CORSO DI PROGRAMMAZIONE IN PYTHON PER L' ANALISI DI DATI AGROMETEOROLOGICI

Bologna, 6-8 giugno 2023

Testo base

Python

Python e' un moderno linguaggio di programmazione, creato da Guido van Rossum attorno al 1990. Ha avuto grande successo, anche grazie alla diffusione di applicativi per internet gratuiti che lo utilizzanvano, come Plone (vedi: https://plone.org/)

E' un linguaggio semplice, facile da imparare, va un po' su tutti i computer (anche sul cellulare) ed e' correlato da un sacco di strumenti per fare un po' di tutto: dai siti internet al calcolo scientifico; e' gratuito.

Il nome deriva da uno show comico trasmesso dalla BBC: "Monty Python's Flying Circus", di cui Van Rossum era appassionato.

Il linguaggio e' attualmente in evoluzione, e gestito dalla "Python Software Foundation", (vedi: Python.org) una organizzazione non commerciale, presieduta da van Rossum e sponsorizzata da entita' come Google. Facebook, Microsoft, NVidia, IBM, Red Hat, Oracle etc. etc. etc.

All'inizio lo sviluppo del linguaggio era molto contenuto e si cercava che ogni versione fosse ben compatibile con le precedenti. Le cose sono cambiate con la versione 3.0, del 2008; essenzialmente per la necessita' di introdurre la gestione di caratteri non solo latini, ma anche giapponesi, cinesi, cirillici etc. Python usa ora la codifica UNICODE che comprende tutti i caratteri possibili.

La versione 3 non e' compatibile con la precedente versione 2; introduce nuovi tipi di dati e ci sono vari cambiamenti, come la diversa sintassi del comando di stampa, che la versione 3 riconosce come errori.

Siccome c'era molto software importante scritto per la versione 2 la transizione e' stata molto lunga, ed ancora si trova in giro software per la versione 2, per cui occorre fare attenzione alla versione di Python che si usa. La Python Fundation ha supportato la versione 2.7 fino al 2020; il linguaggio e' attualmente (2022) alla versione 3.10 e gli vengono fatte nuove aggiunte ogni anno.

Caratteristiche del linguaggio

Una importante caratteristica di Python e' che strutture complesse, come liste e dizionari, sono gia' comprese nel linguaggio come tipi di dati, con tutti gli strumenti per gestirle. Per questo motivo Python e' particolarmente adatto ad applicazioni in cui si devono trattare insiemi di dati eterogenei complicati, con un misto di valori numerici e stringhe. Siccome il linguaggio ha gia' dentro gli strumenti adatti non si deve scrivere software apposito o inserire librerie di software specializzate.

Python e' un linguaggio interpretato, ma con produzioe di un "bytecode".

Un linguaggio interpretato e' tradotto in linguaggio macchina una istruzione per volta, e puo' essere usato in modo interattivo. L'uso di un linguaggio interpretato e' comodo ed immediato, ma occorre evitare strutture cicliche ove le stesse istruzioni vengono ripetute molte volte, dato che devono essere reinterpretate ogni volta.

Nei linguaggi compilati invece il programma viene tradotto tutto insieme (fase di "compilazione"), quindi collegato ad eventuali librerie esterne (fase di "link") ed infine eseguito come un'unica entita'. In caso di modifiche si deve ricompilare tutto, in cambio la compilazione permette una miglior ottimizzazione del programma.

Un programma Python puo' essere eseguito in modo interattivo, ma anche essere messo in un file, ed eseguito, come un'unica procedura del computer. In questo caso, durante la traduzione in linguaggio macchina, viene prodotto un "bypte code", ovvero una versione del programma semi-compilata, posta in un file con estensione "pyc". Se il programma viene eseguito una seconda volta, senza modifiche, si utilizza direttamente la versione pre-compilata, con notevole vantaggio in termini di prestazioni.

Altre caratteristiche importanti sono il fatto di essere un linguaggio ad oggetti (cioe' fatto di parti indipendenti e riutilizzabili); la possibilita' di integrarlo con altri linguaggi ed estenderlo. Poi ha un sistema di "garbage collection", per liberare memoria inutilizzata; la gestione di errori ("eccezioni"); il "dynamic typing", cioe' il fatto che il tipo degli oggetti e' controllato solo durante l'esecuzione del programma; in questo modo si possono scrivere parti di programma in modo indipendente dal tipo di dato che tratteranno (polimorfismo). Poi abbiamo

l'introspezione, ovvero la possibilita' di esaminare e variare la struttura interna del programma; la possibilita' di ridefinire gli operatori ("overloading"); la possibilita' di inserire l'interprete Python in programmi scritti in altri linguaggi ("embedding"). Un programma Python puo' anche modificare se stesso mentre corre (*metaprogramming*).

Python contiene gia' una vasta libreria di funzioni, ma moduli Python per fare praticamente qualunque cosa sono liberamente disponibili. Ci sono anche software specifici per gestire i moduli ausiliari ed installarli nel computer.

Riferimenti

La letteratura relativa a Python e' sterminata, anche su internet si trova molto, troppo per essere elencato. Un vecchio corso di Python, e' sul mio sito internet: www.helldragon.eu tratta anche argomenti non inseriti in questo testo, come la programmazione ad oggetti.

Il sito ufficiale di Python e': Python.org ove si trova anche tutta la documentazione , aggiornata.

Il Python Package Index (PyPI): pypi.python.org e' l'archivio ufficiale dei pacchetti ausiliari. Ce ne sono decine e decine di migliaia, per fare qualunque cosa.

Il sito della comunita' italiana e' Python.it qui si trovano informazioni e documentazione in italiano. Qui si trova anche un buon elenco di testi su Python.

Sono stati pubblicati moltissimi libri su Python, di molti testi inglesi ci sono traduzioni in italiano, ma non sempre sono aggiornate. Siccome, con la versione 3, Python e' abbastanza cambiato occorre porre attenzione alla data di pubblicazione, e che coprano la versione 3 del linguaggio.

Fra gli innumerevoli testi segnalo:

- **** Programmare con Python. Guida completa, di Marco Buttu, LSWR 2014**.**

  E' un libro recente, in italiano, ben fatto, ma un po' difficile per i neofiti.

- **** Python for Data Analysis, by by Wes McKinney, O-Reilly 3rd edition.**

  Questo libro e' stato pubblicato nel 2022, ma e' anche disponibile on-line sul sito dell'autore: https://wesmckinney.com/book/

Installazione di Python

Per il corso serve avere installato Python della versione 3 (non la 2) ed i pacchetti ausiliari *numpy*, *pandas* e *matplotlib*.

E' utile un ambiente di sviluppo grafico e comodo, fra questi c'e' *Idle* che e' distribuito ed installato assieme a Python. Un altro e' *spyder*, scaricabile da: https://www.spyder-ide.org/ , Nel corso utilizzeremo *spyder*

Python e' disponibile praticamente per tutte le architetture, sia su Windows che su Mac o Linux, ci sono anche diverse *app* per cellulari android, ad esempio: "Pydroid" o "QPython", ma vi propinano pubblicita'.

Ci sono diversi modi per installare Python, un sistema comodo, che consiglio, e' usare la distribuzione: "*Anaconda*", che permette un uso grafico, ma richiede 3.5 GB sul disco; si recupera da: https://www.anaconda.com/

Il alternativa c'e' "*Miniconda*" che richiede solo 400 MB sul disco ed ha un'interfaccia testuale, vedi: https://docs.conda.io/en/latest/miniconda.html

Se si ha poco spazio sul disco si puo' anche fare un'installazione minimale, che richiede un 100MB e fornisce la shell di Python e l'interfaccia Idle, che viene installata assieme al linguaggio. In questo caso si scarica Python direttamente da: python.org o da python.it

Installazione di Anaconda su Windows

Si scarica l'installer da: https://www.anaconda.com/ e si esegue. Si puo' scegliere se installarlo per tutto il sistema o solo per un utente; in questo caso tutto viene messo nella cartella: C:/Users/nomeutente/anaconda. Nella finestra "advanced Options" si possono lasciare i default.

Nei menu dei programmi di Windows vi ritrovate ora: le voci: "*Anaconda prompt*", "*Anaconda power shell*", "*Spyder*".

"Anaconda prompt" e "Anaconda power shell" permettono di dare comandi per gestire il sistema, corrispondono al "command prompt" ed alla "poewer shell" di Windows. In particolare permettono di usare il programma "*conda*" per la gestione dei pacchetti ausiliari. Ad esempio per aggiornare o installare i pacchetti ausiliari si danno i comandi:

```
conda install numpy
conda install matplotlib
conda install pandas
```

Spyder e' un ambiente grafico per Pyton.

Installazione minimale su Windows

Un file eseguibile, che installa python si puo' scaricare da http://www.python.it/download/ Oppure dal sito ufficiale www.python.org .

Se si ha una computer a 64 bit (come quasi tutti quelli di oggi) conviene scegliere la versione a 64 bit; ovvero un file tipo: "*python3.10.5-amd64.exe*" piuttosto che "*python3.10.5.exe*" che e' a 32 bit.

Per Windows XP occorre una versione precedente alla 3.5. per Window 7 occorre una versione precedente alla 3.9, le varie versioni si trovano tutte su python.org

Fra le opzioni di installazione, raggiungibili eseguendo o ri-eseguendo il file di installazione, conviene attivare:

- aggiungere Python al path
- abilitare python launcher

- associare files con python
- aggiungere python alle variabili di ambiente
- la precompilazione della libreria standard puo' esere utile per rendere Python piu' veloce, ma non e' essenziale.

L'installer fornisce 4 componenti:

- **Python :** *la semplice shell*
  (interfaccia interattiva al linguaggio)
- **Idle :** *un'interfaccia che ci consente di scrivere*
  e gestire files contenenti programmi Python
- I manuali : tutta l'ampia documentazione sul linguaggio
- Documentazione sui moduli, tramite links al sito ufficiale.

Una volta istallato python, si installano i moduli ausiliari. Occorre aprire l'applicativo "Command prompt" e nella finestra che appare dare i comandi:

```
python -m pip install numpy
python -m pip install matplotlib
python -m pip install pandas
```

spyder si scarica da: https://docs.spyder-ide.org/current/installation.html

Installazione su sistemi Linux

Quasi tutte le distribuzioni Linux hanno Python gia' installato, si puo' usare il gestore di pacchetti preferito per controllare di aver installata la versione 3 del linguaggio ed i pacchetti *numpy*, *matplotlib* e *pandas* che servono per il corso. Anche *spyder* e' in genere compreso nelle distribuzioni Linux.

Installazione su macOS

Sul sito di python c'e' anche un installer per Mac: https://www.python.org/downloads/macos/

Come usare Python

Usando la shell del linguaggio

Il linguaggio si puo' usare in modo interattivo. Se avete Spyder, o Idle, avete anche una finestra in cui scrivere istruzioni Python, che verranno interpretate ed eseguite riga per riga. Il risultato, o l'oggetto ottenuto dall'istruzione, appaiono nella finestra stessa.

L'uso interattivo di Python si ha anche dando il comando "*python*" nella finestra del "command propt" di Windows.

In Linux, si fa lo stesso in una finestra a comandi (un terminale). Per certe distribuzioni, che hanno anche la versione 2 di Python, si deve dare il comando: "*python3*".

Invece di usare la finestra a comandi di python si puo' usare quella di IPython, una estensione all'interprete che ha alcune aggiunte utili. In questo caso si da il comando: "*ipython*", o "*ipython3*" a seconda del sistema.

Spyder utilizza IPython.

Esempio di Uso interattivo in Linux:

```
$ python3
>>> 3+2
5
>>> a=16
>>> a
16
>>> Cntrl/D per finire
```

Programmi python in un file

Si puo' mettere un programma Python in un file e poi chiamare Python per eseguire il file.

Un file con un programma Python: *file.py* si esegue con:

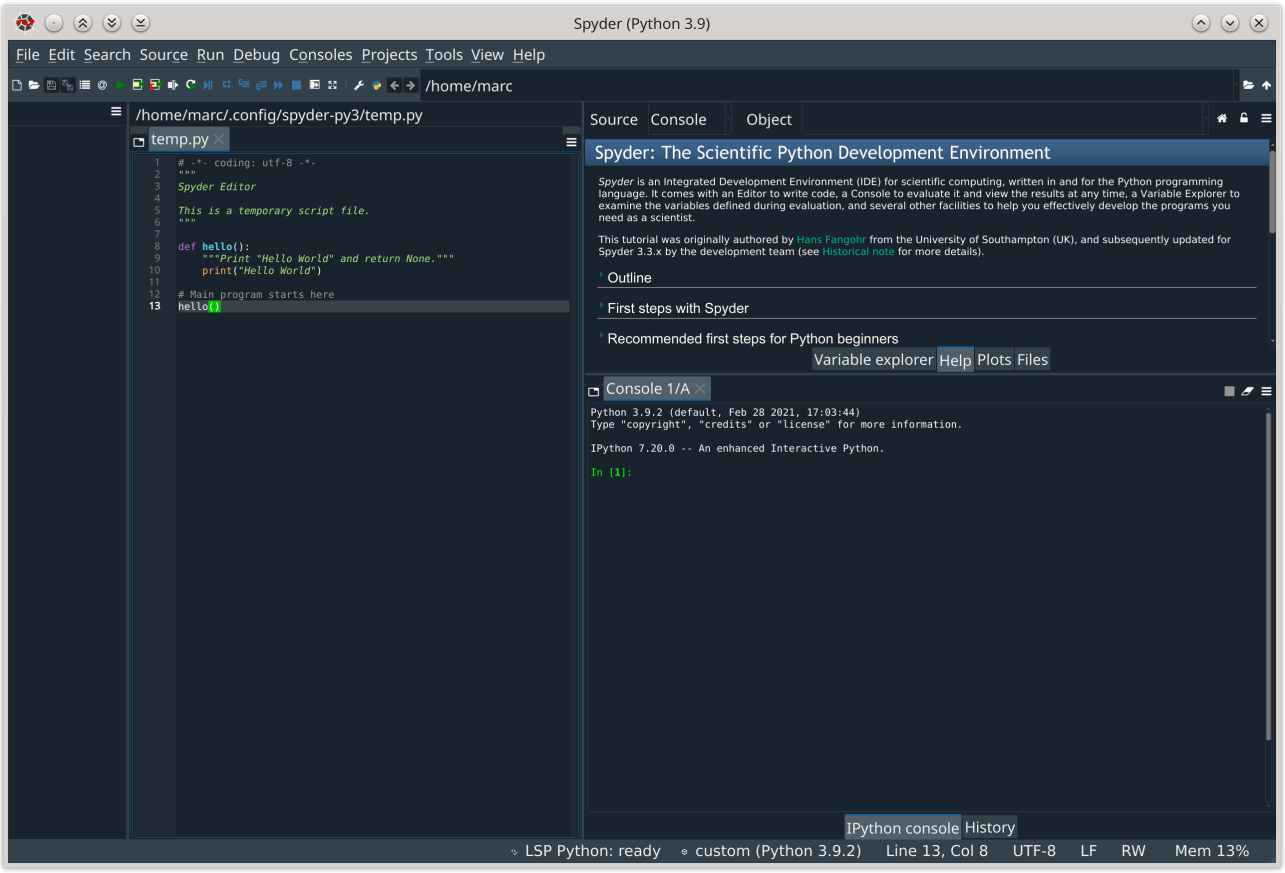
```
$ python3  file.py
```

In Linux un file eseguibile con, nella prima linea: "*#!/usr/bin/python*" , puo' essere eseguito direttamente come un comando.

Interfacce grafiche

Interfacce grafiche come Idle, o Spyder, rendono disponibile una shell per eseguire programmi o singoli comandi, ed un editor per scrivere file con programmi Python.

Nella figura che segue l'interfaccia grafica di *spyder*: a destra un editor per fare files con istruzioni python,in alto a sinistra una finestra per aiuti, grafici etc. in basso a sinistra avete una finestra per dare direttamente istruzioni python, o far correre il programma che editate nella finestra a destra.



Spyder: interfaccia grafica

Sintassi

Variabili

Python e' **case sensitive**

I nomi delle variabili iniziano con un carattere alfabetico e contengono caratteri, numeri o underscore: "_" . Nomi che iniziano con "_" hanno significati speciali, ed alcuni sono definiti da Python stesso.

Una linea che termina con il carattere: "****" continua nella successiva; espressioni fra parentesi possono occupare piu' righe e linee vuote vengono ignorate. Piu' istruzioni possono stare sulla stessa riga, se separate da punto e virgola "**;**"

I commenti sono identificati dal carattere: "**#**", ed il commento va dal carattere a fine linea. Nell'ambiente Unix, se si crea un file eseguibile e si mette: **#!/usr/bin/python** all'inizio del file, si segnala al sistema che il file e' un programma da eseguire con Python.

Caratteri bianchi all'inizio di una riga sono utilizzati per definire blocchi logici del programma.

Oggetti

Python e' orientato alla programmazione ad oggetti, nel senso che tutti i tipi di variabili e tutte le entita' su cui Python opera sono,o si comportano, come oggetti.

La programmazione ad oggetti separa il programma in parti indipendenti, isolate, accessibili tramite un'interfaccia ben precisa. Quando si programma ad oggetti prima si definisce la struttura dell'oggetto, poi dell'oggetto si possono avere tante copie indipendenti (istanze). Un oggetto e' accessibile tramite sue funzioni o sue variabili (attributi).

Oggetti si possono costruire partendo da altri oggetti, ed aggiungendo proprieta'; il nuovo oggetto e' il *child* (figlio) che *eredita* attributi da un oggetto padre (parent).

La sintassi di Python per gli oggetti e' di indicare il nome dell'oggetto e poi il nome dell'attributo, separato da un punto. Ad esempio *car.wheel* , *car.go()* indicano rispettivamente l'attributo *wheel* dell'oggetto *car* e la sua funzione *go*.

Nel corso non parleremo di programmazione ad oggetti, tuttavia occorre tener conto che tutto quello con cui abbiamo a che fare alla fine e' un oggetto, e che incontreremo sempre questa notazione ed useremo spesso il termine *oggetto*.

Oggetti e riferimenti

Quando in Python diamo un'istruzione del tipo:

```
a=3
```

possiamo pensare di star creando una variabile di nome "*a*", che vale 3, ma in realta' "*a*" e' un riferimento (un indirizzo) che punta al valore "*3*", internamente Python lavora tutto a riferimenti c'e' un sistema che conta i riferimenti, e quando un oggetto non ha piu' riferimenti che puntano a lui viene distrutto. Questo si chiama "*garbage collection*".

Oggetti mutabili ed immutabili

Oggetti base in Python (fra cui numeri e caratteri), sono **immutabili**, nel senso che una volta creati non si possono cambiare; le variabili sono riferimenti a questi oggetti, e maneggiare le variabili non tocca gli oggetti cui si riferiscono. Salvo che quando un oggetto non ha piu' riferimenti e viene eliminato.

Oggetti **mutabili** sono oggetti composti, costruiti internamente con insiemi di riferimenti ad oggetti immutabili. Questi si possono modificare durante il programma.

Keywords

Python ha pochi comandi (keywords), questi non si possono usare come nomi di variabili; in Python 3.10 sono:

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

Tipi

In Python ci sono alcuni tipi base, come interi, float, caratteri etc., ma ogni oggetto definisce implicitamente un tipo di variabili. Alle variabili non e' assegnato un tipo a priori ed il tipo delle variabili non va dichiarato, ma quando ad una variabile viene assegnato un oggetto viene definito anche il tipo della variabile. Questo viene chiamato "*run-time binding*", "*late binding*" o "*dynamic binding*", e permette al programma di essere scritto in modo indipendente dai tipi di variabili (*polimorfismo*). Ovviamente non si riescono a fare operazioni che non hanno senso fra tipi diversi (come sommare lettere e numeri), ed il Python in questi casi da errori, che possono essere rilevati solo all'esecuzione del programma, visto non c'e' una compilazione e l'interprete non sa i tipi delle variabili e non fa controlli fino all'esecuzione dell'istruzione.

Python, se puo' , effettua automaticamente la conversione fra tipi nelle operazioni numeriche.

Python ha anche tipi che sono **sequenze**; le sequenze sono insiemi di elementi ordinati, cui ci si puo' riferire tramite indici interi. Gli indici partono da zero, analogamente a quanto accade nel linguaggio C, non da uno. Indici negativi partono dal fondo della sequenza. Gli indici sono rappresentati con numeri, o variabili, fra parentesi quadre. Ad esempio: se la variabile 'a' rappresenta una sequenza, il suo primo elemento sara': a[0], il suo secondo elemento a[1] e cosi' via.

I tipi di base sono:

- **int** : interi con segno (immutabili),

  possono avere un numero illimitato di cifre; se preceduti da 0O oppure 0o sono in notazione ottale, se preceduti da 0x oppure 0X sono in notazione esadecimale, se preceduti da 0b oppure 0B sono binari

  Esempi:

  7 ; 2147483647 ; 0o177 ; 0b100110111 ; 0xdeadbeef 792281625142643375935439503360

  oct(x) visualizza il numero x in ottale,

  hex(x) in esadecimale

  bin(x) in binario

- **float** : numeri reali, in doppia precisione, (immutabili),

  l'esponente e' preceduto dalla lettera **E** on **e** e segue la parte frazionaria.

  Esempi:

  3.14 ; 10. ; .001 ; 1e100 ; 3.14e-10 ; 0e0

- **complessi** : sono somma di reale e di immaginario (immutabili),

  la parte immaginaria e' seguito da **j** oppure **J**

Esempi:

1+3.14j ; 10.j ; 1.e2+10j ; 0+.001j ; 1e100j ; 3+3.14e-10j

- **decimali e frazionari** :

  sono tipi numerici supportati dalla libreria standard, ove sono create le classi relative. I decimali hanno un numero fisso di cifre decimali, i frazionari sono frazioni, usati per rappresentare senza approssimazione i numeri razionali.

- **bool** : booleani :

  possono essere veri o falsi; assumono uno dei 2 valori: "*True*", "*False*". Il numero "0" e' considerato falso, altri numeri sono considerati veri, una stringa e' vera, un oggetto vuoto e' falso.

- **None** : e' una variabile speciale che indica l'assenza di un valore:

  E' usata specialmente per stringhe ed e' il tipo: "*NoneType*"; la variabile None e' considerata falsa.

- **str** : stringhe :

  sono sequenze di di caratteri (immutabili). Sono in codifica unicode in Python3, in codifica ascii in Python 2. Le stringhe, intese come sequenze di caratteri, permettono di accedere a singoli caratteri tramite un indice intero. Le stringhe sono rappresentate con caratteri fra apici doppi o semplici.La stringa fra doppi apici puo' contenere apici semplici e viceversa

  Esempi:

  "abcd" ; '123AAQH' ; "123'34"

- **bytes** : sequenze di interi, nel range 0-255, (immutabili).

  Esistono solo in Python3, ma non in Python2. Siccome le stringhe in Python3 sono in codifica Unicode non sono piu' uno strumento adatto a trattare piccoli interi, per questo e' stato introdotto questo tipo di dati.

- **byte array** : e' un tipo analogo al tipo byte, ma mutabile

- **list** : sono sequenze di oggetti eterogenei (mutabili).

  Sono rappresentate come elementi separati da virgole, racchiusi fra parentesi quadre. Le liste possono contenere ogni tipo di oggetto, liste comprese.

  Esempi:

  [0,1,2,3,4] ; [1,2,'abc/,12.5E3] ; [0,1,2,['a','b','c'],32.4]

- **tuple** : sono sequenze di oggetti eterogenei immutabili.

  Sono analoghe alle liste, ma sono rappresentate con valori racchiusi fra parentesi tonde. Le tuple possono contenere liste, che sono mutabili, la lista nella tupla puo' mutare, ma non essere tolta dalla tupla, che e' immutabile.

  Esempi:

  (0,1,2,3,4) ; (1,2,'abc/,12.5E3) ; (0,1,2,('a','b','c'),32.4)

- **dict** : dizionari, od array associativi.

  Sono insiemi i cui elementi non hanno come indice un numero, ma sono individuati da un oggetto, detto chiave (key). Gli elementi di un dizionario non hanno un ordine definito, come le sequenze. Anche i dizionari contengono oggetti eterogenei, ma le chiavi devono essere oggetti immutabili; siccome internamente i dizionari sono rappresentati come "hash tables", le chiavi devono essere oggetti "hashable", cioe' adatti ad essere trasformati in indirizzi dagli algoritmi interni di Python.

  I dizionari sono rappresentati come coppie "chiave:valore", con elementi separati da virgole e fra parentesi graffe. Un elemento del dizionario e' individuato dalla chiave, messa fra parentesi quadre dopo la variabile che si riferisce al dizionario

  Esempio:

D={'key1':3,'key2':6,1:'abc'}

qui D['key1'] si riferisce al numero 3, D[1] alla stringa 'abc'

- **set** : insiemi.

  Sono gli insiemi della matematica insiemistica. Sono composti di oggetti non sono ordinati che non sono recuperabili con un indice; su di loro si possono fare operazioni come: unione, intersezione, differenza etc. etc.

  Sono oggetti mutabili, ma che contengono oggetti immutabili ed in un unica copia. Sono rappresentati da elementi fra parentesi graffe, separati da virgole.

  Esempio:

  a={1,2,3,'cc',1,2}

  a contiene: {'cc', 1, 2, 3} : non ci sono elementi multipli

- **frozenset** : sono come i set, ma immutabili.

  Si possono creare da una lista, tupla o dizionario, con la funzione frozenset.

  Esempio:

  a=frozenset([1,2,3])

- **range** : sono oggetti che permettono di iterare su sequenze di interi.

  Sono creati con la funzione range, che ha come argomenti il primo, l'ultimo valore (escluso) ed il passo della sequenza.

  Esempio:

  range(2) e' la sequenza: 0,1

  range(2,4) e' la sequenza 2,3

  range(1,5,2) e' la sequenza 1,3

Stringhe, liste, tuple, range, sono sequenze. Le sequenze, ma anche i dizionari e gli insiemi sono detti **iterable** , sono cioe' oggetti i cui elementi si possono estrarre uno per volta, scorrendo sulle componenti dell'oggetto. Questo perche' sono oggetti che implementano tutte le funzioni per fare questo. Questo modo di classificare gli oggetti in base alla loro interfaccia e' chiamato "**duck typing**": qui e' il comportamento di un oggetto che definisce il suo tipo. E' un concetto che si ritrova nei moderni linguaggi interpretati; una specie di polimorfismo, ma questa e' un'idea ancora piu' generale e l'interfaccia stessa diventa la definizione del tipo di dato.

La funzione **type(oggetto)** dice di che tipo e' l'oggetto, la funzione **id(oggetto)** e' un identificatore unico per l'oggetto, in pratica: l'indirizzo. La funzione **isistance(ogetto,tipo)** e' un test sul tipo:

Esempio:

type([]) produce: <class 'list'>

isinstance([], list) produce: True

id([]) e' l'indirizzo della lista vuota, tipo: 140377705648640

Esistono funzioni specifiche per la conversione di valori fra tipi diversi. nelle operazioni aritmetiche ed i confronti.

Docstring:

All'inizio del file, o di una funzione o classe, e' buona pratica mettere una stringa descrittiva, anche su piu' linee. Questa finisce nella variabile **__doc__** dell'oggetto o della funzione e serve da documentazione, E' mostrata dalla funzione help con l'oggetto come argomento. Questa stringa descrittiva si chiama "*docstring*".

Operatori

In Python ha molti operatori per eseguire calcoli numerici o logici. Certi operatori hanno significati diversi se operano con oggetti diversi

Operatori di assegnazione

Oltre al segno di eguale, che definisce una variabile o ne cambia il valore, ci sono operatori che effettuano un'operazione su una variabile e poi assegnano il nuovo valore Ad esempio: "*a += 10*" aggiunge 10 ad "a", equivale ad: "*a=a+10*".

In Python si possono fare assegnazioni multiple, ma il numero di oggetti a sinistra e destra deve essere lo stesso: ad esempio: "a,b = b,a" scambia i valori di a e b

Operatori aritmetici

Operazioni fra interi danno interi, se uno dei 2 operandi e' un numero con la virgola (float) viene prodotto un float. Python, quando puo', fa conversioni automatiche fra tipi diversi. Gli operatori aritmetici sono elencati nella tabella seguente.

Operatore	Funzione	Esempi
**	Elevamento a potenza	a**3 ; anche funzione pow(x,y)
*	Moltiplicazione	a*b ; 3*2 => 6
/	Divisione	a/b ; 5/2 => 2.5
//	Divisione intera	a//b ; 5//2.0 => 1.0
%	Resto	a%b ; 5/2. => 1.0
+	Addizione	a+b ; 2+5 => 7
-	Sottrazione	a-b ; 2-5 => -3

Questi operatori hanno significato diverso se applicati ad oggetti diversi, ad esemio l'addizione si usa anche per concatenare stringhe, o sequenze.

Operatori logici

Operatore	Significato	Esempio
or	or logico	x or y
and	and logico	x and y
not	negazione	not x

Gli operatori "*not*" ed "*or*" **restituiscono uno degli argomenti** non, semplicemente, False o True.

```
x or y      Vale x se x e' True,  altrimenti valuta y e restituisce y
x and y     Vale x se x e' False, altrimenti valuta y e restituisce y

x or  y or  z     restituisce il primo vero  (o l'ultimo)
x and y and z     restituisce il primo falso (o l'ultimo)
```

Operatori logici per i confronti.

Restituiscono : True o False

Operatore	Funzione	Esempi
>	maggiore	a > b
<	minore	a < b
<=	minore od eguale	a <= b
>=	maggiore od eguale	a >=b
==	eguale	a == b
!=	diverso	b != b

Si puo' controllare se una variabile e' in un intervallo con la sintassi compatta del tipo: 1 < a < 3

Operatori logici per l'appartenenza

L'operatore "**in**" restituisce True o False a seconda che un oggetto faccia parte di una sequenza, dizionario o set, l'operatore "**is**" controlla se due riferimenti puntano allo stesso oggetto.

Operatori bit a bit

Questi operatori agiscono sui numeri seguendo una logica binaria e cambiano i singoli bit delle variabili. Questi operatori operano su interi con segno o booleani.

Questi operatori utilizzano i simboli:

<< , >> , & , | , ^ , ~ , *

Sequenze

Alcuni tipi di dati sono sequenze di valori eterogenei, cui si puo' far riferimento tramite un indice intero (per liste, stringhe, tuple), o anche usando un generico identificativo (dizionari). Ci sono istruzioni per iterare sui valori delle sequenze, per trasformarle, per combinarle insieme etc. etc. le sequenze possono essere annidate, ovvero sequenze di sequenze.

Python e' quindi ideale per trattare insiemi complicati di dati eterogenei, quando si riesce a farli corrispondere a queste strutture annidate.

Liste

Gli elementi di una lista hanno un ordine e sono accessibili tramite un indice numerico, che inizia da 0. Una lista puo' crescere in modo dinamico, cioe' non si deve decidere all'inizio quanto sara' grande.

Le liste si indicano con una serie di valori, separati da virgole e racchiusi fra parentesi quadre, ad esempio, per definire una lista A , composta dai numeri da 10 a 90:

```
A=[10,20,30,40,50,60,70,80,90]
```

Il primo elemento e': A[0], il secondo A[1] e cosi' via. Si possono estrarre parti di una lista con una sintassi semplice, si ottiene una seconda lista:

A[2:4] elementi dal numero 2 (il terzo) al quattro (escluso) ovvero una lista coi valori 30 e 40.

Indici negativi partono contando dalla fine: A[-1] vale 90 A[-4,-2] sono i numeri: 60 e 70.

Se uno degli indici e' omesso di default si intende l'inizio (o la fine) della sequenza.

A[:3] e' la lista coi numeri: [10,20,30]

A[3:] sono i numeri dal 40 al 90.

Si puo' indicare un passo con cui si selezionano gli elementi:

A[0:8:2] sono i valori [10,30,50,70]

A[::3] sono i valori: [10,40,70]

Il comando *del* elimina elemanti della lista, ad esempio del A[3] elimina l'elemento di indice 3, ovvero il valore 40. L'elemento di indice 3 diventa il successivo, di valore 50.

Ci sono molte funzioni per le liste:

abbiamo: *len(A)* ;la lunghezza della lista, *min(A)* , *max(A)*, i valori massimo e minimo, poi funzioni per ordinare una lista, per mettere gli elementi in ordine inverso, per aggiungere elementi, etc.

Come detto, sui possono fare liste di liste, ad esempio:

```
A=[10,20,30,['a','b','c'],50]
```

qui l'elemento A[3] e' la lista ['a','b','c','d'] A[3][1] indica l'elemento 1 dell'elemento 3, ovvero 'b'

Stringhe

Le stringhe sono a tutti gli effetti liste di caratteri, e sono rappresentate come una sequenza di caratteri fra apici.

Possono essere utilizzati indifferentemente apici singoli, oppure il doppio apice. Se una stringa e' delimitata da apici singoli puo' contenere doppi apici e viceversa, ad esempio sono stringhe valide:

```
'12345"6789'
"12345'6789"
```

Una stringa puo' essere vuota; una stringa vuota e' definita da:

```
a=''
```

Una stringa puo' essere composta da piu' linee, se delimitata da tre apici o tre doppi apici, esempio:

```
''' questa stringa
continua in questa riga '''

"""
altra stringa multilinea
anche questa linea fa parte della stringa """
```

Stringhe una dietro l'altra sono concatenate, anche se sono separate da spazi:

```
a='abcd' 'efg'
a== "abcdefg"
```

Entro le stringhe hanno significato particolare alcune sequenze precedute da backslash : "****" Alcune di queste sono un residuo delle codifiche che venivano utilizzate il controllo del carrello per le stampanti a modulo continuo, altre sono usate per inserire valori in diverse codifiche.

Ad esempio:

\n : indica che si va a capo **\t** : tabulazione orizzontale

La decodifica di questi caratteri speciali non avviene nelle stringhe "raw", che sono indicate dalla lettera *r* prima della stringa, ad esempio:

a=r"12\n4g"

La stampa di questa da una linea sola, invece a="12\n4g" produrrebbe 2 linee

L'operatore "**+**" concatena le stringhe; l'operatore "*****" ripete una stringa un certo numero di volte:

Esempi:

```
a='0123456'
b='abcdefg'

a+b  e' la stringa: '0123456abcdefg'
b*2  e' la stringa: 'abcdefgabcdefg'
```

L'operatore "**in**" da risultato: *True* se una sottostringa e' compresa in una stringa:

Esempi:

```
'0' in a    vale: True
'01' in a   vale: True
'09' in a   vale: False
```

Ci sono molte funzioni per le stringhe; per fare maiuscoli i caratteri, per farli minuscoli, per mettere maiuscolo il primo carattere di ogni parola, per separare una stringa contenente una frase in parole singole, per unire stringhe.

Dizionari

I dizionari somigliano a liste, ma gli indici (detti keys o chiavi) sono stringhe, od altri oggetti Python immutabili, ad esempio tuple. Si possono costruire strutture complesse con dizionari e liste annidate.

Gli elementi di un dizionario non hanno un ordine definito, e per individuare gli elementi si utilizza una coppia di parentesi quadre, che contengono la chiave dell'elemento.

Un dizionario e' rappresentato da coppie di chiavi e valori, separati da virgole, e racchiusi fra parentesi graffe.

Esempio:

```
D={'chiave':3,'altrachiave':6,1:'abcd'}

D['chiave']       : vale 3
D['altrachiave']  : vale 6
D[1]              : vale 'abcd'
```

Come per le liste abbiamo che:

```
D={}             : crea un dizionario vuoto
D['a']=799.0     : aggiunge elemento, se la chiave esiste gia' lo modifica

del D['b']       : elimina l'elemento con chiave 'b'

len(D)           : dimensione del dizionario: vale 3 se D ha 3 elementi

D.pop('c')       : estrae dal dizionario il valore con chiave 'c' e lo elimina.

D.update ( { 1: 10, 2: 20, 3:30 } ) :  unisce dizionari

D.clear()         : svuota il dizionario
```

Se abbiamo dizionari di dizionari possiamo reperire gli elementi del dizionario interno con la sintassi seguente:

```
D3={'cibo':{'carne':1,'frutta':2}} : dizionario che contiene un dizionario

D3['cibo']['frutta']              : e' l'elemento di valore 2
```

La funzione "**in**" e' riferita alle chiavi, ad esempio

```
D4={'a':0,'b':1,'c':3}

'a' in D4      :  e' True
0 in D4        :  e' False, 0 non e' una chiave di ricerca.
```

Tuple

Le tuple sono sequenze simili alle liste, ma, a differenza delle liste, sono sequenze immutabili e non possono essere cambiate nel corso del programma. Contengono oggetti eterogenei, identificati da un indice numerico. Le tuple sono immutabili, ma i loro elementi, se sono mutabili, possono essere cambiati, per cui se un elemento di una tupla e' una lista questa puo' essere alterata e svuotata, ma non puo'essere tolta dalla tupla. Tuple che contengono solo oggetti immutabili possono essere usate come chiavi in un dizionario.

Le tuple sono rappresentate come insiemi di valori separati da virgole e racchiusi fra parentesi tonde; per il resto abbiamo operatori e notazioni analoghe a quelle delle liste, dei dizionari e delle stringhe:

Esempi:

```
T=(1,3.5,'c')   : definisce una tupla di 3 elementi

TT=()           : e' una tupla vuota

T[0]=33         : da errore, la tupla non e' mutabile

T[1]            : il secondo elemento ( nel nostro 3.5)

len(T)          : numero elementi

T.index(3.5)    : indice dell'elemento 3.5 ( e' al posto 1 )
```

L'operatore di somma: '+' concatena tuple in una nuova tupla, l'operatore: '*' ripete una tupla un certo numero di volte:

Esempio:

```
T3=(2,3)
T4=(5,5)
T5=T3+T4
T5        : e' la tupla:  (2, 3, 5, 5)

T6=T3*2
T6        : e' la tupla: (2, 3, 2, 3)
```

Tuple possono essere mutate in liste e viceversa:

```
L=list(T6)  :  muta la tupla in una lista: [2, 3, 2, 3]

TT=tuple(L) : fa il contrario, muta una tupla in una lista.
```

Sets e Frozensets

I sets sono stati introdotti con Python 2.4, e sono insiemi non ordinati, mutabili, di oggetti immutabili (numeri, caratteri, tuple , NON liste e dizionari). Gli oggetti sono unici nel set (non ce ne sono 2 uguali). I set sono gli insiemi della matematica insiemistica e su di essi si possono fare operazioni di unione, intersezione, differenze etc.

Esempi:

```
a=set()             : crea set  vuoto

s={1,2,3}           : crea un set di 3 valori

a=set()             : crea set  vuoto


b=set(['a','b','c']) : set possono essere costruiti da liste
c=set('abf')         : crea il set: {'h','g','f'}

d = b | c  :        set unione : set(['a', 'c', 'b', 'f'])

e = d & b  :        set intersezione set(['a', 'c', 'b'])

d - b      :        set differenza set(['f'])
```

Istruzioni

Blocchi logici

I blocchi logici in Python sono identificati con indentazione (rientro): tutte le istruzioni del blocco hanno davanti lo stesso numero di spazi, ed il blocco finisce quando l'indentazione del blocco cessa.

I blocchi dentro altri blocchi (nested o annidati) hanno ulteriore indentazione rispetto al blocco che li contiene. Si consiglia di usare spazi bianchi e non tabulazione, visto che gli spazi rappresentati dal tasto di tabulazione possono essere diversi a seconda dei computer e dei programmi usati. In genere si usano 4 spazi bianchi per un rientro.

Assegnazione

Abbiamo gia' visto l'istruzione di assegnazione, che crea un oggetto e gli assegna una variabile, che e' un riferimento all'oggetto.

Esempi:

```
a  = 3
b  = 6.0
a *= b
c=(a+b) * 3
a=b=3
```

Esecuzione condizionale

Il costrutto: "*if.. then .. else*" che ritroviamo in tutti i linguaggi: se la prima condizione e' vera (quella che segue l'if) allora e' eseguito il primo blocco, altrimenti si prosegue alla condizione successiva; se nessuna e' vera viene eseguita la condizione che segue l'istruzione "*else*". Le clausole "*elif*" ed "*else*" sono opzionali

I blocchi sono delimitati dall'indentazione ed iniziano con due punti "*:*"

```
if a==b:
   c=d
   e=f
elif a>b:
   c=f
   e=d
else:
   c=0
   e=0
```

Nell'esempio successivo abbiamo un caso di clausole if annidate (nested if). In Python bisogna sempre fare molta attenzione ai rientri. Altri linguaggi usano le parentesi, che sono una scelta piu' comune e meno soggetta ad errori:

```
if a==b:
   c=d
   if e==f:
      k=0
   else:
      k=1
elif a>b:
   c=f
   e=d
else:
   c=0
   e=0
```

Se c'e' una sola istruzione la si puo' mettere nella stessa linea della condizione:

```
if a==b: k=0
```

Istruzioni cicliche

L'istruzione ciclica di Python e' il ciclo "*while"*, che esegue un blocco finche' la condizione e' vera. La condizione e' testata all'inizio del blocco. Alla fine del blocco, **in ogni caso**, viene eseguito il blocco alla istruzione "**else"**. Anche le istruzioni "*while"* possono essere annidate:

```
while x:
   i+=1
   x-=1
else:
   y=i
```

questa incrementa i e cala x di uno, fino a che x e' zero (falso) quando questo accade il ciclo termina e viene eseguita l'istruzione all'else

Entro un ciclo ci sono istruzioni particolari che alterano la sequenza:

```
break       interrompe il ciclo
continue    passa al giro successivo
pass        non fa nulla , ad esempio:

       while 1:pass  # e' un loop infinito
```

Se, nel blocco del while, viene eseguita una istruzione break si esce dal ciclo **senza eseguire il blocco all'else**

Iterabili ed iteratori

Anche qui abbiamo istruzioni cicliche, ma con una logica diversa.

Liste, dizionari, tuple, sets hanno la caratteristica di essere "*iterabili*" (iterable). Un insieme di oggetti e' iterabile se, su di esso, si puo' definire un oggetto che, in sequenza, assume un diverso valore fra quelli dell'insieme. Questo oggetto che "itera" sulla sequenza e' detto "iteratore".

L'istruzione che crea un iteratore e cicla su tutti gli elementi di un insieme e' l'istruzione "*for*".

```
for i in [1,2,3,4]:
   k+=i
else:
   print('fine')
```

Il riferimento "*i*" assume, in sequenza, i valori della lista, e per ogni valore che "*i*" assume si esegue il blocco che segue l'istruzione "*if*". A fine blocco viene eseguito il blocco della istruzione "*else*", a meno che un'istruzione break non interrompa il ciclo.

Un ciclo for su un dizionario restituisce le chiavi, nell'esempio che segue viene stampato 'a', poi 'b', infine 'c'

```
D={'a':0,'b':1,'c':3}
for i in D:
   print(i)
```

Per iterare su una coppia (chiave, valore) di un dizionario bisogna utilizzare una tupla:

```
for (key, value) in D.items():
   print(key, '=>', value)
```

Nel caso seguente iteriamo su una tupla ed usiamo una tupla come iteratore:

```
T = [(1, 2), (3, 4), (5, 6)]
for (a, b) in T:
     print(a, b)
```

Per avere un ciclo su numeri interi, si utilizza la funzione "*range*", oppure la funzione "*enumerate*":

```
for i in range(3):    # produce la sequenza: 0,1,2
     print(i)


for i,a in enumerate(['a','b','c']):
     k[i]=a
```

"*enumerate*" restituisce una tupla (numero crescente, elemento dell'enumerabile);p ai diversi giri abbiamo qui per la tupla con "*i*" ed "*a*":

(1,'a') ; (2,'b') ; (3,'c')

List comprehensions

Sono cicli che assegnano una serie di valori ad una lista, opzionalmente i valori possono essere filtrati da una condizione "*if*":

```
[x for x in range(5) if x % 2 == 0]
```

produce una lista con i numeri pari fino a 5: [0, 2, 4]

Si possono avere cicli annidati:

```
[x + y for x in range(3) for y in [10, 20, 30]]
```

Qui x va da 0 a 3, e, per ogni x, y va da 10 a 30 ed abbiamo la lista;

[10, 20, 30, 11, 21, 31, 12, 22, 32]

Funzioni per iterabili

Ci sono diverse funzioni, utili per lavorare con iterabili.

La funzione **list** crea un lista, a partire da una serie di valori, o da un iterabile.

La funzione **dict** produce un dizionario, a partire da una sequenza di tuple di 2 elementi: (chiave,valore).

range rppresenta una sequenza di interi, il primo argomento e' il primo valore, il secondo l'ultimo (escluso), il terzo e' il passo.

map applica una funzione ad un iterabile

a=map(abs,[-1,2,-3])

list(a) vale: [1, 2, 3]

zip unisce a 2 a 2 due iterabili:

```
Z = zip((1, 2, 3), (10, 20, 30))

list(Z)  vale:  [(1, 10), (2, 20), (3, 30)]
```

La funzione **filter** applica ad un iterabile una funzione logica elemento per elemento e tiene solo gli elementi per cui la funzione e' True:

```
list( filter((lambda x: x>0 ),[1,2,3,-1]) )  produce [1, 2, 3]
```

"*lambda*" e' un modo di definire una funzione in un unica istruzione, come vedremo dopo.

Funzioni exec ed eval

La funzione "*exec*" permette di eseguire una stringa come istruzione Python:

```
exec('e=3')      # e , che non era definito, diventa 3
```

La funzione "*eval*" valuta un'espessione Python, quelle che possono stare a destra di una assegnazione, e ne ritorna il risultato:

```
eval('abs(-3)+2')  # produce il valore 5
```

Funzioni

Le funzioni sono parti di programma a se stanti, che eseguono operazioni in base acerte variabili, che sono date in input (argomenti) e restituiscono un valore.

La sintassi per la definizione di una funzione e' del tipo:

```
def nomefunzione(a,b,c):
    '''  docstring:
    descrizione funzione
    '''
    d=a
    e=b+c
    return d+e
```

L'istruzione **def** crea la funzione e le assegna un nome

Il corpo della funzione e' un blocco che inizia con: *":"* ed e' identificato da un rientro,

Fra parentesi ci sono gli argomenti della funzione; le variabili che sono passate alla funzione per il calcolo.

La *docstring* e; una stringa, eventualmente su piu' linee, che descrive la funzione, e' facoltativa. Viene conservata nell'attributo: *__doc__* della funzione.

La funzione ha anche l'attributo: *__name__* che contiene il nome della funzione stessa.

L'istruzione return specifica il valore che viene restituito dalla funzione. In caso l'istruzione "*return*" non compaia nella funzione, viene restituito l'oggetto vuoto: "None".

Per utilizzare una funzione si usa la sintassi:

```
g=nomefunzione(r,s,t)
```

Il valore che la funzione restituisce e' assegnato alla variabile: *g*.

Argomenti

Le funzioni possono avere valori di default per gli argomenti. Ad esempio una funzione definita con:

```
def somma(a='uno',b='due'):
   return a+b
```

puo' essere chiamata semplicemente con:

```
somma()
```

ed effettuera' la somma dei due argomenti di defaul; siccome sono stringhe le concatena e restitruisce la stringa: *"unodue"*.

Una funzione puo' anche essere chiamata dando valori ai parametri per nome (keyword arguments), con una sintassi tipo:

```
somma(a='sei')
```

in questo caso alla variabile "a" entro la funzione, viene assegnata la stringa "sei".

Una funzione puo' essere definita in modo che i suoi argomenti siano visti, entro la funzione, come una tupla o come un dizionario; le definizioni della funzione avranno in questi casi rispettivamente la sintassi:

```
def func(*nome):

def func(**nome):
```

Nel caso del dizionario gli argomenti sono passati per nome ed i nomi diventano le chiavi del dizionario:

```
def func(*a):
      print(a)
      '''
      Qui in a finisce una tupla di argomenti,
      la chiamata puo' avere numero variabile di argomenti
      func(1,2,3) stampa la tupla: (1,2,3)
      '''


def func(**d):
      print(d)
      '''
      Qui gli argomenti finiscono in un dizionario
      gli argomenti sono passati per nome
      ed i nomi delle variabili sono le chiavi
      func(a=1,b=2,c=3) stampa:   {'a': 1, 'c': 3, 'b': 2}
      '''


def func(a,*b,**d):
      print(a)
      print(b)
      print(d)
      '''
      Vanno prima gli argomenti posizionali,
      poi quelli per la tupla, infine quelli per il dizionario.
      Chiamata come :  func(1, 2,3,4, s=10,q=20 )
      stampa:  a=1  ;  b=[2,3,4] ; d={s:10,q:20}
      '''
```

Questi modo di passare gli argomenti possono essere combinati, in questo caso, nelle chiamate e nella funzione, vanno prima gli argomenti posizionali, poi quelli che fiscono in una tupla, ed infine, con passaggio per nome, quelli che finiscono nel dizionario:

```
def func(a,b,c):  # esempio di funzione
    print(a)
    print(b)
    print(c)


func(1,2,3)           # chiamata con argomenti passati per posizione
func( b=2,a=1,c=3)    # argomenti passati per nome
func(1,c=3,b=2)       # argomenti passati in parte per posizione, in parte per nome
```

Vediamo un esempio piu' complesso di una funzione con argomenti passati in modo diverso:

```
def func(a,b,c=1,d=2,*aa,**bb):
    print( "Argomenti posizionali: a,b: ",a,",",b)
```

```
    print( "Argomenti con default: c,d:",c,",",d)
    print( "tupla con argomenti restanti:",aa)
    print( "dizionario con argomenti keyword restanti:",bb)
    print()
    return
```

Qui gli argomenti *a,b* vanno forniti, ed al minimo vanno forniti due argomenti; gli argomenti: c,d hanno valori di default, altri argomenti finiscono in una tupla *aa*, eventuali argomenti nella forma: *chiave=valore1,chiave2=valore2* finiscono nel dizionario *bb*.

Campo di validita' della funzione (scope della funzione)

Una funzione e' valida dal punto del programma in cui si incontra in poi; quando la funzione viene incontrata viene "eseguita", nel senso che il suo nome (che in realta' e' un riferimento) diviene valido ed ad esso sono associate le operazioni contenute nel corpo della funzione.

Una funzione puo' essere definita entro una funzione, ed allora e' vista solo li'.

Campo di validita' delle variabili (scope delle variabili)

Una variabile definita in una funzione non e' vista da fuori della funzione. Puo' avere lo stesso nome di una variabile esterna senza confusione.

Una variabile definita nel blocco in cui la funzione e' chiamata e' vista entro la funzione, ma non puo' essere modificata entro la funzione.

Se, entro una funzione, una variabile e' definita come **global** sara' **vista anche nel blocco in cui la funzione e' chiamata**.

La dichiarazione di global e' del tipo:

```
global a,b,c
```

Questa regola di scope e' chiamata LEGB: Local, Enclosing, Global, Build-in ed e' il modo di cercare i nomi di Python

In ogni caso una variabile e' locale al file in cui si trova.

Funzioni lambda

Sono funzioni di una sola istruzione, senza nome, con sintassi:

lambda argomento1,argomento1,argomento3: espressione

Esempio:

```
f= lambda x,y : x+y

f(2,3)    produce 5
```

Le Lambda sono usate in contesti particolari, ove e' comodo mettere una piccola funzione in una sola riga.

Files

Ci sono molte funzioni per trattare i files. Di default i files sono intesi come files di testo, in codifica UTF-8

Input/Output da terminale

Ad un programma Python, anche nell'esecuzione interattiva, sono associati uno "standard" output ed uno "standard" input, da cui il programma legge e scrive di default.

Per scrivere sullo standard output si usa la funzione "*print*". In Python 2 "*print*" invece di una funzione era un comando.

La funzione print ha un numero arbitrario di argomenti e li stampa in fila su una linea. Ogni comando di stampa stampa una sola linea, a meno che le stringhe stampate non abbiano dentro il carattere: "\n" , che viene interpretato come un fine linea.

La lettura da terminale si puo' fare, in Python 3, con la funzione "*input*", che legge una linea e la mette in una stringa. La funzione input puo' avere come argomento un "*prompt*" che viene stampato prima della lettura.

Esempi:

```
print(a,b)                  : scrive 2 variabili separate da uno spazio

print("%s xxxx %s" % (a,b)) : scrive a e b separati da xxxx

a=input("=> ")              : legge una linea e la mette nella stringa "a"
                              prima di leggere stampa: "=> "
```

Uso di files

Per accedere ad un file si usa la funzione "*open*", che crea un oggetto file e ritorna un riferimento ad esso. L'oggetto file ha funzioni per accedere al contenuto del file. Ci sono anche funzioni per leggere il file tutto in una volta e metterne il contenuto in una stringa, e funzioni per farne una lista di stringhe con le singole linee.

L'output e' befferizzato, cioe' non scritto subito sul file, ma posto un un'area di memoria apposita (buffer) e scaricato sul file, tutto insieme, in un secondo momento, per ottimizzare i tempi di calcolo. La funzione "*flush*" scarica il buffer subito.

Il file ha un puntatore che ricorda dove si e' arrivati nella lettura, che, a diversi comandi di lettura, si sposta in avanti nel file. La funzione seek sposta il puntatore, permettendo di saltare parti del file o di rileggere contenuti gia' letti.

Il file viene chiuso con la funzione "*close*", che elimina il riferimento al file.

Nella funzione che apre un file si da in argomento il nome del file ed un un carattere che specifica se e' aperto solo per lettura, per scrittura o modifica.

Esempi:

```
f= open('filetest','w')  : apre un file di nome filetest per scriverci.
                           e crea un oggetto f, di tipo "file"

f1=open('filetest2','r') : apre un file di nome filetest per leggerlo
f2=open('filetest3','r+'): per leggere e scrivere
f3=open('filetest4','a') : per aggiungere in fondo al file

f1.close()               : chiude il file. L'oggetto 'f' viene distrutto
```

Una volta che un file e' aperto si puo' leggere o scrivere, ci sono diverse funzione per la lettura e scrittura

```
stringa1=f1.readline()   : legge una linea dal file f1 e la mette in stringa1
stringa2=f1.read()       : mette in stringa2 tutto il file f1

stringhe=f1.readlines()  : legge tutto il file e ne fa una lista
                           ogni elemento della lista e' una linea del file

f.writelines(stringhe)   : scrive, di seguito, le stringhe della
                         lista. Per essere scritte su diverse linee le
                         stringhe devono finire con: "\n"

f.write('stringa')       : scrive sul file una stringa
f.write('stringa\n')     : scrive sul file una linea
                           (\n e' il carattere di fine linea)
```

Moduli

Programmi python possono essere organizzati in files, contenuti in una gerarchia di cartelle.

Un file costituisce un "modulo", un insieme di istruzioni e dati auto-consistente. Un modulo ha un nome che e' il nome del file, senza il suffisso, che per un file con istruzioni Python e': "*.py*".

Nomi con spazi o caratteri speciali non sono accettati.

Diversi moduli sono organizzati in "**packages**", che occupano una directory. Il nome del package e' il nome della directory. Per essere considerata un package una directory deve contenere un file di nome *__init__.py*, che puo' anche essere vuoto, ma in genere contiene istruzioni che che inizializzano il package.

Un package puo' contenere subpackages, in sottocartelle.

Python cerca i files con i moduli nella directory corrente, poi in una serie di cartelle definite dall'installazione di Python. La variabile *"path"* del modulo *"sys"* contiene questa lista di cartelle.

Per utilizzare un modulo in un programma bisogna "importarlo". In modo che Python possa eseguire il codice del modulo, costruire le classi in esso contenute ed organizzare i nomi degli oggetti.

Il comando per importare un modulo in un file *A.py* e':

```
import A
```

a questo punto oggetti definiti nel file *A.py* possono essere utilizzati riferendosi ad essi con un nome tipo: A.nomeoggetto

L'istruzione *import* importa moduli una volta sola nel programma, ulteriori istruzioni *import* per lo stesso modulo non vengono eseguite.

Per riferirsi agli attributi del modulo usando un prefisso a scelta, invece del nome del modulo, si usa la sintassi:

```
import A as newname
```

Qui ci si riferisce ad un attributo con: "newname.attributo" , invece che "A.attributo" , che non vale piu'.

L'istruzione *from* permette di importare solo alcuni oggetti da un modulo, ed i loro nomi vengono riconosciuti senza il nome del modulo come prefisso.

La sintassi e':

```
from nomefile import nome,altronome
```

Per usare un nome diverso per un oggetto importato:

```
from nomemodulo import nomeoggetto as altronome, nomeoggetto2 as altronome2
```

Per importare tutti i nomi del modulo nel namespace corrente:

```
from nomefile import *
```

Se il file con il modulo viene modificato occorre reimportare il file e rieseguire l'istruzione *from*, ma l'istruzione *import* non re-importa moduli; per far questo c'e' una funzione, che fa parte del modulo *imp*:

```
import imp

imp.reload(nomemodulo)
```

Se i moduli sono in una gerarchia di packages (e di directory) si importano con istruzioni tipo:

```
import nomepackage1.nomepackade2.modulo
```

Se si vuole importare un package si usa import con il nome della directory del package e Python esegue il file __init__.py che trova in questa cartella. Spesso questo file importa i singoli file del package.:

```
import nomedir
```

Se si importa un sub-package con un'istruzione tipo:

import nomedir.nomesubdir.nomesubdir

Vengono eseguiti nell'ordine, i files __init__.py che Python trova nelle diverse sotto-cartelle.

Per aggiungere cartelle alla lista di quelle ove python cerca i moduli si fa:

```
import sys
sys.path.append('/dir/subdir/')
```

Libreria standard

Python viene distribuito assieme ad un'ampia collezione di moduli, che costituiscono la libreria standard.

Il modulo "**sys**" contiene variabili e funzioni relative all'interfaccia di sistema che fa partire l'interprete Python, fra queste:

```
import sys

  sys.argv                        : argomenti del programma principale
  sys.exit()                      : esce dal programma
  sys.modules                     : moduli caricati
  sys.path                        : search path dei moduli
  sys.ps1 sys.ps2                 : prompt del python
  sys.stdin,sys.stderr,sys.stdout : input/output di default
```

Il modulo "**os**" ha funzioni di interfaccia con il sistema operativo, ha funzioni per gestire i files etc. etc. puo' fare quasi tutto quello che si fa da una shell di Unix:

```
import os

   os.system('pwd')          : esegue comando di shell
   os.environ                : variabili di ambiente
   os.putev(nome,valore)     : aggiunge variabile di ambiente
   os.uname()                : nel sistema in Unix
```

Per le funzioni trigonometriche e varie funzioni matematiche c'e' il modulo "**math**" , i moduli "**time**" e "**datetime**" serono per data e ora. "**pickle**" e "**json**" sono moduli per la serializzazione (trasformare strutture complesse in stringhe), Ci sono poi moduli per accesso a database, per leggere e scrivere files in formato csv, per costruire applicazioni di rete e tanti altri.

Moduli ausiliari si trovano in rete, in http://pypi.python.org/ c'e' un vasto indice di moduli. Ci sono programmi appositi (easy_install, pip), per recuperare ed installare packages da questo archivio.

Il modulo *rpy2* permette di usare il linguaggio R in un programma Python.

NumPy

Le strutture di Python sono eterogenee, questo e' molto comodo per tante applicazioni, ma poco efficiente per applicazioni numeriche intensive, dato che, per operare su un dizionario od una lista, Python deve, per ogni elemento, cercare di che tipo e' e la sua posizione in memoria.

Numpy e' un pacchetto ausiliario che rimedia a questo, introducendo strutture multidimensionali, con valori tutti dello stesso tipo. Il sito di riferimento di NumPy e': numpy.org

La struttura principale di NumPy e' l'oggetto ndarray (con alias : *array*), il nome significa n-dimensional array. Un ndarray e' una lista di liste, ma con elementi tutti dello stesso tipo ed alcuni attributi particolari. Internamente e' ordinato per righe, cioe' una linea dietro l'altra, analogamente a come sono ordinate le matrici nel linguaggio C.

Nella tabella seguente alcuni attributi degli ndarray.

Attributo	Significato
ndarray.ndim	Il numero di dimensioni dell'array 1: per un vettore; 2 per una matrice
ndarray.shape	Il numero di righe e colonne espresse con una tupla (righe,colonne)
ndarray.size	Il numero totale di elementi
ndarray.dtype	Il tipo degli elementi: str,int,float,boolean,bool,complex, datetime64, timedelta64 etc. etc.
ndarray.itemsize	le dimensioni, in byte, di un elemento

Creazione array, esempi

```
import numpy as np

c=np.ndarray([3,4])      # crea array 3x4, non inizializzato,
e=np.empty( (3,4) )      # ha dentro valori a caso

# Per creare array monodimensionali da liste,
# NumPy riconsosce che sono interi e gli assegna il
# parametro dtype giusto

b=np.array([11,12,13,21,22,23])

# nell'esempio seguente gli si dice cche sono dei float, a 64 bits

bb=np.array([11,12,13,21,22,23],np.float64)

# per veder gli atributi:

bb.ndim     # mi da 1: ho una dimensione
bb.dtype    # mi da il tipo: float64
bb.shape    # mi da la tupla (6,) e' una sola linea di 6 elementi
bb.size     # vale 6: ho 6 elementi
bb.itemsize # vale 8, ho 8 bytes di 8 bit, in totale 64 bits
```

Qui creo una matrice (array a 2 dimensioni) da una lista di liste Ho alcuni interi e alcuni float, NumPy converte tutto in float per avere dati omogenei:

```
x=np.array( [ (1.5,2,3), (4,5,6) ] )
x.dtype  # da: dtype('float64')

# Si possono fare array con valori preimpostati:

z=np.zeros((3,4))                # array 3x4 ,con 0
u=np.ones((2,3),dtype=np.int16 ) # array 2x3, con 1

ar=np.arange(10,30,5) # numeri da 10 a 30, passo 5
q=np.linspace(0,2,9)  # numeri da 0 a 2, 9 valori equispaziati

dd=np.array(['aa','ba','ca'])    # array di caratteri
```

Reshape degli array

Si possono cambiare le dimensioni di un array

```
#Qui da un array monodimensionale creo una
#matrice  2x3 (2 linee, 3 colonne)

b=np.array([11,12,13,21,22,23])  # creo ndarray monodimensionale
k=b.reshape(2,3)                 # creo matrice 2x3 (2 linee, 3 colonne)

k.shape # mi da la tupla:  (2, 3)
```

Trasformare array

Ci sono molte funzioni per trasformare gli array, vediamo alcuni esempi:

```
p=np.array([1,2,3,4])
w=np.array([10,20,30,40])

aa=np.concatenate((p, w))

#ottengo: array([1,2,3,4,10,20,30,40])
```

vstack ed **hstack** prendono in argomento una tupla di array, e li combinano in verticale od orizzontale, nell'esempio seguente oteniamo, rispettivamente: una matrice con gli array come righe, ed un array monodimensionale con i due array di seguito

```
np.vstack((p, w))
np.hstack((p, w))
```

Le funzioni **vsplit**, **hsplit** dividono un array in orizzontale o verticale nell'esempio che segue separiamo una matrice in ua lista di array (sempre a 2 dimensioni) con le righe o le colonne

```
s=np.array([[1,2],[10,20]])
np.vsplit(s,2)
np.hsplit(s,2)
```

Per copiare un array:

```
s2=s.copy()
```

s2=s NON fa una copia, ma da due nomi all'array

Altre funzioni utili sono: **sort**, per ordinare un array, **flip** che scambia righe o colonne, (la prima diventa l'ultima e viceversa).

```
s.T  # e' la trasposta

a.flatten() # un vettore con gli elementi in fila
```

La funzione unique fa un vettore senza doppioni

```
ff=np.array([1,1,2,2])
v=np.unique(ff)
```

Elementi e parti di array

Per riferirsi a singoli elementi o parti dell'array si puo' usare usa la stessa sintassi che si usa per le liste di liste di Python, ma una notazione preferita e piu' efficiente e' usare gli indici separati da una virgola.

Esempi:

```
a=np.array([[1,2,3],[10,20,30],[100,200,300]])

a[2][0] # e' la terza riga, prima colonna
a[2,0]  # una notazione piu' efficiente.

a[2]    # la terza riga, come array monodimensionale
```

Per gli slices abbiamo la sintassi con i due punti; al solito il primo valore e' compreso, il secondo escluso.

```
a[0:1,2:3] # prima linea (indice 0), terza colonna.
a[0:1,...] # tutta la prima lineaa[0:1]
           # anche: a[0:1] oppure: a[0:1,:]
```

Indice negativi partono dalla fine di linee o colonne:

```
a[-1,-1] # E' l'ultimo elemento
```

Si puo' definire un passo per la selezione, se il passo e' negativo si va all'indietro

```
s=np.array([1,2,3,4,5])
s[0:s.size:2]    # con passo 2, sono i dispari

a=np.array([1,2,3])
a[::-1]          # ribalto l'array (indietro con passo 1)
```

Si possono selezionare elementi con espressioni logiche. Ad esempio un array di valori booleani si puo' usare per selezionare elementi:

```
z=np.array([1,2,3,4,5])
j=np.array([True,False,True,False,True])

z[j]  # produce:   array([1, 3, 5])
```

Una relazione logica applicata ad un array produce un array booleano, delle stesse dimensioni, per cui possiamo usare espressioni logiche come indici:

```
a=np.array([1,2,10,20,40,50,100,200])

b=a[a>20]                    # valori oltre 20
c = a[(a > 20) & (a < 100)] # valori fra 20 e 100
```

La funzione **any** da True se qualche elemento dell'array e' vero, la funzione **all** se sono tutti veri. Le funzioni **less, equal, not_equal, greater greater_equal less_equal** confrontano uno ad uno gli elementi di due array, producendo array di valori logici:

```
a=np.array([1,100])
aa=np.array([2,10])

np.less(a,aa)   # produce: array([ True, False])
```

Per selezionare elementi si possono usare come indici array di indici:

```
j=np.array([1,1,3,3])
aaa=np.array([10,20,30,40,50,60])

aaa[j]   # e'  array([20, 20, 40, 40])
```

Algebra con Array

Si possono fare operazioni algebriche con gli array, senza dover iterare sugli elementi, che, in un linguaggio interpretato, e' un'operazione relativamente lenta. Vediamo alcuni esempi:

```
n=np.array([[0,0,1],[0,1,0],[1,0,0]])
a=np.array([[1,2,3],[10,20,30],[100,200,300]])
b=np.array([1.1,2.1,3.1])
c=np.array([10.1,20.1,30.1])

b+c   # somma gli elementi: array([11.2, 22.2, 33.2])
b*10  # moltiplica ogni elementi per 10
b*c   # moltiplica elementi fra loro:  array([11.11, 42.21, 93.31])

a[0]-1 # la prima riga, meno uno:  array([0, 1, 2])
a-b    # toglie il vettore b ad ogni linea di a
n*b    #  moltiplica ogni riga di a per b
```

Il prodotto di un vettore per una matrice si fa con la funzione **dot**; **vdot** e' il prodotto vettoriale:

```
np.dot(n,b)
np.dot(n,n)  # prodotto fra matrici: righe per colonne
```

Ci sono un sacco di funzioni per gli arrays, possono avere come parametro axis=0 oppure axis=1, per dare un valore per ogni riga o per ogni colonna di una matrice; altrimenti danno un valore singolo od un array monodimensionale.

```
a.sum()       # somma degli elementi
a.min()       # l'elemento minimo
a.max()       # l'elemento massimo
a.min(axis=0) # array di elementi minimi di ogni colonna:
a.min(axis=1) # array di elementi minimi di ogni riga:

a.cumsum()       # cumulativa
a.cumsum(axis=1) # cumulativa per righe
a.cumprod()      # prodotto cumulativo (produce array 1-dim)

a.mean()    #  media
a.var()     #  varianza
a.std()     #  deviazione standard
a.trace()   #  traccia (somma sulla diagonale)
```

Operare su array con funzioni

Si puo' applicare un funzione a un array, ottenendo un array di risultati, che operano sui i valori corrispondenti degli array in agomento.

Per far questo occorre *vettorializzare* la funzione.

Esempio:

```
def myfunc(a,b):
   return a*b

vfun=np.vectorize(myfunc) # ora ha in argomenti array e produce array

a=np.array([1,2,3])
b=np.array([10,20,30])

vfun(a,b)   # produce:  array([10, 40, 90])
```

Iteratori

Quando si puo' conviene evitare di iterare sugli elementi di un array, ma si puo' fare; per iterare sugli elementi di un vettore si usa il *for* di Python, per una matrice restituisce ad ogni giro una riga, come array monodimensionale:

```
a=np.array([[1,2],[10,20]])
for i in a:
  print(i)
```

Per iterare sui singoli elementi della matrice (flat e' un iteratore non una funzione):

```
for i in a.flat:
  print(i)
```

Matplotlib

Si tratta di tool grafici per Python, il sito di riferimento e': matplotlib.org ove si trova la documentazione e molti esempi.

Qui non tratteremo immagini e grafici a tre dimensioni, ma matplotlib ha funzioni per trattare immagini ed inserire immagini nei grafici. Il toolkit: mplot3d permette grafici in 3 dimensioni.

Matplotlib usa Numpy, nel senso che grafica vettori di NumPy od oggetti (come liste) che possono essere convertiti in vettori di Numpy.

Per usare matplotlib occerre importare il pacchetto omonimo ed importare anche la sua componente pyplot, che fornisce comandi sintetici per i grafici:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
```

Oggetti di base in matplotlib sono:

Figure	L'intero spazio su cui si fa il grafico
Axes	Identifica una regione, nella *Figure* ove fare il grafico. E' il principale oggetto per gestire la grafica. I comandi per la grafica sono attributi di questo oggetto, come anche il titolo del grafico e le *labels* degli assi.
Axis	Sono i singoli assi del grafico, con i tick mark, e le scritte relative
Artist	Un termine che indica tutte le componenti software che eseguono parti del disegno

All'inizio occorre creare una "*Figure*"; non e' necessario farlo in modo esplicito; routines di pyplot, come *subplots* , o anche altri comandi per fare grafici, lo fanno automaticamente, *subplots* crea anche oggetti Axes associati alla Figure ed, eventualmente divide l'area della Figure in diversi rettangoli in cui fare grafici, ogni rettangolo con i suoi oggetti Axes.

```
fig    = plt.figure()     # Una figura vuota
fig,ax = plt.subplots()   # Una figura con un oggetto Axes
fig,axs= plt.subplots(2,2)# Una figura divisa in 2 parti, con i loro Axes
```

Il creatore della Figure ha una serie di parametri di tipo keyword, opzionali, alcuni sono illustrati nella tabella seguente.

figsize=(width, height)	Dimensioni della figura, in pollici
dpi=100.0	Risoluzione, in punti per pollice
facecolor='white'	Colore dello sfondo
edgecolor='white'	Colore del bordo
linewidth=1.0	Spessore della linea al bordo (pixels)

La Figure ha molte funzioni per gestire il grafico, ne elenchiamo alcune, anche se in genere, per operare sul grafico, si utilizzano le funzioni simili della componente marplotlib.pyplot (nella tabella "fig" e' un'istansa di Figure).

fig.add_axes()	Aggiunge oggetto Axes (assi del plot)
fig.figimage ()	Aggiunge un'immagine
fig.get_axes()	Ottiene una lista degli axes

fig.legend()	mette una leggenda nella figura
fig.savefig(nomefile)	Mette la figura in un file
fig.text(x,y,testo)	Aggiunge un testo in posizione x,y

L'oggetto Axes definisce il sistema di coordinate per il plot , il disegno dei singoli assi (oggetto Axis), testo, linee della figura etc. I principali comandi per i grafici sono funzioni dell'oggetto Axes, nella tabella seguente alcuni di essi, con "ax" e' indicata un'istanza di Axes (non sono elencati tutti gli argomenti).

ax.plot(x,y,formato)	Un grafico degli array in argomento Il formato e' una stringa che dice disegnare i punti: colore e forma
ax.scatter(x,y)	Per scatter plots, ha anche argomenti per il formato ed i colori dei punti
ax.bar(x,h,width=0.8)	Per plot a torri
ax.barh(x,h,width=0.8)	Per plot a torri in orizzontale
ax.pie(x)	Grafici a torta
ax.hist(x)	Istogrammi
ax.contour(x,y,z)	Grafici a livelli

Abbiamo anche molte funzioni di Axes per definire l'aspetto del grafico, alcune sono elencate nella tabella seguente.

ax.text(x,y,stringa)	Mette un testo in posizione x,y
ax.arrow(x,y,dx,dy)	Mette una freccia
ax.set_xscale('log') ax.set_yscale('log')	Per assi logaritmici
ax.set_title('titolo')	Il titolo del frafico
ax.set_yticks([..]) ax.set_xticks([..])	Posizioni dei segni sugli assi
ax.set_xlabel('string') ax.set_ylabel('string')	Nomi degli assi

I formati per il comando plot sono stringhe di 3 o 4 caratteri, il primo indica il colore, il secondo come sono disegnati i punti, il terzo le linee. Il default e': "b-" una linea blu.

Per i colori abbiamo: "b" per blu, "g" per verde, "r" rosso, "c" ciano, "m" magenta, "y" giallo, "k": nero, "w": bianco.

Per i punti abbiamo".": punto, "o": cerchi, "< > ^ v" per triangoli nelle diverse direzioni, "8" per ottagoni. "s": quadrati, 'p': segno piu', "*" : stelline, "x X" : croci ; "1 2 3 4 ": per segni a v.

Per le linee: "-" : linea , "__" : trattini. "-." tratini e punti , ":" puntini.

matplotlib.pyplot

L'uso di matplotlib e' molto semplificato dal sub-package pyplot. Questo modulo contiene funzioni che creano in automatico la Figure ed hanno valori di default per gli Axes, per cui tutto e' molto piu' semplice.

Nella tabella che segue vediamo alune funzioni di pyplot. plt sta per matplotlib.pyplot, qui si assume che abbiamo importato pyplot con: "import matplotlib.pyplot as plt".

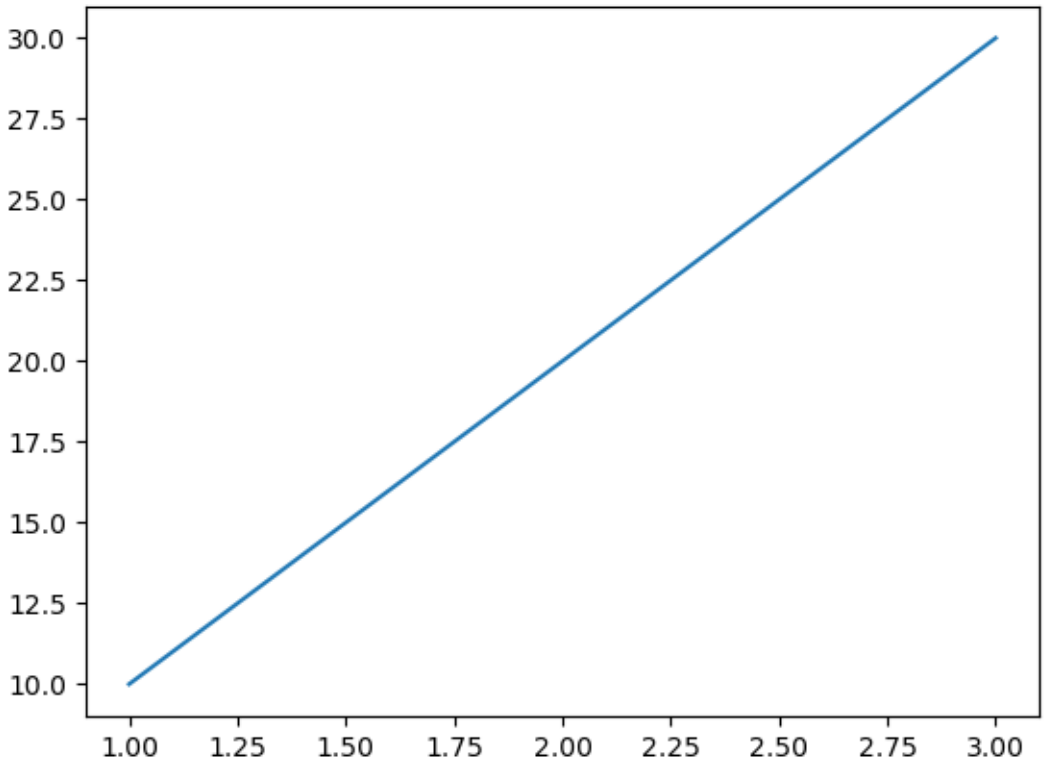
plt.plot(x,y,'formato') plt.scatter(x,y) plt.bar(x,h) plt.pie(x) plt.hist(x) plt.contour(x,y,z)	Fanno diversi tipi di grafici analoghe alle funzioni omonime dell'oggetto Axes
plt.xscale('log') plt.yscale('linear')	Scale logaritmiche o lineare (default: 'linear')
plt.xlabel('stringa) plt.ylabel('stringa)	Per nomi degli assi
plt.xticks([x val],('labels')) plt.yticks([y val],('labels'))	valori e segni sugli assi, in argomento una lista di posizioni ed una lista, o tupla di stringhe
plt.axis([xmin,xmax,ymin,ymax])	Estensione degli assi
plt.suptitle('stringa')	Titolo generale della Figure per figure con piu' grafici
plt.title('stringa')	Titolo per un singolo grafico
fig=plt.figure() ax1=plt.subplot(2,1,1) ax2=plt.subplot(2,1,2)	Si puo' ottenere una figura, dividerla in parti e definire oggetti Axes per ogni parte
plt.clf()	Svuota la Figure
plt.cla()	Resetta Axes
plt.text(x,y,'testo')	Mette un testo
plt.grid(True)	Mostra un reticolo sulla figura

Le funzioni di matplotlib hanno molti parametri, in genere i default sono sufficienti, ma e' anche possibile definire minimi dettagli del grafico.

Vediamo ora alcuni esempi di grafici semplici.

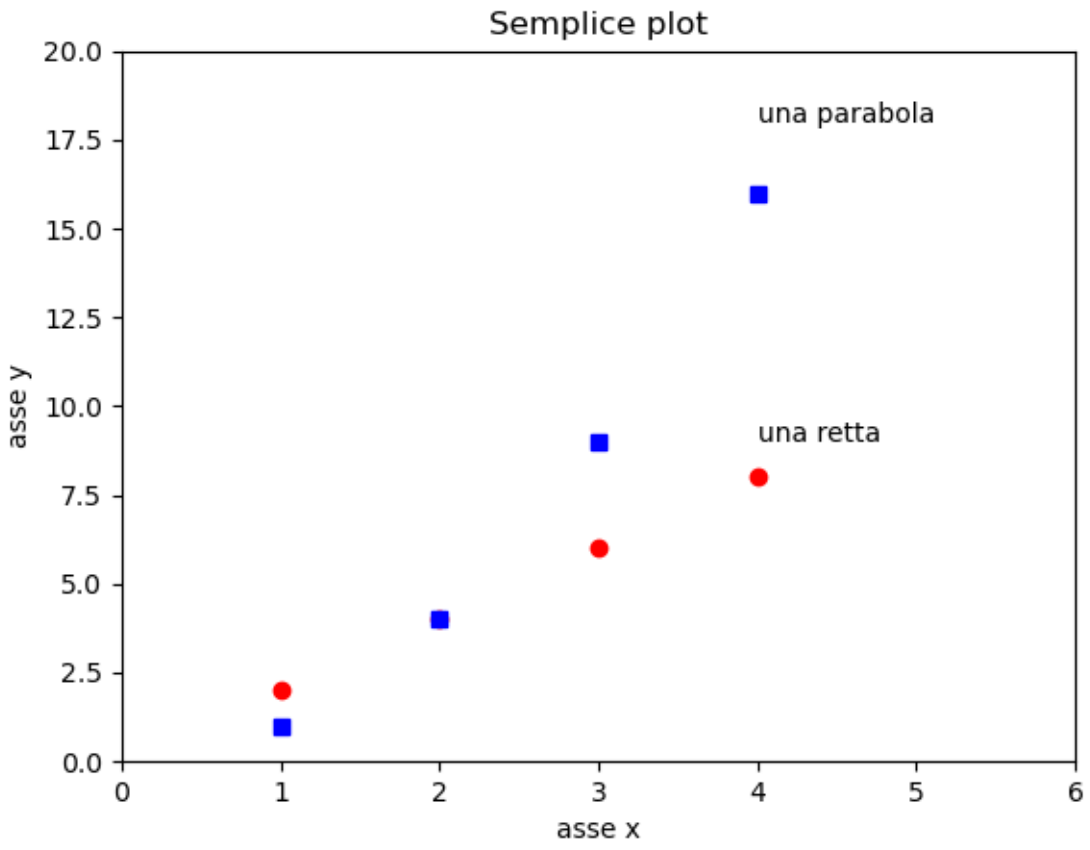
Nell'esempio seguente matplotlib usa dei valori di default per fare tutto lui, figura, assi ,thick mark etc. La funzione show() mostra il grafico in una sua finestra, ma poi lo resetta, per cui il salvataggio su file (funzione savefig) va fatto prima:

```
import matplotlib.pyplot as plt

plt.plot([1,2,3],[10,20,30])
plt.savefig('figA1.png')
plt.show()
```



In questo secondo esempio abbiamo uno "scatter plot" grafichiamo insieme due funzioni, specificando un colore e simbolo per ognuna, di diamo un titolo, nomi e dimensioni agli agli assi, mettiamo un testo:

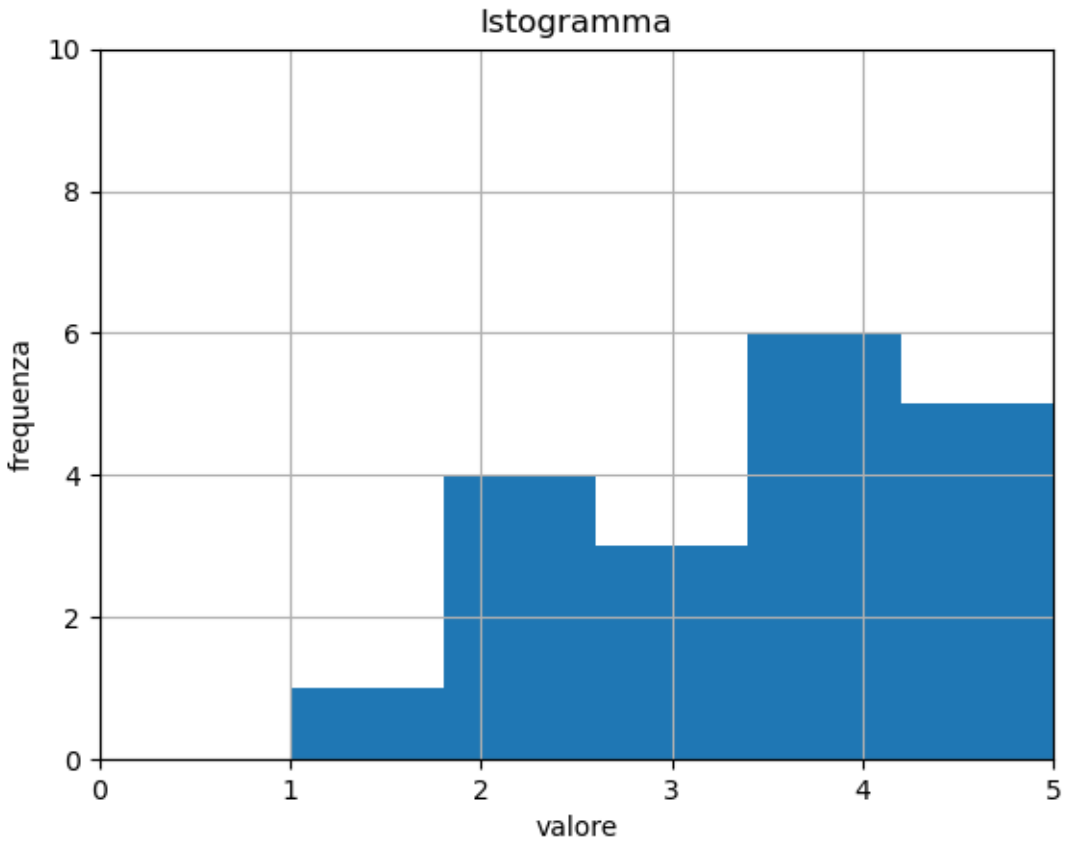
```
import matplotlib.pyplot as plt

x=[1,2,3,4]
y=[2,4,6,8]
z=[1,4,9,16]

plt.plot(x, y, 'ro',x,z,'bs')
plt.axis([0,6,0,20])
plt.title('Semplice plot')
plt.xlabel('asse x')
plt.ylabel('asse y')
plt.text(4,18,'una parabola')
plt.text(4,9,'una retta')
plt.savefig('figA2.png')
plt.show()
```



Nell'esempio seguente facciamo un istogramma Il secondo argomento della funzione *hist* dice in quanti gruppi raggruppare i valori di x la funzione restituisce 3 oggetti:

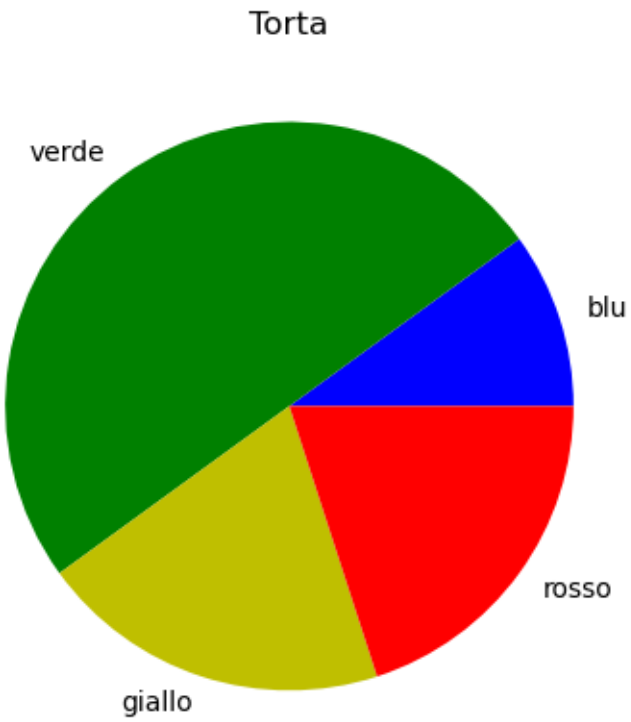
- n: una lista con l'istogramma
- **bins: i 6 valori che delimitano l'istogramma**
  (ha diviso l'intervallo 1:5 grande 4, in 5 parti)
- patches: lista dei rettangoli dell'istogramma

```
import matplotlib.pyplot as plt

x = [1,2,2,2,2,3,3,3,4,4,4,4,4,4,5,5,5,5,5]

n,bins,patches = plt.hist(x,5)
plt.xlabel('valore')
plt.ylabel('frequenza')
plt.title('Istogramma')
plt.axis([0,5,0,10])
plt.grid(True)          # mette un reticolo
plt.show()
```



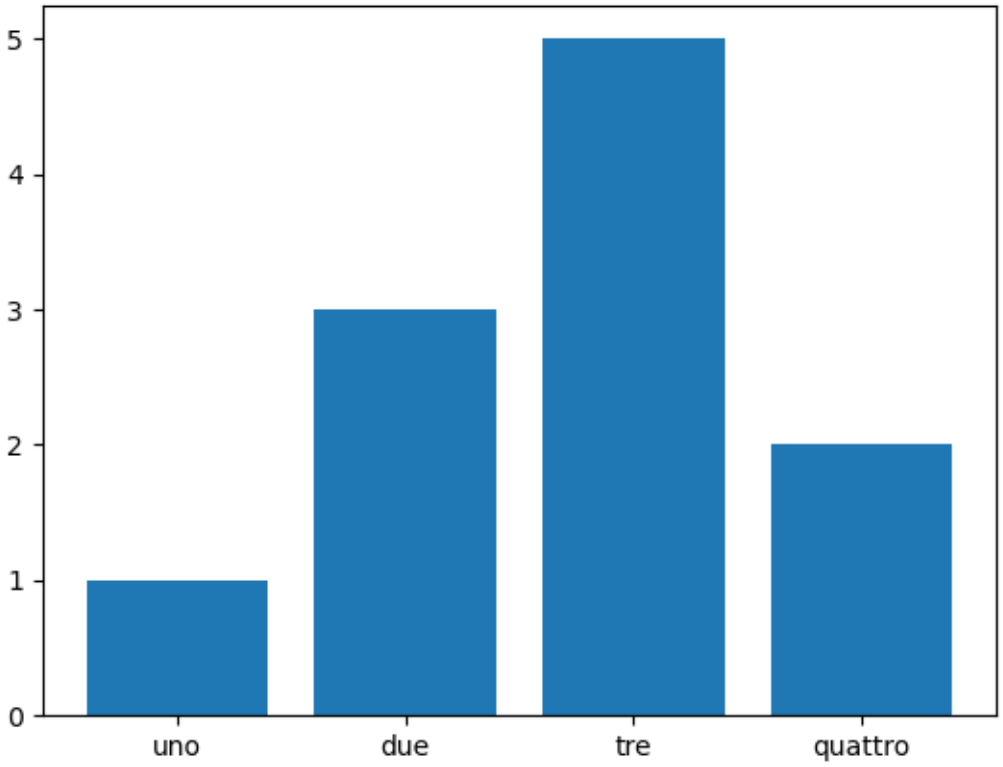
Vediamo ora un grafico a torta. patches: sono i disegni delle fette, testo: una lista con le labels:

```
import matplotlib.pyplot as plt

x = [10,50,20,20]

patches,testi=plt.pie(x,colors=('b','g','y','r'),
   labels=('blu','verde','giallo','rosso'))
plt.title('Torta')
plt.show()

In questo esempio abbiamo un grafico a torta.
patches: sono i disegni delle fette,
testo: una lista con le labels
```



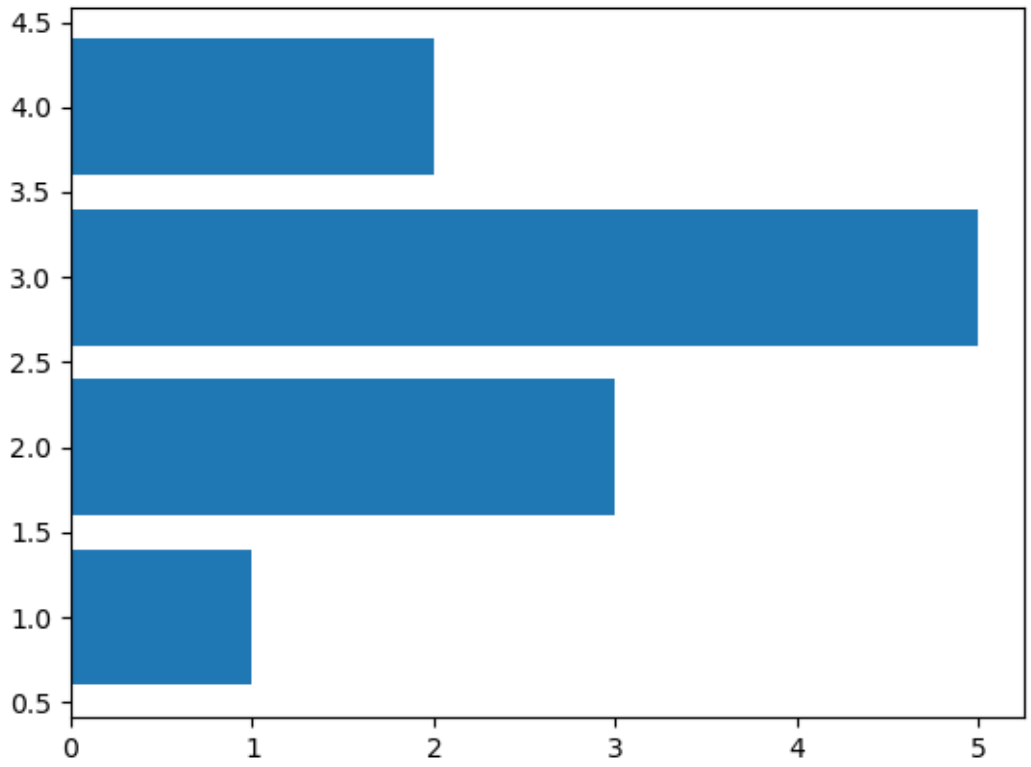
Qui abbiamo un "bar plot", diamo esplicitamente valori ai "tick marks" dell'asse x:

```
import matplotlib.pyplot as plt
x=[1,2,3,4]
h=[1,3,5,2]
plt.xticks(x,('uno','due','tre','quattro'))
plt.bar(x,h)
plt.show()
```



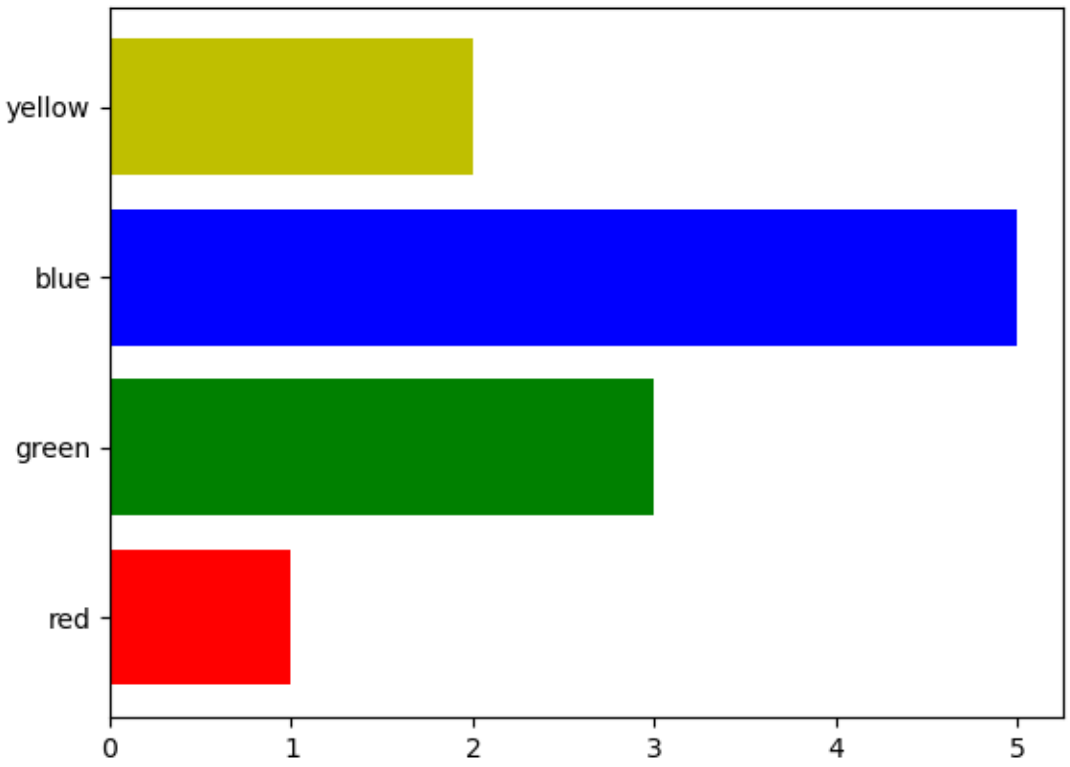
Ancora un "bar plot, ma con barre orizzontali:

```
import matplotlib.pyplot as plt
x=[1,2,3,4]
h=[1,3,5,2]
plt.xticks(x,('uno','due','tre','quattro'))
plt.barh(x,h)
plt.show()
```



Bar plot orizzontale, con barre di diverso colore:: e label sull'asse y:

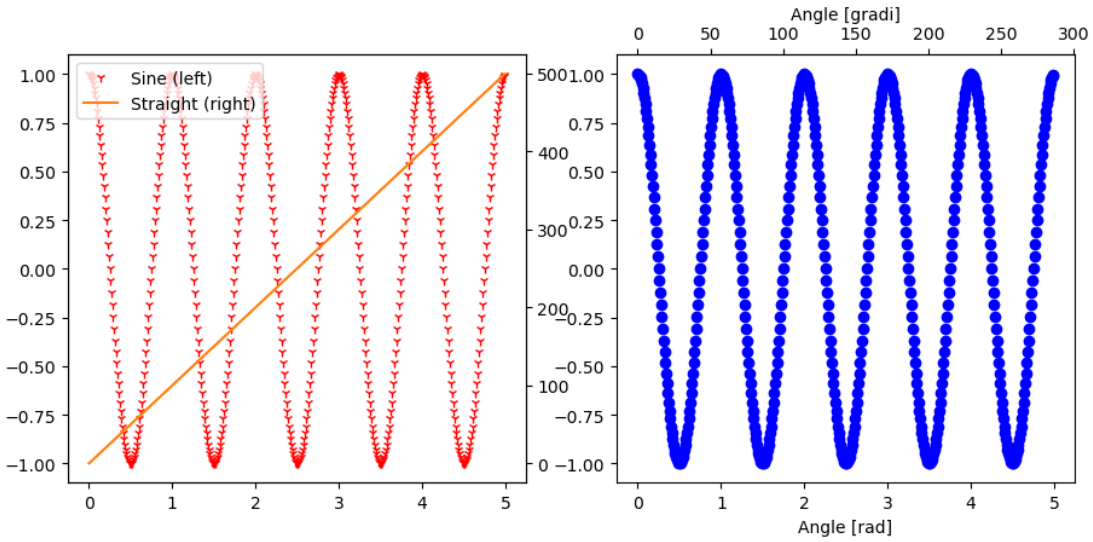
```
import matplotlib.pyplot as plt
x=[1,2,3,4]
h=[1,3,5,2]
plt.yticks([1,2,3,4],('red','green','blue','yellow'))
plt.barh(x,h,color=['r','g','b','y'])
plt.show()
```



Vediamo ora un esempio piu' complicato (preso dal tutorial di matplotlib, ma modificto)

```
import matplotlib.pyplot as plt
import numpy as np

# qui usiamo numpy e definiamo x ed y di una funzione

x = np.arange(0.0, 5.0, 0.01)
y = np.cos(2 * np.pi * x)

# Qui dividiamo la figura in 2 parti: individuate
# da un reticolo di 1 linea e 2 colonne
# ( i primi 2 argomenti di subplot),
# facciamo una figura allungata,con il parametro figsize
```

```
fig,ax= plt.subplots(1, 2,figsize=(7, 2.7))

# Siccome abbiamo 2 grafici, la funzione subplot ci
# restituisce un array di due oggetti Axes,

ax1=ax[0]  # axes del primo grafico
ax2=ax[1]  # axes del secondo grafico


# faccio un primo grafico, plot mi torna  una lista
# di oggetti che rappresentano le linee da disegnare

l1= ax1.plot(x,y,'r1')

ax1b = ax1.twinx()  # mi creo un Axes ausiliario a destra

# un secondo grafico, con una altra scala, associato all'asse a destra

l2 = ax1b.plot(x,range(len(x)), 'C1')

# metto una legenda, associata alle linee.
ax1b.legend([l1[0], l2[0]], ['Sine (left)', 'Straight (right)'])

# ora il secondo grafico, nel secondo riquadro della figura

ax2.plot(x,y,'bo')
ax2.set_xlabel('Angle [rad]')  # label per asse x

# Ora faccio un asse secondario in alto, i cui valori sono
# ottenuti dai valori del primo asse con una funzione che
# definisco, assieme all'inversa.

ax2b = ax2.secondary_xaxis('top', functions=(np.rad2deg, np.deg2rad))
ax2b.set_xlabel('Angle [gradi]')
```



Per ulteriori esempi e dettagli sui grafici ai rimanda al sito: matplotlib.org

Pandas

Il pacchetto Pandas estende Numpy, creando strutture a una o due dimensioni, in cui righe e colonne hanno un nome che le identifica.

Le strutture ad una dimensione sono dette **Series** quelle a due dimensioni **DataFrames**.

I nomi e i tipi delle righe e delle colonne sono in oggetti di tipo **Indexes**. Le colonne sono oggetti: **columns**, le righe: **index**

I DataFrasmes sono particolarmente utili, in pratica e' come maneggiare direttamente una tabella di un database, o un foglio excel.

Come per gli array di si possono fare operazioni algebriche con Seried e DataFrames, qui pero' si combinano fra loro elementi che hanno lo stesso nome per righe e colonne, anche se il loro ordine e' diverso.

Pandas ha anche strumenti per la gestione di serie temporali, tratta valori mancanti nelle tabelle, integra matplotlib per la grafica e scipy per algoritmi lmatematici.

Il sito di riferimento e': https://pandas.pydata.org ove si trova anche tutta la documentazione

Il testo di riferimento e' il libro di Wes McKinney (il creatore di Pandas):

> "Python for Data Analysis" by Wes McKinney 3nd Edition , O'Reilly
>
> ISBN-13: 978-1098104030 , ISBN-10: 109810403X
>
> Il libro e' on-line: https://wesmckinney.com/book/

Strutture per i dati: Series

Una *Series* e' un array monodimensionale di numpy, quindi con elementi tutti dello stesso tipo, che possono essere interi, float, stringhe o qualunque altro oggetto Python.

A questo array e' associato in *index*, che ha un valore per ogni dato ed e' un identificativo del dato, di default sono una sequenza di interi, cha pertono da zero.

Sono ammessi dati mancanti, ovvero indici senza un dato corrispondente, in questo caso il dato e' indicato con: NaN (not a number).

Una Series puo' essere costruita a partire da una lista, un dizionario, una array di numpy, o anche un singolo valore:

```
import numpy as np
import pandas as pd

k= pd.Series(dtype='int')              # una serie vuota, di interi
k= pd.Series(dtype='int',name='vuota') # una Series puo' avere un nome

# una serie di interi, con lettere come indici

s= pd.Series([0,1,2,3,4],index=['a', 'b', 'c', 'd', 'e'])

# numeri a caso, fra 0 ed 1, con interi come indice

s1 = pd.Series(np.random.rand(5), index=[1,2,3,4,5])

# utilizzando un dizionatio le *key* diventano gli indici

s2 = pd.Series( {'a' : 0., 'b' : 1., 'c' : 2.} )
```

```
# utilizzzando un singolo valore

s3=pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
```

Elementi e parti di una Series (*slices*), possono essere ottenuti utlizzando indici interi, fra quadre, oppure il valore dell'indice:

```
s= pd.Series([0,10,20,30,40],index=['a', 'b', 'c', 'd', 'e'])

s[1]  # vale 10

s[:3] # una Series con i primi 3 elementi (i numeri da 0 a 2)

s['b'] # vale 10

s['b':'e'] # una serie con fli elementi con indici da 'b' ad 'e' (compresi)
```

L'operatore *in* opera sull'index:

```
'c' in s  # e' True
10 in s   # e' False
```

for itera normalmente sull'array, non sull'indice, per iterare sull'indice bisogna fare:

```
for  i in s.index:
   print(i,s[i])    #stampa  indice e valore dell'elemento
```

Le operazioni per gli array di numpy si applicano alle Series:

```
s[s > s.median()]  # Series dei valori maggiori della mediana

s[s == s.mean()]   # Series con il valor medio

s[[1,2]]           # Uso di liste come indici (si ottengono Series)
s[['b','c']]

np.sum(s)         # ma somma dei valori

s+s               # Series con elementi di valore doppio
s*2
```

Usando indici con elementi si ha un errore se l'elemento non e' presente, la funzione get invece produce l'oggetto '*None*':

```
s[f]       # da errore keyerror
s.get('f') # da "None"
```

Serie possono esere convertire in dizionari:

```
s.to_dict()
```

E' possibile cambiare l'indice riassegnando l'oggetto Index:

```
s.index=['a','b','c','d','e']
```

Strutture per i dati:Dataframe

Queste strutture a due dimensioni, hanno identificativi per le linee, in *index* come se Series, ma anche in identificativo per le colonne: *columns*.

Un dataframe si puo' costruire a partire da un dizionario di serie, un dizionario di liste, da liste di dizionati, da dizionari di dizionari etc. etc.

Come primo esempio creiamo una dataframe da un dizionario di liste:

```
import pandas as pd
d = {'col1': [1, 2], 'col2': [3, 4]}
df = pd.DataFrame(data=d)

    col1  col2
  0    1     3
  1    2     4
```

Abbiamo creato un dataframe con due colonne, una colonna per lista, gli indici del dizionario diventano i nomi delle colonne:

Creiamo da un dizionario di Series:

```
d = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
     'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}

df = DataFrame(d)

        one  two
     a  1.0  1.0
     b  2.0  2.0
     c  3.0  3.0
     d  NaN  4.0
```

Abbiamo creato un dataframe con due colonne, una colonna per ogni chiave del dizionerio, le chiavi del dizionario diventano i nomi delle colonne. Qui abbiamo dato gli indici, che di default sono interi crescenti che partono da zero.

Creiamo un dataframe da una lista di dizionari:

```
data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

df2=pd.DataFrame(data2)

     a   b    c

  0  1   2   NaN
  1  5  10  20.0
```

Ogni dizionario diventa una riga, le chiavi sono nomi di colonne, i dati mancanti sono indicati con: *NaN*;

La funzione **DataFrame.from_records** costruisce il dataframe a partire da tuple; **DataFrame.from_dict** costruisce il dataframe da un dizionario, procedendo per righe o per colonne. **df.to_dict** converte un dataframe in un dizionario:

```
dd={ 'd1':{'da':1,'db':2},'d2':{'da':3,'dc':4}}
df3=pd.DataFrame.from_dict(dd,orient='index')

          da   db   dc
      d1   1  2.0  NaN
      d2   3  NaN  4.0

df4=pd.DataFrame.from_dict(dd,orient='columns')

          d1   d2
      da  1.0  3.0
      db  2.0  NaN
      dc  NaN  4.0
```

Gli attributi index e columns sono oggetti tipo: *Index* con la lista dei nomi dell righe e colonne; il nome delle colonne deve essere unico, invece righe diverse possono avere nomi uguali:

```
Ad esempio prendiamo il dataframe df con:

     one  two
  a  1.0  1.0
  b  2.0  2.0
  c  3.0  3.0
  d  NaN  4.0

df.index
Index(['a', 'b', 'c', 'd'], dtype='object')

df.columns
Index(['one', 'two'], dtype='object')
```

Un dataframe ha anche l'attributo dtypes che e' una Series con i tipi di dato di ogni colonna e, come indici, i nomi delle colonne:

```
one    float64
two    float64
```

Altri attributi sono *shape* : tupla con numero righe e colonne, *axes* : lista di oggetti index con nomi righe e colonne, values: una array di numpy con i valori,

```
df.shape
 (4, 2)

df.axes

 [Index(['a', 'b', 'c', 'd'], dtype='object'),
 Index(['one', 'two'], dtype='object')]
```

La funzione **df.describe()** fa una statistica dei valori delle colonne, **df.info()** da una descrizione del dataframe.

Creando un dataframe da un dizionario e dando index e columns in modo esplicito, dal dizionario si prendono solo i valori indicati, e si mette NaN quando i valori indicati mancano:

```
w=pd.DataFrame(d, index=['d', 'b', 'k'], columns=['two', 'three'])

   two three
d  4.0   NaN
b  2.0   NaN
k  NaN   NaN
```

Per copiare una dataframe:

```
dff=df.copy()
```

Come per le serie, l'assegnazione: *dff=df* crea un secondo nome per il dataframe., ma non lo copia.

Come per le Series in un dataframe possono mancare alcuni valori e questi sono identificati con: *NaN* . La funzione **dropna** elimina righe o colonne con valori mancanti, **fillna** mette un valore al posto dei mancanti:

```
df.dropna(axis=0,how='any') # elimina righe con qualche dato mancante
df.dropna(axis=1,how='all') # elimina colonne con tutti dati mancanti

df.fillna(100)      # fa copia di df con 100 al posto dei mancanti
```

La funzione **fillna** ha diversi parametri per gestire i dati mancanti: metterci valori diversi per ognuno, propagarli lungo una riga o colonna, rimpiazzarne solo alcuni.

df.T e' la trasposte del dataframe, con righe e colonne scambiate.

La funzione **rename** cambia nomi ad indici e colonne, anche la funzione reindex riordina indici e colonne:

```
dff=pd.DataFrame({'c1':[1,2],'c2':[3,4]})

      c1  c2
   0   1   3
   1   2   4


dff.rename(index={0:'zero',1:'uno'},columns={'c1':'col1'})

         col1  c2
   zero     1   3
   uno      2   4


dff.reindex(index=[1,0],columns=['c2','c1','c3'])

      c2   c1  c3
   1  4.0  2.0 NaN
   0  3.0  1.0 NaN

 Mette NaN ove mancano valori.
```

Dataframe, selezione righe e colonne

I nomi delle colonne permettono di selezionarne una, vista come una Series, o piu' colonne, viste come Datframe:

```
df['nomecol']           #: una singola Series, di nome 'nomecol'
df[['nomecol','nomecol']]# un DataFrame con solo le cols selezionat
df[['nomecol']]         # un dataframe di una sola colonna

df.nomecol              # la colonna e' vista come attributo
```

la sintassi con **:** serve a selezionare righe:

```
df=pd.DataFrame({'c1':[1,2],"c2":[3.4]},index=['a','b'])

      c1  c2
   a   1   3
   b   2   4

df[0:1]

        c1  c2
     a   1   3

df['a':'b']

          c1  c2
       a   1   3
       b   2   4
```

Per selezionare sia righe che colonne:

```
df[0:1]['c1']

   a    1

df[0:1][['c1','c2']]

      c1  c2
   a   1   3
```

Anche gli attributi **loc** ed **iloc** selezionano righe:

```
df.iloc[0]     # la prima riga, vista come una serie,
df.loc['a']    # la righa con index: *a*, e' una serie
               # i nomi delle colonne sono gli indici
   c1    1
   c2    3
```

df.loc e df.iloc possono anche essere usati per selezionare sia righe che colonne:

```
df.loc['a':'b','c2']  # il primo indice e' la riga
```

```
   a    3
   b    4
   Name: c2, dtype: int64

df.loc['a':'b',['c1','c2']]
df.iloc[0:2,[0,1]]

    c1  c2
 a   1   3
 b   2   4

df.loc['a','c1']  # il primo indice e' la riga
df.iloc[0,0]      # il secondo la colonna
```

Per selezionare un singolo elemento si possono usare "*at*" od "*iat*":

```
df.at['a','c2']  # vale: 3.4; elemento della riga: 'a' , colonna: 'c2'
df.iat[0,1]      # vale: 3.4; elemento della prima riga, seconda colonna
```

Operazioni logiche fra colonne sono Series booleane:

```
df['c1']>3

   a    False
   b    False
```

Espressioni logiche sulle colonne possono essere usate come indici per selezionare righe di un DataFrame:

```
df[df['c1']>1]  # righe con la colonna c1 >1

        c1  c2
     b   2   4

df[[True,False]]  # prima riga True, la prendo, l'altra no

      c1  c2
   a   1   3
```

Dataframe, modifica righe e colonne

Vediamo alcuni esempi; definiamo un dataframe:

```
import pandas as pd
d={c1:[11,21],c2:[12,22]}
df=pd.DataFrame(d,index=['r1','r2'])

        c1  c2
    r1  11  12
    r2  21  22
```

Per eliminare colonne:

```
df.drop(columns='c1')     # questo elimina colonne (fa una copia di df)
df.drop('c1',axis=1)
del df['c1']     # questo non fa una copia di *df*, ma lo modifica
```

Per aggiungere colonne, o cambiarene il contenuto:

```
df['c3'] = [13,23]                  # aggiungo una colonna (alla fine)
df['c4'] = pd.Series(['a','b','c']) # aggiungo una Series come colonna
df['c5'] = 3                        # nuova colonna, ogni elemento con: '3'

df.insert(1,'c1',[12,22])  # aggiungo una colonna in posizione: *'0'*

        c1  c2  c3   c4  c5
    r1  12  12  13  NaN   3
    r2  22  22  23  NaN   3

df['c1']=df['c1']*100    # moltiplica i valori delle colonna per 100
```

Per eliminare righe:

```
df.drop(['r1','r2'])            # elimina righe in una copia del dataframe
df.drop('r1',axis=0)
df.drop(index=['r1','r2'])
df.drop(index='r1',inplace=True] # qui non crea una copia, modifica 'df'
```

Per cambiare valori di righe:

```
df.loc['r1']=1333        # mette un unico valore in tutta la riga
df.loc['r1']=[1,2,3,4,5] # ridefinisc i valori di una riga

df.loc['r1','c2']=1234   # cambia un singolo valore
```

Si possono fare modifiche ad un dataframe usando condizioni logiche come indici:

```
df[df<20]=0        # tutti glie lementi minori di 20 diventano 0

df[df==0]=np.nan   # al posto di '0' mette: NaN: valore mancante.
```

Algebra con DataFrame

Come per le Series, quando si fanno operazioni sui dataframe vengono fatti corrispondere valori con righe e colonne di stessi nomi; mettendo: 'NaN' in caso di valori non corrispondenti:

```
df+df1       # somma elemento per elemento
df.add(df2)
```

La somma di una serie ad un Daframe opera per righe, nel senso che gli indici della serie corrispondono alle colonne del dataframe e la serie e' sommata ad ogni riga; analoga cosa vale per le altre operazioni fra serie e dataframe:

```
df
        c1  c2
    r1  11  12
    r2  21  22

df.iloc[0]   # una serie, con indici i nomi delle colonne
    c1    11
    c2    12
    Name: r1, dtype: int64

df+df.iloc[0]  # i valori della serie sono sommati ad ogni riga

        c1  c2
    r1  22  24
    r2  32  34
```

Per sommare una serie ad ogni colonna si usa la funzione add, con il parametro axis=0

```
s=pd.Series([100,200],index=['r1','r2'])

df.add(s,axis=0)

        c1   c2
   r1  111  112
   r2  221  222
```

Le operazioni con singoli valori operano su tutti gli elementi del Data Frame:

```
df * 5 + 2   # ogni elemento e' moltiplicato per 5 e aumentato di 2
1 / df       # applicato ad ogni elemento
df ** 2      # eleva al quadrato ogni elemento

df.T : trasposta ; scambia righe e colonne
```

Se il dataframe contiene valori logici si possono utilizzare operatori logici, ottenendo sempre dataframe booleani:

```
df1 & df2
df1 | df2
df1 ^ df2

-df1      : muta veri in falsi e viceversa
```

Per applicare una funzione generica alle colonne di una dataframe, ed ottenere una serie con un valore per ogni colonna, si usa la funzione apply:

```
df.apply(lambda x: max(x))

La Series ottenuta ha come indici i nomi delle
colonne e contiene il valore massimo di ogni colonna.
```

Dataframe ed iteratori

L'istruzione for itera sui nome delle colonne:

```
for col in df:
  print(col)

c1
c2
```

Per ottenere, a ogni iterazione, una colonna:

```
for colname,col in df.iteritems():

    # qui colname e' il nome della colonna
    # col e' una serie, con la colonna
```

Per ottenere, ad ogni iterazione, una serie con una riga:

```
for indice,riga in df.iterrows():

  # qui indice e' il nome della riga
  # riga: la Series contenente la riga
```

Funzioni per i dataframe

Una funzione utile e' reindex, che permette di gestire nomi di righe e colonne:

```
df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])

df.reindex_like(df2) # mette indici di altro oggetto
```

Per applicare una generica funzione ad ogni elemento di un dataframe si usa "applymap", ad esempio:

```
f = lambda x: len(str(x))
df.applymap(f)
```

Per ordinare le righe di una dataframe in base ai valori degli indici si puo' usare la funzione '*sort_index*':

```
dff=pd.DataFrame({3:[20,22],2:[10,40]},index=[300,200])

         3   2
   300  20  10
   200  22  40
```

```
dff.sort_index()  # ordino le righe

        3   2
  200  22  40
  300  20  10

dff.sort_index(axis=1) # ordino le colonne

        2   3
  300  10  20
  200  40  22
```

Si puo' ordinare un dataframe in base ai valori di una colonna, qui ordino le righe in base ai valori delle colonna: '2', in ordine inverso:

```
dff.sort_values(by=[2],ascending=False)

         3   2
   200  22  40
   300  20  10
```

Le funzioni concat, join, merge, combinano insieme dataframe:

- concat: unisce dataframes mettendoli uno sotto l'altro od accanto all'altro
- join unisce le colonne di dataframe selezionando le righe in base agli indici
- merge unisce le colonne di dataframe selezionando le righe in base a valori di una colonna comune

Per i dettagli si rimanda alla documentazione di pandas.

Indice