
```

La riga 1 riporta la parola chiave `if` seguita dalla condizione (`a_A < 0`) e infine dal carattere :. Le successive righe 2 e 3 sono indentate, ovvero sono precedute da quattro caratteri di spaziatura. L'*indentazione* consente a Python di capire che le righe 2 e 3 costituiscono un blocco di codice. Questo blocco di codice viene eseguito soltanto se la condizione posta a riga 1 è vera, quindi soltanto se la variabile `a_A` contiene un valore minore di 0. In tale caso viene eseguita la riga 2 che stampa una stringa e la riga 3 che chiude il programma con la funzione `exit()`.

Suggerimento L'indentazione in Python è obbligatoria per identificare i blocchi di codice all'interno delle selezioni, ma non solo: vale lo stesso per i cicli, le funzioni o anche le classi, che non vedremo in questa scheda. All'interno del codice sorgente la tipologia di indentazione deve essere univoca per cui i caratteri ammessi sono lo spazio oppure la tabulazione. Normalmente si indenta utilizzando la tabulazione che convenzionalmente viene sostituita da quattro caratteri di spaziatura. Il carattere dopo il quale l'interprete Python si aspetta sempre un blocco indentato è il : .

Possiamo fare in modo che il programma esegua un blocco di codice oppure un altro in maniera esclusiva, abbinando l'istruzione `else a if`, come nell'esempio sotto.

```
1 if (a_A < 0):
2     print("La forza di attrito tra il corpo A e il piano è troppo elevata")
3     exit()
4 else:
5     print("Il programma può continuare")
```

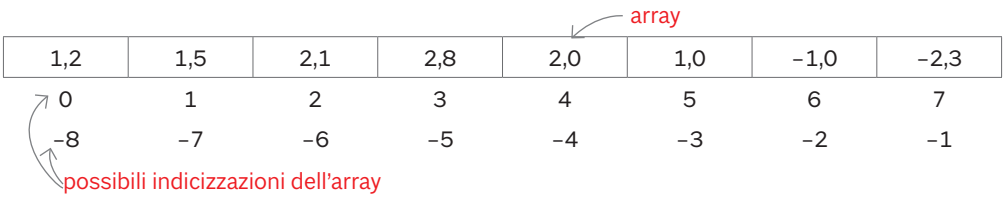
Le righe 1, 2 e 3 sono identiche al caso precedente, ma ora se la condizione (`a_A 1 0`) è falsa, il programma esegue il blocco di codice successivo all'istruzione `else`, quindi stampa che il programma può continuare.

IL MODULO NUMPY

Numpy è un modulo di Python per la manipolazione di array e matrici dotato di una vasta collezione di funzioni matematiche capaci di operare sugli array. In questa sezione tratteremo solo gli aspetti di numpy salienti ai fini delle schede successive, ma ricordiamo che numpy è uno tra i moduli di Python più potenti in circolazione e che ha avuto il maggior successo nella comunità scientifica. Per tutti i dettagli rimandiamo al sito ufficiale *https://numpy.org/doc/stable/index.html*.

Che cos'è un *array*? Possiamo immaginare un array come un contenitore, le cui caselle sono dette celle o elementi dell'array. Ciascuno degli elementi si comporta come una variabile tradizionale e tutte le celle sono variabili di uno stesso tipo, detto tipo base dell'array. Per esempio, useremo spesso array di tipo base float, nei quali ogni cella è di tipo float, ovvero un numero decimale. La lunghezza di un array è il numero di celle che esso contiene.

La figura sotto rappresenta un array di tipo base float.



La lunghezza dell'array è 8 e ogni cella contiene un valore decimale, per esempio la prima cella contiene il valore 1,2.

Ogni cella è identificata da un indice definito dalla posizione della cella all'interno dell'array. La prima cella ha indice 0, la seconda ha indice 1 e così via; questa numerazione delle celle si chiama *indicizzazione*.

In Python esiste una seconda modalità di indicizzazione che fa uso dei numeri interi negativi: l'ultima cella ha indice −1, la penultima ha indice −2 e così via. Vediamo un esempio di utilizzo dell'indicizzazione per accedere ai valori contenuti nelle celle dell'array.

```python
import numpy as np

array = np.array([1.2, 1.5, 2.1, 2.8, 2.0, 1.0, -1.0, -2.3])
print(array[0])
print(array[-8])
print(array[7])
print(array[-1])
```

Prima di utilizzare `numpy` dobbiamo importarne il modulo come a riga 1. Sempre a riga 1 definiamo l'alias `np` che ci consente di fare riferimento a `numpy` semplicemente scrivendo `np` nel codice.

La riga 3 inizializza l'array array che contiene i valori della figura sopra. Ricordiamo che in Python il separatore delle cifre decimali è il punto, mentre la virgola delimita le diverse celle dell'array. Osserviamo che la prima cella dell'array è delimitata da una [a sinistra, mentre l'ultima da una] alla sua destra. Tale costrutto in Python si chiama *lista*, quindi possiamo dire che `[1.2, 1.5, 2.1, 2.8, 2.0, 1.0, -1.0, -2.3]` è una lista. Per ottenere un array dobbiamo passare la lista alla funzione `np.array` come nella riga 3.

Le funzioni `print` delle righe 4, 5, 6 e 7 stampano i valori della prima e dell'ultima cella facendo uso delle due diverse indicizzazioni. L'indice deve essere inserito dopo il nome dell'array e delimitato da parentesi quadre.

La funzione `np.array` genera un array a partire da una lista, ma esistono numerose altre funzioni per generare array. Nelle schede successive useremo spesso `np.linspace`. Questa funzione genera un array di numeri (in genere decimali) equi-spaziati tra loro su un intervallo specificato. Per utilizzare questa funzione dobbiamo conoscere:

- gli estremi inferiore e superiore dell'intervallo;
- il numero di elementi dell'array, ovvero quanti numeri equi-spaziati sull'intervallo ci occorrono.

Illustriamo il concetto con un esempio.

```python
import numpy as np

array = np.linspace(1, 2, 5)
print(array)
```

Dopo aver importato `numpy` alla riga 1, creiamo un nuovo array usando la funzione `np.linspace` come in riga 3. A essa sono passati tre numeri: 1 è l'estremo inferiore, 2 è l'estremo superiore e 5 è il numero di elementi dell'array che verrà creato. La riga 4 stampa l'array ed eseguendo il programma verifichiamo che l'array contiene i valori 1. , 1.25, 1.5, 1.75, 2. Si tratta proprio di 5 numeri nell'intervallo tra 1 e 2, equi-spaziati tra loro.

CREARE ARRAY PSEUDO-CASUALI

Possiamo creare array di *numeri pseudocasuali* mediante le funzione del sotto-modulo np.random. Tra queste ci tornerà utile `np.random.uniform` che sorteggia uniformemente numeri pseudocasuali all'interno di un intervallo. Essa riceve in input:

- gli estremi inferiore e superiore dell'intervallo all'interno del quale prelevare i numeri;
- il numero di elementi dell'array, ovvero quanti numeri pseudocasuali vogliamo inserire nell'array.

Suggerimento I numeri pseudocasuali sono generati da algoritmi deterministici secondo una sequenza che approssima le proprietà statistiche di una sequenza generata da un processo fisico realmente casuale. Il modulo `numpy.random` dispone di alcuni di questi algoritmi.

Per esempio, nel codice sotto creiamo un array x che contiene 4 numeri pseudocasuali estratti nell'intervallo compreso tra −1 e 1.

```
1 import numpy as np
2
3 x = np.random.uniform(-1,1,4)
4 print(x)
```

Eseguendo il programma verifichiamo che l'array x cambia in ogni esecuzione.

INDICIZZAZIONE BOOLEANA

È possibile definire condizioni come quelle viste per il costrutto di selezione `if..else..` anche nel caso degli `array`. Per esempio, dato l'array x del codice riportato sopra, possiamo definire la condizione `x > 0`. Le condizioni come questa sono valutate automaticamente per ogni cella dell'array, quindi per alcune celle la condizione è vera, per altre falsa. Possiamo affermare che `x > 0` è anch'esso un array in cui ogni cella può contenere il valore `True` oppure il valore `False`.

Il programma seguente, che è un'estensione di quello sopra, ci consente di verificare ciò.

```
1 import numpy as np
2
3 x = np.random.uniform(-1,1,4)
4 print(x)
5 print(x>0)
6 print(np.count_nonzero(x>0))
7 print(x[x>0])
```

Dopo aver stampato tutti i valori contenuti nell'array con la riga 4, stampiamo il risultato della condizione `x > 0` mediante la riga 5. Il risultato è un array di `True e False`, a seconda dei valori degli elementi di x, per esempio: `True False False False`.

La riga di codice 6 fa uso della funzione `np.count_nonzero` per contare quanti valori True sono presenti nell'array `x > 0`: convenzionalmente il valore `False` corrisponde a 0 e non viene considerato nel conteggio. In questo caso la riga 6 stampa 1, perché abbiamo un solo `True`.

Infine, la riga 7 stampa l'array `x[x>0]`: esso è l'array che contiene i soli elementi dell'`array x` per i quali la condizione è vera, quindi nel nostro caso contiene soltanto il primo elemento di x.

FARE MATEMATICA CON GLI ARRAY

Numpy è molto potente perché ci consente di eseguire operazioni matematiche sugli array nello stesso modo in cui le eseguiamo su variabili numeriche. Ci consente anche di scrivere espressioni matematiche che coinvolgano sia variabili numeriche sia array.

Python è dotato degli operatori che ci servono per l'aritmetica di base. Tra questi ricordiamo le quattro operazioni aritmetiche associate rispettivamente agli operatori +, -, *, / e poi l'operatore di elevamento a potenza **.

Numpy contiene un gran numero di funzioni matematiche predefinite, capaci di operare sia su variabili numeriche sia sugli array. Tra queste ricordiamo quelle che ci serviranno nelle prossime schede:

- `np.sin e np.cos` per calcolare le funzioni trigonometriche seno e coseno;
- `np.sqrt` per calcolare la radice quadrata;
- `np.sum` per calcolare la somma di tutti gli elementi di un array.

Vediamo un primo esempio che è riportato nel codice sotto.

```
1  import numpy as np
2
3  t = np.linspace(0, 10, 11)
4  v = 2.0
5  s1 = v*t
6
7  a = 9.81
8  s2 = v*t + 0.5*a*t**2
9
10 print(s1)
11 print(s2)
```

Abbiamo definito un array `t` alla riga 3, una variabile numerica `v` alla riga 4 e una seconda variabile a alla riga 7. Le righe 5 e 6 creano due nuovi array, `s1` e `s2`, a partire da espressioni matematiche che coinvolgono `t`, `v` e `a`. Nel primo caso, gli elementi dell'array `s1` sono pari al risultato del prodotto degli elementi dell'array `t` per il valore di `v`: il prodotto è svolto elemento per elemento. Eseguiamo il programma per verificarlo.

Nel secondo caso, alla riga 8, il primo addendo dell'espressione che definisce `s2` equivale all'array `s1` già calcolato. Il secondo addendo è invece ottenuto elevando al quadrato tutti gli elementi dell'array `t`, poi moltiplicandoli per il valore di `a` e per 0,5.

Notiamo che sia `s1`, sia `s2` sono array che hanno la stessa lunghezza di `t`.

Vediamo ora un ultimo esempio.

```
1  import numpy as np
2
3  array = np.linspace(0, 5, 6)
4  print(np.sqrt(array))
5
6  alfa = np.pi/2
7  print(np.sin(alfa))
```

Riga 3 crea un array `array` mediante le funzione `np.linspace`. La riga 4 stampa il risultato dell'applicazione della funzione radice quadrata all'array. Il risultato, come possiamo vedere eseguendo il programma, è un array che ha per elementi le radici quadrate degli elementi di `array`.

Infine, nella riga 6 definiamo la variabile numerica `alfa` che contiene il valore della costante matematica pi greco diviso per 2. Numpy contiene varie costanti predefinite e `np.pi` corrisponde al pi greco. La riga 7 stampa il valore del seno applicato alla variabile numerica `alfa` e possiamo verificare, con l'esecuzione del codice, che il risultato è correttamente pari a 1.

IL MODULO MATPLOTLIB

Matplotlib è un modulo specifico per la creazione di grafici.

Tra le prerogative con le quali è stato creato matplotlib troviamo quella di creare grafici di alta qualità scrivendo un codice semplice e chiaro in Python.

Nell'ultimo decennio matplotlib è divenuto uno tra i moduli grafici più utilizzati in tutti gli ambiti tecnici e scientifici, dalla biologia all'ingegneria, dalla fisica alla scienza dei dati. Il sito ufficiale di matplotlib è *https://matplotlib.org*.

CREARE UN GRAFICO

La procedura di creazione di un grafico è la medesima per tutti i tipi di grafici. Vediamo un esempio pratico. Prima di creare il grafico ricordiamo di importare il modulo matplotlib con l'istruzione import `matplotlib.pyplot as plt` che assegna l'alias `plt` al nome del modulo.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.linspace(0,1,10)
5  y = x**2
6
7  figura, grafico = plt.subplots(figsize=(10,6))
8  grafico.plot(x, y)
9  plt.show()
```

Le righe 4 e 5 creano due array, `x` e `y`: il primo contiene 10 numeri equi-spaziati tra 0 e 1, mentre il secondo contiene i loro quadrati.

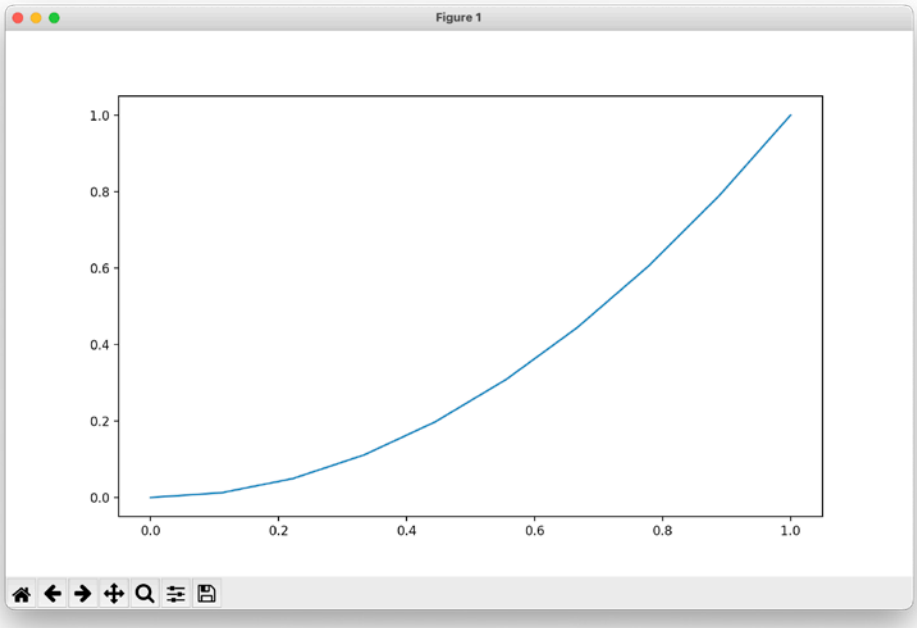
La riga 7 crea la figura `figura`, all'interno della quale sarà collocato il grafico, mentre il grafico vero e proprio è un oggetto che viene assegnato alla variabile `grafico`. La creazione è implementata dalla funzione `plt.subplots`, alla quale passiamo le dimensioni della figura `figsize=(10,6)`. In questo esempio 10 è la larghezza della figura espressa in pollici e 6 è l'altezza sempre in pollici. Un pollice equivale a 2,54 cm.

Nelle schede successive non opereremo mai su `figura` ma soltanto su `grafico`.

Nell'esempio che stiamo discutendo vogliamo rappresentare i valori dell'array `y` in funzione dei valori dell'array `x`. Notiamo che i due array hanno la stessa lunghezza quindi per ogni elemento di `x` abbiamo un elemento di `y` e viceversa. Ciascuna di queste coppie di elementi definisce una coppia di coordinate che rappresenta un punto nel piano: in totale abbiamo 10 punti. Se i due array avessero lunghezze diverse, non sarebbe possibile creare un grafico in questo modo.

La riga 8 rappresenta i 10 punti in un grafico: notiamo che per ogni punto è tracciato il segmento che lo congiunge al punto successivo. Abbiamo ottenuto un grafico a linea.

La riga 9 occorre per mostrare il grafico all'interno dell'apposita finestra che Thonny apre quando eseguiamo il programma. Possiamo osservare la finestra nella figura sotto.



La finestra mostra il grafico voluto e presenta alcuni tasti in basso a sinistra. I primi 5 tasti da sinistra consentono la navigazione e lo zoom del grafico, mentre il tasto a destra (icona del floppy disk) permette di salvare il grafico all'interno di un file di tipo immagine.

In questo esempio la funzione `grafico.plot` ha creato un grafico a linee. Nella maggior parte delle schede utilizzeremo questa funzione, mentre, per creare altri tipi di grafici si utilizzano altre funzioni, ma la struttura del codice non cambia. Ecco una tabella che riassume le funzioni usate nelle prossime schede e i loro parametri principali.

Funzione	Descrizione	Parametri più importanti
`plot()`	Rappresenta un array di ordinate *y* in funzione di un array di ascisse *x*, tramite linee o punti.	• x • y • marker • color • alpha • linestyle • linewidth • label
`scatter()`	Rappresenta un array di ordinate *y* in funzione di un array di ascisse *x* tramite punti in un diagramma a dispersione, con la possibilità di personalizzare lo stile di ciascun punto.	• x • y • c • s • alpha • marker
`errorbar()`	Rappresenta un array di ordinate *y* in funzione di un array di ascisse *x*, tramite linee o punti. Rappresenta anche le barre di errore, sia lungo le ascisse sia lungo le ordinate.	• x • y • xerr • yerr • ecolor • elinewidth • capsize • marker • linestyle

CREARE PIÙ GRAFICI ALL'INTERNO DI UNA FIGURA

È anche possibile posizionare molti grafici all'interno della stessa figura. Vediamo con un esempio come inserire due grafici.

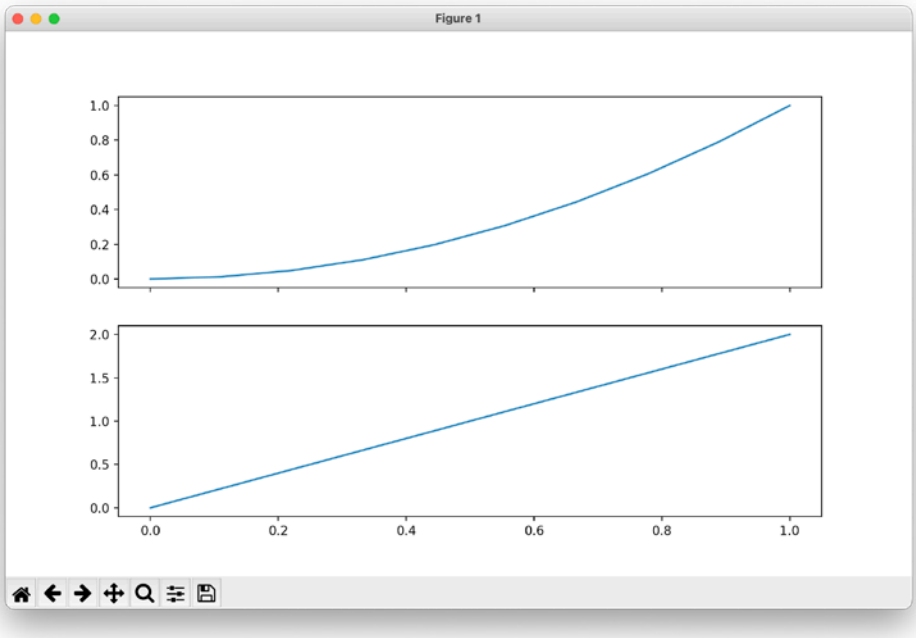
```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.linspace(0,1,10)
5  y = x**2
6  z = 2*x
7
8  fig, grafici = plt.subplots(nrows=2, ncols=1, figsize=(10,6), sharex=True)
9  grafici[0].plot(x, y)
10 grafici[1].plot(x, z)
11 plt.show()
```

Gli array `x` e `y` sono analoghi a quelli dell'esempio precedente, l'array `z` è un nuovo array i cui elementi sono gli elementi di `x` moltiplicati per 2.

La riga 8 crea la figura `fig` e un array di grafici che abbiamo chiamato grafici. Matplotlib gestisce i grafici multipli mediante array di grafici. I due grafici sono posizionati all'interno della figura secondo una griglia immaginaria. In questo caso, dato che abbiamo due grafici, decidiamo di posizionarli su due diverse righe della griglia (`nrows=2` di riga 8) e sulla stessa colonna (`ncols=1` di riga 8). Infine, vogliamo che i due grafici condividano lo stesso asse delle ascisse (`sharex=True`).

Poiché `grafici` è un array, possiamo accedere ai singoli grafici mediante l'indicizzazione: il primo grafico è `grafici[0]`, il secondo è `grafici[1]`. Quindi realizziamo i due grafici mediante i codici di riga 9 e 10: il primo rappresenta y rispetto a x, il secondo rappresenta z rispetto a x.

Il risultato è il seguente.



Notiamo che l'asse *x* è comune a entrambi i grafici.

ALTRI ELEMENTI GRAFICI

Per rendere comprensibile e completo un grafico occorre inserire altri elementi grafici come etichette descrittive degli assi, un titolo, una legenda oppure una griglia. Per inserire questi elementi matplotlib ci mette a disposizione numerosi metodi: i principali che useremo nelle prossime schede sono riassunti nella tabella sotto.

Metodo	Descrizione	Parametri più importanti
`set_xlabel()`	Assegna un'etichetta all'asse cartesiano *x*	• label • fontsize
`set_ylabel()`	Assegna un'etichetta all'asse cartesiano *y*	• label • fontsize
`set_title()`	Assegna un titolo al grafico	• label • fontsize
`minorticks_on()`	Mostra le tacche secondarie sugli assi	
`legend()`	Inserisce una legenda	• fontsize • loc
`grid()`	Inserisce una griglia	• which • axis • linewidth • linestyle
`set_aspect("equal")`	Disegna il grafico con le stesse proporzioni di scala per i due assi	–

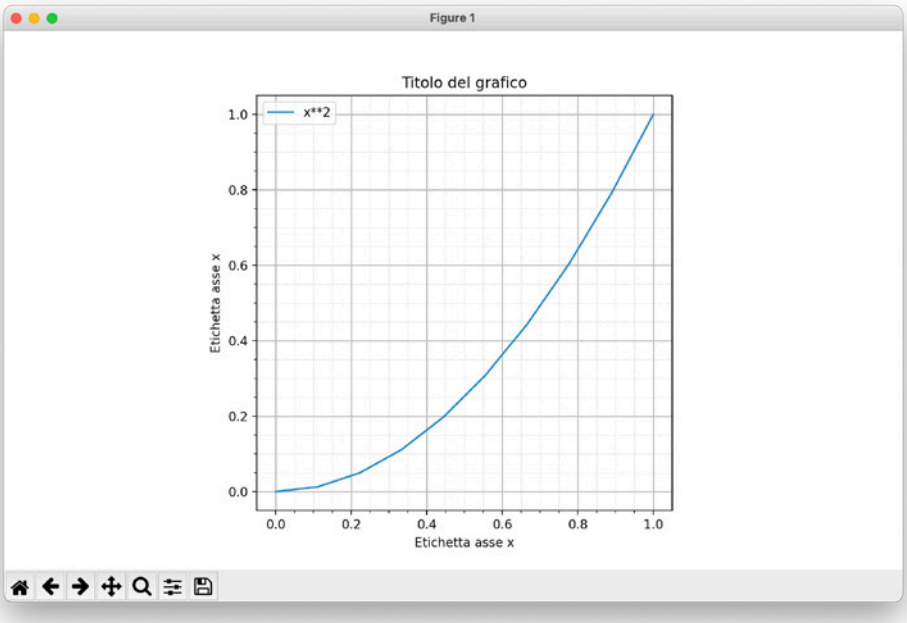
Vediamo un esempio pratico nel quale aggiungiamo elementi grafici al grafico del primo esempio della sezione su matplotlib.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.linspace(0,1,10)
5  y = x**2
6
7  figura, grafico = plt.subplots(figsize=(10,6))
8  grafico.plot(x, y, label="x**2")
9  grafico.set_xlabel("Etichetta asse x")
10 grafico.set_ylabel("Etichetta asse x")
11 grafico.set_title("Titolo del grafico")
12 grafico.minorticks_on()
13 grafico.grid(which='major', axis='both', linewidth=1.0)
14 grafico.grid(which='minor', axis='both', linestyle ="--", linewidth=0.2)
15 grafico.set_aspect("equal")
16 grafico.legend()
17 plt.show()
```

Ora abbiamo aggiunto:

- una etichetta per l'asse *x* (riga 9);
- una etichetta per l'asse *y* (riga 10);
- un titolo (riga 11);
- le tacche secondarie sugli assi (riga 12);
- una griglia in corrispondenza delle tacche primarie che ha linee di spessore pari a 1.0 (riga 13);
- una griglia in corrispondenza delle tacche secondarie che ha linee di spessore pari a 0.2 e ha uno stile tratteggiato, `linestyle="--"` (riga14);
- riga 15 che imposta proporzione uguali per entrambi gli assi;
- una legenda (riga 16). La legenda usa come etichetta la stringa `label="x**2"`, passata alla funzione `plot` nella riga 8.

Il risultato è il seguente.



Nelle schede successive adotteremo alcuni di questi elementi aggiuntivi.

Un altro elemento grafico utile in alcune circostanze è costituito dalle figure geometriche. Per esempio, proviamo a disegnare un rettangolo e un cerchio colorati all'interno del grafico.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4
5  figura, grafico = plt.subplots(figsize=(10,6))
6  rettangolo = plt.Rectangle((0.2,0.2),0.3,0.5,color="red")
7  cerchio = plt.Circle((0.6, 0.4), 0.2, color="blue")
8  grafico.add_artist(rettangolo)
9  grafico.add_artist(cerchio)
10 grafico.set_aspect("equal")
11 plt.show()
```

In questo caso non usiamo array di numpy poiché indicheremo esplicitamente le coordinate dei nuovi elementi grafici.

La riga 6 definisce un rettangolo mediante la funzione `plt.Rectangle`, alla quale passiamo le coordinate del vertice in basso a sinistra `(0.2,0.2)`, la base `0.3`, l'altezza `0.5` e il colore rosso. La riga 7 si occupa della definizione del cerchio con la funzione `plt.Circle`. A essa passiamo le coordinate del centro `(0.6,0.4)`, il raggio `0.2` e il colore blu.

La riga 8 aggiunge il rettangolo al grafico e la riga 9 aggiunge il cerchio. Ecco il risultato.