Tutto Python per principianti

Vittorio Albertoni

Introduzione

Nel lontano febbraio del 1991, quando ancora Internet era riservata a scienziati e ricercatori delle Università, comparve un messaggio, in inglese, i cui principali passaggi tradotti in italiano sono i seguenti:

«Ho pubblicato una versione beta del mio linguaggio Python su alt.sources.
...
Python è un linguaggio di programmazione interpretato estensibile che combina una notevole potenza con una sintassi molto chiara.
Questa è la versione 0.9 (la prima versione beta), livello di patch 1.
Python può essere usato al posto degli script shell, Awk o Perl,...
...
Per favore provalo e mandami i tuoi commenti...
...
Sono l'autore di Python:
Guido van Rossum
CWI, dipartimento CST
Kruislaan 413
1098 SJ Amsterdam
Paesi Bassi
E-mail: gu...@cwi.ne
...
La fonte Python è protetta da copyright, ma è possibile utilizzarla e copiarla liberamente purché non si modifichi o si rimuova il copyright.»

Nasceva così, in pieno stile software libero, un linguaggio di programmazione che oggi domina nel campo della ricerca scientifica e non solo.

E' con un messaggio dello stesso tipo che, qualche mese più tardi, nell'agosto del 1991, Linus Torvalds lanciò quello che divenne il sistema operativo GNU/Linux.

Python e Linux, due bei coetanei: il pitone e il pinguino.

Mentre il pinguino pare sia stato scelto in quanto animale che godeva della simpatia di Linus Torvalds, il pitone proviene in realtà dal nome con cui Guido van Rossum battezzò il suo linguaggio in onore del

gruppo di comici inglesi Monty Python, nei cui confronti egli nutriva sconfinata ammirazione.

In entrambi i casi abbiamo un inventore che affida la propria invenzione ad una comunità affinché la commenti e la arricchisca.

E' grazie a questa formula che Python è diventato un'eccellenza tra i linguaggi di programmazione e Linux è diventato un'eccellenza tra i sistemi operativi.

E non a caso, pertanto, il miglior ambiente in cui utilizzare Python è il sistema operativo Linux. Anche se Python funziona benissimo in Windows e in Mac OS, con Linux si ha l'impressione di essere a casa propria.

* * *

Python è, come si suol dire, un linguaggio all purpose, cioè ci si può fare di tutto. La sua fortuna si deve tuttavia alla potenza ed alla facilità con cui ci consente di fare calcolo.

In questa guida per principianti mi tengo ancorato a queste caratteristiche.

Il principiante che può affrontare con profitto i primi quattro capitoli basta che sappia usare un computer ed abbia un'idea di massima di cosa si intenda quando si parla di programmare un computer.

Per gustare anche gli altri capitoli occorrerebbe altresì un preparazione di base in matematica generale e statistica.

Indice

Capitolo 1

Il mondo di Python

1.1 Quale Python

La corrente versione del linguaggio Python è la 3. La versione 2 è la versione del passato ed è stata ormai soppiantata dalla versione 3.

Purtroppo tra le due esistono differenze tali che uno script creato in una versione non viene eseguito nell'altra, per cui, per utilizzare programmi scritti per la versione 2, occorre disporre di Python 2, salvo apportare le necessarie modifiche per poterli utilizzare con l'interprete Python 3.

Per chi ha la fortuna di utilizzare il sistema operativo Linux, o il suo cugino di discendenza Unix Mac OS X, vi sono modi per gestire senza confusioni entrambe le versioni, in quanto gli eseguibili degli interpreti hanno nomi diversi (python2 o semplicemente python per python2 e python3) e c'è modo di indicare l'eseguibile nello script[1]. Per chi usa il sistema operativo Windows, purtroppo, l'interprete si chiama python.exe in entrambi i casi e la differenza la fa la directory in cui si trova l'interprete stesso (Python2 o Python3), con tutte le complicazioni che derivano dalla gestione dei path nelle variabili d'ambiente.

Bello, soprattutto per chi usa Windows, sarebbe avere la sola versione 3, quella del futuro, su cui far girare programmi scritti per la versione 2 opportunamente adattati. Chiunque può provvedere all'adattamento tenendo presenti le differenze che riepilogo nell'ultimo paragrafo

[1] Ciò può avvenire creando nella prima riga dello script la così detta shebang line, cioè scrivendo all'inizio dello script i caratteri #! seguiti dall'indirizzo dell'interprete. Per esempio, volendo utilizzare l'interprete Python3, che si trova nella directory /usr, sottodirectory /bin, si scrive nella prima riga `#!/usr/bin/python3`. Nel sistema operativo Windows questa riga non viene letta.

del prossimo Capitolo 2, dopo che il lettore avrà acquisito le basi del linguaggio.

* * *

In questo libro ci occupiamo di Python 3.

1.2 Il pacchetto base e la sua installazione

Sistema Linux

Il pacchetto base di Python è da sempre preinstallato sui sistemi operativi Linux.

La versione preinstallata è l'ultima disponibile al momento dell'uscita della distro: ormai è la versione 3.

Se la o le versioni installate non ci aggradano più con il passare del tempo e non vogliamo aggiornare il sistema operativo, possiamo procurarci il source del pacchetto che vogliamo installare dal sito *www.python.it* e installarlo, dopo averlo decompresso, con il solito procedimento

```
./configure
make
make test
sudo make install.
```

La dotazione di base Python che troviamo in Linux ha tutto ciò che serve per creare programmi anche complessi (nel caso di Python sarebbe più proprio chiamarli script, in quanto si tratta di istruzioni che vengono interpretate ogni volta per essere eseguite). Per quanto riguarda il calcolo, se non bastano i normali e più ricorrenti operatori per addizione, sottrazione, moltiplicazione, divisione ed elevamento a potenza, la dotazione di base comprende un modulo, chiamato **math**, che ci offre tutta una serie di funzioni precostituite.

In Python si chiama modulo una raccolta di funzioni riunite in una libreria.

Sistema Mac OS X

Mac, che un tempo preinstallava la versione 2, non preinstalla più Python.

Per installare Python occorre procurarsi l'installer nella sezione *Download* del sito *www.python.it*: le ultime versioni richiedono almeno la versione 10.6 di Mac OS X[2].

Sistema Windows

Windows non preinstalla Python.

Possiamo procurarci l'installer nella sezione *Download* del sito *www.python.it*.
Le ultime versioni di Python richiedono almeno Vista. Se vogliamo installare su XP dobbiamo procurarci la versione 3.4[3].

La dotazione di base che installiamo su Windows, come quella per Linux, per quanto riguarda i calcoli contiene solo il modulo **math**, ma contiene, in più, la IDLE e il modulo **tkinter**.

* * * * *

Per lavorare bene con Python è consigliabile installare la IDLE (Integrated Development and Learning Environment). Su Windows, come ho appena detto, l'installazione avviene automaticamente quando lanciamo l'installer di Python. Su Linux e Mac OS X dobbiamo farlo noi ricorrendo al repository della nostra distro, nel caso di Linux, o, nel caso di Mac OS X all'indirizzo *https://www.python.org/download/mac/tcltk/*.

L'utilità di installare l'IDLE sta innanzi tutto nel fatto che, facendolo, automaticamente arricchiamo il pacchetto base del modulo **tkinter**, che ci consente di sviluppare in maniera abbastanza semplice applicazioni con interfaccia grafica. A parte questa utilità secondaria (il modulo lo potremmo installare altrimenti, come vedremo) l'IDLE ci offre una shell di Python con simpatiche e utilissime prestazioni di autoinserimento di funzioni e parametri molto utile per l'apprendimento del linguaggio. La IDLE contiene pure un editor per scrivere i programmi, ma questo non ci offre nulla in più di quanto ci offra un qualsiasi editor di testo.

La comunità Python ha sviluppato e continua a sviluppare programmi e librerie per Python.

---

[2]Per trovare versioni precedenti alle ultime uscite che ci offre *www.python.it* occorre andare nella sezione *Download*s su *https://www.python.org*

[3]vedi nota precedente

1.3 Il repository di Python

Nel momento in cui scrivo, una catalogazione di quasi 100.000 oggetti, tra programmi e librerie, la troviamo all'indirizzo

https://pypi.python.org/pypi

PyPI sta per Python Package Index.

Per esplorare il contenuto di questo enorme deposito possiamo cliccare su BROWSE PACKAGES in alto a sinistra della pagina web e scegliere le parole chiave nella successiva o successive pagine.

Per verificare se un determinato pacchetto si trova nel deposito possiamo scriverne il nome o parte del nome nella finestrella di search in alto a destra della pagina web.

Una volta individuato il package che ci interessa, cliccando sul suo nome apriamo una finestra in cui troviamo una più o meno ampia descrizione del pacchetto.

Questi pacchetti possono essere programmi che fanno qualche cosa una volta lanciati con l'interprete Python che abbiamo sul computer. Ad esempio, nel repository possiamo trovare programmi di utilità, giochi, ecc.

Di maggior interesse penso tuttavia siano le librerie (moduli) che ci consentono di espandere le potenzialità del pacchetto base.

Per arricchire il nostro Python di base con una libreria per il calcolo matriciale, per esempio, ci serve il modulo **NumPy**.

Se vogliamo un arricchimento per calcoli scientifici, altre a NumPy ci serve **SciPy**.

Se vogliamo inoltrarci nel machine learning, oltre ai due moduli precedenti ci serve **Scikit-learn** (importabile negli script come sklearn).

Di grande utilità, per sviluppare grafici 2D, il modulo **Matplotlib**.

Per lavorare su serie temporali e grandi moli di dati abbiamo il modulo **Pandas**.

Installando tutte queste librerie attrezziamo il nostro Python per elaborazioni matematiche e statistiche di altissimo livello.

Se, più che alla scienza, siamo interessati al gioco, possiamo avere bisogno della libreria **python-chess**, utile per sviluppare applicazione di gioco a scacchi o la libreria **pgoapi** per sviluppare applicazioni pokemon, ecc.

Per sviluppare applicazioni dotate di interfaccia grafica, in alternativa o in aggiunta al già citato modulo **tkinter**, facile da usare ma forse un tantino povero nei risultati estetici, troviamo **PyQt4** e **wxPython**.

Ovviamente, per usare i moduli aggiuntivi dobbiamo anche sapere come funzionano e il nostro indice PyPI, insieme ad una sommaria descrizione del pacchetto, ci indica il sito web di chi o della comunità che l'ha sviluppato, in modo che ci sia possibile trovare documentazione.

Per pacchetti dell'importanza di NumPy o SciPy, come per le librerie Qt4 e wxWidgets, troviamo parecchi manuali e manualetti navigando su Internet.

Il problema è l'installazione, a volte complicata, soprattutto per i programmi più che per le librerie, dal dover rispettare certe dipendenze, dal dover tener conto della versione Python con cui vogliamo utilizzare il modulo contenuto nel pacchetto, ecc.

Tutti questi problemi si possono superare utilizzando un installatore che si chiama **pip**.

Installazione e uso di pip

Sistema Linux

Dalle versioni Python 2.7.9 e Python 3.4 pip è installato di default.

Nelle distro Linux della famiglia Debian (Ubuntu e derivate, Mint, MX) lo possiamo comunque installare sia come generico pip (che agisce sulla versione Python di default) sia come pip specializzato per la versione 3 di Python, con i comandi a terminale impartiti con collegamento Internet attivo

`sudo apt-get install python-pip`
`sudo apt-get install python3-pip`[4].

Per installare un pacchetto contenuto in PyPI si usa il comando a terminale
`pip install <nome_pacchetto>`
oppure
`pip3 install <nome_pacchetto>`.

Per disinstallare un pacchetto precedentemente installato con pip si usa il comando a terminale
`pip uninstall <nome_pacchetto>`
oppure
`pip3 uninstall <nome_pacchetto>`.

Non dovrebbe essere richiesto di impartire questi comandi come superuser; in caso di difficoltà far precedere `sudo` ai comandi.

[4]Nelle versioni più recenti di Linux non si usa più apt-get ma semplicemente apt.

Con il comando
`pip list`
oppure
`pip3 list`
possiamo vedere l'elenco dei pacchetti catalogati in PyPI che abbiamo installato.

L'installazione di pip può avvenire anche scaricando il file **get-pip.py** da
https://bootstrap.pypa.io/get-pip.py
e, posizionati nella directory dove abbiamo messo il file scaricato, dando il comando a terminale
`python get-pip.py`
oppure
`python3 get-pip.py`.

In ogni caso l'upgrade di pip si fa con il comando a terminale
`pip install -U pip`
oppure
`pip install --upgrade pip`.

Sistema Mac OS X

Su Mac non è installato pip.

Oltre che con get-pip.py, nel modo appena visto per Linux, possiamo installare pip su Mac impartendo i seguenti comandi a terminale
`xcode-select --install`
`sudo easy_install pip`.

L'upgrade lo facciamo con
`sudo pip install --upgrade pip`.

Per l'installazione, la disinstallazione e la listatura dei pacchetti valgono gli stessi comandi visti per Linux.

Sistema Windows

Se installiamo Python con l'installer come abbiamo visto nel paragrafo precedente, ci troviamo l'eseguibile pip.exe nella directory C:\PythonXX-\scripts e da lì possiamo lanciarlo per installare, disinstallare o listare i pacchetti con gli stessi comandi visti per Linux.

Rammento che quello che in Linux e Mac si chiama terminale per lanciare questi comandi, in Windows è la finestra che si apre con il comando `cmd`.

* * * * *

Alcuni dei principali pacchetti di moduli contenuti nel repository, come NumPy, SciPy, PyQt4, ecc. sono comunque scaricabili da Internet in archivi compressi e sono installabili anche nel modo che vediamo nel paragrafo che segue.

1.4 Alternative al repository e a pip

Come ho appena detto, alcuni pacchetti che fanno parte del repository si trovano facilmente su Internet in archivi compressi che si prestano all'installazione manuale e altri pacchetti, che possiamo trovare su Internet, per i più svariati motivi, non sono catalogati nel repository.

In questi casi, procuratoci l'archivio che contiene il pacchetto (generalmente si tratta di un archivio compresso con estensione .tar.gz o .zip), lo scompattiamo e, con privilegio da superutente, lo trasferiamo nella directory adatta, che ora vedremo dove si trova, e da questo momento il modulo è importabile nei nostri script Python.

Unica avvertenza quella di sapere se il pacchetto è adatto alla versione Python che abbiamo installato.

La directory in cui mettere l'archivio è `dist_packages` o `site_packages` e si trova ai seguenti indirizzi

- in Linux e Mac OS `/usr/lib/pythonXX` o `/usr/local/lib/pythonXX`
- in Windows `C:\pythonXX\Lib`.

dove pythonXX sarà, per esempio, python3.5, python3.7, python2.7, ecc., a seconda della versione Python installata.

Può accadere, poi, che qualche modulo sia installabile, per chi usa linux, ricorrendo ai vari installatori (apt per Debian e famiglia Ubuntu e derivate, compreso Mint, yum per Fedora e Red Hat, YaST per SuSE) che attingono dai repository delle varie distro.

Esiste, infine, la possibilità di affidarsi ad installatori predisposti, come Anaconda, attraverso i quali dedichiamo una zona del nostro computer al mondo Python, ma, per le finalità che qui ci proponiamo basta che ci affidiamo a pip[5].

[5]Chi fosse interessato ad Anaconda può consultare l'allegato PDF «python_anaconda» all'articolo «Software libero per data scientists» dell'aprile 2019, archiviato in Software libero nel blog dell'autore, all'indirizzo *www.vittal.it*.

1.5 Il problema della trasferibilità degli script

Il programma Python, in realtà, è uno script, cioè è una serie di istruzioni scritte in linguaggio pseudo-umano, il linguaggio Python, che vengono eseguite dal computer essendo interpretate nel momento stesso dell'esecuzione. Sul computer dove vengono interpretate deve essere installato tutto ciò che serve per interpretarle, cioè il pacchetto Python di base e tutte le librerie cui eventualmente lo script faccia ricorso, quelle importate nella stesura dello script stesso. Uno script che fa ricorso al modulo NumPy può essere eseguito solo su un computer sul quale sia installato non solo il pacchetto base di Python ma anche il modulo NumPy, a sua volta scritto in linguaggio Python.

Tutti i vantaggi fruiti al momento della stesura dello script, con la possibilità di sperimentare le istruzioni nel momento stesso in cui le scrivevamo e con l'accesso ad una miriade di librerie di funzioni preconfezionate che ci facilitava il compito, li paghiamo nel momento in cui non ci accontentiamo di essere noi i soli utenti dello script, sul nostro computer, ma vogliamo trasferirlo ad altri, affinché lo possano utilizzare sul loro computer.

Un modo per superare il problema sarebbe quello di indurre il destinatario dello script ad attrezzarsi per poterlo eseguire: basterebbe che anche lui installasse le librerie necessarie. Né sarebbe difficile sapere quali debbano essere: basterebbe caricare lo script in un editor di testo e vedere, nelle prime righe, quali moduli esso importa.

Un modo più comodo per il destinatario dello script è quello di trasferirgli un file che contenga tutto quanto serve per eseguire lo script stesso. Alcuni chiamano questo file "eseguibile": propriamente non è un eseguibile in linguaggio macchina del tipo di quelli prodotti compilando programmi scritti in C o in Pascal; è più simile all'altrettanto così detto "eseguibile" archivio .jar del linguaggio Java. Con la differenza, comunque, che l'archivio .jar richiede la presenza di Java sul computer, mentre il file che produciamo nel caso di Python non richiede nemmeno la presenza dell'interprete di base Python, per cui assomiglia veramente ad un eseguibile propriamente detto. Tutto ciò si paga con la pesantezza in termini di byte: ma, con le tecnologie di cui disponiamo ora, sia per trasferire sia per archiviare file, è un problema relativo.

Lo strumento principe per produrre questo file è **PyInstaller**, un pacchetto che, essendo contenuto nel PyPI, è installabile con pip con il comando a terminale

`pip install pyinstaller`.

Per produrre il file, posizionati nella directory che contiene lo script, si scrive a terminale il comando

`pyinstaller --onefile` <nome_script>`.py`

e nella stessa directory, con altre cose che possiamo cancellare, troviamo una nuova directory `dist` che contiene un file che ha lo stesso nome dello script, senza l'estensione .py, e che è il così detto "eseguibile" concentrato in un solo file.

L'indicazione dell'opzione `--onefile` è consigliabile in quanto, altrimenti, nella nuova directory `dist` non troveremmo solo il file che ci interessa, unico, ma una sottodirectory, con il nome dello script di partenza contenente un file "eseguibile" che deve viaggiare accompagnato da tutto quanto contiene la sottodirectory stessa, il che sarebbe parecchio scomodo da trasferire e da maneggiare.

Da tener presente che un file prodotto con PyInstaller su un sistema a 32 bit non gira su un sistema a 64 bit e viceversa. Così come un file prodotto con PyInstaller su un sistema Linux non gira su Windows e viceversa.

Altra avvertenza: su Linux il file "eseguibile" va reso tale sul computer di destinazione con `chmod +x` <nome_file> oppure attraverso il gestore file, cliccando destro sul file e agendo su PROPRIETÀ -> PERMESSI.

In alternativa a PyInstaller, che funziona su tutti i sistemi operativi Linux, Mac OS X e Windows, ricordo l'esistenza, per il sistema Windows, di **py2exe** e, per il sistema Mac OS X, di **py2app**, che fanno la stessa cosa.

Detto questo, devo segnalare l'esistenza di una corrente di pensiero che considera una cattiva soluzione quella del ricorso a questi software e che raccomanda, soprattutto nel caso di script impegnativi che utilizzino librerie importanti, l'installazione di quanto serve sul computer destinatario del trasferimento dello script.

Capitolo 2

Basi del linguaggio

2.1 Come funziona

Python è un linguaggio interpretato e interattivo.

Interpretato significa che il codice che scriviamo viene eseguito direttamente senza essere compilato in linguaggio macchina, come avviene con altri linguaggi (C, Pascal).

Interattivo significa che possiamo anche vedere eseguite le nostre istruzioni intanto che le scriviamo.

Quello che abbiamo installato secondo quanto indicato nel Capitolo precedente è l'interprete e viene avviato con il comando a terminale `python` (se abbiamo installato anche Python 2 il comando per l'interprete di Python 3 è `python3`)[1].

Se ci proponiamo di realizzare il nostro primo programma Ciao mondo possiamo sperimentarlo in due modi.

Possiamo scrivere in un file di testo, con un qualsiasi editor di testo, questa istruzione

```
print('Ciao mondo')
```

e salvare il file con estensione `.py`: ciaomondo.py.
Successivamente, e tutte le volte che vogliamo, posizionandoci nella directory dove abbiamo salvato il file e scrivendo a terminale
`python ciaomondo.py` (o `python3 ciamondo.py`)
vedremo eseguito il nostro programma (meglio chiamato script) che produrrà la voluta scritta.

[1] Il terminale è una finestrella nella quale possiamo scrivere istruzioni con cui interagire con il computer. In Linux e Mac OS X si chiama proprio Terminale e troviamo modo di avviarlo dai menu a disposizione. In Windows si chiama Prompt dei comandi e si avvia con il comando `cmd` o `command`.

L'altro modo è quello di avviare l'interprete scrivendo a terminale il comando

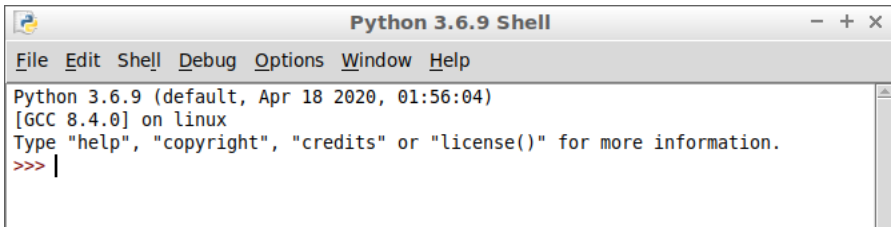
`python` (o `python 3`)

col che il terminale si trasforma in terminale Python, la shell Python, una conchiglia all'interno della quale si usa il linguaggio Python, e vi possiamo scrivere l'istruzione

```
print('Ciao mondo')
```

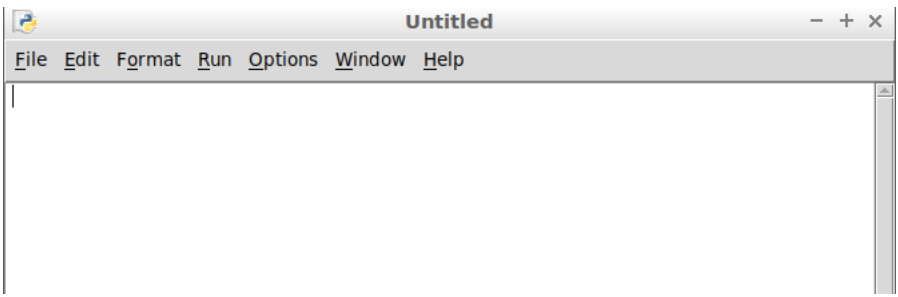
vedendola immediatamente eseguita alla pressione del tasto INVIO.

Se usiamo la IDLE, quando la lanciamo ci troviamo di fronte questa finestra



che è la shell di Python, nella quale ci possiamo divertire con l'interattività di Python.

Da menu FILE ▷ NEW FILE apriamo invece l'editor



nel quale possiamo scrivere il nostro programma, salvarlo con FILE ▷ SAVE AS, e lanciarlo con RUN ▷ RUN MODULE.

2.2 Tipi e tipizzazione

In Python, oltre ai soliti tipi che troviamo in tutti i linguaggi (numeri, stringhe) abbiamo anche tipi molto speciali che possono diventare utili in programmi di una certa complessità (liste, tuple, dizionari).

2.2.1 Numeri

Python tratta numeri interi, numeri in virgola mobile (dotati del separatore decimale `.`) e numeri complessi.

Per la dimensione di un numero intero (`int`) non c'è limite, salvo quello fisico della memoria del computer.

Per i numeri in virgola mobile (`float`) abbiamo la solita precisione limitata a 17 cifre, virgola compresa.

Nei numeri complessi (`complex`) parte reale e parte immaginaria, dotate di segno, sono scritte di seguito e la parte immaginaria è contrassegnata dal suffisso `j`.

Esempi:
`728` è un numero intero,
`7.5` è un numero in virgola mobile,
`-1+0.78j` è un numero complesso.

2.2.2 Stringhe

La stringa (`str`) è una sequenza di caratteri racchiusa tra apici semplici (`'`) o doppi (`"`). Se la racchiudiamo tra apici doppi, all'interno possiamo collocare apici singoli senza che questi la chiudano.

La stringa è immutabile, cioè non è possibile aggiungervi altri caratteri o toglierne.

Esempi:
`'Pippo'` è una stringa,
`"l'altro giorno"` è una stringa.

2.2.3 Tuple

La tupla (`tuple`) è una sequenza di elementi eterogenei, racchiusi tra parentesi tonde e separati da una virgola (,).

La tupla è immutabile, cioè non è possibile aggiungervi altri elementi o toglierne.

Esempio:
`(1, 24, 'giuseppe')` è una tupla.

2.2.4 Liste

La lista (`list`) è una sequenza di elementi eterogenei, racchiusi tra parentesi quadre e separati da una virgola (,).

La lista può essere modificata, cioè è possibile aggiungervi altri elementi e toglierne quando si vuole.

Esempio:
`[15, 'Ciao', (13, "Vittorio"), [16, 'Elena']]` è una lista.
Come si vede la lista, come la tupla, può contenere altre liste e tuple.

2.2.5 Dizionari

Il dizionario (`dict`) è una sequenza di elementi eterogenei, racchiusi tra parentesi graffe, contraddistinti da una chiave accoppiata a ciascun elemento.

Chiave e relativo elemento sono separati da due punti (:) e gli accoppiamenti sono separati da virgola (,).

Esempio:
`{'a': 12, 4: (22, 'pippo'), 'c':'Giuseppe'}` è un dizionario in cui `'a'`, `4` e `'c'` sono le chiavi.

2.2.6 Insiemi

L'insieme (`set`) viene creato dalla parola chiave `set` seguita, tra parentesi tonde, da una qualsiasi sequenza di elementi. Gli elementi dell'insieme sono tutti diversi uno dall'altro e sono racchiusi tra parentesi graffe.

Esempio:
`set('Vittorio')` crea l'insieme delle lettere che compongono la parola Vittorio, scartando le doppie, con il risultato `{'V', 'r', 't', 'i', 'o'}`.

2.2.7 Valori booleani

I valori booleani (`bool`) sono `True` (vero) e `False` (falso).

2.2.8 Nulla

Il nulla (`NoneType`) è identificato dalla parola `None`.

* * *

Come scriviamo qualche cosa, Python, basandosi su ciò che scriviamo e su come lo scriviamo, assegna uno dei suoi tipi a ciò che scriviamo.

Esiste anche una funzione, la funzione `type()`, attraverso la quale possiamo verificare quale tipo Python assegna alla cosa che scriviamo.

Per esempio, se scriviamo nella shell di Python `type(122)`, viene ritornato <class 'int'>, a dimostrazione del fatto che Python ha capito che abbiamo scritto un numero intero.
Se scriviamo `type(True)` viene ritornato `<class 'bool'>`, a dimostrazione del fatto che Python ha capito che abbiamo scritto un valore booleano.
Se scriviamo `type([1, 4, 'ciao'])` viene ritornato `<class 'list'>`, a dimostrazione del fatto che Python ha capito che abbiamo scritto una lista.

2.3 Variabili

Le variabili sono delle locazioni di memoria, delle scatole, alle quali diamo un nome, destinate a contenere valori di un certo tipo.

In Python le variabili si creano nel momento in cui servono, senza bisogno, come avviene in quasi tutti gli altri linguaggi di programmazione, di dichiararle prima.

La tipizzazione della variabile è dinamica ed avviene automaticamente al momento dell'assegnazione del valore, in quanto Python, come abbiamo visto alla fine del precedente paragrafo, riconosce il tipo da come scriviamo il valore stesso.

L'assegnazione del valore viene fatta con l'operatore = e la sintassi
`<nome_variabile> = <valore>`.
`nome = 'Vittorio'` crea la variabile `nome` e le assegna il valore di tipo stringa `Vittorio`
`raggio = 6.5` crea la variabile `raggio` e le assegna il valore di tipo float `6,5`
`l = [15, 22, 8.5]` crea la variabile `l` e le assegna la lista `[15, 22, 8.5]`.

Il valore può essere espresso in modo letterale, come negli esempi, o in forma di espressione matematica, più o meno coinvolgendo altre variabili.

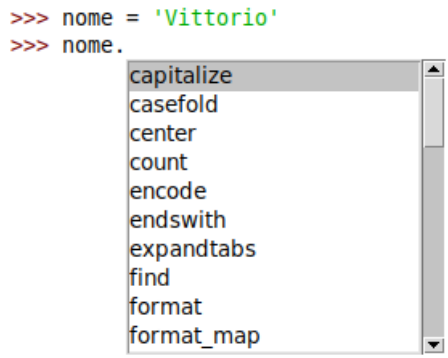
Il fatto che Python utilizzi la tipizzazione dinamica non vuol dire che sia un linguaggio che non dia importanza ai tipi: al contrario, Python è un linguaggio fortemente tipizzato.
Per esempio non è possibile sommare un valore numerico ad un valore di tipo stringa e non è possibile fare con una tupla ciò che si può fare con una lista.

Infatti, a seconda del tipo assunto da una variabile, Python associa all'oggetto variabile tutta una serie di proprietà e metodi.

Esiste un comando, `dir`, che elenca proprietà e metodi di un oggetto passato come argomento con la sintassi
`dir(<argomento>)`
dove `<argomento>` può essere qualsiasi oggetto riconosciuto da Python (un modulo, una variabile, un oggetto creato da noi).

Meglio ancora, utilizzando la IDLE, possiamo vedere e scegliere i metodi associati ad una variabile scrivendo il nome della variabile stessa seguito da un punto (`.`) e attendere fino a quando ne vedremo l'elenco.

In questa illustrazione vediamo tutto ciò con riferimento ad una variabile stringa appena creata



Scorrendo l'elenco possiamo scegliere il metodo che ci interessa ed agire. Se, per esempio, volessimo convertire la nostra stringa in caratteri maiuscoli, scorrendo l'elenco dovremmo scegliere il metodo `upper` e passare l'istruzione `nome.upper()`.
In questo modo produrremmo la stringa `'VITTORIO'` lasciando inalterata la stringa originaria contenuta nella variabile nome.
Se l'intenzione è di sostituirla l'istruzione deve essere
`nome = nome.upper()`.
Se l'intenzione è di avere una nuova variabile con le lettere maiuscole accanto a quella con le lettere minuscole possiamo usare l'espressione `nome_maiuscolo = nome.upper()`.

Nell'economia di questo manuale non posso illustrare tutti i metodi che Python associa alle variabili nei vari tipi, anche perché probabilmente sconfinerei dall'intenzione di presentare qui le basi del linguaggio.

Utilizzando la IDLE, che non a caso sta per Integrated Development and Learning Environment, possiamo facilmente sperimentare i vari metodi che ci vengono offerti e stupirci di fronte alla ricchezza ed alla potenza di Python.

Può accadere di dover modificare il tipo di una variabile, per esempio per poter applicare ad essa metodi che non sarebbero propri del tipo assegnato dinamicamente al momento della sua creazione.

Abbiamo a disposizione le funzioni `int`, `float`, `str`, `list` e `tuple` per fare alcune trasformazioni.

Esempi:

sia `l` una variabile di tipo lista `[1, 6, 8]`
con `tuple(l)` la leggiamo come tupla e con `t = tuple(l)` la trasformiamo nella tupla `t`
in ogni caso la variabile `l` rimane tale e quale

sia `x` una variabile numerica a virgola mobile `13.7`
con `str(x)` la leggiamo come stringa `'13.7'` e con `s = str(x)` la trasformiamo nella stringa `s`

Se la variabile contiene una stringa una tupla, o una lista, ciascun elemento della sequenza è individuabile attraverso il suo indice posto tra parentesi quadre dopo il nome della variabile: gli indici sono interi positivi in successione, con il valore zero attribuito al primo elemento.
Esempio:
se `l` è una variabile di tipo lista `[1, 6, 8]`
`l[1] corrisponde a 6.`

Se la variabile contiene un dizionario, fungono da indice le chiavi.
Esempio:
se d è una variabile di tipo dizionario `{'a': 12, 4: (22, 'pippo'), 'c':'Giuseppe'}`
`d['c'] corrisponde a Giuseppe.`

2.4 Operatori

Gli operatori collegano tra loro operandi di varia natura in espressioni che forniscono un risultato.

2.4.1 Operatori aritmetici

In ordine di esecuzione sono i seguenti:
`**` per l'elevamento a potenza tra numeri,
`*` per la moltiplicazione tra numeri,
`/` per la divisione tra numeri,
`%` per il modulo (resto della divisione intera),
`+` per la somma tra numeri o il concatenamento di stringhe,
`-` per la sottrazione tra numeri.

2.4.2 Operatori di confronto

Servono per confrontare due valori e il risultato che restituiscono è un valore booleano. Sono i seguenti:
`==` uguale,
`!=` non uguale,
`<` minore,
`<=` minore o uguale,
`>` maggiore,
`>=` maggiore o uguale.
Gli operandi assoggettati al confronto devono avere lo stesso tipo.

2.4.3 Operatori logici

Forniscono come risultato un valore booleano e sono i seguenti:
`&` per l'AND logico,
`|` per l'OR logico.

2.4.4 Operatori sugli insiemi

Agiscono tra operandi di tipo `set` e sono:
`|` per l'unione,
`-` per la differenza,
`&` per l'intersezione,
`^` per la differenza simmetrica (elementi presenti in uno dei due insiemi ma non in entrambi).

2.5 Interattività con l'utente

L'interfacciamento con l'utente nel terminale avviene attraverso due funzioni: `print` e `input`.

2.5.1 Output

La funzione per l'output è `print` con la sintassi
`print(`<cosa_scrivere>`)`
dove <cosa_scrivere> può essere indicato con una stringa, una espressione matematica o il nome di una variabile che contiene il valore che vogliamo scrivere, anche combinati tra loro separati con una virgola (`,`).

Esempi:
`print('ciao')` scrive `ciao`,
`print(5)` scrive `5`,
`print(3*7.5)` scrive `22.5`,
`print('3 per 8 fa', 3*8)` scrive `3 per 8 fa 24`,
data la variabile x contenente il valore `16`,
`print(x)` scrive `16`,
`print('la variabile x vale', x)` scrive `la variabile x vale 16`.

Con la direttiva di formattazione
`'%.`<cifre_decimali>`f' %`
possiamo stabilire le cifre decimali da scrivere arrotondando l'ultima.

Per esempio:
`print(7.168 * 4.68)` scrive `33.54624`
`print('%.2f' %(7.168*4.68))` scrive `33.55`
data la variabile `p` contenente il valore `3.141592653589793`
`print('%.5f' %p)` scrive `3.14159`

2.5.2 Input

La funzione per l'input è `input` con la seguente sintassi
`input(`<eventuale_stringa_messaggio>`)`
dove <eventuale_stringa_messaggio> può servire per chiedere all'utente che cosa inserire.

L'esecuzione del programma resta sospesa fino a quando l'utente non ha inserito da tastiera il dato richiesto.

Il dato inserito viene letto come stringa.

Se deve essere letto come numero il dato letto va reso argomento della funzione `eval()`.

Esempi:

`print(2**eval(input('esponente di due ')))` se inseriamo 3 scrive 8

`raggio = eval(input('raggio '))` inserisce un numero nella variabile `raggio`

`x = input('Digita un numero ')` inserisce il numero digitato nella variabile `x` come stringa (se la variabile `x` deve entrare in un calcolo numerico dobbiamo preventivamente modificarne il tipo con `float(x)`).

2.6 Istruzioni complesse e indentazione

Capita spesso, soprattutto in programmi non banali, di dover gestire istruzioni in blocco e, in altri linguaggi di programmazione, queste istruzioni vengono in genere racchiuse tra parentesi graffe.

Python risolve il problema in maniera del tutto originale, attraverso l'indentazione, con la sintassi

```
<istruzione_apertura_blocco>:
    <istruzione>
    <istruzione>
    ....
<seguito_oltre_blocco>
```

La prima istruzione apre il blocco e termina con due punti (`:`) e il blocco è costituito dalle istruzioni rientranti sotto questa istruzione.

Come si esce dall'indentazione il blocco finisce.

2.7 Strutture di controllo

Come ogni altro linguaggio di programmazione, Python ha dei comandi per condizionare l'esecuzione di certe istruzioni al verificarsi di determinate condizioni oppure per la ripetizione dell'esecuzione di una o più istruzioni.

2.7.1 Esecuzione condizionale

if

L'istruzione `if`, chiamata istruzione di esecuzione condizionale (in inglese if è il nostro se), ci dà modo di assoggettare l'esecuzione di un blocco di istruzioni al verificarsi di una determinata condizione: se la condizione è vera viene eseguito il blocco di istruzioni, altrimenti si prosegue l'esecuzione del programma saltando il blocco stesso.

La sintassi è la seguente

```
if <condizione>:
    <istruzioni>
```

dove <condizione> è una qualsiasi espressione che relaziona due valori attraverso operatori di confronto: se la condizione si verifica vengono eseguite la o le istruzioni indicate, altrimenti si passa oltre.

L'istruzione `if` si presta anche all'esecuzione condizionale a due rami. Per ottenere questo dobbiamo abbinarla all'istruzione `else` con questa sintassi

```
if <condizione>:
    <istruzioni>
else:
    <istruzioni>
```

In questo caso se la condizione si verifica vengono eseguite le istruzioni contenute nel primo blocco altrimenti vengono eseguite quelle contenute nel secondo blocco dopo else (altrimenti). In ogni caso proseguendo poi nell'esecuzione del resto del programma.

Possiamo infine gestire l'esecuzione condizionale a più rami abbinando a `if` l'istruzione `elif` con questa sintassi

```
if <condizione>:
    <istruzioni>
elif <condizione>:
    <istruzioni>
elif <condizione>:
    <istruzioni>
else:
    <istruzioni>
```

2.7.2 Ripetizione

for

Si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni per un numero definito di volte.

La sintassi per l'uso di questa istruzione in Python è varia.

La situazione più semplice è quella che utilizza una sorta di contatore che è il costrutto `range`.

```
for i in range(n):
    <istruzioni>
```

esegue <istruzioni> `n` volte.

Numero delle iterazioni ed elementi su cui eseguirle possono anche essere determinati utilizzando un qualsiasi oggetto che sia una sequenza ed è il numero degli elementi della sequenza a stabilire il range.

Per esempio:

```
for x in 'pippo':
    print('ciao')
```

scrive cinque volte (quante sono le lettere della stringa `'pippo'`) la parola `ciao`.

while

Si usa per ripetere istruzioni fino a quando si verifica una certa condizione.

La sintassi è:

```
while <condizione>:
    <istruzioni>
```

Se la condizione è espressa attraverso l'uso di un contatore otteniamo gli stessi risultati che otteniamo con l'istruzione `for` vista prima

```
i = 1
while i <= 5:
    print "ciao"
    i = i + 1
```

scrive cinque volte la parola `ciao`.

L'istruzione `while`, combinata con `break`, si presta ad interrompere un ciclo al verificarsi di un certo evento, come l'inserimento di un certa stringa da tastiera:

con

```
while 1:
    x = input()
    if x == 'fine':
        break
```

essendo il valore 1 sempre vero, viene richiesto un input fino a quando non si scrive la parola `fine`.

2.8 Semplici programmi

Quanto visto finora ci consente di creare i nostri primi semplici script.

Questo calcola media, varianza e scarto quadratico medio di una serie di numeri:

```
lista = []
sx = 0
qx = 0
n = 0
print('Inserisci i dati (f per finire)')
while 1:
    xs = input()
    if xs == 'f':
        break
    x = float(xs)
    lista.append(x)
    sx = sx + x
    qx = qx + x * x
    n = n + 1
m = sx / n
v = qx / n - m * m
sqm = v ** (1/2)
print("dati inseriti:")
for i in lista:
    print(i)
print('media:  ', '%0.4f' %m)
print('varianza:  ', '%0.4f' %v)
print('scarto quadratico medio:  ', '%0.4f' %sqm)
```

Notare che le variabili `lista`, `sx`, `qx` e `n` sono inizializzate al valore 0 prima del ciclo in quanto nel ciclo vengono lavorate come già esistenti (`lista.append`, `sx = sx + x`, ecc.).
Non avendo ancora visto come si estrae una radice quadrata, quella della varianza per determinare lo scarto quadratico medio è calcolata elevando la varianza a 1/2.
I dati in output comprendono l'elencazione dei numeri inseriti e i risultati sono espressi arrotondando a quattro cifre decimali.

Quest'altro script rivolge un saluto personalizzato al nome inserito con la tastiera:

```
nome = input('Come ti chiami?  ')
print('Ciao,', nome, '!')
La seconda riga potrebbe anche essere
print('Ciao, ' + nome + '!')
```

utilizzando l'operatore di concatenamento di stringhe.

Il seguente calcola l'area di un triangolo di cui si chiedono base e altezza:

```
b = eval(input('base:  '))
h = eval(input('altezza:  '))
a = b * h / 2
print("L'area di un triangolo di base", b, 'e altezza', h, 'è:', a)
```

2.9 Funzioni

La funzione è una sezione del programma in cui è racchiusa una serie di istruzioni da eseguire e che vengono eseguite ogni volta che la funzione è chiamata.

Per semplificare la scrittura di un certo programma, potrebbe essere utile raggruppare alcune istruzioni all'interno del programma stesso, creando funzioni richiamabili, in modo da suddividere i compiti e da evitare di scrivere più volte le stesse cose.

La sintassi per definire una funzione è

```
def <nome_funzione> (<nome_parametro>, <nome_parametro>,...):
    <istruzioni>
```

dove <nome_funzione> è il nome che intendiamo dare alla funzione, ed è il nome che useremo per richiamarla, <nome_parametro> è il nome del o dei parametri (dati) da inserire quando la richiamiamo.
Le <istruzioni> sono quelle relative alle elaborazioni da effettuare sui parametri per ottenere il risultato.
Se la funzione deve restituire un valore numerico, l'ultima istruzione è `return` con indicazione del risultato.

La sintassi per chiamare una funzione ed eseguirla è

```
<nome_funzione> (<parametro>, <parametro>,...)
```

Esempi:

definita la seguente funzione

```
def saluta(nome):
    print('Ciao ' + nome)
```

con l'istruzione `saluta('Giuseppe')` scriviamo `Ciao Giuseppe`.

definita la funzione

```
def area_triangolo(base, altezza):
    return base * altezza / 2
```

con l'istruzione `area_triangolo(12, 6)` otteniamo il risultato `36`.

Come esercizio esemplificativo supponiamo di voler scrivere un programma che calcoli Combinazioni e Disposizioni, semplici senza ripetizione, di n numeri presi k a k.

Le formule che dobbiamo utilizzare sono
$C(n,k) = \frac{n!}{(n-k)!k!}$ per le combinazioni,
$D(n,k) = \frac{n!}{(n-k)!}$ per le disposizioni,
nelle quali ricorre spesso il calcolo del fattoriale.
Il fatto che la necessità di questo calcolo ricorra spesso e comporti una non semplice scrittura di istruzioni per effettuarlo suggerisce di isolare queste istruzioni in una funzione dedicata al calcolo del fattoriale in modo da poter richiamare queste istruzioni in blocco quando serve.
La funzione per calcolare il fattoriale può essere scritta così:

```
def fattoriale (n):
    if n < 2:
       return 1
    return n * fattoriale (n - 1)
```

E' un bell'esempio di formula ricorsiva (funzione che richiama sé stessa) che ci dimostra come il linguaggio Python supporti la ricorsività.
Tra poco scopriremo che, nel modulo `math`, Python dispone della funzione `factorial()` per calcolare il fattoriale, per cui ci saremmo potuti evitare questa fatica.
Il programma per calcolare combinazioni e disposizioni di n numeri presi k a k utilizzando la funzione scritta da noi diventa il seguente:

```
def fattoriale (n):
    if n < 2:
        return 1
    return n * fattoriale(n-1)
n = eval(input('numero elementi:  '))
k = eval(input('presi:  '))
c = fattoriale(n)/(fattoriale(n-k) * fattoriale(k))
d = fattoriale(n)/fattoriale(n-k)
print('combinazioni:  ', c)
print('disposizioni:  ', d)
```

2.10 Moduli

Praticamente un modulo è una raccolta di funzioni.

Tra i moduli che fanno parte della dotazione basica di Python mi soffermo su quello che offre la maggiore utilità per un principiante.

2.10.1 Math

Come tutti i moduli, per essere utilizzato va importato con una particolare istruzione, da scrivere prima di ogni altra nello script, e lo si può fare in tre modi:

. `import math`

. `import math as <abbreviazione>`

. `from math import *`

In ogni caso l'operazione ci mette a disposizione le funzioni racchiuse nel modulo.

Nel primo caso la funzione che ci interessa si richiama come funzione membro dell'oggetto `math` con la sintassi

`math.<funzione>`.

Nel secondo caso la funzione che ci interessa si richiama come funzione membro dell'oggetto ribattezzato con la sintassi

`<abbreviazione>.<funzione>`.

Nel terzo caso possiamo richiamare la funzione che ci interessa semplicemente nominandola.

In ogni caso, se si tratta di funzione, si devono mettere tra parentesi tonde i parametri richiesti.

Esempio:
tra le funzioni del modulo `math` c'è `sqrt()` per estrarre la radice quadrata.
Per estrarre la radice quadrata di 9:
. se abbiamo importato con `import math` scriviamo `math.sqrt(9)`,
. se abbiamo importato con `import math as m` scriviamo `m.sqrt(9)`,
. se abbiamo importato con `from math import *` scriviamo `sqrt(9)`.

Il modulo `math` ci offre innanzi tutto i valori delle due più famose costanti e e π , rispettivamente richiamabili con `e` e `pi`.

Come per ogni altro modulo, per sapere quali funzioni contiene il modulo, dopo averlo importato con

```
>>> import math
```

digitiamo

```
>>> dir(math)
```

e ci si presenta l'elenco.

Tra le funzioni di uso più ricorrente rammento le seguenti.

algebriche

`exp()` ritorna una potenza del numero e
`log()` ritorna il logaritmo naturale di un numero
`log10()` ritorna il logaritmo in base 10 di un numero
`sqrt()` ritorna la radice quadrata di un numero

`pow(base, esponente)` ritorna base elevata a esponente
`factorial()` ritorna il fattoriale di un numero
`trunc()` ritorna la parte intera di un numero
`floor()` ritorna l'intero inferiore di un numero decimale
`ceil()` ritorna l'intero superiore di un numero decimale

Forse balza all'occhio la mancanza di una funzione che determini il valore assoluto di un numero. In realtà Python dispone della funzione `abs()` al di fuori dal modulo `math`.

trigonometriche

`sin()` ritorna il seno
`asin()` ritorna l'arcoseno
`cos()` ritorna il coseno
`acos()` ritorna l'arcocoseno
`tan()` ritorna la tangente
`atan()` ritorna l'arcotangente

Gli argomenti per le funzioni trigonometriche vanno espressi in radianti.

Utili le seguenti funzioni di trasformazione:
`degrees(radianti)` trasforma radianti in gradi
`radians(gradi)` trasforma gradi in radianti

iperboliche

`sinh()` ritorna il seno iperbolico
`asinh()` ritorna l'arcoseno iperbolico
`cosh()` ritorna il coseno iperbolico
`acosh()` ritorna l'arcocoseno iperbolico
`tanh()` ritorna la tangente iperbolico
`atanh()` ritorna l'arcotangente iperbolico

2.10.2 Ricchezza di Python

Quello che abbiamo visto è il modulo di supporto al calcolo scientifico che troviamo nella dotazione di base.

Un altro modulo molto utile che troviamo installato se abbiamo installato la IDLE è il modulo `tkinter`, che ci consente di sviluppare applicazioni dotate di interfaccia grafica (GUI). Di questo ci occuperemo nel prossimo Capitolo.

Come abbiamo visto nel Capitolo 1 esiste poi una grande quantità di altri moduli e, nei Capitoli 5 e 6 parleremo di quelli predisposti per la ricerca scientifica e per la data science.

2.11 Oggetti

Ovviamente Python supporta la programmazione a oggetti.

Tanti più esperti di me raggiungibili in rete possono ricordare o spiegare al lettore che cosa si intende per programmazione a oggetti.

Io provo a farlo capire con un esempio.

Supponiamo di avere spesso bisogno di determinare volume e superficie del cubo di cui ci è noto lo spigolo.

Se vogliamo evitare di scrivere tutte le volte le necessarie formule, anche perché non è detto che ce le ricordiamo (a volte si ha a che fare con formule che più difficilmente di quelle per calcolare la superficie di un cubo sono rimandabili a memoria) possiamo creare delle funzioni che le contengono, come abbiamo fatto prima per calcolare il fattoriale, oppure possiamo ricorrere alla programmazione a oggetti. Se seguiamo quest'altra strada scriviamo una Classe attraverso la quale creare un oggetto Cubo, contenente funzioni (chiamate funzioni membro) attraverso le quali calcolare volume e superficie.

Come si scrive una funzione lo abbiamo visto. Nel caso del nostro esempio la funzione per calcolare il volume del cubo sarà scritta così:

```
def volume_cubo(spigolo):
    return spigolo ** 3
```

Per avere il volume di un cubo di spigolo 5, scriveremo `volume_cubo(5)`

Nella programmazione per oggetti, la classe per creare il cubo, con le funzioni membro per calcolare superficie e volume, sarà

```
class Cubo:
    def __init__(self, spigolo):
        self.s = spigolo
    def superficie(self):
        self.superficie = self.s * self.s * 6
        return self.superficie
    def volume(self):
        self.volume = self.s * self.s * self.s
        return self.volume
```

Utilizzando questa classe possiamo costruire un oggetto cubo con un certo spigolo, ad esempio 5:

```
mioCubo = Cubo(5)
```

e richiamarne le funzioni membro per scrivere superficie e volume:
```
print(mioCubo.superficie())
print(mioCubo.volume())
```
Il primo metodo inserito nella classe, dal nome prestabilito `__init__`, è il metodo costruttore, il cui primo parametro, che si usa chiamare `self`, ma potrebbe essere battezzato a piacere, ci permette di accedere all'oggetto che si sta per creare e, usato negli altri metodi, di accedere all'oggetto creato. L'altro parametro per il costruttore è lo spigolo, l'unico che, in questo caso, riteniamo adeguato per qualificare il cubo da costruire in vista delle grandezze da calcolare.
Se avessimo a che fare con una piramide, i parametri dovrebbero almeno essere: lato della base, numero dei lati della base e altezza della piramide.

Dopo di che abbiamo le due funzioni membro per determinare superficie e volume del nostro cubo, che andremo a costruire indicandone lo spigolo.

Il codice della classe e quello del suo utilizzo andrebbero inclusi nello stesso script.

Se si preferisce memorizzare a parte il codice della classe, lo si salva in un file con estensione `.py`, nel nostro caso, per esempio, `Cubo.py` nella stessa directory in cui salveremo lo script che lo utilizzerà.
La prima riga di questo script sarà
```
from Cubo import *.
```

2.12 Lavorare con file

Esiste la funzione `open()` con la quale possiamo creare un oggetto `file` dotato di funzioni per scriverne, aggiornarne o leggerne il contenuto.

La sintassi è la seguente
```
f = open(<nome_file>, <modalità>)
```
dove `f` è un qualsiasi nome che diamo all'oggetto `file`,
`<nome_file>` è una stringa che indica il percorso al file,
`<modalità>` è una stringa che può assumere i seguenti valori:

`'w'` se vogliamo scrivere sul file, creandolo se non c'è o sostituendolo,

`'a'` se vogliamo aggiungere elementi ad un file esistente senza sostituirlo,

`'r'` se vogliamo leggere il file.

Le più importanti funzioni membro del nostro oggetto file sono
`.write(<stringa>)` per scrivere una stringa nel file (per andare a capo occorre terminare la stringa con `\n`),

`.read()` per leggere tutto il file come unica stringa,
`.readline()` per leggere una riga del file (una volta letta una riga ci si posiziona sulla riga successiva e occorre ripetere il comando per leggerla),
`.readlines()` per leggere tutto il file e inserirne il contenuto in una lista i cui elementi sono costituiti da una riga del file.

Per rendere operativo ciò che abbiamo fatto sul file e chiuderlo posizionandosi sul primo elemento occorre utilizzare il metodo
`.close()`

Esempio:
con `f = open('/home/vittorio/Documenti/prova', 'w')`
creo il file `prova` nella posizione indicata dal percorso,
con `f.write('Pippo\n')` scrivo la sua prima riga con il nome Pippo,
con `f.close()` chiudo il file, confermando quanto scritto in esso,
con `f = open('/home/vittorio/Documenti/prova', 'a')`
apro il file `prova` per aggiungere altri elementi
con `f.write('Pluto\nPaperino\n')`
aggiungo altre due righe,
con `f.close()` chiudo il file, confermando quanto aggiunto,
con `f = open('/home/vittorio/Documenti/prova', 'r')`
apro il file `prova` per leggerne il contenuto
con `f.readline()` leggo la prima riga e, se ripeto il comando subito dopo, leggo la seconda e così via
con `f.close()` chiudo il file e riposiziono tutto.
Se riapro il file in lettura,
con `f.readlines()` ottengo la lista
`['Pippo\n', 'Pluto\n', 'Paperino\n']`
con `f.read()` otterrei la stringa
`'Pippo\nPluto\nPaperino\n'`

Il metodo `write` può scrivere una sola stringa per volta, per cui se vogliamo scrivere più stringhe dobbiamo concatenarle e se vogliamo scrivere numeri dobbiamo lavorare di conversione di tipo.

Per esempio, con il comando
```
f.write('3 moltiplicato 2 fa ' + str(3 * 2) + '\n')
```
scriveremmo nel nostro file la riga
```
3 moltiplicato 2 fa 6
```

Quello qui presentato è il modo di lavorare con i file basico di Python. Se l'esigenza è quella di lavorare su file di grandi dimensioni, contenenti dati da elaborare, non è certo questa la strada da percorrere e Pyhton ci offre ben altro. Per questo rimando a quanto citato nel Paragrafo 2.10.2 Ricchezza di Python.
Esiste, per esempio, il modulo `Pandas` che ci consente di leggere file che contengono tabelle, dati separati da virgola (csv) su più colonne, potendo anche specificare le colonne da leggere.

2.13 Lavorare con database

Con il linguaggio Python possiamo collegarci a un database per eseguirvi operazioni di scrittura o di lettura utilizzando il linguaggio SQL.

Esistono moduli adatti per i più noti database: `sqlite3` per sqlite3, `MySQLdb` per MySQL, `psycopg2` per PostGreSQL, `pyodbc` per MySQL, PostGreSQL, Access, Oracle.
Si tratta di moduli che non fanno parte della dotazione di base ma che vanno installati, per esempio utilizzando `pip` secondo quanto indicato nel Capitolo 1.

Una volta importato il modulo con

```
import <nome_modulo>
```

creiamo una connessione con il database, che chiamiamo con un nome evocativo, per esempio, `db`, con

```
db = <nome_modulo>.connect("<database>")
```

dove `<database>` sta per percorso e nome del database, messi tra apici come stringa.

Poi, per lavorare sul database con cui siamo collegati, utilizzando il metodo `cursor()` della connessione creiamo un cursore, che chiamiamo, per esempio, con la sua iniziale `c`, con

```
c = db.cursor().
```

I principali metodi del cursore così creato ci consentono di eseguire sul database comandi SQL o di leggere i risultati di una query, in particolare:

```
c.execute("<comando_SQL>")
```

esegue un comando SQL indicato come stringa o come variabile stringa;

```
risultato = c.fetchall()
```

crea una lista, chiamata risultato, nella quale ogni elemento è una tupla che rappresenta la riga di una astratta tabella con i dati della query per riga e colonna, come estratti dal database su cui è stata fatta la ricerca, dati a loro volta rappresentati sotto forma di stringhe unicode.

Alla fine del nostro lavoro chiudiamo il database con `db.close()`.

Ad esempio, per estrarre il contenuto della tabella dei Paesi di un database sqlite3 «biblioteca» che contiene titoli di libri di autori italiani, inglesi, statunitensi e russi possiamo usare il seguente script:

```
import sqlite3
db = sqlite3.connect("/home/vittorio/Database/biblioteca")
c = db.cursor()
c.execute("select * from paesi")
risultato = c.fetchall()
print(risultato)
db.close()
```

L'esecuzione dello script fornisce il seguente risultato:
```
[(1, 'Italia'), (2, 'Gran Bretagna'), (3, 'USA'), (4, 'Russia')]
```
Se non ci piace questo modo grezzo di presentare il risultato e lo vogliamo esporre meglio possiamo intervenire inserendo, al posto dell'istruzione
```
print(risultato)
```
il seguente blocchetto:
```
for i in range(len(risultato)):
    ri = risultato[i]
    r = str(ri[0]) + " " + ri[1]
    print r
```
nel quale ogni elemento della lista risultato viene preso come una tupla isolata, ogni elemento della tupla viene accostato all'altro come stringa, con in mezzo un piccolo spazio libero, e ogni stringa risultante viene stampata e vediamo quanto segue:
```
1 Italia
2 Gran Bretagna
3 USA
4 Russia
```

2.14 Da Python 2 a Python 3

Le differenze tra Python 3 e Python 2 sistemando le quali uno script Python 2 può essere eseguito dall'interprete Python 3 sono le seguenti.

Importazione di moduli

- Il modulo `tkinter` si importa con `from tkinter import *`, con la t minuscola (nella versione 2 Tkinter è scritto con la T maiuscola),
- Il modulo `sqlite3`, per la verità fin dalla versione 2.6, ha sostituito il vecchio modulo `PySQlite` e si importa con `import sqlite3` e la connessione si stabilisce con

  `mioDatabase = sqlite3.Connection(<''`path e nome del database`''>)`.

Tipi di dati predefiniti

- il tipo numerico `long` è unificato nel tipo `int`: nella versione 3 tutti gli interi sono `long`;
- il tipo `unicode` è unificato nel tipo `str`; nella versione 3 tutte le stringhe sono `unicode` e l'operatore `u` per renderle tali non c'è più;

- il tipo `basestring` è unificato nel tipo `str`;
- il tipo `xrange` è abolito e `xrange()` è unificato in `range()`;
- `types.InstanceType` è abolito;
- `types.UnboundedMethodType` è abolito.

In generale

- il comando `print` della versione 2 diventa la funzione `print()` nella versione 3 e non si fa più `print 'ciao'` ma si fa `print('ciao');`
- per scrivere in un file non si usa più il comando `print>>`<nome_file> ma si usa il metodo write dell'oggetto file: `<nome_file>.write();`
- il risultato della divisione con l'operatore `/` è sempre un float, anche se la divisione è fatta tra interi. Se serve che il risultato sia un intero occorre renderlo tale con il casting `int();`
- la funzione `input()` incamera una stringa; per incamerare un numero, nella versione 3 occorre scrivere `eval(input());`
- la funzione `raw_input()` è abolita ed è unificata nella funzione `input();`
- la funzione `cmp()` è abolita e va sostituita utilizzando i confronti elementari `(x>y)`, `(x<y)`. Negli ordinamenti si devono usare le chiavi.

Capitolo 3

Grafica

Scopo di certi pacchetti grafici è quello di programmare software dotato della così detta Graphical User Interface (GUI), cioè di un'interfaccia grafica che ci consenta di interagire con il computer attraverso la visualizzazione di ciò che facciamo, utilizzando un puntatore grafico (mouse) per scegliere cosa fare o, ove possibile, addirittura toccando lo schermo con un dito.

Per arricchire di tutto questo i programmi, o, meglio, gli script Python esistono almeno quattro pacchetti di questo tipo, che ci mettono a disposizione altrettanti framework: PyQt, wxPython, PyGTK e Tkinter.

I primi tre si rifanno rispettivamente ai toolkit e alle librerie Qt, wxWidgets e GTK+.

Tkinter deriva dalle librerie Tcl/Tk ed è da sempre la libreria standard per Python, anche se pare che il creatore stesso di Python, Guido van Rossum, ammetta che i risultati che si ottengono utilizzando wxWidgets siano graficamente migliori. Probabilmente Guido è più abile di me ad installare wxPython su Linux: tra noi comuni mortali molti ci hanno provato e poi si sono detti «ma chi me lo fa fare?». PyGTK è molto vecchio e sta per essere soppiantato da PyGObject, ormai di casa con Python 3 per Linux ma non ancora pronto per Windows e Mac. Qt non è software libero. Meno male che c'è sempre Tkinter.

Tra l'altro, tra tutte le librerie citate, Tkinter è sicuramente la più facile da usare, la più leggera, è molto stabile e, dopo gli arricchimenti introdotti con il modulo ttk nella versione 8.5, fornisce risultati grafici di tutto rispetto.

Ho pertanto scelto di proporre in questo libro Tkinter.

Una delle migliori guide all'uso di Tkinter, a mio giudizio, è quella di John W. Shipman e la troviamo all'indirizzo

www.nmt.edu/tcc/help/pubs/tkinter/tkinter.pdf.

E' scritta in inglese, la considero adatta per professionisti e qui non ho intenzione di scimiottarla in italiano. Qui propongo una modalità di utilizzo di Tkinter che è molto più semplice di quella proposta da Shipman, meno professionale, ma più avvincente per i dilettanti evoluti ai quali sono solito rivolgere i miei lavori.

La guida di Shipman può eventualmente servire per avere più dettagli sugli strumenti a disposizione, per approfondimenti su argomenti che io toccherò di striscio o non toccherò affatto e per chi voglia un approccio con lo stile di programmazione per classi, più professionale di quello che seguirò io.

In 1.2 Il pacchetto base e sua installazione abbiamo visto come installare il modulo Tkinter.

Per utilizzarlo dobbiamo importarlo con
`from tkinter import *`.

3.1 I widget di Tkinter

Secondo la definizione che ne dà Wikipedia un widget è un componente grafico dell'interfaccia utente di un programma, che ha lo scopo di facilitare all'utente l'interazione con il programma stesso. Il termine deriva dalla contrazione dei termini "window" e "gadget" ed ha una valenza un tantino dispregiativa, dal momento che un gadget è solitamente una cosa di poco conto e poco seria: probabilmente ciò deriva dal fatto che il termine è stato coniato nell'epoca in cui le finestre con i pulsanti e l'uso del mouse facevano la loro prima apparizione in un mondo in cui l'informatica si riteneva seria solo se praticata su terminali a riga di comando.

In sostanza i widget sono i mattoncini con i quali possiamo costruire una GUI (Graphical User Interface).

Si tratta, anche nel linguaggio informatico, di veri e propri oggetti, dotati di metodi per svolgere determinati compiti.

Ne propongo qui una rassegna, limitandomi a quelli di utilità più ricorrente.

3.1.1 Widget contenitori

Sono considerati widget ma, in realtà, sono dei recipienti destinati a contenere widget.

Sono sostanzialmente due: il canvas e il frame.

Il primo è pensato per applicazioni orientate al disegno e alla grafica, il secondo è più adatto per applicazioni scientifiche e da ufficio.

Canvas

Il canvas (termine inglese che in italiano significa canovaccio o tela) è un'area rettangolare in cui possiamo mettere di tutto (anche altri widget) ma che è particolarmente adatta per disegnarci sopra, come avviene appunto per la tela del pittore.

L'oggetto informatico canvas è infatti dotato di parecchi metodi per disegnare.

La costruzione dell'oggetto canvas che vogliamo concretamente usare avviene con l'istruzione

`<identificatore>=Canvas(<genitore>, <opzione> = <valore>, ...)`

dove

`<identificatore>` è il nome che diamo al nostro canvas e servirà per richiamarne i metodi;

`<genitore>` identifica il contenitore in cui viene inserito il canvas;

`<opzione>` e `<valore>` servono per definire determinati attributi del nostro canvas.

Le opzioni possibili sono molte e per la conoscenza di tutte rimando alla guida di Shipman che ho citato nella premessa a questo Capitolo. Ci basti conoscere queste tre fondamentali:

* la larghezza del canvas, esprimibile con `width = <numero_pixel>`
* l'altezza del canvas, esprimibile con `height = <numero_pixel>`
* il colore di fondo del canvas, esprimibile con `bg = <colore>`

La via più semplice per indicare il colore è quella di indicarne il nome, racchiuso tra apici, scegliendolo tra quelli riconosciuti da Tkinter, elencati nella tabella di figura 1 alla pagina seguente.

Il colore di default, nel caso non lo si indichi, è un grigio chiaro (lightgrey).

Se vogliamo costruire un canvas, chiamato c, di 300 pixel per 400, con sfondo giallo dobbiamo scrivere

```
c = Canvas(<genitore>, width = 300, height = 400, bg = 'yellow')
```

Come avviene per tutti i widget, dopo che il canvas è stato costruito, ogni opzione è modificabile con il metodo `config()`.

Se vogliamo allargare a 400 pixel il canvas, chiamato c, che abbiamo appena costruito e modificarne lo sfondo rendendolo verde dobbiamo scrivere

```
c.config(width = 400, bg = 'green')
```

Nomi simbolici dei colori in Tkinter

alice blue – AliceBlue – antique white – AntiqueWhite – AntiqueWhite1 … AntiqueWhite4 – agua – aquamarine – aquamarine1 … aquamarine4 – azure – azure1 … azure4

beige – bisque – bisque1 … bisque4 – black – blanched almond – BlanchedAlmond – blue – blue violet – blue1 – blue2 – blue3 – blue4 – BlueViolet – brown – brown1 … brown4 – burlywood – burlywood1 … burlywood4

cadet blue – CadetBlue – CadetBlue1 – CadetBlue2 – CadetBlue3 – CadetBlue4 – chartreuse – chartreuse1 … chartreuse4 – chocolate – chocolate1 … chocolate4 – coral – coral1 … coral4 – cornflower blue – CornflowerBlue – cornsilk – cornsilk1 … cornsilk4 – crymson – cyan – cyan1 … cyan4

dark blue – dark cyan – dark goldenrod – dark gray – dark green – dark grey – dark khaki – dark magenta – dark olive green – dark orange – dark orchid – dark red – dark salmon – dark sea green – dark slate blue – dark slate gray – dark slate grey – dark turquoise – dark violet – DarkBlue – DarkCyan – DarkGoldenrod – DarkGoldenrod1 … DarkGoldenrod4 – DarkGray – DarkGreen – DarkGrey – DarkKhaki – DarkMagenta – DarkOliveGreen – DarkOliveGreen1 … DarkOliveGreen4 – DarkOrange – DarkOrange1 … DarkOrange4 – DarkOrchid – DarkOrchid1 … DarkOrchid4 – DarkRed – DarkSalmon – DarkSeaGreen – DarkSeaGreen1 … DarkSeaGreen4 – DarkSlateBlue – DarkSlateGray – DarkSlateGray1 … DarkSlateGray4 – DarkSlateGrey – DarkTurquoise – DarkViolet – deep pink – deep sky blue – DeepPink – DeepPink1 … DeepPink4 – DeepSkyBlue – DeepSkyBlue1 … DeepSkyBlue4 – dim gray – dim grey – DimGray – DimGrey – dodger blue – DodgerBlue – DodgerBlue1 … DodgerBlue4

firebrick – firebrick1 … firebrick4 – floral white – FloralWhite – forest green – ForestGreen – fuchsia

gainsboro – ghost white – GhostWhite – gold – gold1 … gold4 – goldenrod – goldenrod1 … goldenrod4 – gray – gray0 – gray1 … gray100 – green – green yellow – green1 … green4 – GreenYellow – grey – grey0 … grey100

honeydew – honeydew1 … honeydew4 – hot pink – HotPink – HotPink1 … HotPink4

indian red – IndianRed – IndianRed1 … IndianRed4 – indigo – ivory – ivory1 … ivory4

khaki – khaki1 … khaki4

lavender – lavender blush – LavenderBlush – LavenderBlush1 … LavenderBlush4 – lawn green – LawnGreen – lemon chiffon – LemonChiffon – LemonChiffon1 … LemonChiffon4 – light blue – light coral – light cyan – light goldenrod – light goldenrod yellow – light gray – light green – light grey – light pink – light salmon – light sea green – light sky blue – light slate blue – light slate gray – light slate grey – light steel blue – light yellow – LightBlue – LightBlue1 … LightBlue4 – LightCoral – LightCyan – LightCyan1 … LightCyan4 – LightGoldenrod – LightGoldenrod1 … LightGoldenrod4 – LightGoldenrodYellow – LightGray – LightGreen – LightGrey – LightPink – LightPink1 … LightPink4 – LightSalmon – LightSalmon1 … LightSalmon4 – LightSeaGreen – LightSkyBlue – LightSkyBlue1 … LightSkyBlue4 – LightSlateBlue – LightSlateGray – LightSlateGrey – LightSteelBlue – LightSteelBlue1 … LightSteelBlue4 – LightYellow – LightYellow1 … LightYellow4 – lime – lime green – LimeGreen – linen

magenta – magenta1 … magenta4 – maroon – maroon1 … maroon4 – medium aquamarine – medium blue – medium orchid – medium purple – medium sea green – medium slate blue – medium spring green – medium turquoise – medium violet red – MediumAquamarine – MediumBlue – MediumOrchid – MediumOrchid1 … MediumOrchid4 – MediumPurple – MediumPurple1 … MediumPurple4 – MediumSeaGreen – MediumSlateBlue – MediumSpringGreen – MediumTurquoise – MediumVioletRed – midnight blue – MidnightBlue – mint cream – MintCream – misty rose – MistyRose – MistyRose1 … MistyRose4 – moccasin

navajo white – NavajoWhite – NavajoWhite1 … NavajoWhite4 – navy – navy blue – NavyBlue

old lace – OldLace – olive – olive drab – OliveDrab – OliveDrab1 … OliveDrab4 – orange – orange red – orange1 … orange4 – OrangeRed – OrangeRed1 … OrangeRed4 – orchid – orchid1 … orchid4

pale goldenrod – pale green – pale turquoise – pale violet red – PaleGoldenrod – PaleGreen – PaleGreen1 … PaleGreen4 – PaleTurquoise – PaleTurquoise1 … PaleTurquoise4 – PaleVioletRed – PaleVioletRed1 … PaleVioletRed4 – papaya whip – PapayaWhip – peach puff – PeachPuff – PeachPuff1 … PeachPuff4 – peru – pink – pink1 … pink4 – plum – plum1 … plum4 – powder blue – PowderBlue – purple – purple1 … purple4

red – red1 … red4 – rosy brown – RosyBrown – RosyBrown1 … RosyBrown4 – royal blue – RoyalBlue – RoyalBlue1 … RoyalBlue4

saddle brown – SaddleBrown – salmon – salmon1 – salmon2 – salmon3 – salmon4 – sandy brown – SandyBrown – sea green – SeaGreen – SeaGreen1 – SeaGreen2 – SeaGreen3 – SeaGreen4 – seashell – seashell1 … seashell4 – sienna – sienna1 … sienna4 – silver – sky blue – SkyBlue – SkyBlue1 … SkyBlue4 – slate blue – slate gray – slate grey – SlateBlue – SlateBlue1 … SlateBlue4 – SlateGray – SlateGray1 … SlateGray4 – SlateGrey – snow – snow1 … snow4 – spring green – SpringGreen – SpringGreen1 … SpringGreen4 – steel blue – SteelBlue – SteelBlue1 … SteelBlue4

tan – tan1 … tan4 – teal – thistle – thistle1 … thistle4 – tomato – tomato1 … tomato4 – turquoise – turquoise1 … turquoise4

violet – violet red – VioletRed – VioletRed1 … VioletRed4

wheat – wheat1 … wheat4 – white – white smoke – WhiteSmoke

yellow – yellow green – yellow1 … yellow4 – YellowGreen

Figura 3.1: Nomi dei colori riconosciuti da Tkinter

Ogni pixel del canvas è identificato da una coppia di coordinate, la prima indicante la posizione in orizzontale (convenzionalmente la chiamiamo x), la seconda indicante la posizione in verticale (convenzionalmente la chiamiamo y). L'origine del sistema di coordinate sta nell'angolo superiore di sinistra del canvas, che ha coordinate x = 0 e y = 0.

Per disegnare nel canvas abbiamo a disposizione i seguenti metodi, che possiamo richiamare indicandoli dopo il nome che abbiamo assegnato al canvas e il punto:

`create_line(x1, y1, x2, y2, <opzione> = <valore>, ...)`

disegna una linea che inizia nel punto di coordinate x1, y1 e termina nel punto di coordinate x2, y2, per default di colore nero e di tratto sottile (valore 1 pixel).

Per le tante opzioni disponibili rimando alla già citata guida di Shipman.

Qui rammento l'opzione `fill = <colore>` per avere un colore diverso dal nero e l'opzione `width = <numero_pixel>` per scegliere un tratto più pesante.

Esempio:

```
c.create_line(10, 10, 50, 50, fill = 'red', width = 5)
```

disegna un segmento rosso, di buon spessore tra i punti indicati nel canvas c.

`create_rectangle(x1, y1, x2, y2, <opzione> = <valore>, ...)`

disegna un rettangolo con l'angolo in alto a sinistra nel punto di coordinate x1, y1 e con l'angolo in basso a destra nel punto di coordinate x2, y2, per default tracciato in nero e senza riempimento. Ovviamente, se la distanza tra le x è uguale alla distanza tra le y, si disegna un quadrato.

Tra le opzioni rammento le più utili: `width = <numero_pixel>` per regolare il tratto della linea che disegna la figura, `outline = <colore>` per indicare il colore di questa linea e `fill = <colore>` per indicare il colore di riempimento.

`create_polygon(x1,y1,x2,y2,x3,y3, ... , <opzione> = <valore>, ...)`

disegna il riempimento di un poligono con i vertici nei punti corrispondenti alle coordinate indicate percorse in senso orario e, per default, in colore nero.

Le opzioni sono praticamente le stesse viste per il metodo rectangle e possono servire per cambiare il colore di riempimento e per tracciare e dare un colore alla linea di contorno che per default, non viene tracciata.

`create_oval(x1, y1, x2, y2, <opzione> = <valore>, ...)`

disegna un'ellisse inscritta in un rettangolo con l'angolo in alto a sinistra nel punto di coordinate x1, y1 e con l'angolo in basso a destra nel punto di coordinate x2, y2, per default tracciata in nero e senza riempimento. Ovviamente, se la distanza tra le x è uguale alla distanza tra le y, si disegna un cerchio.

Le opzioni sono le stesse di quelle del metodo rectangle.

`create_arc(x1, y1, x2, y2, <opzione> = <valore>, ...)`

disegna un arco dell'ellisse inscritta in un rettangolo con l'angolo in alto a sinistra nel punto di coordinate x1, y1 e con l'angolo in basso a destra nel punto di coordinate x2, y2, per default tracciata in nero e senza riempimento.

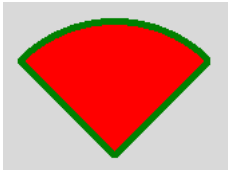
Oltre alle opzioni che abbiamo a disposizione per regolare dimensione e colore dell'arco e per il riempimento, le stesse del metodo rectangle, qui abbiamo in più l'opzione `style`, che può assumere i valori '`pieslice`', '`chord`' e '`arc`', rispettivamente per disegnare figure a fetta di torta (quella di default), ad arco con estremi uniti da corda o semplicemente l'arco.

Inoltre, per orientare a nostro piacere la figura, abbiamo le opzioni `start` e `extent`, il cui argomento è il valore in gradi di un angolo: per `start` l'angolo a partire dal quale tracciare l'arco, per `extent` l'angolo in corrispondenza del quale terminare il tracciamento in senso antiorario.

Esempi:

```
c.create_arc(50,50,250,250,start=45,extent=90, fill='red', width=5,
    outline='green', style='pieslice')
```

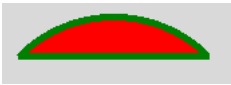
disegna questa figura



L'opzione style='pieslice' si faceva a meno di indicarla, in quanto la pieslice è l'opzione di default.

```
c.create_arc(50,50,250,250,start=45,extent=90, fill='red', width=5,
    outline='green', style='chord')
```

disegna questa figura



```
c.create_arc(50,50,250,250,start=45,extent=90, fill='red', width=5,
    outline='green', style='arc')
```

disegna questa figura

Il riempimento non c'è in quanto non c'è nulla da riempire.

`create_text(x, y, text = ' ... ', <opzione> = <valore>, ...)`

scrive il testo indicato, tra apici, nell'opzione text (per andare a capo inserire \n), in colore nero, utilizzando il font Arial, 10, normale e centrandolo verticalmente e orizzontalmente sul punto di coordinate x e y.

Per utilizzare un font diverso è a disposizione l'opzione con sintassi `font =` seguito, tra parentesi tonde, dal nome del font scritto tra apici, dal numero indicante la dimensione e dal tipo (normal, bold, italic) scritto tra apici, il tutto separato da virgole.

Per utilizzare un colore diverso abbiamo l'opzione `fill = <colore>`.

Per posizionare il testo a nostro piacimento abbiamo l'opzione `anchor =` che accetta i parametri NE, NW, SE e SW ad indicare dove si collochi il punto di coordinate x e y rispetto alla scritta. Indicando, per esempio NW, la scritta si svilupperà verso destra e sotto il punto, che rimarrà così collocato sopra e a sinistra (come dire nord-ovest) della scritta.

Al default corrisponde il parametro CENTER.

Esempio:

```
c.create_text(100, 100, text = 'Ciao', fill = 'green',
    font = ('olivier', 18, 'bold'), anchor = SW)
```

colloca la scritta Ciao, in verde, con il font Olivier, 18, grassetto sviluppandola sopra verso destra al punto di coordinate 100 e 100 pixel.

```
c.create_text(100, 150, text = 'Ciao', fill = 'red', anchor = NE)
```

colloca la scritta Ciao, in rosso, con il font di default Arial, 10, normale sviluppandola sotto e in modo che termini in corrispondenza del punto di coordinate 100 e 150 pixel.

`create_bitmap(x, y, bitmap = <indirizzo>,<opzione> = <valore>, ...)`

inserisce un'immagine bitmap, in colore nero, centrandola sul punto di coordinate x e y.

Con la stessa opzione `anchor =` appena vista per create_text e gli stessi parametri possiamo scegliere altro modo di collocare l'immagine.

Con l'opzione `foreground = <colore>` scegliamo un colore.

Sono disponibili in tkinter alcune icone bitmap preconfezionate che rispondono alle definizioni 'gray75', 'gray50', 'gray25', 'gray12', 'hourglass', 'info', 'questhead', 'question', 'warning' e 'error', richiamabili inserendone il nome tra apici al posto di <indirizzo>. Esse corrispondono, rispettivamente, a questi simboli

Possiamo creare noi stessi icone bitmap in formato .xbm con software grafici come GIMP e utilizzarle. In tal caso al posto di <indirizzo> inseriamo tra apici il percorso verso l'immagine desiderata preceduto da @.

`create_image(x, y, image = <identificatore_immagine>)`

inserisce un'immagine centrandola sul punto di coordinate x e y con le possibilità di personalizzazione consentite dal metodo `anchor` visto per create_text e create_bitmap.

L'immagine accettata deve essere in formato .gif o .png e deve essere preventivamente resa oggetto importabile con il metodo `PhotoImage()` con la sintassi

```
<identificatore_immagine> =
    PhotoImage(file = <percorso al file .gif o .png>)
```

Esempio:

Supponiamo di avere una fotografia di un fungo in formato .jpg, chiamata boleto.jpg e vogliamo inserirla nel canvas.

Innanzi tutto, con un software adatto, tipo GIMP, la dobbiamo convertire nel formato .gif o .png e il nuovo file, per esempio boleto.png, lo archiviamo, per esempio, in /home/vittorio/fotografie.

Ora scriviamo

```
boleto=PhotoImage(file='/home/vittorio/fotografie/boleto.png')
```

e, finalmente, disponendo, sempre per esempio, di un canvas c di 400 per 400 pixel, lanciamo il metodo

```
c.create_image(200, 200, image = boleto)
```

e ci ritroveremo la nostra foto centrata nel canvas (se la fotografia è più grande del canvas risulterà ritagliata).

Quando con i metodi che abbiamo sopra elencato inseriamo disegni e testo nel canvas possiamo usare semplicemente, come visto nei pochi esempi, la sintassi
`<nome_canvas>.<metodo>`
ed otteniamo il risultato voluto.

Dal momento, tuttavia, che con questa istruzione creiamo in realtà un oggetto informatico, sarebbe buona norma assegnare a ciascun oggetto un nome identificatore utilizzando la sintassi
`<identificatore> = <nome_canvas>.<metodo>`
in modo da poter poi utilizzare alcuni metodi che consentono di manipolare l'oggetto stesso.

Questi metodi, sempre richiamabili indicandoli dopo il nome che abbiamo assegnato al canvas e il punto, sono i seguenti:

`delete(<identificatore>)`

rimuove il solo oggetto corrispondente all'identificatore dal canvas.

Per ripulire il canvas da tutto si usa `delete(ALL)`.

`coords(<identificatore>, <nuove_coordinate>)`

cambia le coordinate precedentemente indicate per disegnare l'oggetto corrispondente all'identificatore, con il risultato che l'oggetto verrà spostato.

Esempio:

Se abbiamo costruito un cerchio con l'istruzione

```
cerchio = c.create_oval(100, 100, 200, 200)
```

lo possiamo spostare con l'istruzione

```
c.coords(cerchio, 200, 200, 300, 300)
```

o, meglio ancora, per i motivi detti, con l'istruzione

```
nuovo_cerchio = c.coords(cerchio, 200, 200, 300, 300)
```

Il tutto senza andare a toccare gli altri parametri con i quali avevamo costruito il cerchio originario.

`itemconfig(<identificatore>, <opzione> = <valore>, ...)`

modifica una o più opzioni che caratterizzano l'oggetto corrispondente all'identificatore.

Le opzioni da modificare devono essere presenti nell'oggetto originario.

Esempio:

Se abbiamo un cerchio costruito con

```
cerchio=c.create_oval(100,100,200,200, outline = 'red', fill =
'green')
```

e lo vogliamo riempire di blu scriviamo

```
c.itemconfig(cerchio, fill = 'blue')
```

e ce lo troveremo ridisegnato riempito di blu.

Frame

Il frame (termine inglese che in italiano significa telaio) è uno spazio rettangolare in cui possiamo collocare altri widget.

Mentre il Canvas è dotato di tutta una serie di metodi anche per disegnarci sopra, modificare o cancellare ciò che si è fatto, il Frame non

è dotato di metodi ma si costruisce e basta. Poi vedremo in che modo possiamo ordinatamente collocarvici altri widget. Widget collocati per errore o per i quali si è cambiato idea si possono eliminare con il metodo `destroy()` di ciascun widget, scrivendo
`<identificatore_widget>.destroy())`

La costruzione dell'oggetto frame che vogliamo concretamente usare avviene con l'istruzione
`<identificatore>=Frame(<genitore>, <opzione> = <valore>, ...)`
dove
`<identificatore>` è il nome che diamo al nostro frame;
`<genitore>` identifica il contenitore in cui viene inserito il frame;
`<opzione>` e `<valore>` servono per definire determinati attributi del nostro frame.

Le opzioni possibili sono molte e per la conoscenza di tutte rimando alla già più volte citata guida di Shipman. Per l'economia di questo manuale ci basti conoscere queste tre fondamentali:
* la larghezza del frame, esprimibile con `width = <numero_pixel>`
* l'altezza del frame, esprimibile con `height = <numero_pixel>`
* il colore di fondo del frame, esprimibile con `bg = <colore>`.
Per come indicare il colore rimando a quanto detto per il Canvas a pagina 43. Il colore di default, nel caso non lo si indichi, è un grigio chiaro (lightgrey).

3.1.2 Widget di cui non si può fare a meno per costruire una GUI

Qualsiasi interfaccia utente deve necessariamente ed almeno consentire all'utente di comunicare con il computer, al computer di comunicare con l'utente ed all'utente di far partire o di arrestare una o più attività del computer.

Per poter fare queste cose Tkinter ci mette a disposizione tre widget: uno per fornire dati al computer (che viene chiamato Entry), uno per leggere dati che ci restituisce il computer o per rendere visibili istruzioni per l'utente (cose per le quali torna comodo quello chiamato Label) ed uno per dare il via o per stoppare determinate azioni del computer (cose per le quali va benissimo quello chiamato Button).

Entry

Il widget Entry consiste in una finestrella che serve per immettere una singola linea di testo.

Si costruisce con l'istruzione
`<identificatore>=Entry(<genitore>, <opzione> = <valore>, ...)`
dove `<genitore>` è il contenitore in cui vogliamo inserire il widget.

Tra le tante opzioni disponibili forse la più utile è quella che ci consente di stabilire la larghezza della finestrella in cui inserire il testo, per default di 20 caratteri. Per modificare questa dimensione è disponibile l'opzione `width = <numero_caratteri>`.
Con le opzioni `bg = <colore>` e `fg = <colore>` possiamo stabilire, rispettivamente, un colore per lo sfondo della finestrella e per il carattere scritto. Con l'opzione `justify =` possiamo allineare il contenuto della finestrella, utilizzando i parametri `LEFT`, `CENTER` e `RIGHT`.
Per la rassegna completa delle opzioni rimando alla guida di Shipman.

Il principale metodo del widget Entry è ovviamente quello di leggere e ritornare ciò che è stato scritto nella finestrella; si tratta del metodo `get()`, che ritorna una stringa contenente ciò che è stato scritto. Attenzione che il metodo ritorna una stringa anche se è stato immesso un numero, per cui, se si vuole un numero occorre fare il casting.

Per ripulire il contenuto della finestrella abbiamo il metodo `delete(0, END)`; se al posto di `0` mettiamo l'indice del primo carattere da cancellare e al posto di `END` l'ultimo, possiamo cancellare solo parte del contenuto.

Altro utile metodo è `focus_set()`, con il quale possiamo stabilire se la nostra finestrella debba essere già pronta per ricevere per prima i dati di input. Nel qual caso al suo interno vedremo un cursore lampeggiante.

Per richiamare un funzione una volta confermato l'inserimento si usa il metodo `bind('<Return>', <nome_funzione>)` (la funzione deve avere `event` tra i parametri attesi).
Esempio:
Con l'istruzione
```
inp = Entry(f, width = 10, bg = 'Lightblue')
```
costruiamo nel frame f una finestra di input che chiamiamo inp, predisposta per 10 caratteri, con sfondo azzurro.
Con l'istruzione
```
inp.focus_set()
```
la indichiamo come già predisposta a ricevere dati.
Se nella finestra scriviamo una stringa di testo, ad esempio un nome, possiamo inserire la stringa stessa in una variabile stringa, per esempio chiamata nome, con l'istruzione
```
nome = inp.get()
```
Se nella finestra scriviamo un numero che deve poi essere utilizzato come tale per fare calcoli, per inserire questo numero in una variabile numerica di tipo intero dobbiamo usare l'istruzione
```
n = int(inp.get())
```
e, per inserirlo in una variabile numerica a virgola mobile, l'istruzione
```
n = float(inp.get())
```

Label

Il widget Label consiste in una finestrella per esporre testo o numeri. E' utile sia per scrivere nella più ampia finestra della GUI istruzioni di compilazione e avvertenze varie, sia per scrivere le risposte del calcolatore alle elaborazioni che abbiamo chiesto.

Si costruisce con l'istruzione

`<identificatore>=Label(<genitore>, <opzione> = <valore>, ...)`

dove `<genitore>` è l'identificatore del contenitore in cui vogliamo inserire il widget.

Tra le tante opzioni disponibili quella sicuramente più importante è `text = ' .... '` che espone la stringa di testo racchiusa tra apici.

Altrettanto importanti sono il metodo `fg = <colore>` per dare un colore al testo e quello per la scelta del carattere con cui scriverlo: `font = (<famiglia>, <dimensione>, <tipo>)`. Il font di default è il solito Arial, 10, normale.

Per quanto riguarda il colore di fondo, quello di default è il solito grigio chiaro (lightgrey). Per modificarlo, soprattutto per renderlo magari uguale a quello non di default scelto per il contenitore, abbiamo il metodo `bg = <colore>`.

Attenzione meritano pure l'opzione `width = <valore>` e `height = <valore>` con le quali possiamo fissare un'ampiezza, in numero di caratteri, e un'altezza, in numero di righe, della label. Questo merita attenzione in quanto, una volta fissata l'ampiezza, ciò che scriviamo per l'opzione text non comparirà interamente se ha un numero di caratteri o di righe superiore. Se utilizziamo queste opzioni abbiamo il vantaggio di poter posizionare il testo all'interno della label utilizzando l'opzione `anchor = <parametro>`, dove parametro può essere `CENTER` (quello di default) per un allineamento al centro, `W` per un allineamento a sinistra, `E` per un allineamento a destra. Se la label contempla più righe possiamo usare i parametri `NE` per un allineamento in alto a destra, `SW` per un allineamento in basso a sinistra, ecc. Se non utilizziamo queste opzioni la label ha un'ampiezza e un'altezza elastiche che si adattano al contenuto; in questo caso, per evitare che l'ampiezza vada oltre una dimensione tollerabile, si può usare l'opzione `wraplength = <numero_pixel>`, in modo che quanto esposto vada a capo su una nuova riga una volta raggiunta l'ampiezza indicata.

Anche dopo la costruzione della label possiamo modificare o inserire nuove opzioni con il metodo

`<identificatore>.config(<opzione> = <valore>)`.

E' questo il metodo che utilizziamo, con l'opzione `text = <valore>` per scrivere i risultati di elaborazioni non ancora visibili all'apertura della GUI. Bene sapere che, in questa sede, <valore>, anziché essere una stringa tra apici, può essere il nome di una variabile, sia di tipo stringa sia di tipo numerico, non scritto tra apici. La variabile di tipo numerico non ha bisogno di casting se è l'unica componente della scritta; occorre il casting `string(<variabile_numerica>)` solo se è necessario un concatenamento con una stringa.

Esempio:

Se abbiamo il risultato 8 di un'operazione matematica nella variabile numerica chiamata r da mostrare in una label chiamata l1 possiamo scrivere

```
l1.config(text = r)
```

Se però vogliamo mostrare il risultato con la scritta «il risultato è: 8» dobbiamo scrivere

```
l1.config(text = 'il risultato è:  '  + string(r))
```

Button

Il widget Button è un pulsante sul quale si clicca con il mouse per far fare qualche cosa al computer.

Si costruisce con l'istruzione

`<identificatore>=Button(<genitore>, <opzione> = <valore>, ...)`

dove `<genitore>` è l'identificatore del contenitore in cui vogliamo inserire il widget.

Le opzioni sono molte ma le più importanti sono

`text = ' ...  '` con cui si indica tra apici la stringa di testo esplicativa da scrivere sul pulsante (il pulsante ha dimensione elastica e si adatta alla lunghezza della scritta e si sviluppa anche in altezza se la scritta va a capo, ciò che è possibile inserendo \n nella stringa stessa)

`command = <identificatore_funzione>`, con cui si indica la funzione che deve essere eseguita al click sul pulsante.

Con `width = <numero_caratteri>` e `height = <numero_righe>` è comunque possibile specificare la larghezza e l'altezza del pulsante, se il pulsante deve ospitare una scritta. Per inserire un'immagine anziché in caratteri e righe la dimensione si indica in pixel.

Utili potrebbero risultare anche `bg = <colore>` e `fg = <colore>` per dare un colore diverso dal solito grigio chiaro e nero di default, rispettivamente allo sfondo del pulsante ed alla scritta.

3.1.3 Widget Text

Abbiamo visto che il widget Entry può accettare una sola riga di testo e ciò, per la funzione di acquisizione di un input alla quale è destinato, è più che sufficiente.

Per trattare più grandi quantità di testo, come potrebbe essere necessario in un'applicazione di editing di testo, abbiamo il widget Text.

Si costruisce con l'istruzione

`<identificatore> = Text(<genitore>, <opzione> = <valore>, ...)`

dove `<genitore>` è l'identificatore del contenitore in cui vogliamo inserire il widget.

Le opzioni più importanti riguardano

* l'ampiezza della finestra, che possiamo stabilire con
  `width=<numero_caratteri>` (per default è di 80),
* l'altezza della finestra, che possiamo stabilire con
  `height = <numero_righe>` (per default è di 24),
* il font, che possiamo scegliere con
  `font=(<famiglia>,<dimensione>,<tipo>)`
  (per default è Arial, 10, normal),
* il colore del carattere, che scegliamo con
  `fg = <colore>` (per default è nero),
* il trattamento della riga eccedente la larghezza della finestra. Per andare a capo si preme il tasto Invio della tastiera. Per default, quando si raggiunge il limite destro della finestra si innesca a capo automatico sul carattere senza riguardo all'interezza della parola; il che corrisponde all'opzione `wrap = CHAR`. Per evitare l'interruzione della parola si deve usare l'opzione `wrap = WORD`, in modo che a capo automatico si inneschi dopo l'ultima parola completa battuta prima di raggiungere il limite destro della finestra. L'opzione wrap = NONE esclude che si vada a capo automaticamente e l'eccedenza scritta nella riga rimane nascosta e diventa visibile scorrendo la riga stessa.

3.1.4 Widget Menu

Qualsiasi applicazione che si rispetti è dotata di un menu e proprio Menu si chiama il widget che ci fornisce Tkinter per questo scopo.

Il widget Menu ha la particolarità di non poter essere inserito in un altro widget e può essere pertanto inserito solo nel contenitore radice, che è il genitore di tutto il progetto e ne parleremo in seguito.

Per costruire un menu abbiamo innanzi tutto bisogno di una barra in cui collocarlo; la barra si costruisce con l'istruzione

`<identificatore_barra> = Menu(<identificatore_radice>).`

Poi dobbiamo inserire la barra nel contenitore radice con l'istruzione
`<identificatore_radice>.config(menu = <identificatore_barra>).`

Finalmente diamo avvio alla costruzione del menu nella barra con l'istruzione
`<identificatore_menu> = Menu(<identificatore_barra>).`

Ora aggiungiamo le voci del menu, ciascuna con l'istruzione
`<identificatore_menu>.add_command(label='...',command=<funzione>)`
dove tra gli apici inseriamo una stringa per il nome da dare alla voce di menu e come funzione da eseguire richiamiamo una funzione altrove definita.

Possiamo separare un comando dall'altro con
`<identificatore_menu>.add_separator().`

Chiudiamo con l'istruzione
```
<identificatore_barra>.add_cascade(label='...',
        menu=<identificatore_menu>)
```
che dà la pennellata finale.

Esempio:
Con queste istruzioni costruiamo un menu m nella barra b del contenitore radice r con una sola voce Chiudi che richiama la funzione predefinita quit per chiudere la finestra.
```
b = Menu(r)
r.config(menu = b)
m = Menu(b)
m.add_command(label = 'Chiudi', command = quit)
b.add_cascade(label = 'File', menu = m)
```

3.1.5 Modulo filedialog

Nel menu di certe applicazioni è necessario inserire voci che ci aiutino a caricare o a salvare il lavoro da svolgere o svolto: si pensi, per esempio, ad un editor di testo o a un'agenda calendario.

Per fare questo Tkinter ha un modulo aggiuntivo, che si chiama filedialog e che, per poter essere utilizzato, occorre caricare esplicitamente con l'istruzione
`import tkinter.filedialog as fd`
in modo da rendere disponibile il modulo filedialog e dargli un nome (in questo caso ho scelto fd ma potremmo utilizzare qualsiasi nome). Ora fd è un oggetto e possiamo richiamare i suoi metodi con la classica sintassi della programmazione a oggetti (`fd.<nome_metodo>`).

I due metodi che ci fornisce questo modulo sono
`askopenfilename(<opzione> = <valore>, ... )`
`asksaveasfilename(<opzione> = <valore>, ... ).`

Il richiamo di questi metodi apre la classica finestra di dialogo con cui ci viene chiesto, rispettivamente, dove e con che nome trovare il file da aprire o dove e con quale nome salvare il file da salvare. La chiusura della finestra dopo l'inserimento dei dati richiesti, ritorna il path del file.

Le opzioni più importanti riguardano:

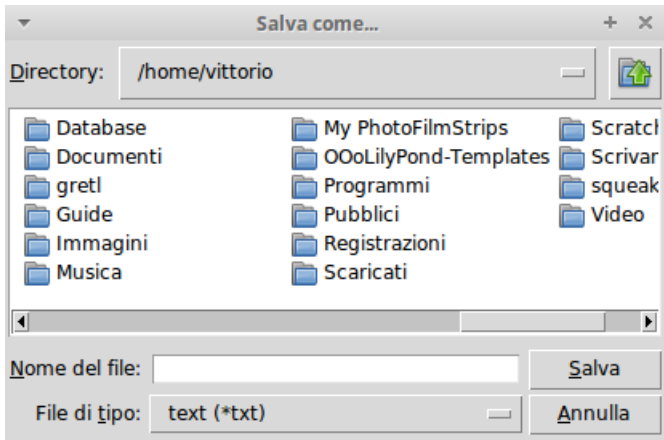
* l'inserimento di un titolo per la finestra di dialogo, che facciamo con
  `title = '...',`
* l'inserimento del tipo e dell'estensione del file, che facciamo con
  `filetypes =` seguito da una lista di tuple di due elementi,
  il primo ad indicare il tipo del file, il secondo ad indicare l'estensione.
  La sintassi completa è

```
filetypes = [(<tipo_file>, <estensione>),
             (<tipo_file>, <estensione>), ...].
```

Esempio:
Con l'istruzione

```
p = fd.asksaveasfilename(title = 'Salva come...', filetypes = [('text', '*txt'),
                         ('python', '*.py')])
```

apriamo questa finestra



Se nella finestrella NOME DEL FILE inseriamo, per esempio, `script_prova` e, aprendo la finestra FILE DI TIPO, cliccando sul rettangolino sulla destra, scegliamo python (*.py), dopo aver chiuso la finestra cliccando su SALVA nella nostra variabile p avremo il path `/home/vittorio/script_prova.py`.

3.1.6 Altri widget

In questo capitolo dedicato ai widget di Tkinter ho presentato quelli più importanti e di più frequente utilizzo. Ce ne sono altri, che possono risultare utili ma non necessari o dei quali si può avere necessità in casi

molto particolari; per il loro uso rimando alla più volte richiamata guida di Shipman.

3.2 La GUI (Graphical User Interface)

Per generare la GUI dobbiamo innanzi tutto importare Tkinter. La più bella istruzione per farlo è

```
from tkinter import *
```

con la quale diciamo a Python di importare tutti i componenti di Tkinter (ricordo, comunque, che ciò non basta per avere a disposizione il modulo filedialog, che, se serve, va importato a parte).

Dobbiamo quindi creare l'oggetto genitore della GUI, che viene generalmente chiamato radice, con l'istruzione

```
r = Tk()
```

dove r sta per un qualsiasi identificatore (c'é chi preferisce root o radice) che richiami in qualche modo il fatto che questo oggetto è la radice, il genitore di tutto.

Questo oggetto altro non è che una finestra; è ovviamente un contenitore ed è mettendovi dentro gli widget che costruiamo la GUI.

E' dotato, tra gli altri, di due importanti metodi:

```
title('...')
```

che serve per far apparire la stringa che inseriamo tra apici nella cornice in alto della finestra e diventerà il titolo dell'applicazione;

```
config(<opzione> = <valore>, ...  )
```

con cui possiamo configurare a nostro piacimento la finestra. Le opzioni sono le stesse che abbiamo a disposizione per il widget Frame e le tre fondamentali sono:

* la larghezza della finestra, esprimibile con

  ```
  width = <numero_pixel>
  ```

* l'altezza della finestra, esprimibile con

  ```
  height = <numero_pixel>
  ```

* il colore di fondo della finestra, esprimibile con

  ```
  bg = <colore>.
  ```

Ci si potrà chiedere come mai, avendo già a disposizione il contenitore radice, è stato previsto anche il widget contenitore Frame. La risposta è semplice: nel contenitore radice possiamo inserire più Frame e più Canvas, di colore diverso, di geometria diversa e la nostra creatività ha così a disposizione più strumenti.

Il contenitore radice è una top-level window ed è solo lì, come dicevo a suo tempo, che possiamo inserire una barra del menu.

Come top-level window è anche quella deputata a mostrare tutta l'applicazione e per ottenere questo, alla fine dello script dell'applicazione stessa, dobbiamo richiamarne il metodo `mainloop()`.

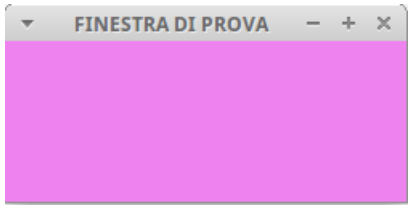
Nella cornice alta della finestra compaiono pure i soliti simboli per minimizzarla, ingrandirla e chiuderla.

Esempio:

Il seguente script

```
#!  /usr/bin/python3
from tkinter import *
r = Tk()
r.title('FINESTRA DI PROVA')
r.config(width = 250, height = 100, bg = 'violet')
r.mainloop()
```

crea questa finestra con colore di fondo violetto



che possiamo chiudere cliccando sulla X in alto a destra.

3.3 La geometria della GUI

Ora viene la parte più delicata e laboriosa della creazione dell'interfaccia grafica perché occorre decidere quali widget ci servono, dove dobbiamo collocarli, sia in senso funzionale sia in senso estetico, come dimensionare e armonizzare il tutto, in modo che la nostra GUI risulti facile da utilizzare e bella da vedere.

Per fare questo, al servizio della nostra creatività e della nostra progettualità (cose che dobbiamo avere noi), tkinter ci offre tre strumenti per collocare i widget nei contenitori, strumenti che corrispondono ad altrettanti metodi di cui è dotato ciascun oggetto widget:

* il metodo `.pack()`,
* il metodo `.grid()`,
* il metodo `.place()`.

Una volta creato il widget, per poterlo vedere nel suo contenitore dobbiamo richiamare uno di questi suoi metodi con la sintassi `<identificatore_widget>.<metodo>`.

Bene chiarire subito che è raccomandabile, per più widget nello stesso contenitore, utilizzare lo stesso metodo; in caso contrario si potrebbero creare conflitti che rendono ingovernabile il rendering di tutta la GUI.

Se per una zona della GUI ci torna comoda la geometria generata dal metodo pack e per un'altra zona ci torna comoda la geometria generata dal metodo grid, creiamo due frame separati, uno per ciascuna geometria.

Geometria con il metodo pack

Con il metodo pack i widget vengono inseriti impacchettati uno via l'altro, nell'impostazione di default dall'alto in basso: il primo inserito sta in alto, il secondo sta sotto di lui, ecc. Possiamo modificare questa impostazione con l'opzione `side =` che accetta i valori precostituiti `LEFT`, `RIGHT`, `BOTTOM` e `TOP` (quello di default).

Per comprendere l'effetto di queste opzioni occorre considerare che, nella geometria pack, il contenitore è elastico e, quando vi si inserisce il primo widget, esso si restringe attorno ad esso. Ciò non toglie che la cavità in cui inserire gli altri widget, anche se non si vede più, rimanga a disposizione. Nell'impostazione di default la cavità disponibile si sviluppa sotto il widget inserito, in quanto questo è nella posizione `TOP`, per cui il successivo inserimento si collocherà sotto il primo widget. Se avessimo inserito il primo widget con l'opzione `side = BOTTOM`, la cavità libera si svilupperebbe verso l'alto, sopra il primo widget inserito, e il successivo si collocherebbe sopra. In entrambi i casi la cavità occupata si sviluppa a sinistra e a destra sotto o sopra del widget nel contenitore e lascia libera tutta la fascia orizzontale sottostante o sovrastante in cui poter lavorare scegliendo sinistra o destra.

Purtroppo quando scegliamo l'opzione `side = LEFT` o `side = RIGHT`, la cavità libera rimane, rispettivamente, a destra o a sinistra del widget e, se dopo aver inserito, per esempio, un widget con l'opzione `side = LEFT`, ne inseriamo uno con l'opzione `side = TOP` ce lo ritroveremo sì al

top ma nella parte destra della finestra e mai più, con la geometria pack nello stesso contenitore, potremo inserirlo al top e al centro.

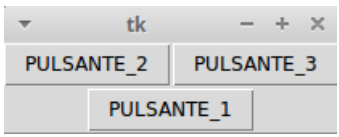
Con questa geometria siamo pertanto costretti a ricorrere spesso a sdoppiamenti dei contenitori.

Esempio:

Il seguente script

```
from tkinter import *
r = Tk()
p1 = Button(r, text = 'PULSANTE_1')
p1.pack(side = BOTTOM)
p2 = Button(r, text = 'PULSANTE_2')
p2.pack(side = LEFT)
p3 = Button(r, text = 'PULSANTE_3')
p3.pack(side = LEFT)
r.mainloop()
```

crea questa finestra



Se volessimo realizzare una finestra così concepita

dovremmo ricorrere a questo script

```
from tkinter import *
r = Tk()
p1 = Button(r, text = 'PULSANTE_1')
p1.pack(side = BOTTOM)
p4 = Button(r, text = 'PULSANTE_4')
p4.pack(side = TOP)
p2 = Button(r, text = 'PULSANTE_2')
p2.pack(side = LEFT)
p3 = Button(r, text = 'PULSANTE_3')
p3.pack(side = LEFT)
r.mainloop()
```

nel quale vediamo come, per tenere al centro il quarto pulsante, dobbiamo impacchettarlo nella cavità prima che questa sia affettata dall'opzione side = LEFT degli altri pulsanti.

Altra alternativa per ottenere lo stesso risultato sarebbe quella di creare un altro contenitore, ad esempio così:

```
from tkinter import *
r = Tk()
zona_alta = Frame()
zona_alta.pack()
p4 = Button(zona_alta, text = 'PULSANTE_4')
p4.pack()
p1 = Button(r, text = 'PULSANTE_1')
p1.pack(side = BOTTOM)
p2 = Button(r, text = 'PULSANTE_2')
p2.pack(side = LEFT)
p3 = Button(r, text = 'PULSANTE_3')
p3.pack(side = LEFT)
r.mainloop()
```

Come si vede, la geometria con il metodo pack non è così semplice da realizzare, soprattutto se pensiamo a GUI che abbiano una certa ricchezza di alternative e l'aspirazione di essere belle da vedere.

A quest'ultimo proposito si sappia che al fitto impacchettamento dei widget, uno a ridosso dell'altro, si può rimediare con le opzioni `padx = <numero_pixel>` e `pady = <numero_pixel>` del metodo `pack()` con le quali otteniamo che il widget venga inserito con attorno un margine, rispettivamente in orizzontale e in verticale, della dimensione in pixel indicata, margine che eredita il colore di sfondo del contenitore. In questo modo possiamo almeno distribuire le cose con un certo senso estetico.

Geometria con il metodo grid

Anche nella geometria grid il contenitore è elastico e si restringe attorno ai widget man mano li inseriamo. Il grande vantaggio del metodo sta nel fatto che la cavità nella quale inseriamo i widget è idealmente divisibile in righe e colonne, a mo' di griglia (grid, appunto), numerate da 0 in su con l'origine in alto a sinistra.

Possiamo scegliere in quale cella della griglia collocare un widget con le opzioni `column=<numero_intero>` e `row=<numero_intero>` che identificano le coordinate della cella in termini di colonna e riga.

Se vogliamo che la cella si estenda su più colonne usiamo l'opzione
`columnspan = <numero_colonne>`
e se vogliamo che si estenda su più righe usiamo l'opzione
`rowspan = <numero_righe>`.

Esempio:

Per realizzare, con il metodo grid, una finestra simile alla seconda contemplata nell'esempio contenuto nel paragrafo precedente lo script è il seguente:

```
from tkinter import *
r = Tk()
p4 = Button(r, text = 'PULSANTE_4')
p4.grid(row=0, column=0, columnspan=2)
p2 = Button(r, text = 'PULSANTE_2')
p2.grid(row=1, column=0)
p3 = Button(r, text = 'PULSANTE_3')
p3.grid(row=1, column=1)
p1 = Button(r, text = 'PULSANTE_1')
p1.grid(row=2, column=0, columnspan=2)
r.mainloop()
```

Anche con il metodo grid abbiamo a disposizione le opzioni
`padx = <numero_pixel>` e `pady = <numero_pixel>`
per costituire attorno ai widget che inseriamo delle cornici distanziatrici.

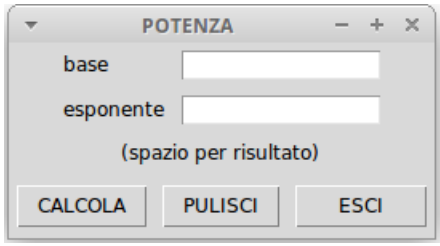
Infine, dal momento che le celle di ogni riga e di ogni colonna si dimensionano prendendo la dimensione della più alta o della più larga, può accadere spesso che un widget inserito in una cella non la occupi tutta. Per default, in questi casi, il widget si centra orizzontalmente e verticalmente nella cella e ciò può creare disallineamenti non graditi. Tutto si può regolare con l'opzione
`sticky =`
che accetta i valori `W` e `E` per allineare orizzontalmente il contenuto della cella, rispettivamente, a sinistra e a destra, lasciandolo centrato in senso verticale, `N` e `S` per allineare verticalmente il contenuto della cella, rispettivamente, in alto e in basso, lasciandolo centrato in senso verticale, con le possibili combinazioni `NW`, `NE`, `SW` e `SE` si allineano i contenuti negli angoli, rispettivamente, in alto a sinistra, in alto a destra, in basso a sinistra e in basso a destra.

Esempio:
Questo script

```
from tkinter import *
r = Tk()
r.title('POTENZA')
zona_alta = Frame(r)
zona_alta.pack()
l1 = Label(zona_alta, text = 'base')
l1.grid(column = 0, row = 0, sticky = W, padx = 4)
e1 = Entry(zona_alta, width = 15)
e1.grid(column = 1, row = 0, padx = 4, pady = 4)
l2 = Label(zona_alta, text = 'esponente')
l2.grid(column = 0, row = 1, sticky = W, padx = 4)
e2 = Entry(zona_alta, width = 15)
e2.grid(column = 1, row = 1, padx = 4, pady = 4)
l3 = Label(zona_alta, text = '(spazio per risultato)')
l3.grid(row = 2, columnspan = 2, pady = 4)
```

```
zona_bassa = Frame(r)
zona_bassa.pack()
b1 = Button(zona_bassa, text = 'CALCOLA')
b1.grid(column = 0, row = 0, padx = 4, pady = 6)
b2 = Button(zona_bassa, text = 'PULISCI')
b2.grid(column = 1, row = 0, padx = 4)
b3 = Button(zona_bassa, text = 'ESCI', width = 7)
b3.grid(column = 2, row = 0, padx = 4)
r.mainloop()
```

crea questa finestra



Si tratta di una GUI destinata ad un'applicazione che calcola la potenza ennesima di un numero.

Richiamo l'attenzione su come la GUI sia stata costruita con due frame, uno per la zona alta e uno per la zona bassa: ho ritenuto utile agire così in quanto la zona alta è utilizzata e stabilizzata su due colonne e, per poter collocare armonicamente i tre pulsanti mi serviva una zona di tre colonne.

Si nota inoltre come la collocazione dei due frame sia stata fatta con il metodo pack e, all'interno dei frame, si sia agito con il metodo grid: a seconda dei casi si usa ciò che si ritiene più funzionale. Purché si evitino conflitti. Se, per esempio, la zona alta della GUI fosse stata disegnata nel contenitore r con il metodo grid invece di creare il frame per la zona alta, non avremmo più potuto impacchettare il frame della zona bassa con il metodo pack in quanto il contenitore r sarebbe già stato strutturato con il metodo grid.

Geometria con il metodo place

La prima grande differenza tra questo metodo e i due che abbiamo visto prima sta nel fatto che il contenitore non si restringe attorno ai widget man mano che li inseriamo ma rimane tutto sempre disteso secondo la dimensione di default o quella che gli abbiamo assegnato.

Ogni widget si inserisce determinando un punto del contenitore al quale ancorarlo e determinando come ancorarlo.

Il punto si determina indicandone le coordinate in pixel con le opzioni
`x = <pixel>` e `y = <pixel>`,
ricordando che l'origine, dove x è uguale a 0 e y è uguale a 0, sta nell'angolo in alto a sinistra del contenitore. Oppure si determina con le opzioni
`relx = <valore>` e `rely = <valore>`,
dove il valore è un numero decimale compreso tra 0 e 1 che indica una frazione, rispettivamente, dell'ampiezza del contenitore e dell'altezza del contenitore: il valore 0.5 indica il punto centrale dell'asse, 0.25 indica un punto che sta al primo quarto dell'asse, ecc.

L'ancoraggio al punto si determina con l'opzione `anchor = <valore>`, dove il valore può essere `N`, `E`, `S`, `W`, `NE`, `NW`, `SE` e `SW` e indica il lato o l'angolo di ancoraggio stesso. Il valore di default è `NW` e sta ad indicare che il widget è ancorato al punto nel suo angolo in alto a sinistra. Se indichiamo `N` il widget viene ancorato al punto con il centro del suo lato superiore, se indichiamo `E` il widget viene ancorato al punto con il centro del suo lato destro, ecc.

Nella geometria con il metodo place non abbiamo a disposizione le opzioni `padx` e `pady` e i distanziamenti tra i widget devono essere creati individuando adeguatamente i punti di ancoraggio.

Secondo alcuni questo è il metodo più semplice dei tre che abbiamo esaminato. Personalmente non condivido questo parere in quanto la difficoltà di individuare esattamente le coordinate dei punti cui ancorare i widget per realizzare una GUI ben fatta mi pare notevole.

Un aiuto potrebbe derivare dalla possibilità di sostituire ai riferimenti in pixel riferimenti in unità di misura più alla nostra portata, come centimetri, millimetri o pollici: per farlo, mentre i pixel si indicano semplicemente con un numero, occorre inserire tra apici il numero immediatamente seguito dalla lettera `c` per indicare centimetri, dalla lettera `m` per indicare millimetri o dalla lettera `i` per indicare pollici: per esempio, il valore di 22 millimetri si indica con '22m'. In questo modo diventa forse più agevole lavorare con le coordinate dei punti.

Rimane il grosso problema dato dal fatto che, per collocazioni dei widget con giusti distanziamenti, occorrerebbe sempre conoscere esattamente gli spazi occupati dai widget che via via si inseriscono nella GUI. Questo problema può essere superato dall'esistenza di due metodi, comuni a qualsiasi widget, che ne ritornano le dimensioni in pixel
`<identificatore_widget>.winfo_width()` ritorna la larghezza,
`<identificatore_widget>.winfo_height()` ritorna l'altezza.

Attraverso questi due metodi possiamo delegare allo script il compito di misurare gli spazi occupati dai widget via via inseriti in modo da sapere dove sia possibile inserire gli altri. C'è però un problema: per poter misurare l'ingombro di un oggetto occorre prima crearlo e, in uno script Tkinter, la creazione degli oggetti avviene solo con l'istruzione di mainloop finale. Il problema si supera inserendo nello script, appena dopo il costruttore del widget e appena prima del metodo per misurarne la larghezza o l'altezza il metodo
`<identificatore_contenitore>.update()`.

Come si vede, i creatori di Tkinter le hanno pensate proprio tutte.

Esempi:

Per realizzare, con il metodo place, una finestra simile alla seconda contemplata nell'esempio contenuto nel paragrafo dedicato al metodo pack lo script è il seguente:

```
from tkinter import *
r = Tk()
r.config(width = 200, height = 100)
b4 = Button(r, text = 'PULSANTE_4')
b4.place(relx = 0.5, anchor = N)
b2 = Button(r, text = 'PULSANTE_2')
b2.place(relx = 0.25, anchor = N, rely = 0.25)
b3 = Button(r, text = 'PULSANTE_3')
b3.place(relx = 0.75, anchor = N, rely = 0.25)
b1 = Button(r, text = 'PULSANTE_1')
b1.place(relx = 0.5, anchor = N, rely = 0.5)
r.mainloop()
```

Come si vede il tutto è basato su posizionamenti indicati in termini relativi.

Per realizzare, con il metodo place, una finestra simile a quella realizzata prima con il metodo grid potremmo utilizzare lo script seguente:

```
from tkinter import *
r = Tk()
r.title('POTENZA')
r.config(width = 250, height = 110)
l1 = Label(r, text = 'base')
l1.place(x = 15, y = 2)
e1 = Entry(r, width = 15)
e1.place(x = 115, y = 2)
l2 = Label(r, text = 'esponente')
l2.place(x = 15, y = 26)
e2 = Entry(r, width = 15)
e2.place(x = 115, y = 26)
l3 = Label(r, text = '(spazio per risultato')
l3.place(x = 15, y = 50)
b1 = Button(r, text = 'CALCOLA')
b1.place(x = 2, y = 74)
b2 = Button(r, text = 'PULISCI')
b2.place(x = 87, y = 74)
```

```
b3 = Button(r, text = 'ESCI', width = 7)
b3.place(x = 165, y = 74)
r.mainloop()
```
In questo caso ho posizionato i widget utilizzando esclusivamente riferimenti assoluti. L'unico modo per riuscire ad azzeccare questi riferimenti è di comporre lo script verificando via via quello che succede lanciando nell'IDLE la parte già scritta e procedere alla rettifiche e alle determinazioni del caso.

Quest'altro è un esempio di script nel quale ho delegato allo script stesso la determinazione degli ingombri.

```
from tkinter import *
r = Tk()
r.title('POTENZA')
r.config(width = 250, height = 110)
l1 = Label(r, text = 'base')
l1.place(x = 15, y = 2)
e1 = Entry(r, width = 15)
e1.place(x = 115, y = 2)
r.update()
ingombro_verticale = 2 + e1.winfo_height()
l2 = Label(r, text = 'esponente')
l2.place(x = 15, y = 4 + ingombro_verticale)
e2 = Entry(r, width = 15)
e2.place(x = 115, y = 4 + ingombro_verticale)
r.update()
ingombro_verticale = ingombro_verticale + e2.winfo_height()
l3 = Label(r, text = '(spazio per risultato)')
l3.place(x = 15, y = 7 + ingombro_verticale)
r.update()
ingombro_verticale = ingombro_verticale + l3.winfo_height()
b1 = Button(r, text = 'CALCOLA')
b1.place(x = 2, y = 10 + ingombro_verticale)
r.update()
ingombro_orizzontale = 2 + b1.winfo_width()
b2 = Button(r, text = 'PULISCI')
b2.place(x = 2 + ingombro_orizzontale, y = 10 + ingombro_verticale)
r.update()
ingombro_orizzontale = ingombro_orizzontale + b2.winfo_width()
b3 = Button(r, text = 'ESCI', width = 7)
b3.place(x = 2 + ingombro_orizzontale, y = 10 + ingombro_verticale)
r.mainloop()
```

I tre esempi presentati utilizzano ciascuno un diverso sistema di indicazione del piazzamento. Ciò non toglie che, nella realtà, si possa agire utilizzando l'un sistema o l'altro alternativamente nello stesso script, a seconda della convenienza.

3.4 Gli abbellimenti ttk

Un difetto che secondo alcuni affligge Tkinter è la vetustà che caratterizza il look delle GUI e lo scarso adattamento che queste GUI riescono a realizzare con lo stile dell'ambiente in cui sono inserite. In poche parole, un'applicazione progettata con Tkinter su un sistema Linux utilizzata, grazie alla portabilità di Python, su Windows 10 fa forse dire a chi la vede «ma da dove viene 'sta roba?».

Siccome questo difetto non ce l'hanno altri pacchetti grafici come Qt o wxWidget, i curatori di Tkinter si sono dati da fare per rimediare ed hanno sviluppato il modulo ttk, un modulo che fa parte di tkinter ma va caricato a parte.

A differenza di quanto avviene in Python 2[1] in Python 3, per utilizzare ttk, occorrono queste due istruzioni

```
from tkinter import *
from tkinter import ttk
```

ed ogniqualvolta si intende utilizzare un widget ttk occorre richiamarne il costruttore con l'istruzione

```
ttk.<widget>
```

in modo che, nella stessa GUI sia possibile usare widget di entrambi i tipi.

Tutti i widget che abbiamo visto nei capitoli precedenti, ad eccezione del widget Text, hanno un loro corrispondente in ttk.

I widget ttk hanno probabilmente un look più moderno ma sono molto meno personalizzabili, almeno in via diretta, rispetto a quelli normali in quanto sono concepiti in maniera del tutto particolare.

L'aspetto di un widget ttk è determinato da due componenti: il tema e lo stile. Il tema riguarda il disegno del widget e lo stile riguarda le altre componenti del suo aspetto (colori, font, ecc.). Tema e stile, una volta determinati, per default si estendono a tutti i widget, di qualsiasi natura, utilizzati nel progetto. Essi sono comunque personalizzabili e se ne possono creare di diversi con altro nome.

La madre di tutto è la classe Style del modulo ttk, di cui possiamo creare un'istanza `s` con l'istruzione

```
s = ttk.Style().
```

Da qui, con l'istruzione

[1] In Python 2 l'importazione avviene, dopo l'istruzione from Tkinter import *, con l'istruzione from ttk import * e questa importazione sovrascrive tutti i widget del normale tkinter con i widget ttk che, a questo punto, diventano quelli di default e sono i soli utilizzabili. Bestialità che fortunatamente è stata eliminata in Python 3.

`s.theme_names()`
possiamo vedere che i temi presenti nel modulo sono quattro: `clam`, `alt`, `default` e `classic`. Quest'ultimo molto simile al tema utilizzato nei widget normali.
Con l'istruzione `s.theme_use()` vediamo qual'è il nome del tema in uso; con la stessa istruzione inserendo tra le parentesi e tra apici il nome del tema possiamo cambiarlo e tutti i widget si uniformeranno a questo tema. Con un po' di prove nell'IDLE possiamo rapidamente avere un'idea di come si presentano i widget nei vari temi.

Se prescindiamo da tutte le opzioni di formattazione (colori, font, ecc.) possiamo costruire i widget ttk con le stesse opzioni che abbiamo visto per i widget normali.

Le cose si complicano con le formattazioni, che si fanno attraverso gli stili.

Il nome dello stile di un widget, per default, coincide con quello del widget preceduto da una T maiuscola: lo stile di default per il widget Button è TButton, ecc.

Per passare allo stile opzioni di formattazione dobbiamo creare un'istanza della classe Style con
`s = ttk.Style()`
e invocarne il metodo `configure`, passandogli innanzi tutto, tra apici, il parametro costituito dal nome dello stile da configurare e poi le opzioni di configurazione.
Per esempio, per dare un colore a un frame
`s.configure('TFrame', background = <colore>)`.
Da questo momento tutti i frame che costruiremo avranno lo stesso colore e, per costruirne altri con altro colore, dobbiamo creare altri stili, con altri colori, e attribuire questi stili ai nostri frame per sostituire quello di default.
Esempio:
Costruiamo un frame dal modulo ttk con

```
from tkinter import *
from tkinter import ttk
r = Tk()
f1 = ttk.Frame(r, width = 250, height = 300)
f.pack()
```

dal momento che il costruttore del frame non accetta l'opzione per il colore di fondo, per avere questo colore del frame dobbiamo proseguire con

```
s = ttk.Style()
s.configure('TFrame', background = 'green')
```

se ora costruiamo un altro frame, sempre dal modulo ttk

```
f2 = ttk.Frame(r, width = 250, height = 100)
f2.pack()
```

avrà anche lui il colore di fondo verde e, se lo vogliamo giallo, dobbiamo costruire un altro stile da frame con il background giallo
`s.configure('altro.TFrame', background = 'yellow')`
e attribuire questo stile all'altro frame
`f2.configure(style = 'altroTFrame')`

Penso che questo esempio sia abbastanza dimostrativo di quanto sia laborioso utilizzare il modulo ttk e come, tutto sommato e per dilettanti come noi, sia forse meglio tenerci il modulo base rinunciando ad abbellimenti che non è poi detto siano così riscontrabili.

3.5 Alcuni esempi

Per dimostrare come si integra la costruzione della GUI con il resto del linguaggio Python propongo alcuni esempi.

Calcolo di potenze e radici

Il seguente script serve per calcolare potenze e radici utilizzando la GUI che avevamo costruito a suo tempo con il metodo grid.

```
#!  /usr/bin/python3
from tkinter import *
def calcola():
    base = float(e1.get())
    esponente = e2.get()
    if '/' in esponente:
        def cerca():
            i = 0
            while i < len(esponente):
                if esponente[i] == '/':
                    return i
                i = i+1
        indice = cerca()
        r = esponente[indice+1:]
        esponente = 1/int(r)
        risultato = base ** esponente
        l3.config(text = risultato)
    else:
        esponente = float(e2.get())
        risultato = base ** esponente
        l3.config(text = risultato)
def pulisci():
    e1.delete(0, END)
    e2.delete(0, END)
    l3.config(text = '')
    e1.focus_set()
r = Tk()
r.title('POTENZA')
zona_alta = Frame(r)
zona_alta.pack()
l1 = Label(zona_alta, text = 'base')
```

```
l1.grid(column = 0, row = 0, sticky = W, padx = 4)
e1 = Entry(zona_alta, width = 15, justify = RIGHT)
e1.grid(column = 1, row = 0, padx = 4, pady = 4)
e1.focus_set()
l2 = Label(zona_alta, text = 'esponente')
l2.grid(column = 0, row = 1, sticky = W, padx = 4)
e2 = Entry(zona_alta, width = 15, justify = RIGHT)
e2.grid(column = 1, row = 1, padx = 4, pady = 4)
l3 = Label(zona_alta)
l3.grid(row = 2, columnspan = 2, pady = 4)
zona_bassa = Frame(r)
zona_bassa.pack()
b1 = Button(zona_bassa, text = 'CALCOLA', command = calcola)
b1.grid(column = 0, row = 0, padx = 4, pady = 6)
b2 = Button(zona_bassa, text = 'PULISCI', command = pulisci)
b2.grid(column = 1, row = 0, padx = 4)
b3 = Button(zona_bassa, text = 'ESCI', width = 7, command = quit)
b3.grid(column = 2, row = 0, padx = 4)
r.mainloop()
```

In carattere diritto il codice tkinter per il disegno della GUI e in carattere corsivo il codice Python.

La funzione calcola() è costruita in modo che si possa accettare l'inserimento di un esponente frazionario, così da poter calcolare, oltre che potenze, anche radici (ricordo che $\sqrt[n]{x} = x^{\frac{1}{n}}$).

Calcolo del fattoriale di un numero

Questo script contempla un'applicazione che calcola il fattoriale di un numero.

```
#!  /usr/bin/python3
import math
def calcola():
    numero = zonaInput.get()
    n = int(numero)
    risultato.configure(text = math.factorial(n))
def pulisci():
    zonaInput.delete(0, END)
    risultato.configure(text = '')
from tkinter import *
r = Tk()
r.title('FATTORIALE')
r.config(bg = 'lightgreen')
descrizioneInput = Label(r, text = "Inserisci il numero di cui
                   vuoi calcolare il fattoriale", bg = 'lightgreen')
descrizioneInput.pack(padx = 5)
zonaInput = Entry(r, justify = CENTER)
zonaInput.focus_set()
zonaInput.pack(pady = 5)
zonaPulsanti = Frame(r, bg = 'lightgreen')
zonaPulsanti.pack()
pulsanteCalcola = Button(zonaPulsanti, text = 'CALCOLA', bg = 'green',
                   command = calcola)
```

```
pulsanteCalcola.grid(column = 0, row = 0, padx = 5)
pulsantePulisci = Button(zonaPulsanti, text = 'PULISCI', bg = 'green',
                    command = pulisci)
pulsantePulisci.grid(column = 1, row = 0, padx = 5)
pulsanteEsci = Button(zonaPulsanti, text = 'ESCI', width = 7,
                    bg = 'green', command = quit)
pulsanteEsci.grid(column = 2, row = 0, padx = 5)
zonaOutput = Frame(r, bg = 'lightgreen')
zonaOutput.pack()
descrizioneOutput = Label(zonaOutput, text = "Fattoriale
                    del numero inserito:", bg = 'lightgreen')
descrizioneOutput.pack(pady = 5)
risultato = Label(zonaOutput, wraplength = 600, justify = LEFT,
                    bg = 'lightgreen')
risultato.pack(pady = 5)
r.mainloop()
```

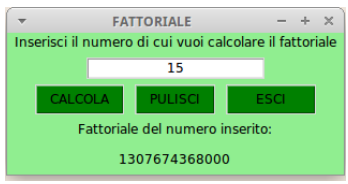
Anche qui abbiamo in cattere diritto il codice dedicato alla costruzione della GUI e in carattere corsivo il codice Python.

Per calcolare il fattoriale di un numero non è che ci sia un gran bisogno di una GUI, ma ho voluto produrre questo esempio soprattutto per dimostrare l'elasticità dei widget Tkinter.

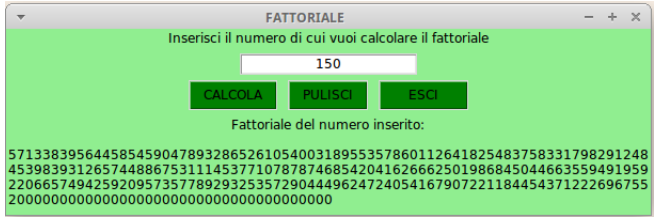
Nell'esempio precedente abbiamo trattato con numeri a virgola mobile (float) che Python scrive per default in notazione scientifica, mai occupando più di 22 caratteri. Le dimensioni della nostra GUI erano pertanto più che sufficienti per evidenziare il risultato del calcolo.

In questo esempio, invece, trattiamo numeri interi e Python, quando scrive numeri interi, per default li scrive tali e quali, in tutta la loro estensione; estensione che nel linguaggio Python, che non conosce stack overflow, può assumere dimensioni ragguardevoli.

Così, se calcoliamo il fattoriale di 15, la nostra GUI ce lo mostra nella normale finestra



Se calcoliamo il fattoriale di 150, che è un numero ben maggiore, la finestra diventa



Questa finestra allarga la label destinata a contenere il risultato fino al limite di 600 pixel indicato nell'opzione wraplength e la allunga fino a contenere l'enorme numero andando accapo.

Un piccolo editor di testo

Un'applicazione per la quale torna utile una GUI è l'editor di testo.

Possiamo costruirne uno molto semplice con questo script

```
#!  /usr/bin/python3
from tkinter import *
import tkinter.filedialog as fd
r = Tk()
r.title('Senza Titolo')
t = Text(r, wrap = WORD)
t.pack(padx = 5, pady = 5)
def apri():
    path = fd.askopenfilename(title = 'Scegli un file')
    if len(path) > 0:
        t.delete('1.0', 'end')
        with open(path, 'U') as f:
            t.insert('1.0', f.read())
        r.title(path)
def salva_come():
    path = fd.asksaveasfilename(title = 'Dove Salvare')
    if len(path) > 0:
        with open(path, 'w') as f:
            f.write(t.get('1.0', 'end'))
        r.title(path)
mb = Menu(r)
r.config(menu=mb)
fm = Menu(mb)
fm.add_command(label = 'Apri...', command = apri)
fm.add_command(label = 'Salva come...', command = salva_come)
fm.add_separator()
fm.add_command(label = 'Esci', command = quit)
mb.add_cascade(label = 'File', menu = fm)
r.mainloop()
```

Al solito abbiamo con carattere corsivo la parte di codice Python e con carattere diritto quella Tkinter.

E nella pagina seguente vediamo il nostro editor

Master Mind

Dove la GUI assume un'importanza fondamentale è nei giochi. Per questa esigenza il mondo Python, con l'insieme di moduli PyGame, ci offre di fare ben altro di ciò che possiamo fare con Tkinter. Per alcuni giochi da tavolino dove non vi sia l'esigenza di movimento può tuttavia tornare utile anche Tkinter.

Questo script contempla un'edizione semplificata, costruita utilizzando rigorosamente solo ciò che abbiamo visto in questo libro, del gioco Master Mind, con metodi grid e pack che si combinano, con frame e canvas che si alternano.

```
#!  /usr/bin/python3
from random import *
def inizializza():
    areaTentativi.delete('pallina')
    areaRisposte.delete('quadratino')
    combinazione.delete('pallina')
    messaggio.configure(text = '')
    global listaGioco
    global listaTentativo
    global tentativi
    tentativi= 0
    global x1
    x1 = 28
    global x2
    x2 = 28
    global y1
    y1 = 10
    global y2
```

```
    y2 = 12
    listaBase = ['R', 'V', 'G', 'B']
    listaGioco = [1,2,3,4]
    for i in range(4):
        p = int(random() * 4)
        listaGioco[i] = listaBase[p]
def tentativo(event):
    nomiColori = {'R':'RED', 'V':'GREEN', 'G':'YELLOW', 'B':'BLUE'}
    stringaColori = 'RVGB'
    stringaTentativo = stringaInput.get()
    verifica = ''
    if len(stringaTentativo) != 4:
        messaggio.config(text = 'DEVI INSERIRE ALMENO 4 CARATTERI')
        verifica = 'NO'
    for i in range(4):
        if stringaTentativo[i] not in stringaColori:
            verifica = 'NO'
            break
    if verifica == 'NO':
        messaggio.config(text = 'HAI INSERITO MALE QUALCOSA')
    else:
        messaggio.config('text' = '')
        global listaTentativo
        listaTentativo = [1,2,3,4]
        global x1
        global y1
        global tentativi
        for i in range(4):
            listaTentativo[i] = stringaTentativo[i]
        for i in range(4):
            areaTentativi.create_oval(x1, y1, x1+15, y1+15,
                                      fill = nomiColori[listaTentativo[i]])
            x1 += 20
            areaTentativi.addtag_all('pallina')
        stringaInput.delete(0,4)
        x1 = 28
        y1 += 20
        tentativi += 1
        xx = 28
        yy = 5
    if listaTentativo == listaGioco:
        messaggio.config(text = 'MOLTO BENE!\nHAI VINTO CON %d TENTATIVI\n
                                      LA COMBINAZIONE ERA:' % (tentativi))
        for i in range(4):
            combinazione.create_oval(xx, yy, xx+15, yy+15,
                                      fill = nomiColori[listaGioco[i]])
            xx += 20
            combinazione.addtag_all('pallina')
    elif tentativi == 9:
        messaggio.config(text = 'HAI PERSO!\nLA COMBINAZIONE ERA:')
        for i in range(4):
            combinazione.create_oval(xx, yy, xx+15, yy+15,
                                      fill = nomiColori[listaGioco[i]])
            xx += 20
            combinazione.addtag_all('pallina')
    elif verifica != 'NO':
        risposta()
```

```
def risposta():
    risposta = []
    for i in range(4):
        if listaTentativo[i] == listaGioco[i]:
            risposta.append('BLACK')
        elif listaTentativo[i] != listaGioco[i] and
                                            listaTentativo[i] in listaGioco:
            risposta.append('WHITE')
    risposta.sort()
    global x2
    global y2
    for i in range(len(risposta)):
        areaRisposte.create_rectangle(x2, y2, x2+11, y2+11, fill = risposta[i])
        x2 += 20
        areaRisposte.addtag_all('quadratino')
    x2 = 28
    y2 += 20
from tkinter import *
pianoDiGioco = Tk()
zonaApertura = Frame(pianoDiGioco)
zonaApertura.grid()
descrizione = Label(zonaApertura)
descrizione.configure(text = 'Colori in gioco:  ', foreground = 'blue')
descrizione.pack()
strisciaColori = Canvas(zonaApertura, width = 150, height = 20)
strisciaColori.pack()
strisciaColori.create_text(30, 10, text = 'R', font = ('Helvetica','18','bold'),
                   fill = 'red')
strisciaColori.create_text(60, 10, text = 'V', font = ('Helvetica','18','bold'),
                   fill = 'green')
strisciaColori.create_text(90, 10, text = 'G', font = ('Helvetica','18','bold'),
                   fill = 'yellow')
strisciaColori.create_text(120, 10, text = 'B', font = ('Helvetica','18','bold'),
                   fill = 'blue')
regole = Label(zonaApertura, text = '9 tentativi a disposizione\nquadratino nero:
                   colore giusto posto giusto\nquadratino bianco:  colore giusto
                   posto sbagliato', fg = 'blue') regole.pack(pady = 5)
zonaGioco = Frame(pianoDiGioco)
zonaGioco.grid()
areaTentativi = Canvas(zonaGioco)
areaTentativi.configure(width = 130, height = 200)
areaTentativi.grid(row = 0, column = 0, sticky = N)
areaRisposte = Canvas(zonaGioco)
areaRisposte.configure(width = 130, height = 200)
areaRisposte.grid(row = 0, column = 1, sticky = N)
zonaInput = Frame(pianoDiGioco)
zonaInput.grid()
descrizione_1 = Label(zonaInput)
descrizione_1.configure(text='Scegli quattro colori indicando',foreground='BLUE')
descrizione_1.grid(sticky = N)
descrizione_2 = Label(zonaInput)
descrizione_2.configure(text = 'la loro iniziale maiuscola.',foreground='BLUE')
descrizione_2.grid(sticky = N)
descrizione_3 = Label(zonaInput)
descrizione_3.configure(text = 'Poi premi il tasto INVIO.',foreground='BLUE')
descrizione_3.grid(sticky = N)
stringaInput = Entry(zonaInput)
```

```
stringaInput.configure(justify = CENTER, font = ('Helvetica', '16', 'bold'))
stringaInput.focus_set()
stringaInput.bind('<Return>', tentativo)
stringaInput.grid(sticky = N)
zonaMessaggi = Frame(pianoDiGioco)
zonaMessaggi.grid()
messaggio = Label(zonaMessaggi)
messaggio.configure(justify = CENTER, foreground = 'RED')
messaggio.grid(sticky = N)
combinazione = Canvas(zonaMessaggi)
combinazione.configure(width = 130, height = 25)
combinazione.grid(sticky = N)
zonaSceltaFinale = Frame(pianoDiGioco)
zonaSceltaFinale.grid()
pulsanteAltroGioco = Button(zonaSceltaFinale)
pulsanteAltroGioco.configure(text = 'ALTRO GIOCO', padx=1, pady=1,
            background = 'GREEN', width = 13, command = inizializza)
pulsanteAltroGioco.grid(row = 0, column = 0, sticky = N, padx = 4, pady = 4)
pulsanteChiudi = Button(zonaSceltaFinale)
pulsanteChiudi.configure(text = 'CHIUDI', padx = 1, pady = 1,
            background = 'GREEN', width = 8, command = quit)
pulsanteChiudi.grid(row = 0, column = 1, sticky = N, padx = 4, pady = 4)
pianoDiGioco.title('MASTER MIND')
inizializza()
pianoDiGioco.mainloop()
```

Capitolo 4

Sintesi e riconoscimento vocale

Sintesi vocale e riconoscimento vocale sono le tecnologie attraverso le quali possiamo interagire con una macchina utilizzando la voce.

Con la sintesi vocale otteniamo che un computer, ricevendo un input da file, produca un output verso un altoparlante imitando la voce umana. E' ciò che si chiama TTS, che sta per Text To Speech.

Con il riconoscimento vocale otteniamo che un computer, ricevendo un input espresso con voce umana, produca un output consistente in un testo scritto su file, come risultato di una dettatura. E' ciò che si chiama STT, che sta per Speech To Text.

Esistono tre pacchetti Python che ci mettono in grado di fare queste cose con estrema facilità: PyAudio, SpeechRecognition e Pyttsx3.

Se lavoriamo su Linux o Mac prima di tutto occorre installare una libreria che difficilmente è installata di default sul computer, la libreria `portaudio19-dev`: senza di essa non riusciamo ad installare e a far funzionare PyAudio.
Su Linux la installiamo con
`sudo apt install portaudio19-dev`
oppure riccorrendo a Synaptic.
Su Mac la installiamo con
`brew install portaudio`.

Installiamo poi i pacchetti con `pip`, in questo ordine
`pip` (o `pip3`) `install pyaudio`
`pip` (o `pip3`) `install speechrecognition`
`pip` (o `pip3`) `install pyttsx3`

Dobbiamo, infine, installare un sintetizzatore vocale e possiamo utilmente scegliere eSpeak.

All'indirizzo *http://espeak.sourceforge.net/download.html* troviamo i compilati per Windows e Mac e il source per Linux.

Per Linux possiamo più semplicemente installare con

```
sudo apt install espeak
```

o attraverso il gestore di pacchetti Synaptic[1].

4.1 TTS

Il modulo Python che rende possibile il Text To Speech sui tre più diffusi sistemi operativi è Pyttsx3.

All'indirizzo *https://pyttsx3.readthedocs.io/en/latest/engine.html* si trova il manuale in lingua inglese, utile per chi voglia andare oltre le istruzioni basilari di cui parlo qui.

Pyttsx3 usa per default la lingua inglese del Regno Unito e può essere configurato per usare un'altra lingua: tra le tante cito gli identificatori per le principali lingue europee (`italian`, `english`, `german`, `french`, `spanish`). Per l'inglese esiste la variante per gli USA (`english-us`).

Il modulo si importa con

```
import pyttsx3
```

Si costruisce un oggetto parlante, che possiamo chiamare recitante, con

```
recitante = pyttsx3.init()
```

Se scriviamo nella IDLE il nome di questo oggetto seguito da un punto ci viene mostrato l'elenco dei suoi metodi.
Tra questi i più importanti sono:
`say()` che accetta tra le parentesi tonde la stringa da recitare,
`runAndWait()` che dà il via alla recitazione.

E' quanto basta per ottenere la recitazione di una frase in lingua inglese UK.

Se vogliamo utilizzare un'altra lingua, prima di utilizzare il metodo `say()`, abbiamo a disposizione il metodo `setProperty()` che accetta tra le parentesi tonde il nome di una proprietà e il suo valore, separati da virgola.

[1] Chi ha la fortuna di lavorare su Linux può utilizzare eSpeak da terminale utilizzando i comandi descritti all'indirizzo *http://espeak.sourceforge.net/commands.html*. Sempre sul sistema Linux, è possibile installare il componente di collegamento con Python con `sudo apt install python3-espeak`. In tal modo su può usare eSpeak come fosse un modulo Python, importandolo con `from espeak import espeak`.

Nel caso della lingua, la proprietà è la stringa `'voice'` e il valore è una stringa contenente l'identificatore della lingua.
Altra proprietà che possiamo utilmente modificare è la velocità di lettura espressa da un numero intero che indica le parole lette in un minuto. Il valore di default è 200 e va un tantino svelto.
In questo caso la proprietà è la stringa `'rate'` e il valore è un numero intero, per esempio 150 per rallentare un po' la velocità di lettura.

Esempi:

Il seguente script

```
import pyttsx3
recitante = pyttsx3.init()
recitante.say('lieutenant')
recitante.runAndWait()
```

ci fa sentire la strana pronuncia inglese UK della parola lieutenant.

Quest'altro

```
import pyttsx3
recitante = pyttsx3.init()
recitante.setProperty('voice', 'italian')
recitante.say('Ciao, Vittorio!')
recitante.runAndWait()
```

ci fa sentire un saluto in lingua italiana.

Quest'altro ancora ci recita i versi della canzone tedesca Lili Marleen contenuti in un file di testo, con una velocità un tantino inferiore a quella di default

```
import pyttsx3
recitante = pyttsx3.init()
recitante.setProperty('voice', 'german')
recitante.setProperty('rate', 160)
f = open('/home/vittorio/Documenti/lili_marleen.txt', 'r')
recitante.say(f.read())
recitante.runAndWait()
```

Se facciamo un po' di prove ci accorgiamo che la qualità della lettura non è gran che, soprattutto se lavoriamo non con semplici messaggi ma con testi di una certa dimensione e di una qualche espressività.

4.2 STT

Il modulo Python che rende possibile il Speech To Text sui tre più diffusi sistemi operativi è SpeechRecognition.

All'indirizzo *https://pypi.org/project/SpeechRecognition/* si trovano la library reference in lingua inglese e numerosi esempi. A questa documentazione può ricorrere chi voglia andare oltre le istruzioni basilari di cui parlo qui.

Al fine di non reinventare la ruota, SpeechRecognition si avvale di alcuni sistemi di riconoscimento vocale raggiungibili attraverso la rete: i due migliori sono quello di Google e quello di IBM. Qui mostrerò l'utilizzo di quello di Google.

Per questo motivo, perché SpeechRecognition funzioni, occorre utilizzarlo con collegamento internet attivo.

Il modulo si importa utilmente con la seguente formulazione

```
import speech_recognition as sr
```

Si costruisce l'oggetto di riconoscimento, che possiamo chiamare riconoscitore, con

```
riconoscitore = sr.Recognizer()
```

Se scriviamo nella IDLE il nome di questo oggetto seguito da un punto ci viene mostrato l'elenco dei suoi metodi.
Tra questi i più importanti sono:
`listen()` per catturare il segnale da microfono,
`record()` per catturare il segnale da file audio,
`recognize_google()` per convertire il segnale in testo scritto.

Il segnale viene catturato con una istruzione `with` che specifica la provenienza del segnale stesso, con questa sintassi
`with` <origine_ segnale> `as` <sorgente>:

<istruzione_di_cattura>

e il segnale così catturato viene convertito in testo, passando alla funzione `recognize_google()` come primo parametro quanto è stato catturato e come secondo parametro la lingua con la sintassi
`language =` <codice_ISO_lingua>
(rammento i codici per le principali lingue europee: `it`, `en`, `de`, `fr`, `es`).
Esempi:

Il seguente script accetta la dettatura di un testo, conferma quanto il computer ha capito e chiede se deve memorizzare quanto ha capito in un file di testo:

```
import speech_recognition as sr
riconoscitore = sr.Recognizer()
with sr.Microphone() as source:
    audio = riconoscitore.listen(source)
testo = riconoscitore.recognize_google(audio, language = "it")
print('Ho capito:  \n', testo)
scelta = input("Vuoi salvare in un file di testo?  (si/no)")
if scelta == 'si' or scelta == 'SI':
    f = open('/home/vittorio/Documenti/testo.txt', 'w')
    f.write(testo)
    f.close()
    print('Ho salvato nel file testo.txt.')
else:
    print('Non ho salvato il testo.')
print('Ciao')
```

Quest'altro fa le stesse cose ascoltando un file audio:

```
import speech_recognition as sr
riconoscitore = sr.Recognizer()
voce = sr.AudioFile('/home/vittorio/Documenti/testo.wav')
with voce as source:
    audio = riconoscitore.record(source)
testo = riconoscitore.recognize_google(audio, language = "it")
print('Ho capito:  \n', testo)
scelta = input("Vuoi salvare in un file di testo?  (si/no)")
if scelta == 'si' or scelta == 'SI':
    f = open('/home/vittorio/Documenti/testo.txt', 'w')
    f.write(testo)
    f.close()
    print('Ho salvato nel file testo.txt.')
else:
    print('Non ho salvato il testo.')
print('Ciao')
```

* * *

A conclusione del capitolo propongo questo esempio di dialogo tra uomo e computer: il computer chiede il nome all'interlocutore e, uditolo da microfono, recita un saluto personalizzato.

```
import pyttsx3
import speech_recognition as sr
recitante = pyttsx3.init()
recitante.setProperty('voice', 'italian')
recitante.say('Come ti chiami?')
recitante.runAndWait()
riconoscitore = sr.Recognizer()
with sr.Microphone() as source:
    audio = riconoscitore.listen(source)
testo = riconoscitore.recognize_google(audio, language = "it")
f = open('/home/vittorio/Documenti/saluto.txt', 'w')
f.write('Ciao' + testo)
f.close()
f = open('/home/vittorio/Documenti/saluto.txt', 'r')
recitante.say(f.read())
recitante.runAndWait()
```

Capitolo 5

Python per le ricerche scientifiche

Con quanto visto nei capitoli precedenti siamo in grado, pur da programmatori alle prime armi, di costruire programmi anche non banali, volendo, arricchendoli di interfaccia grafica.

Tutto ciò che vedremo da qui in poi si basa sostanzialmente su ciò che abbiamo visto ma farlo con quei semplici strumenti richiederebbe una maestria da programmatori superesperti e la scrittura di centinaia, anche di migliaia, di righe di codice per ogni programma.

Fortunatamente, nella comunità di Python, c'è chi ha scritto una volta per tutte queste righe di codice per metterci a disposizione librerie, quelle che nel mondo Python si chiamano moduli, che possiamo utilizzare richiamando le funzioni che contengono senza riscriverne il codice, consentendoci di fare, anche da programmatori alle prime armi, cose più complesse di quelle viste finora.

Nel Paragrafo 2.9 abbiamo visto come si possano creare funzioni riutilizzabili: ne abbiamo creata una, per calcolare il fattoriale di un numero, che ritroviamo anche in un modulo predisposto dalla comunità Python, il modulo math. In questo caso la funzione era relativamente semplice da scrivere ed anche se non l'avessimo trovata già predisposta poco male. Ma non sempre è così semplice.

Come supporto di Python alla ricerca scientifica, il modulo math, inserito nella dotazione di base, è un utile primo passo.

Sempre la dotazione di base ci offre strutture di dati complesse assimilabili a ciò che in informatica va sotto il nome di array (`liste`, `tuple`) e, pur con parecchio sforzo, già utilizzando queste strutture potremmo sviluppare calcoli vettoriali e matriciali.

E' proprio per semplificare e rendere più efficienti queste tipologie di calcolo che è stato creato il modulo **numpy**, che è diventato il fulcro di una serie di altri moduli per il calcolo scientifico (**matplotlib** e **scipy**).

Dal calcolo numerico, cui sono dedicati tutti i richiamati moduli, passiamo al calcolo simbolico, di cui ha altrettanto bisogno il ricercatore scientifico, con il modulo **sympy**.

Questo capitolo è dedicato ad un approccio di base a questi moduli.

5.1 NumPy

NumPy sta per Numeric Python e, prendendo la definizione di Wikipedia, è una libreria open source per il linguaggio di programmazione Python, che aggiunge supporto a grandi matrici e array multidimensionali insieme a una vasta collezione di funzioni matematiche di alto livello per poter operare efficientemente su queste strutture dati.

All'indirizzo *https://numpy.org/* troviamo tutto ciò che riguarda NumPy e il suo utilizzo: documentazione, tutorial, ecc.

Prerequisito per l'utilizzo di NumPy è avere installato Python.

Prima di installare NumPy possiamo verificare se non sia per caso già installato digitando nella shell di Python `import numpy`: se non esce alcun segnale di errore significa che NumPy è già installato.

L'installazione può facilmente avvenire con pip, secondo quando visto nel Capitolo 1.

Per usare NumPy dobbiamo importarlo all'inizio dello script e, dei tre modi per poterlo fare, il più efficiente è

```
import numpy as np
```

che richiede poi il richiamo delle funzioni attraverso la sintassi `np.<nome_funzione>`[1].

5.1.1 ndarray

Sta per n-dimension-array ed è la struttura dati di base per NumPy. E' un oggetto che rappresenta un array multidimensionale e omogeneo di dimensioni fisse.

Se l'array è ad una dimensione si tratta di un vettore; se è a due dimensioni si tratta di una matrice e teoricamente non c'è limite alle

[1] L'importazione con i comandi `import numpy` o `from numpy import *`, pur possibile, è sconsigliata in quanto può dare origine a confusione sui nomi delle funzioni e, almeno nel primo caso, allunga anche i tempi di riferimento.

dimensioni, anche se sul piano pratico si va difficilmente oltre queste due.

Avendo importato il modulo NumPy come indicato prima, l'array si crea con la funzione

`np.array()`

il cui argomento, da mettere tra le parentesi tonde, può essere una tupla o una lista.

Costruiamo il vettore

```
1 2 3
```

con

`np.array((1,2,3))`

oppure

`np.array([1,2,3])`.

Costruiamo la matrice

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

con

`np.array(((1,2,3), (4,5,6), (7,8,9)))`

oppure

`np.array([[1,2,3], [4,5,6], [7,8,9]])`.

Come si vede, il vettore è rappresentato da una sola tupla o da una sola lista, mentre la matrice è rappresentata da una tupla di tuple o da una lista di liste i cui elementi corrispondono alle righe della matrice.

La funzione array accetta un ulteriore argomento facoltativo con il quale possiamo stabilire il tipo di dato numerico (`int` o `float`). Se esso non viene indicato, automaticamente viene desunto da quanto inserito: se un solo dato è di tipo float, tutti saranno di tipo float.

Esistono particolari funzioni per creare array di un certo tipo.

`np.arange(<partenza>, <arrivo>, <step>)`

crea un vettore che contiene i numeri compresi tra partenza e arrivo - 1 cadenzati secondo lo step. Se si indica solo l'arrivo si crea un vettore che contiene i numeri da 0 a arrivo - 1. Se non si indica lo step i numeri saranno indicati per unità consecutive.

`np.arange(5)` crea il vettore `0 1 2 3 4`

`np.arange(1, 5)` crea il vettore `1 2 3 4`

`np.arange(1, 3, 0.5)` crea il vettore `1 1,5 2 2,5`

`np.arange(1,11,2)` crea il vettore `1 3 5 7 9`

`np.linspace(<partenza>, <arrivo>, <quantità_punti>)`

crea un vettore che suddivide l'enumerazione tra partenza e arrivo in tanti punti quanti indicati.

np.linspace(1,5,5) crea il vettore 1 2 3 4 5

np.linspace(1,5,3) crea il vettore 1 3 5

`np.zeros(<elementi>)`

crea un vettore con il numero di elementi indicato come argomento, tutti uguali a 0; se indichiamo gli elementi con una tupla contenente il numero di righe e di colonne di una matrice creiamo una matrice di zeri.

`np.zeros(4)` crea il vettore `0 0 0 0`

`np.zeros((2,3))` crea la matrice $\left|\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right|$

`np.ones(<elementi>)`

crea un vettore con il numero di elementi indicato come argomento, tutti uguali a 1; se indichiamo gli elementi con una tupla contenente il numero di righe e di colonne di una matrice creiamo una matrice di 1.

`np.ones(4)` crea il vettore `1 1 1 1`

`np.ones((2,3))` crea la matrice $\left|\begin{array}{ccc} 1 & 1 & 1 \\ 1 & 1 & 1 \end{array}\right|$

Anche queste funzioni accettano il parametro opzionale per stabilire il tipo di dato (`int` o `float`). Se esso non viene indicato viene attribuito il tipo float.

L'oggetto ndarray può essere inserito in una variabile.

`a = np.array([1,2,3])`
crea la variabile `a` contenente il vettore `1 2 3`,
`m = np.array([[1,2,3,4], [5,6,7,8]])`
crea la variabile `m` contenente la matrice

$$\left|\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{array}\right|$$

Ogni elemento del ndarray è identificato dal nome della variabile seguito, tra parentesi quadre, dall'indice di posizione dell'elemento: l'indice di posizione è un solo numero intero per i vettori e corrisponde alla

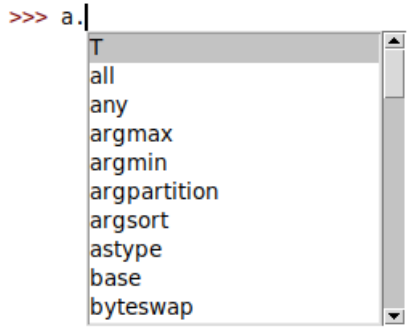
conta degli elementi da sinistra a destra partendo da zero; per le matrici è una tupla il cui primo elemento è l'indice di riga e il secondo è l'indice di colonna sapendo che l'angolo in alto a sinistra della matrice corrisponde a (0, 0).
Date le variabili appena create,
`a[1]` corrisponde a `2`,
`m[(1,2)]` oppure semplicemente `m[1,2]` corrisponde a `7`.

Ogni elemento del ndarray è a sua volta una variabile e ne può essere modificato il valore.
`a[1] = 5` assegna al secondo elemento dell'array `a` il valore `5` e l'array `a` diventa `1 5 3`.

Attraverso lo slicing possiamo identificare zone dell'array che sono a loro volta degli ndarray. Per un array `x`
`x[2:4]` identifica un array composto dal terzo e dal quarto elemento di un vettore (gli elementi compresi tra quello di indice 2 e quello di indice 4-1).
`x[1,0:2]` identifica un array composto dal primo e dal secondo elemento della seconda riga di una matrice.
`x[2,2:]` identifica un array composto da tutti gli elementi a partire dal terzo fino alla fine della terza riga di una matrice.

Se esploriamo il tipo della variabile che contiene l'array con la funzione `type` ci viene indicato che la variabile è di tipo `ndarray`.
`type(m)` restituisce `<class 'numpy.ndarray'>`.

Se nell'IDLE dove abbiamo creato un array assegnandolo ad una variabile digitiamo il nome di questa variabile seguito da un punto apriamo il lungo elenco di proprietà e metodi dell'oggetto ndarray



Agendo sulla barra di scorrimento sulla destra vediamo che l'elenco è molto lungo. Qui illustro solamente le proprietà e i metodi di uso più

ricorrente e rimando alla documentazione ufficiale eventuali approfondimenti.

Proprietà

Possiamo esplorare alcune proprietà della variabile contenente ndarray attraverso le seguenti funzioni, richiamabili con la sintassi <variabile>.<funzione>

`dtype`

ritorna il tipo dei dati che formano l'array.

Con riferimento alle variabili create nel precedente paragrafo,

`a.dtype` ritorna `dtype('int32')`.

`ndim`

ritorna la dimensione dell'array.

Sempre con riferimento alle variabili create nel precedente paragrafo,

`a.ndim` ritorna `1`,

`m.ndim` ritorna `2`.

`shape`

applicata alle matrici, ritorna una tupla contenente il numero delle righe e il numero delle colonne

`m.shape` ritorna `(2, 4)`.

Metodi

Esistono molti metodi di manipolazione e di estrazione di dati.

In generale questi metodi non alterano l'array su cui vengono applicati, in modo che il contenuto della variabile che li contiene non cambia. Per riutilizzare i risultati delle manipolazioni occorre creare nuove variabili che li contengano.

Unica eccezione, il seguente metodo

`sort()`

ordina gli elementi dell'array.

Essendo un array `a` composto dagli elementi `1 5 3 2`

con `a.sort()`

esso diventa `1 2 3 5`

Tra i tanti altri metodi cito i seguenti, più comunemente utilizzati.

`copy()`

crea una copia dell'array.

Si potrebbe pensare di fare una copia semplicemente inserendo il contenuto di una variabile in un'altra variabile: così facendo, tuttavia, una variazione apportata al contenuto agirebbe su entrambe le variabili.

Se creiamo la variabile copia con questa funzione ciò non accade.

Per esempio, tornando all'array `a` che abbiamo appena utilizzato, con `b = a` prima di applicare la funzione `sort()` avremo due array, `a` e `b`, contenenti gli elementi `1 5 3 2`.

Con `c = a.copy()` avremo anche l'array `c`, con gli stessi elementi `1 5 3 2`.

Dopo `a.sort()`, entrambe le variabili contenenti gli array `a` e `b` diventano `1 2 3 5`, mentre la variabile contenente l'array `c` rimane `1 5 3 2`.

Allo stesso modo, se ora modifichiamo il primo elemento dell'array `a` con `a[0] = 7`, gli array contenuti in `a` e `b` diventano `7 2 3 5` mentre l'array contenuto in `c` rimane sempre `1 5 3 2`.

`tolist()`

converte l'array in una lista.

Se l'array è una matrice viene prodotta una lista di liste.

Dal momento che il ndarray ha dimensione fissa e non esistono suoi metodi per aggiungere elementi, la conversione in lista torna utile per poterlo fare con il metodo `append` dell'oggetto `list`.

Se, per esempio, abbiamo la seguente matrice `m`

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{vmatrix}$$

e vogliamo aggiungervi la riga `7 8 9`,

innanzi tutto convertiamo la matrice in una lista di liste con

```
ml = m.tolist()
```

aggiungiamo a questa lista la lista `[7,8,9]` con

```
ml.append([7,8,9])
```

e poi, utilizzando la nuova lista di liste, ricostruiamo la matrice m con

`m = np.array(ml)`, ottenendo così la nuova matrice `m`

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

`transpose()`

genera la trasposta di una matrice.

Con

```
mt = m.transpose()
```

otteniamo la matrice `mt`, trasposta della matrice `m` creata nel precedente esempio

$$\begin{vmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{vmatrix}$$

`flatten()`

trasforma una matrice in un vettore.

Data la matrice `m`

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$$

con `a = m.flatten()` otteniamo il vettore `a` con elementi `1 2 3 4`.

`reshape()`

trasforma un vettore in una matrice.

Tra le partentesi tonde va inserito, come argomento, una tupla indicante il numero di righe e di colonne della matrice che si vuole ottenere.

Dato il vettore a dell'esempio precedente `1 2 3 4`, con `a.reshape((2,2))` oppure, semplicemente, `a.reshape(2,2)` otteniamo la matrice

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$$

Ovviamente il numero di elementi del vettore deve essere coerente con le dimensioni della matrice che vogliamo ottenere: da un vettore di 3 o di 5 elementi non potremo ottenere una matrice 2x2.

`min()`

ritorna il valore minimo degli elementi che costituiscono un vettore o una matrice.

`max()`

ritorna il valore massimo degli elementi che costituiscono un vettore o una matrice.

`mean()`

ritorna la media del valore degli elementi che costituiscono un vettore o una matrice.

`std()`

ritorna lo scarto quadratico medio tra il valore degli elementi che costituiscono un vettore o una matrice.

`sum()`

ritorna la somma dei valori degli elementi che costituiscono un vettore o una matrice.

`prod()`

ritorna il prodotto dei valori degli elementi che costituiscono un vettore o una matrice.

`cumsum()`

ritorna un vettore i cui elementi sono la somma cumulativa dei valori degli elementi che costituiscono un vettore o una matrice.

Data la matrice `m`

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$$

`m.min()` ritorna `1`
`m.max()` ritorna `4`
`m.mean()` ritorna `2,5`
`m.std()` ritorna `1.1180339887498949`
`m.sum()` ritorna `10`
`m.prod()` ritorna `24`
`m.cumsum()` ritorna il vettore `1 3 6 10`.

Attraverso lo slicing possiamo riferire questi calcoli a zone di un vettore o di una matrice:

`x[1,0:3].mean()` calcola la media tra i primi tre elementi della seconda riga di una matrice `x`.
Nel caso della nostra matrice `m`
`m[0,0:].mean()` ritorna `1,5`.

5.1.2 Operazioni matematiche

Inserendo gli operatori `+ - * / **` tra uno o più ndarray che abbiano le stesse dimensioni otteniamo un nuovo ndarray i cui elementi sono il risultato dell'applicazione dell'operatore elemento per elemento.

Dati i vettori

```
a = np.array((1,2,4))
b = np.array((3,1,2))
```

`a+b` ritorna un vettore composto da 4 3 6,
`a/b` ritorna un vettore composto da 0,3333 2 2,
`a**b` ritorna un vettore composto da `1 2 16`.

Date le matrici

```
m = np.array(((2,3,1),(4,2,3)))
```

$$\begin{vmatrix} 2 & 3 & 1 \\ 4 & 2 & 3 \end{vmatrix}$$

```
n = np.array(((1,2,3),(3,2,2)))
```

$$\begin{vmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \end{vmatrix}$$

`m-n` ritorna la matrice

$$\begin{vmatrix} 1 & 1 & -2 \\ 1 & 0 & 1 \end{vmatrix}$$

`m*n` ritorna la matrice

$$\begin{vmatrix} 2 & 6 & 3 \\ 12 & 4 & 6 \end{vmatrix}$$

che, attenzione, non è il prodotto matriciale ma il prodotto membro a membro.

Le stesse operazioni si possono fare tra un ndarray e uno scalare. In questo caso viene generato un nuovo ndarray i cui elementi sono costituiti dal risultato dell'operazione tra ciascun elemento e lo scalare.
Se prendiamo il nostro vettore `a` di prima
`a*3.5` ritorna `3,5 7 14`,
`a**2` ritorna `1 4 16`.
Se prendiamo la nostra matrice n di prima
`n/2` ritorna la matrice

$$\begin{vmatrix} 0.5 & 1 & 1.5 \\ 1.5 & 1 & 1 \end{vmatrix}$$

Allo stesso modo possiamo passare il ndarray come argomento ad una qualsiasi funzione simile a quelle contenute nel modulo math di Python di base, che il modulo NumPy replica in modo da renderla applicabile ad un ndarray (gli argomenti che possiamo passare ad una funzione del modulo math sono solo scalari). Otteniamo così un nuovo ndarray i cui elementi sono il risultato della funzione applicata a ciascun elemento.
Dato il vettore
`v = np.array((4,16,36,12))`
`np.sqrt(v)` ritorna `2 4 6 3,464`.

Per i nostri calcoli NumPy ci fornisce i valori a doppia precisione delle costanti matematiche π e e:
`np.pi` ritorna `3.141592653589793`,
`np.e` ritorna `2.718281828459045`.

5.1.3 Calcolo matriciale

Alla base del calcolo matriciale sta il prodotto scalare.

Il prodotto scalare tra due vettori aventi le stesse dimensioni corrisponde alla somma dei prodotti tra gli elementi dei vettori aventi lo stesso indice.
Se il vettore `a` è `1 2 3` e il vettore `b` è `3 5 0`
il prodotto scalare tra `a` e `b` è `(1*3)+(2*5)+(3*0) = 3+10+0 = 13.`

NumPy ha una funzione per calcolarlo, la funzione `dot()`.
Nel caso dei due vettori di cui sopra
`np.dot(a,b)` ritorna `13`.

Se abbiamo due matrici, di cui la prima ha un numero di colonne pari al numero di righe della seconda, attraverso il prodotto scalare calcoliamo il prodotto matriciale tra le due matrici, che è una matrice i cui elementi sono i prodotti scalari tra le righe della prima matrice e le colonne della seconda matrice.
Il prodotto matriciale tra `m` = $\begin{vmatrix} 1 & 2 \\ 5 & 4 \end{vmatrix}$ e `n` = $\begin{vmatrix} 2 & 6 \\ 1 & 3 \end{vmatrix}$ è una matrice che ha
all'indice [0,0] `(1*2)+(2*1) = 4`
all'indice [0,1] `(1*6)+(2*3) = 12`
all'indice [1,0] `(5*2)+(4*1) = 14`
all'indice [1,1] `(5*6)+(4*3) = 42`
cioè $\begin{vmatrix} 4 & 12 \\ 14 & 42 \end{vmatrix}$.

A questo risultato arriviamo passando alla funzione dot(), come argomenti, le due matrici `m` e `n`: `np.dot(m,n)`.

E qui si innesta l'algebra lineare.

NumPy ha il sotto-modulo `linalg` che contiene utili funzioni.

`inv()`

calcola la matrice inversa.

Data la matrice `m` di prima,

`np.linalg.inv(m)` ritorna $\begin{vmatrix} -0.66666667 & 0.33333333 \\ 0.83333333 & -0.16666667 \end{vmatrix}$

`det()`

calcola il determinante.

Sempre con la matrice m di prima,

`np.linalg.det(m)` ritorna `-6`

Ne avremmo abbastanza per risolvere sistemi di equazioni lineari, moltiplicando l'inversa della matrice dei coefficienti per il vettore dei termini noti, ma Numpy, con il sotto-modulo linalg, ci leva anche questo disturbo con la funzione `solve()`, passando alla quale la matrice dei coefficienti e il vettore dei termini noti otteniamo la soluzione del sistema.

Dato il sistema

$$\begin{cases} 2x - y = 2 \\ 3y + 2z = 16 \\ 5x + 3z = 21 \end{cases}$$

costruiamo la matrice dei coefficienti

```
m = np.array(((2,-1,0),(0,3,2),(5,0,3)))
```

e il vettore dei termini noti

```
n = np.array((2,16,21))
```

Possiamo ottenere il vettore della soluzione con

```
s = np.dot(np.linalg.inv(m),n)
```

oppure con

```
s = np.linalg.solve(m,n)
```

in entrambi i casi ottenendo il vettore `s = 3 4 2` dove `3` è il valore risolutivo per la `x`, `4` è il valore risolutivo per la `y` e `2` è il valore risolutivo per la `z`.

* * *

Se qualcuno vorrà approfondire i segreti di NumPy più di quanto abbia fatto io in questo rapido excursus troverà molto altro, soprattutto vedrà come esistano apparentemente molti modi di fare la stessa cosa. Per esempio scoprirà che per costruire una matrice di tipo diverso da `ndarray` esiste la funzione `matrix()`, che genera un oggetto di tipo `matrix` che ha praticamente tutte le proprietà e i metodi di `ndarray` con la sola

differenza che non può andare oltre le due dimensioni. Per cui non è del tutto vero che è la stessa cosa, anche, se per fare ciò che abbiamo visto finora, lo è.

5.2 Matplotlib

Matplotlib è una libreria Python per la visualizzazione di dati in modalità grafica.

All'indirizzo *https://matplotlib.org/* troviamo tutto ciò che riguarda Matplotlib e il suo utilizzo: documentazione, tutorial, ecc.

Prerequisito per l'utilizzo di Matplotlib è avere installato Python e il modulo Numpy.

Come per il modulo NumPy, l'installazione può facilmente avvenire con pip.

Con Matplotlib possiamo realizzare qualsiasi tipo di grafico e per descrivere le sue potenzialità servirebbe un intero volume.

Qui mi limito a descrivere l'applicazione di Matplotlib più vicina al calcolo scientifico, quella della visualizzazione di grafici di funzione.

Per fare questo si utilizza il sottomodulo `pyplot`, che si importa così `import matplotlib.pyplot as plt`.

Le più importanti funzioni di questo sottomodulo sono
`plot()` per creare il grafico,
`title(<stringa>)` per dargli un titolo,
`grid()` per visualizzare una griglia,
`show()` per mostrare il grafico sullo schermo.

Gli argomenti della funzione `plot()` sono, nell'ordine:
la variabile contenente un array per l'asse della ascisse,
la variabile contenente l'espressione della funzione,

oltre, facoltativi:
`color = "<nome_colore>"` per indicare il colore del grafico,
`linewidth = "<numero> "` per indicare lo spessore della linea.
Se questi parametri facoltativi non sono indicati viene tracciato un grafico in blu con linea sottile (pari all'intero 1).

Prima dell'eventuale utilizzo della funzione `show()` possiamo utilizzare la funzione
`savefig("<path_e_nome_file>.png")` per salvare il grafico in formato `.png`.
Cosa che possiamo fare anche utilizzando un pulsante contenuto nella finestra del grafico mostrata dalla funzione `show()`.

Il seguente script traccia il grafico della funzione $y = x^2$ tra -5 e 5, in colore verde, con il titolo «Grafico di funzione», con griglia visibile, lo memorizza in un file in formato .png e lo visualizza sullo schermo.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-5,5,50)
y = x**2
plt.plot(x, y, color = "green")
plt.title("Grafico di funzione")
plt.grid()
plt.savefig("/home/vittorio/Immagini/grafico.png")
plt.show()
```

Il terzo argomento passato alla funzione `linspace()` di NumPy per creare l'array della variabile indipendente deve essere un numero abbastanza elevato in modo da creare una bella curva continua nel grafico.

Se vogliamo dare un titolo agli assi abbiamo a disposizione le funzioni
`xlabel(<stringa>)` per dare un titolo all'asse delle ascisse,
`ylabel(<stringa>)` per dare un titolo all'asse delle ordinate.

Se vogliamo regolare le dimensioni degli assi da visualizzare possiamo usare la funzione `axis()`, il cui argomento è una lista contenente, nell'ordine, il limite sinistro dell'asse x, il limite destro dell'asse x, il limite inferiore dell'asse y e il limite superiore dell'asse y.

5.3 SciPy

SciPy sta per Scientific Python ed è una collezione di funzioni e algoritmi matematici usati in campo scentifico, costruita su NumPy.

All'indirizzo *https://scipy.org/* troviamo una serie di link a tutte le librerie Python utilizzabili in campo scientifico, comprese le due viste finora. Scegliendo il link SCIPY LIBRARY troviamo tutto ciò che riguarda SciPy e il suo utilizzo: documentazione, tutorial, ecc.

Prerequisito per l'utilizzo di SciPy è avere installato Python e il modulo NumPy.

Come per gli altri moduli, l'installazione può facilmente avvenire con pip.

La raccolta delle funzioni è organizzata in più sotto-moduli:
`cluster` contiene algoritmi di clustering,
`constants` contiene costanti matematiche e fisiche,

`fftpack` contiene funzioni della Trasformata di Fourier,
`integrate` per integrazione ed equazioni differenziali ordinarie,
`interpolate` per interpolazione e smoothing spline,
`io` per Input e Output,
`linalg` contiene funzioni di algebra lineare,
`ndimage` per image processing n-dimensionale,
`odr` per la regressione delle distanze ortogonali,
`optimize` contiene funzioni di ottimizzazione e di ricerca di radici,
`signal` per elaborazione di immagini,
`sparse` per matrici sparse e funzioni associate,
`spatial` per strutture dati spaziali e algoritmi,
`special` contiene altre funzioni speciali,
`stats` per distribuzioni statistiche e funzioni correlate.

Per usare SciPy occorre importarlo e il modo migliore per farlo è quello di limitare l'importazione al sotto-modulo che serve con la sintassi
`import scipy.`<sottomodulo> `as` <sigla>.
Con
`import scipy.constants as c`
ci teniamo a disposizione il sotto-modulo `constants` e possiamo richiamarne le proprietà e le funzioni con la sintassi
`c.`<proprietà/funzione>.

Della quindicina di sotto-moduli che compongono SciPy propongo qui un'introduzione ai sei di più ricorrente utilizzo.

Per un panorama completo e più approfondito rimando alla SciPy Reference Guide che possiamo consultare e scaricare in formato PDF dal sito web di SciPy.

5.3.1 Costanti

Il sotto-modulo constants contiene costanti matematiche e fisiche.

Le costanti matematiche, in realtà, sono solo due: pi greco (π) e rapporto aureo (ϕ).
Avendo importato il sotto-modulo `constants` con la sigla `c`,
`c.pi` ritorna `3.141592653589793`,
`c.golden` ritorna `1.618033988749895`.

Moltissime le costanti del mondo della fisica, per il cui elenco completo rimando alla Reference Guide.
Come esempio cito
`c.g` che ritorna `9.80665`, la costante di gravitazione universale,

`c.Avogadro` che ritorna `6.022140857e+23`, la costante di Avogadro.

5.3.2 Algebra lineare

Il sotto-modulo `linalg` contiene praticamente le stesse funzioni dell'omonimo sottomodulo di `numpy` che abbiamo visto prima:
`inv()` calcola la matrice inversa,
`det()` calcola il determinante,
`solve(A,b)` calcola il valore delle incognite in un sistema lineare,
e rimando alle esemplificazioni fatte prima.

Altra funzione utile, pure questa contenuta anche nel sotto-modulo `linalg` di `numpy` ma di cui non ho parlato, è la funzione `lstsq(A,b)` che calcola il valore delle incognite in un sistema lineare non compatibile con il metodo dei minimi quadrati.

Sappiamo che per risolvere in maniera esatta un sistema di equazioni lineari occorre che esso sia compatibile, cioè che il numero delle equazioni sia uguale al numero delle incognite. In questo modo la matrice dei coefficienti è una matrice quadrata e se ne può calcolare l'inversa per applicare il metodo risolutivo $A^{-1}b = x$ oppure se ne può calcolare il determinante per applicare il metodo risolutivo di Cramer. In ogni caso, trovate le soluzioni x, si verifica che $b - Ax = 0$. Ed è ciò che avviene utilizzando la funzione `solve()`.

Con la funzione `lstsq()` possiamo risolvere in maniera approssimata un sistema di equazioni non compatibili e le soluzioni saranno tali per cui $b - Ax$ non è uguale a zero ma è del valore minimo possibile secondo il principio dei minimi quadrati.

Importato NumPy con
```
import numpy as np
```
e il sotto-modulo `linalg` di SciPy con
```
import scipy.linalg as ln
```
e dato il sistema
$$\begin{cases} 2x + 3y - 5z = 5 \\ 3x + 2y + 2z = 2 \end{cases}$$
costruiamo la matrice dei coefficienti con
```
A = np.array(((2,3,-5),(3,2,2)))
```
e il vettore dei termini noti con
```
b = np.array((5,2)),
```
infine, con
```
ln.lstsq(A,b)
```
otteniamo una lista il cui primo elemento è l'array
```
array([ 0.56074766, 0.58411215, -0.42523364])
```

che elenca, nell'ordine, i valori delle incognite `x`, `y` e `z`.
La lista contiene altre cose che lasciamo ai matematici professionisti.

5.3.3 Interpolazione

L'interpolazione è il processo con cui valutiamo il più probabile valore ignoto tra due valori noti.

Il sotto-modulo `interpolate` di SciPy offre parecchie funzioni per attuare questo processo. Qui illustro la più semplice: `interp1d`, nelle due varianti opzionali lineare e cubica.

Supponiamo di avere otto misurazioni della temperatura di una giornata a determinate ore:

ora	gradi
3	14
5	13
7	14
10	16
12	18
14	19
18	17
23	16

Importiamo il modulo numpy con
import numpy as np,

importiamo il sotto-modulo interpolate di scipy con
import scipy.interpolate as int.

Creiamo gli array dei dati denominando `x` l'array delle ore e `y` quello delle temperature:

```
x = np.array((3,5,7,10,12,14,18,23)),
y = np.array((14,13,14,16,18,19,17,16)).
```

Troviamo la linea interpolante con
f1 = int.interp1d(x, y, kind = 'linear') per l'interpolazione lineare,
f2 = int.interp1d(x, y, kind = 'cubic') per l'interpolazione cubica.

Se non si utilizza l'opzione kind viene effettuata l'interpolazione lineare.

Se vogliamo conoscere la più probabile temperatura alle ore 16, ora per la quale non esiste la rilevazione, passiamo questo valore come argomento alla funzione interpolante:
`f1(16)` ritorna `18`,
`f2(16)` ritorna `18.426358355165736`

a dimostrazione del fatto che la dinamica dell'andamento della temperatura desunta dalle osservazioni è diversa a seconda del tipo di linea interpolante scelta.

Possiamo avere contezza grafica di tutto ciò.

Importiamo il plotter di Matplotlib con

`import matplotlib.pyplot as plt`.

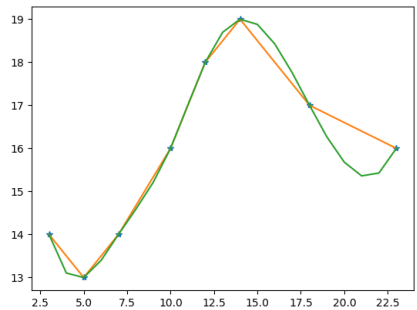
Creiamo un nuovo array per l'asse delle ascisse scandito su tutte le ore del nostro intervallo di misurazione con:

`xnew = np.linspace(3,23,21)`.

Ora costruiamo un grafico che mostri i dati osservati, la linea di interpolazione lineare f1 e la curva di interpolazione cubica f2 con:

`plt.plot(x,y,'*',xnew,f1(xnew),xnew,f2(xnew))`

e con `plt.show()` lo mostriamo



I valori osservati sono contrassegnati da un asterisco (notare la sintassi con cui lo si è scelto con l'opzione '*').

La linea arancione corrisponde all'interpolante lineare.

La curva verde corrisponde all'interpolante cubica.

* * *

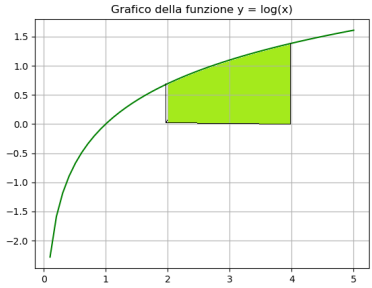
Qui ho presentato la più semplice applicazione del sotto-modulo - `interpolate` di SciPy attraverso la funzione `interp1d` che lavora su due variabili.

Se siamo in presenza di tre variabili abbiamo a disposizione la funzione `interp2d`.

Ma vi sono tante altre funzioni per le quali rimando alla Reference Guide.

5.3.4 Integrazione

Nel calcolo numerico l'integrazione consiste praticamente nel determinare l'area compresa tra un segmento dell'asse delle ascisse e la corrispondente linea del grafico di una funzione: la così detta integrazione definita.



In questo esempio, che riguarda la funzione $y = log(x)$, essa corrisponde alla zona colorata di verde e si esprime in questo modo

$$\int_2^4 log(x)\, dx$$

Il sotto-pacchetto `integrate` contiene parecchie funzioni per il calcolo integrale (integrali semplici, doppi, tripli, a quadratura gaussiana, ecc.) e rimando alla Reference Guide per approfondimenti. Qui mostro l'uso della funzione più semplice, quella che calcola l'area di cui abbiamo parlato, che si chiama `quad()`.

Importiamo il sotto-pacchetto con

```
import scipy.integrate as i
```

Importiamo anche il modulo `math` di Python per utilizzarne la funzione `log()`

```
import math
```

Definiamo la funzione di cui calcolare l'integrale definito con

```
def f(x):
    return math.log(x)
```

e, per stare all'esempio da cui siamo partiti, ne calcoliamo l'integrale tra 2 e 4 così

```
i.quad(f, 2, 4).
```

Il risultato è la seguente tupla

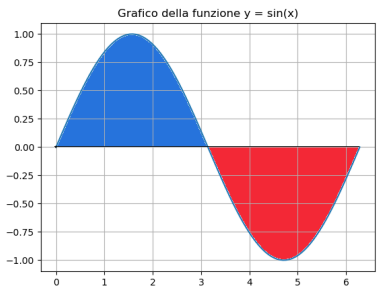
```
(2.158883083359672, 2.396841706619659e-14)
```

il cui primo membro è il ricercato valore dell'area e il secondo, come si vede un numero molto piccolo, indica l'errore di approssimazione del calcolo.

Sugli integrali definiti va ricordato che l'area calcolata in corrispondenza a valori negativi della funzione è negativa e, se calcoliamo l'integrale per tratti in cui la funzione è in parte positiva e in parte negativa, l'area negativa va a sottrarsi a quella positiva.

Se vogliamo calcolare l'area effettiva, comprendente sia la parte positiva sia la parte negativa, dobbiamo effettuare il calcolo delle aree separatamente per le zone con segno diverso della funzione, sommando ai valori positivi il valore assoluto dei valori negativi.

Prendiamo, per esempio, la funzione $y = sin(x)$



L'area contrassegnata in colore blu, corrispondente a valori positivi della y, ha valore positivo mentre quella contrassegnata in colore rosso, corrispondente a valori negativi della y, ha valore negativo.
Dal momento che le due aree sono uguali, se calcoliamo l'integrale definito della funzione nell'intervallo tra 0 e 2π che le comprende entrambe, il risultato è zero.

Pertanto, se vogliamo sapere l'area effettiva delle due zone messe insieme possiamo procedere così.

Importiamo ciò che ci serve con

```
import scipy.integrate as i
import numpy as np
```

L'importazione di `numpy` anziché di `math` ci serve per la funzione che calcola il valore assoluto. Dal momento che il risultato da indicare come argomento della funzione `absolute()`, presente in entrambi i moduli, è una tupla, la funzione del modulo `math` non va bene in quanto accetta come argomento un solo valore. Da che ci siamo utilizziamo anche la funzione `sin()` di NumPy.

Definiamo la funzione con

```
def f(x):
    return np.sin(x)
```
Ora, con
`i.quad(f,0,2*np.pi)`
otteniamo l'area `2.2579663352318637e-16`
mentre con
`i.quad(f,0,np.pi) + np.absolute(i.quad(f,np.pi,2*np.pi))`
otteniamo l'area `4.00000000e+00`.
Praticamente 0 nel primo caso e 4 nel secondo.

Precisazione finale.

Negli esempi ho utilizzato le funzioni `log()` e `sin()`, che sono funzioni predefinite nei moduli Python, ma la funzione da integrare possiamo anche costruircela noi.

Per esempio, per calcolare

$$\int_0^3 x^2 - 2x + 1 \, dx$$

costruiamo la funzione con
```
def f(x):
    return x**2-2*x+1
```
e con
`i.quad(f,0,3)`
otteniamo il risultato
`(2.9999999999999996, 3.330669073875469e-14).`

5.3.5 Ottimizzazione

In matematica l'ottimizzazione consiste nella ricerca degli input per una funzione obiettivo ai quali corrisponda l'output minimo o massimo della funzione stessa.

Il sotto-pacchetto `optimize` contiene moltissime funzioni per la ricerca dei minimi di una funzione. Qui mostro l'uso delle funzioni più semplici e rimando alla Reference Guide per approfondimenti.

Il fatto che gli algoritmi siano tutti finalizzati alla minimizzazione non ci deve scoraggiare se abbiamo come obiettivo la ricerca di un massimo: basta, infatti, che anteponiamo segno negativo ai valori restituiti dalla funzione obiettivo e la ricerca del minimo si converte nella ricerca di un massimo.

Le semplici funzioni che utilizziamo come principianti sono:

`minimize_scalar()` per la ricerca del minimo assoluto,
`minimize()` per la ricerca di tutti i minimi, compresi quelli relativi.

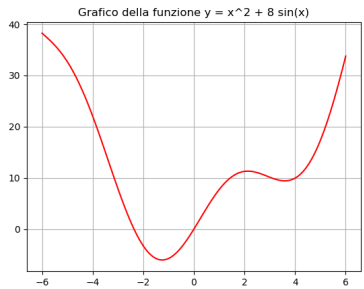
Nel primo caso basta che passiamo alla funzione, come argomento, la funzione obiettivo. Nel secondo caso dobbiamo passare come argomenti, oltre alla funzione obiettivo, separato da virgola, un valore tentativo che avvii la ricerca iterativa.

Vediamo come funziona.

Sia questa la nostra funzione obiettivo

$$x^2 + 8\sin(x)$$

il cui grafico è il seguente



Importiamo il sotto-pacchetto con

```
import scipy.optimize as opt
```

Importiamo anche il modulo `math` di Python per utilizzarne la funzione `sin()`

```
import math
```

Definiamo la funzione obiettivo con

```
def f(x):
    return x**2+8*math.sin(x)
```

e con

```
opt.minimize_scalar(f)
```

veniamo informati del fatto che la ricerca ha avuto successo e che il minimo assoluto della funzione è

```
-6.0294035242910322
```

corrispondente al valore della x di

```
-1.2523532340863732
```

Allo stesso risultato perveniamo con

```
opt.minimize(f,-3)
```

avviando l'iterazione da sinistra con il valore tentativo -3.
Se ci portiamo verso destra, per esempio con il valore tentativo 4,
`opt.minimize(f,4)`
scopriamo che ha avuto successo la ricerca di un altro minimo, questa volta relativo, con valore della funzione
`9.41977563692598`
corrispondente al valore della x di
`3.59530513`

Per ricercare i massimi della funzione, basta che anteponiamo il segno - alla funzione obiettivo definendola così

```
def f(x):
   return -(x**2+8*math.sin(x))
```

e con
`opt.minimize_scalar(f)`
veniamo informati del fatto che la ricerca ha avuto successo e che esiste un massimo con valore della funzione
`11.318349214841467` (rendendo positivo il risultato)
corrispondente al valore della x di
`2.1333322446613039`

Il sotto-modello `optimize` contiene anche funzioni per la ricerca delle radici della nostra funzione obiettivo, cioè dei punti in cui il grafico della funzione si incrocia con l'asse delle ascisse, i così detti zeri della funzione. La più semplice è
`fsolve()`
alla quale dobbiamo passare come parametri la funzione obiettivo e, separato da virgola, un valore tentativo per avviare l'iterazione.

Riferendoci alla funzione obiettivo dei precedenti esempi, con
`opt.fsolve(f, -10)`
avviando l'iterazione da sinistra, troviamo una prima radice per x uguale a
`-2.36630242`
e con
`opt.fsolve(f, -1)`
procedendo verso destra, troviamo una seconda radice per x uguale a
`0`
e, corrispondendo la nostra funzione ad una equazione di secondo grado, non ci saranno altre radici.

5.3.6 Statistica

Già tra i metodi del `ndarray` di NumPy abbiamo visto essere presenti alcune funzioni di statistica descrittiva (valore minimo, valore massimo, media dei valori e scarto quadratico medio).

Ma per la statistica abbiamo il sotto-modulo di SciPy `stats`, che contiene un gran numero di strumenti per descrivere campioni statistici, per lavorare con distribuzioni di probabilità e per eseguire diverse tipologie di test statistici.

Possiamo importarlo con la formula sin qui usata
`import scipy.stats as st.`

Dal momento che tutti i dati su cui lavoriamo devono essere nel formato `ndarray`, dobbiamo sempre importare anche NumPy con
`import numpy as np`.

Con un'unica funzione, `describe()`, produciamo in blocco alcuni indicatori di statistica descrittiva.
Dato l'array
```
a = np.array((2,4,6.5,12,7,6,8,15.8))
```
con
```
st.describe(a)
```
produciamo una tupla contenente il numero degli elementi (`8`), i valori minimo e massimo (`2.0` e `15.8`), la media (`7.6625`), la varianza (`19.3112`), l'indice di asimmetria (`0.6898`) e l'indice di curtosi (`-0.3922`).

Per un dilettante potrebbe bastare, ma c'è molto di più e servirebbe un intero libro per parlarne. Se nell'IDLE di Python, avendo importato il sotto-pacchetto come `st` scriviamo `st` seguito da un punto (`st.`) possiamo scorrere il lungo elenco delle funzioni disponibili e lo statistico professionista può apprezzarne l'abbondanza.

Sempre a livello dilettantesco possiamo costruire questi due array
```
x = np.array((34,45,58,62,78,112,148))
y = np.array((78,89,112,123,138,190,220))
```
e trovare la correlazione esistente tra di loro calcolando l'indice `r` di Pearson con la funzione
```
st.pearsonr(x,y)
```
che genera la tupla
```
(0.99335214818166018, 6.8961823195309017e-06
```
dove il primo elemento è l'indice `r` e il secondo è il possibile errore di determinazione.

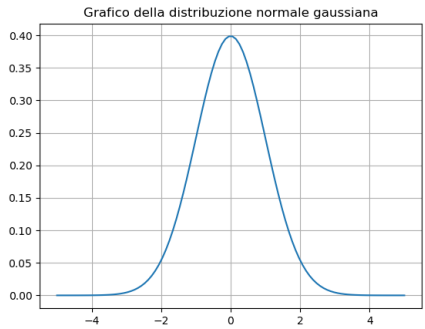
Con gli stessi array possiamo determinare la retta di regressione con la funzione
```
st.linregress(x,y)
```

che genera una tupla contenente il coefficiente angolare (slope), l'intercetta, il coefficiente di determinazione e i possibili errori.

Come esempio di funzioni relative a distribuzioni di probabilità cito il più classico, che ha a che fare con la curva gaussiana, detta anche normale.
Per crearla abbiamo a disposizione la funzione `norm()` che, utilizzata senza indicare alcun argomento, crea la normale vera e propria. Tra le parentesi possiamo indicare due argomenti per personalizzare una gaussiana: il primo per indicare il valore medio della distribuzione e il secondo, separato da virgola, per indicare la deviazione standard della distribuzione.
Con il seguente script

```
import scipy.stats as st
import numpy as np
import matplotlib.pyplot as plt
y = st.norm()
x = np.linspace(-5, 5, 100)
px = y.pdf(x)
plt.plot(x, px)
plt.title("Grafico della distribuzione normale gaussiana")
plt.grid()
plt.show()
```

otteniamo il grafico della gaussiana



Il passaggio fondamentale è quello con cui viene costruito l'oggetto «distribuzione normale» con

```
y = st.norm()
```

Tutto il resto riguarda la costruzione del grafico utilizzando una variabile `x` da mettere sulle ascisse e la funzione, chiamata `px`, che rappresenta la densità di probabilità corrispondente ai valori della `x`.
La densità di probabilità si ottiene con il metodo `pdf()` (probability density function) dell'oggetto `y`.
Altri importanti metodi dell'oggetto `y` sono
`mean()` che fornisce la media della distribuzione
nel caso della normale classica `y.mean()` ritorna `0`;
`std()` che fornisce la deviazione standard
nel caso della normale classica `y.std()` ritorna `1` (il σ);
`cdf()` che fornisce la probabilità cumulativa (cumulative distribution function).
Quest'ultima funzione è molto utile in quanto per differenza tra la probabilità cumulativa di un valore superiore e la probabilità cumulativa di un valore inferiore si ottiene la probabilità che un valore sia compreso in un intervallo: in questo modo si ottengono i valori leggibili nella tavola della funzione $\Theta(\lambda)$. Per differenza tra 1 e questa probabilità si ottiene la probabilità che un valore sia fuori dall'intervallo.
Così, sempre utilizzando l'oggetto `y` che rappresenta la distribuzione normale,
`y.cdf(1)-y.cdf(-1)` restituisce `0.68268949213708585`
probabilità corrispondente a σ
`y.cdf(2)-y.cdf(-2)` restituisce `0.95449973610364158`
probabilità corrispondente a 2σ
`y.cdf(3)-y.cdf(-3)` restituisce `0.99730020393673979`
probabilità corrispondente a 3σ
`y.cdf(4)-y.cdf(-4)` restituisce `0.99993665751633376`
probabilità corrispondente a 4σ, la perfezione.
Con questi metodi possiamo determinare la probabilità che si verifichi un certo dato in presenza di una qualsiasi distribuzione per la quale siano noti la media e la deviazione standard (scarto quadratico medio).
Poniamo, per esempio, di essere in presenza di una serie di dati la cui media sia 5 e lo scarto quadratico medio sia 2.
Costruiamo la distribuzione normale caratterizzata da questi dati con
d = st.norm(5,2).
Se vogliamo conoscere la probabilità che la distribuzione presenti dati con scarto dalla media inferiore a 1 facciamo la differenza
`d.cdf(6)-d.cdf(4)` e troviamo il risultato `0.38292492254802624`
che rappresenta la probabilità che si riscontrino dati compresi tra 4 e 6

e da cui deduciamo che dati inferiori a `4` o superiori a `6` si potranno riscontrare con probabilità `0.6170750774519738` (complemento a `1` della precedente probabilità).

Per finire questa carrellata esemplificativa e tutt'altro che esauriente vediamo un test statistico, il test χ^2 di Pearson per valutare se la differenza tra due distribuzioni statistiche sia accidentale o significativa. Abbiamo tre distribuzioni in altrettanti ndarray:

```
a = np.array((38,45,36,47,58,42,35))
b = np.array((42,51,40,61,87,68,48))
c = np.array((39,45,37,45,59,43,36))
```

Con

```
st.chisquare(a,b)
```

otteniamo il valore del test, `27.8286`, e la relativa probabilità, `0.000101`, che, essendo inferiore al valore critico di `0,05` indica che la differenza tra la distribuzione `a` e la distribuzione `b` è significativa.

Con

```
st.chisquare(a,c)
```

otteniamo il valore del test, `0.2095`, e la relativa probabilità, `0.9998`, che, essendo superiore al valore critico di `0,05` indica che la differenza tra la distribuzione `a` e la distribuzione `c` è accidentale.

* * *

A chi voglia avere un'idea di cosa offra Python per il calcolo scientifico posso dire che ciò che ho illustrato in questo manualetto penso corrisponda si e no ad un 25%.

5.4 SymPy

La differenza tra il calcolo numerico e il calcolo simbolico sta nel fatto che il calcolo numerico tratta solo numeri ed esprime sempre un risultato numerico, molto spesso approssimato, mentre il calcolo simbolico tratta numeri e simboli ed esprime un risultato numerico solo se si tratta di un numero esatto e non approssimato, altrimenti anche il risultato è espresso in simboli.

Così, se lavorando con Python ed avendo importato il modulo `math` per il calcolo numerico, calcoliamo la radice quadrata di 12,25 otteniamo il risultato 3,5. Sempre lavorando con Python ma avendo importato il modulo `sympy` per il calcolo simbolico, dal momento che 3,5 è esattamente la radice quadrata di 12,25, otteniamo ancora il risultato 3,5.

Se però calcoliamo, per esempio, la radice quadrata di 8, con il calcolo numerico arriviamo al risultato approssimato 2.8284271247461903 ma con il calcolo simbolico arriviamo al risultato 2 per la radice quadrata di 2 (dalla scomposizione $\sqrt{8} = \sqrt{4}\sqrt{2} = 2\sqrt{2}$): tutto quanto si può calcolare numericamente in maniera esatta ($\sqrt{4}$) viene calcolato e il resto ($\sqrt{2}$) rimane espresso in simboli.

Se, come nel caso della radice di 8, il simbolo è numerico ($\sqrt{2}$) abbiamo modo di forzare, se serve, la sua traduzione in numero approssimato; se il simbolo è un vero e proprio simbolo, come x, a o β, esso rimane tale e quale.

Come avviene, per esempio, nel calcolo del prodotto notevole $(\sqrt{2} + a)(\sqrt{2} - a)$ che porta al risultato $2 - a^2$, dove il prodotto tra i due simboli di numeri irrazionali diventa un numero intero (2) e rimane in simbolo la parte simbolica vera e propria (a^2).

Il modulo SymPy ci dà modo di inserire il calcolo simbolico in script Python fatti da noi in modo da poter eseguire elaborazioni per le quali dovremmo altrimenti ricorrere a software dedicati al calcolo simbolico.

Obiettivo di questo paragrafo è innanzi tutto illustrare le più ricorrenti funzioni di elaborazione che ci mette a disposizione SymPy e poi far vedere come si può fare questo inserimento.

La descrizione completa di tutto ciò che si può fare con SymPy la troviamo all'indirizzo *https://www.sympy.org* nella guida completa a SymPy, disponibile anche in formato PDF, purtroppo solo in lingua inglese.

All'indirizzo *https://live.sympy.org* troviamo anche una shell per la sperimentazione on-line di Sympy.

Il modo più semplice per installare Sympy è ancora una volta pip.

Come avviene per tutti i moduli Python, per utilizzare Sympy dobbiamo importarlo e, contrariamente a quanto fatto per i moduli visti prima, lo possiamo fare con:

```
from sympy import *
```

Nel caso di Sympy, infatti, le funzioni sono abbastanza esclusive e non c'è pericolo di fare confusione con funzioni denominate allo stesso modo nei moduli importati.

5.4.1 Operazioni di base

Dichiarazione dei simboli

I simboli non numerici che intendiamo usare nelle nostre elaborazioni vanno preventivamente dichiarati.

La sintassi per farlo è:

`<simbolo>, <simbolo>, ... = symbols('<simbolo> <simbolo> ...')`
dove <simbolo> è una lettera dell'alfabeto latino direttamente prodotta dalla tastiera o una lettera dell'alfabeto greco prodotta con le dizioni `alpha, beta, gamma, delta, epsilon, zeta, eta, theta, iota, kappa, lamda, mu, nu, xi, omicron, pi, rho, sigma, tau, upsilon, phi, chi, psi, omega`.
Esempio:
`a, beta, x, w = symbols('a beta x w')`
rende disponibili per le nostre elaborazioni i simboli a, β, x e w.

I simboli di SymPy sono oggetti dotati di funzioni membro: se nell'IDLE scriviamo un simbolo precedentemente dichiarato e lo facciamo seguire da un punto (`.`) vediamo comparire l'elenco delle sue funzioni membro.

Costruzione delle espressioni

Le espressioni matematiche si compongono utilizzando i simboli dichiarati, funzioni riconosciute da SymPy e numeri, combinati con gli operatori aritmetici del linguaggio Python (`+ - * / **`).

L'espressione può essere inserita in una variabile, creando così un oggetto espressione dotato di funzioni membro.
Esempio:
`e = sin(x)**2 + cos(x)**2 - x + 2*x*y`
crea l'oggetto `e` contenente l'espressione matematica $sin^2(x) + cos^2(x) - x + 2xy$

Possiamo creare un'espressione da una stringa utilizzando la sintassi:
`<nome_espressione> = sympify('<stringa>')`
`<nome_espressione> = sympify(<nome_stringa>)`
La stringa può essere un nome o una frase qualsiasi ma, ovviamente, a noi servirà che sia un'espressione matematica.
Nel primo caso scriviamo la stringa tra le parentesi e tra apici: cosa che potevamo fare senza disturbare stringhe e sympify.
Nel secondo caso importiamo una variabile stringa già esistente con un suo nome.
Attenzione a non confondere le funzioni `sympify` e `simplify` che, come vedremo, fanno cose diverse.
Questa funzione ci dà modo di acquisire come espressione matematica elaborabile in SymPy un'espressione generata al di fuori di SymPy, per esempio introdotta dall'utente da tastiera come risposta all'invito di input da uno script Python.
Esempi:
`e = sympify('4*x**2-5')`

crea l'oggetto `e` contenente l'espressione matematica $4x^2 - 5$.
`e = sympify(formula)`
crea l'oggetto `e` che contiene l'espressione contenuta in una stringa denominata formula.

Sostituzione di simboli

Le espressioni matematiche di SymPy sono immutabili.

E' tuttavia possibile generare altre espressioni matematiche sostituendo un simbolo di una espressione matematica con un altro simbolo o valore numerico con la sintassi

`<espressione>.subs(<parte_da_sostituire>,<parte_in_sostituzione>).`

L'espressione di partenza rimane sempre la stessa.

Esempi:
`e.subs(x, y)`
applicato all'espressione `e` dell'ultimo esempio del paragrafo precedente genera l'espressione $4y^2 - 5$.
`e.subs(x, 2)`
sempre applicato all'espressione `e`, genera l'espressione matematica $4 * 2^2 - 5$, cioè il numero 11.
In ogni caso l'espressione `e` rimane $4x^2 - 5$.

Se vogliamo salvare l'espressione modificata dobbiamo assegnare il risultato della sostituzione ad una nuova espressione.
`new_e = e.subs(x, y)`
crea una nuova espressione, chiamata `new_e`, $4y^2 - 5$.

Dall'espressione numerica al numero

Quando SymPy compie calcoli su numeri fornisce il risultato in una espressione numerica.

Se questo risultato è un numero che esiste, l'espressione numerica coincide con il numero stesso e non ci accorgiamo nemmeno che è un'espressione numerica.

Se il risultato è un numero che non esiste e che non si può calcolare, come avviene per i numeri irrazionali, l'espressione numerica è, in realtà, un simbolo.

Tutto ciò lo abbiamo visto nella premessa, esemplificando il calcolo della radice quadrata di 8.

La funzione `sqrt(8)` di Sympy fornisce il risultato $2\sqrt{2}$.

L'oggetto che contiene questo risultato ha una funzione membro, che si chiama `evalf()`, che forza la traduzione dell'espressione numerica simbolica in numero approssimato. Tra le parentesi la funzione accetta un numero intero corrispondente al numero di cifre decimali, virgola

compresa, che si vogliono vedere. Se non si indica questo parametro, per default vengono fornite 16 cifre, virgola compresa.
Esempio:
`a = sqrt(8)` attribuisce alla variabile `a` il risultato $2\sqrt{2}$
`a.evalf()` fornisce il risultato 2.82842712474619
`a.evalf(4)` fornisce il risultato 2.828

Ogni espressione ha la funzione membro evalf(); per esempio `(sqrt(2)*x).evalf()` fornisce il risultato $1.4142135623731x$.

Agli stessi risultati si perviene utilizzando la funzione N() i cui argomenti sono l'espressione da valorizzare e, eventualmente, il numero di decimali richiesti.
L'esempio appena fatto potrebbe essere scritto `N(sqrt(8)*x)` e il precedente `N(a,4)`.

Bene sapere che in SymPy esiste anche la funzione `eval()` che fornisce il valore di una espressione passata come argomento in forma di stringa.
Praticamente `eval()` fa ciò che fa `sympify()`, che abbiamo visto prima.
Sia `eval()` che `sympify()` hanno la funzione membro `evalf()` per eventualmente forzare i risultati numerici.
Esempio:
`eval('sqrt(8)').evalf(16)` fornisce il risultato 2.828427124746190
`sympify('sqrt(8)').evalf()` fornisce il risultato 2.82842712474619

Visualizzazione

Per visualizzare il contenuto di variabili e i risultati che via via otteniamo dalle nostre elaborazioni, dal momento che lavoriamo in Python, abbiamo ovviamente a disposizione la funzione `print()`, con la quale, tuttavia, otteniamo la visualizzazione delle espressioni così come si inseriscono da tastiera.

SymPy è però dotato della funzione di pretty printing `pprint()`, con la quale otteniamo visualizzazioni più adeguate.

La differenza la vediamo in questo esempio

```
>>> i = Integral(sqrt(1/sin(x)**2), x)
>>> i
Integral(sqrt(sin(x)**(-2)), x)
>>> print(i)
Integral(sqrt(sin(x)**(-2)), x)
>>> pprint(i)
⌠
⎮      ___________
⎮     /     1
⎮    /    ------  dx
⎮   /        2
⎮  \/    sin (x)
⌡
```

Abbiamo inserito nella variabile `i` l'espressione per scrivere un integrale e vediamo come l'ha acquisita SymPy: la frazione con la potenza al denominatore è sostituita dal solo denominatore con potenza negativa. Se stampiamo il contenuto della variabile `i` con la funzione `print()` otteniamo tale e quale questo contenuto, scritto secondo la sintassi Python. Se lo stampiamo con la funzione `pprint()` otteniamo la visualizzazione del contenuto secondo la ricorrente simbologia matematica.

Se lavoriamo in SymPy, inoltre, sia la funzione `print()` sia la funzione `pprint()` possono produrre il codice Latex per la scrittura dell'espressione con la sintassi

```
print(latex(<espressione>))
pprint(latex(<espressione>))
```

Tornando alla nostra espressione `i` possiamo ottenere il codice Latex per scriverla come si deve

```
>>> print(latex(i))
\int \sqrt{\frac{1}{\sin^{2}{\left(x \right)}}}\, dx
```

codice che, inserito in un editor Latex tra due simboli `$`, produce questo risultato

$$\int \sqrt{\frac{1}{\sin^2(x)}}\, dx$$

5.4.2 Elaborazione delle espressioni

Semplificazioni

Quando creiamo un'espressione dalla tastiera o importando una stringa, SymPy la acquisisce apportando già le più immediate semplificazioni.

Per esempio, se digitiamo la seguente istruzione

```
e = sin(x)**2 + cos(x)**2 - x + 2*x*y + 4*x - (x*y)/2
```

l'espressione che viene creata da SymPy è

$sin^2(x) + cos^2(x) + 3x + \frac{3}{2}xy$

nella quale abbiamo già $3x$ al posto di $-x + 4x$ e $\frac{3}{2}xy$ al posto di $2xy - \frac{xy}{2}$

Ma la nostra espressione contiene un'altra possibile semplificazione, meno immediata: la somma dei quadrati di $sin(x)$ e $cos(x)$, infatti, è uguale a 1, qualunque sia il valore di x.

Per eseguire le semplificazioni meno immediate SymPy ci offre la funzione `simplify()` il cui argomento può essere un'espressione creata al momento o il nome di un'espressione esistente.

Esempi:

```
simplify(sin(x)**2 + cos(x)**2)
```

fornisce il risultato 1;

`simplify(e)` con riferimento all'espressione `e` del precedente esempio fornisce il risultato
$1 + 3x + \frac{3}{2}xy$.

Espansioni polinomiali

Per espandere un'espressione polinomiale abbiamo la funzione `expand()`, il cui argomento è un'espressione inserita direttamente o richiamata.
`expand((a+b)**3)` produce l'espressione $a^3 + 3a^2b + 3ab^2 + b^3$
Se abbiamo l'espressione `e = (x - 3)*(x + 1)`,
`expand(e)` produce l'espressione $x^2 - 2x - 3$
`expand((a+b)*(a-b))` produce l'espressione $a^2 - b^2$

Fattorizzazioni polinomiali

La fattorizzazione è l'opposto dell'espansione e la realizziamo con la funzione `factor()`, il cui argomento è un'espressione inserita direttamente o richiamata.
factor(a**2 + 2*a*b + b**2) produce l'espressione $(a + b)^2$
Se abbiamo l'espressione `e = x**2 - 2*x -3`,
factor(e) produce l'espressione $(x - 3)(x + 1)$

Operazioni sui polinomi

Per sommare, sottrarre, moltiplicare ed elevare a potenza possiamo usare i soliti operatori `+ - *` e `**` di Python.
Esempi:
Siano A e B i due polinomi
`A` = $x^4 - 2x^3 - 8x + 16$
`B` = $2x^4 + 8x$
`A + B` ritorna $3x^4 - 2x^3 + 16$
`A - B` ritorna $-x^4 - 2x^3 - 16x + 16$
`A * B`, assoggettando il risultato a expand(), ritorna $2x^8 - 4x^7 - 8x^5 + 16x^4 - 64x^2 + 128x$
`B ** 2`, assoggettando il risultato a expand(), ritorna $4x^8 + 32x^5 + 64x^2$

Per la divisione abbiamo a disposizione la funzione `div()`, i cui argomenti, separati da virgola, sono il polinomio dividendo e il polinomio divisore, che ritorna una tupla contenente il quoziente e il resto.
Esempio:
Siano DD e D i due polinomi
`DD` = $x^6 + 3x^4 - 4x + 2$
`D` = $x^2 - x + 3$
`div(DD,D)` ritorna $\left(x^4 + x^3 + x^2 - 2x - 5, 17 - 3x\right)$
per cui il quoziente è $x^4 + x^3 + x^2 - 2x - 5$

e il resto è $17 - 3x$

5.4.3 Calcolo

SymPy conosce tutte le funzioni matematiche che possiamo immaginare: per rendersene conto basta consultare la documentazione che troviamo all'indirizzo *https://live.sympy.org*, specificamente il capitolo dedicato al Functions Module.

Al mio lettore dilettante con una preparazione matematica basica ricordo le più ricorrenti:
. tutte le funzioni trigonometriche,
. tutte le funzioni iperboliche,
. `Abs()` per il valore assoluto,
. la funzione esponenziale `exp()`,
. `log()` oppure `ln()` per il logaritmo naturale,
. `log(x, b)` oppure `ln(x, b)` per il logaritmo in una qualsiasi base `b`,
. `sqrt()` per la radice quadrata,
. `root(x, n)` per la radice ennesima,
. `factorial()` per il fattoriale.

SymPy conosce pure le due costanti numeriche fondamentali:
. `pi` che è la costante π,
. `E` che è la base dei logaritmi naturali *e*.

Attenzione a parte meritano poi i seguenti strumenti di calcolo simbolico. Anche per questi mi limito ai più ricorrenti.

Derivate

Per calcolare la derivata usiamo la funzione `diff()` che accetta due argomenti:
. la funzione da derivare, scritta tale e quale o richiamata attraverso una variabile che la contiene,
. la variabile rispetto alla quale calcolare la derivata (se la funzione contiene un solo simbolo possiamo omettere questo argomento).
Esempi:
`diff(log(x))` ritorna $\frac{1}{x}$
`diff(a*log(x), x)` ritorna $\frac{a}{x}$
`diff(a*log(x), a)` ritorna $log(x)$
se abbiamo la variabile `v` che contiene l'espressione $log(x)sin^2(x)$
diff(v, x) ritorna $2\log(x)\sin(x)\cos(x) + \frac{\sin^2(x)}{x}$

La funzione `diff()` è funzione membro dell'oggetto espressione e può essere richiamata come tale: in questo caso accetta il solo argomento relativo alla variabile di derivazione.
Esempi:
I quattro esempi che abbiamo appena visto possono essere risolti anche con questa sintassi:
`log(x).diff()`
`(a*log(x)).diff(x)`
`(a*log(x)).diff(a)`
`v.diff(x)` o, essendoci un solo simbolo variabile, semplicemente `v.diff()`.
Attenzione a racchiudere sempre tra parentesi eventuali espressioni complesse, ad evitare che la funzione venga richiamata solo in relazione all'ultimo pezzo inserito.

Tutto ciò che abbiamo visto finora calcola la derivata prima.

Le derivate di ordine successivo possiamo calcolarle derivando via via la derivata precedente, ma SymPy ci offre di meglio. Infatti possiamo calcolare direttamente una derivata di un ordine qualsiasi inserendo più volte la variabile di derivazione, separando con virgole, come argomento di `diff()` oppure inserendo il numero corrispondente al grado di derivazione dopo la variabile di derivazione, separato da virgola.
Esempi:
`diff(x**3, x, x)` ritorna $6x$ che è la derivata seconda di x^3
`diff(x**3, x, 3)` ritorna 6 che è la derivata terza di x^3

Nel caso di funzioni di più variabili possiamo ottenere le derivate parziali inserendo come argomento della funzione `diff()` solo la variabile di derivazione e possiamo ottenere la derivata totale inserendo come argomenti della funzione `diff()` tutte le variabili.
Esempi:
`diff(cos(x)*sin(y), x)` ritorna $-sin(x)sin(y)$ che è la derivata parziale rispetto a x
`diff(cos(x)*sin(y), y)` ritorna $cos(x)cos(y)$ che è la derivata parziale rispetto a y
`diff(cos(x)*sin(y), x, y)` ritorna $-sin(x)cos(y)$ che è la derivata totale.

Integrali

Per il calcolo dell'integrale usiamo la funzione `integrate()`.

Integrale indefinito L'integrale indefinito è l'antiderivata e ci fornisce la funzione primitiva.

Per calcolarlo passiamo alla funzione `integrate()` due argomenti:
. la funzione da integrare, direttamente o richiamando la variabile che la contiene,
. la variabile rispetto alla quale integrare (se la funzione contiene un solo simbolo possiamo omettere questo argomento).

Esempi:
`integrate(1/x)` ritorna $log(x)$
`integrate(a/x, x)` ritorna $alog(x)$
`integrate(log(x), a)` ritorna $alog(x)$
Se abbiamo la variabile v che contiene l'espressione $2\log(x)\sin(x)\cos(x) + \frac{\sin^2(x)}{x}$
`integrate(v, x)` ritorna $\frac{-log(x)cos(2x)}{2} + \frac{log(x)}{2}$ semplificabile in $log(x)sin^2(x)$.
A seconda della potenza del nostro processore il calcolo di questo integrale può richiedere anche alcuni minuti.

La funzione `integrate()` è funzione membro dell'oggetto espressione e può essere richiamata come tale: in questo caso accetta il solo argomento relativo alla variabile di derivazione.
Esempi:
I quattro esempi che abbiamo appena visto possono essere risolti anche con questa sintassi:

```
(1/x).integrate()
(a/x).integrate(x)
log(x).integrate(a)
v.integrate(x)
```

Integrale definito Per calcolare l'integrale definito dobbiamo passare alla funzione `integrate()`, oltre ai due argomenti necessari per il calcolo dell'integrale indefinito, altri due argomenti: il limite inferiore e il limite superiore, secondo la sintassi:

```
integrate(<espressione>, (<variabile>, <limite_inferiore>,
          <limite superiore>))
```

Così l'integrale definito tra 1 e 3 di $cos(x)$ si calcola con l'istruzione

```
integrate(cos(x), (x, 1, 3))
```

il cui risultato è $-sin(1) + sin(3)$ che, assoggettato a `evalf()`, è il numero -0.700350976748029.

Limiti

La funzione per il calcolo dei limiti è `limit()` che richiede tre argomenti:
. la funzione di cui calcolare il limite, inserita direttamente o richiamando la variabile che la contiene,
. la variabile in relazione alla quale calcolare il limite,
. il valore cui far tendere la variabile per calcolare il limite.

Così, il limite di cos(x) per x tendente a 0 si calcola con l'istruzione `limit(cos(x), x, 0)` che ritorna 1.
Se abbiamo la variabile v che contiene l'espressione 1/x, possiamo calcolarne il limite per x tendente a 0 con l'istruzione

`limit(v, x, 0)` che ritorna infinito, evidenziato in SymPy con il simbolo oo (doppia o minuscola).
`limit(v, x, oo)` ritorna 0.

La funzione `limit()` è funzione membro dell'oggetto espressione e può essere richiamata come tale: in questo caso accetta i soli argomenti relativi alla variabile in riferimento alla quale calcolare il limite e al valore cui essa debba tendere.
Esempio:
il limite di $sin(x)$ per x tendente a 1 lo possiamo calcolare con l'istruzione
`sin(x).limit(x, 1)` con risultato $sin(1)$, che assoggettato a `evalf(4)`, è il numero 0.8414.
Allo stesso risultato perverremmo con l'istruzione `(sin(x).limit(x,1)).evalf(4)` che dimostra quale varietà di combinazioni di istruzioni ci offre SymPy.

Per valutare il limite da una parte sola possiamo passare anche, tra virgolette, l'argomento + (limite a destra) o - (limite a sinistra) dopo quello relativo al valore cui debba tendere la variabile.
Esempio:
`limit(1/x, x, 0, '-')` ritorna - oo ($-\infty$) che è il limite di 1/x per x tendente a zero a sinistra
`limit(1/x, x, 0, '+')` ritorna oo (∞) che è il limite di 1/x per x tendente a zero a destra.

5.4.4 Risolutori

Il più semplice risolutore che ci offre SymPy e che ritengo più che sufficiente per noi principianti è la funzione `solve()` e risolve le equazioni algebriche, comprese polinomiali e trascendenti.

Gli argomenti da passare alla funzione sono:
. l'equazione da risolvere implicitamente uguagliata a zero: ciò significa, per esempio, che se l'equazione è $x^2 - x = 2$ l'argomento da inserire è $x^2 - x - 2$,
. la variabile in relazione alla quale si intende trovare la soluzione.

Sicché l'equazione indicata si risolve con l'istruzione
`solve(x**2-x-2, x)` che ritorna la lista delle due soluzioni possibili [-1, 2]

L'equazione da risolvere può essere inserita direttamente, come abbiamo fatto qui, o richiamando la variabile ove è inserita.

Se, per esempio, abbiamo la variabile `v` che contiene l'equazione $x^2 - x - 3$, possiamo risolverla con
`solve(v, x)` e, in questo caso, la lista delle soluzioni possibili è $\left[\frac{1}{2} - \frac{\sqrt{13}}{2}, \frac{1}{2} + \frac{\sqrt{13}}{2}\right]$.

Per approssimare numericamente i risultati dobbiamo applicare `evalf()` a ciascuno di essi o utilizzare per ciascuno di essi la funzione `N()` (purtroppo tutto ciò non è applicabile a liste ma solo a oggetti singoli).

Altro esempio la risoluzione dell'equazione $ax - 3x^2 = 0$ che possiamo risolvere rispetto a `x` o ad `a` ottenendo i seguenti risultati:
`solve(a*x-3*x**2, x)` ritorna le possibili soluzioni $[0, \frac{a}{3}]$,
`solve(a*x-3*x**2, a)` ritorna la soluzione $3x$.

Se l'equazione non ammette soluzioni reali vengono indicate le soluzioni complesse, come avviene, per esempio, cercando le soluzioni di $x^2 + x + 2 = 0$ con
`solve(x**2+x+2, x)` che ci dà la lista delle soluzioni [-1/2 - sqrt(7)*I/2, -1/2 + sqrt(7)*I/2], cioè $-0.5 - 1.3228756555i$ e $-0.5 + 1.3228756555i$ come possiamo ricavare sottoponendo le due espressioni a `evalf()`.

A volte la lista delle soluzioni ne comprende di reali e complesse e spesso le radici sono espresse in termini molto grezzi e da semilavorato rispetto all'aspettativa di vederle espresse con un numero, con la complicazione che il tutto si trova in liste alle quali, come ho già detto, non sono direttamente applicabili il metodo `evalf()` e la funzione `N()`.

In alcuni casi, infine, pure in presenza di equazioni che hanno più di una soluzione, ne viene indicata solo una.
Per esempio l'equazione trascendente $e^x - 5x + 1$ ha due soluzioni:
0,544880... e 2,396138....
Se la passiamo alla funzione `solve()` ci viene indicata solo la prima delle due.

5.4.5 Matrici

Per costruire una matrice usiamo la funzione `Matrix()` il cui argomento è la lista dei vettori riga che la compongono.
`Matrix([[3, 4], [2, 5]])`
costruisce la matrice $\begin{vmatrix} 3 & 4 \\ 2 & 5 \end{vmatrix}$
`Matrix([[2, 5]])`
costruisce il vettore riga $\begin{vmatrix} 2 & 5 \end{vmatrix}$
`Matrix([2, 5])`
costruisce il vettore colonna $\begin{vmatrix} 2 \\ 5 \end{vmatrix}$

Ovviamente, essendo in SymPy, possiamo utilizzare anche simboli:
`Matrix([[a, 4], [2, b]])`
costruisce la matrice $\begin{vmatrix} a & 4 \\ 2 & b \end{vmatrix}$

Per conoscere le dimensioni di una matrice abbiamo a disposizione il metodo `shape`.

Se M è la matrice $\begin{vmatrix} a & 2 & b \\ 2 & b & 3 \end{vmatrix}$

`M.shape` fornisce il risultato (2, 3), essendo 2 il numero delle righe e 3 il numero delle colonne.

Con gli operatori `+`, `-`, `*`, `/` e `**` possiamo fare la somma, la differenza, la moltiplicazione, la divisione e l'elevamento a potenza di matrici.

Per calcolare la matrice inversa dobbiamo elevare la matrice quadrata ad esponente `-1` (se compiamo questa operazione su una matrice non quadrata generiamo un errore).

L'oggetto matrice quadrata ha la funzione membro `det()` per calcolarne il determinante (se richiamiamo questa funzione in presenza di una matrice non quadrata generiamo un errore).

Esempi:

Siano A = $\begin{vmatrix} x & 2 \\ 1 & 3 \end{vmatrix}$ e B = $\begin{vmatrix} 3 & 2 \\ 4 & 1 \end{vmatrix}$

`A + B` = $\begin{vmatrix} x+3 & 4 \\ 5 & 4 \end{vmatrix}$

`A * 3` = $\begin{vmatrix} 3x & 6 \\ 3 & 9 \end{vmatrix}$

`A * B` = $\begin{vmatrix} 3x+8 & 2x+2 \\ 15 & 5 \end{vmatrix}$

`A ** 2` = $\begin{vmatrix} x^2+2 & 2x+6 \\ x+3 & 11 \end{vmatrix}$

`A ** -1` = $\begin{vmatrix} \frac{-3}{2-3x} & \frac{2}{2-3x} \\ \frac{1}{2-3x} & \frac{-x}{2-3x} \end{vmatrix}$

`B ** -1` = $\begin{vmatrix} -\frac{1}{5} & \frac{2}{5} \\ \frac{4}{5} & -\frac{3}{5} \end{vmatrix}$ `(B ** -1).evalf()` = $\begin{vmatrix} -0.2 & 0.4 \\ 0.8 & -0.6 \end{vmatrix}$

`A.det()` = $3x-2$

`B.det()` = -5

Matrice aumentata Comunemente, per risolvere un sistema di equazioni lineari, si moltiplica l'inversa della matrice dei coefficienti per il vettore dei termini noti e si ottiene il vettore dei valori delle incognite.

Dato il sistema di equazioni

$$\begin{cases} 3x + 4y = 25 \\ 2x + y = 20 \end{cases}$$

possiamo creare la matrice dei coefficienti

`A = Matrix([[3,4],[2,1]])`
il vettore colonna dei termini noti
`b = Matrix([25,20])`
ed eseguire
`A**-1 * b`
ottenendo il vettore $\begin{vmatrix} 11 \\ -2 \end{vmatrix}$ della soluzione: x = 11 e y = -2

Se usiamo un foglio di calcolo o altri moduli Python come NumPy e SciPy è questo il modo con cui procediamo alla soluzione dei sistemi di equazioni lineari.

SymPy ci offre un modo originale, avvalendosi di una funzione che utilizza la matrice aumentata `M = [A|b]` del sistema di equazioni, una matrice che contiene sia la matrice dei coefficienti sia il vettore dei termini noti.
Nel caso del sistema sopra ipotizzato costruiamo questa matrice aumentata così
`A = Matrix([[3,4,25],[2,1,20]]),`
ottenendo A = $\begin{vmatrix} 3 & 4 & 25 \\ 2 & 1 & 20 \end{vmatrix}$

La funzione prevista per calcolare la soluzione utilizzando la matrice aumentata è
`solve_linear_system()`
i cui argomenti sono, separati da virgole, il nome della matrice aumentata e il simbolo delle incognite.
Nel nostro caso `solve_linear_system(A, x, y)`
che produce il risultato nel dizionario `{x: 11, y: -2}`

5.4.6 Integrazione con Python

In questo capitolo vedremo alcuni esempi di come il modulo SymPy si possa utilmente integrare in script Python.

Risoluzione di equazioni

Nel paragrafo 4.4.4, parlando della funzione `solve()` di SymPy, ho detto che questo formidabile strumento, per chi sia interessato a vedere le soluzioni di una equazione espresse in bei numeri, ha il difetto di essere molto grezzo nel presentare le soluzioni che trova. Peraltro non bisogna dimenticare che SymPy è innanzi tutto uno strumento di calcolo simbolico e i risultati che ritornano le sue funzioni devono essere utilizzabili

per ulteriori elaborazioni, meglio facendolo in maniera diretta senza inutilmente passare attraverso intermedie conversioni numeriche ottenute con approssimazioni. Ed è per questo che `solve()` non si preoccupa di ottenere sempre e comunque risultati numerici, anche quando ha a che fare con soli numeri.

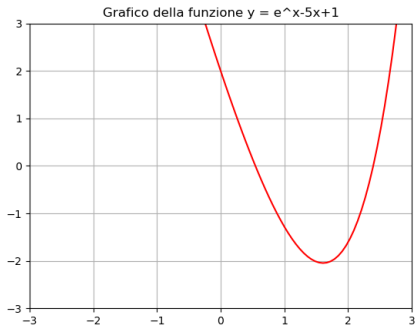
Questi sono esempi di come `solve()` si esprime.
Per l'equazione $x^2 - 2x - 3 = 0$, `solve(x**2-2*x-3, x)`, dal momento che le radici sono numeri belli e buoni, ritorna la lista di soluzioni [-1, 3];
per l'equazione $3x - 2 = 0$, `solve(3*x-2, x)`, dal momento che la radice è una frazione che richiede un'approssimazione, ritorna la soluzione [2/3];
per l'equazione $3^{x-1} - 4 = 0$, `solve(3**(x-1)-4, x)` ritorna la soluzione [log(12)/log(3)];
per l'equazione $e^x + 5a + 1$, `solve(exp(x)+5*a +1, x)` ritorna la soluzione [log(-5*a - 1)];
per l'equazione $x + log(x)$, `solve(x+log(x), x)` ritorna la soluzione [LambertW(1)], la più ermetica di tutte.

Se siamo interessati a soluzioni espresse in numeri, anche per quelle strane equazioni che hanno radici non immediatamente esprimibili in numero salvo approssimazioni di calcolo, possiamo costruirci uno script come questo, che utilizza la funzione `solve()` di SymPy:

```
from sympy import *
def avvio():
    print('Inserisci la parte sinistra di una equazione uguagliata a zero')
    print("(usa la sintassi Python e nomina con x l'incognita)")
    global espressione
    espressione = input()
    risolvi()
def risolvi():
    x = symbols('x')
    e = sympify(espressione)
    l = solve(e, x)
    n = len(l)
    for i in range(n):
        s = N(l[i-1])
        print('soluzione', i+1, ':  ')
        pprint(s)
    print("Vuoi risolvere un'altra equazione?  (si/no)")
    r = input()
    if r == 'si' or r == 'SI' or r == 'Si':
        avvio()
    else:
        input('Premi INVIO per terminare')
avvio()
```

In carattere corsivo è evidenziato il codice che utilizza il modulo SymPy, in carattere diritto è evidenziato il normale codice Python.
Lo script, dopo aver chiesto all'utente di inserire l'equazione da risolvere, la risolve utilizzando la funzione `solve()` di SymPy e poi applica la funzione `N()` di SymPy a ciascuna delle soluzioni. In questo modo potremo vedere le soluzioni espresse in formato numerico, pur conservando l'eventuale presenza di simboli non numerici.
Per le equazioni viste prima avremo questi risultati:
per l'equazione $x^2 - 2x - 3 = 0$
soluzione 1 : 3.00000000000000
soluzione 2 : -1.00000000000000
per l'equazione $3x - 2 = 0$
soluzione 1 : 0.666666666666667
per l'equazione $3^{x-1} - 4 = 0$
soluzione 1 : 2.26185950714292
per l'equazione $e^x + 5a + 1 = 0$
soluzione 1 : log(-5*a - 1)
per l'equazione $x + log(x) = 0$
soluzione 1 : 0.567143290409784
e, per altre, a volontà:
per l'equazione $(1 - sin(x))^2 + cos(x) = 0$
soluzione 1 : 0.785398163397448 + 1.38432969165679*I
soluzione 2 : 1.57079632679490
soluzione 3 : 0.785398163397448 - 1.38432969165679*I
per l'equazione $(1 + x)^3 - 2 = 0$
soluzione 1 : -1.62996052494744 + 1.09112363597172*I
soluzione 2 : 0.259921049894873
soluzione 3 : -1.62996052494744 - 1.09112363597172*I
per l'equazione $x^7 - 3x^4 + 4x^2 - x = 0$
soluzione 1 : 1.09802749885925 + 0.346632626761893*I
soluzione 2 : 0
soluzione 3 : -1.07697753213803
soluzione 4 : 0.263663043733714
soluzione 5 : -0.691370254657094 - 1.47587345054391*I
soluzione 6 : -0.691370254657094 + 1.47587345054391*I
soluzione 7 : 1.09802749885925 - 0.346632626761893*I
per l'equazione $ax^2 + bx + c = 0$
soluzione 1: $\frac{-0{,}5(b+(-4ac+b^2)^{0{,}5})}{a}$
soluzione 2: $\frac{0{,}5(-b+(-4ac+b^2)^{0{,}5})}{a}$
che sono un altro modo di scrivere la nota formula risolutiva

$\frac{-b \pm \sqrt{b^2-4ac}}{2a}$
per l'equazione $e^x - 5x + 1 = 0$
soluzione 1 : 0.544880440159982
Come si vede, se non ci sono di mezzo altri simboli, ora le soluzioni sono espresse come numeri.
Purtroppo in molti casi abbiamo soluzioni reali mischiate con soluzioni complesse e questo disturba un tantino.
La cosa peggiore è che per l'ultima equazione ($e^x - 5x + 1 = 0$) ci viene indicata una sola radice, quando le radici sono due, come si può vedere dal seguente grafico:



una tra 0 e 1, quella trovata da `solve()`, l'altra tra 2 e 3, che non è stata evidenziata.

* * *

Dal momento che SymPy è bravo a calcolare le derivate, possiamo costruire noi un risolutore, sempre integrando il nostro script con SymPy, che, invece di sfruttare la funzione `solve()`, sfrutta quest'altra sua capacità.

Esiste infatti un metodo per risolvere equazioni di qualsiasi tipo, detto metodo delle tangenti, ideato da Newton[2] , per applicare il quale occorre calcolare la derivata dell'espressione che rappresenta l'equazione.

Questo script è un esempio di applicazione di questo metodo.

```
from sympy import *
def avvio():
    print('Inserisci la parte sinistra di una equazione uguagliata a zero')
    print("(usa la sintassi Python e nomina con x l'incognita)")
```

[2]Un'ottima esposizione di questo metodo, detto metodo di Newton-Raphson, si trova su Wikipedia

```
    global espressione
    espressione = input()
    global s
    s = 0.1
    risolvi()
def risolvi():
    ss = 0
    test = 0
    contatore = 0
    while 1:
        x = symbols('x')
        global s
        e = sympify(espressione)
        de = diff(e, x)
        ss = s - N(e.subs(x,s))/N(de.subs(x,s))
        test = s
        s = ss
        contatore = contatore + 1
        try:
            if int(ss*1000000) == int(test*1000000):
                break
        except TypeError:
            str(ss)
            ss = 'Non ho trovato alcuna soluzione reale'
            break
        if contatore == 100:
            str(ss)
            ss = 'Non ho trovato alcuna soluzione reale'
            break
    print('soluzione:  x = ', ss)
    print()
    print("Vuoi cercare altre soluzioni per la stessa equazione?(si/no)")
    r1 = input()
    if r1 == 'SI' or r1 == 'Si' or r1 == 'si':
        ripeti()
    elif r1 != 'SI' or r1 != 'Si' or r1 != 'si':
        print("Vuoi cercare soluzioni per un'altra equazione?(si/no)")
        r2 = input()
        if r2 == 'SI' or r2 == 'Si' or r2 == 'si':
            avvio()
        else:
            input('Premi INVIO per terminare')
def ripeti():
    print('Inserisci una radice tentativo')
    global s
    s = float(input())
    risolvi()
avvio()
```

E' evidenziato in carattere corsivo il codice che utilizza il modulo SymPy e in carattere diritto è evidenziato il normale codice Python.
Dopo aver importato il modulo chiediamo all'utente di inserire l'equazione e la acquisiamo come stringa Python (espressione).
Indi definiamo una variabile globale `s`, di fondamentale importanza per trovare le radici della nostra equazione; la definiamo come variabile globale in quanto dovremo utilizzarla in più di una funzione.
Questa variabile rappresenta un tentativo di soluzione e viene utilizzata per avviare un ciclo nel quale, a partire da questo tentativo, si genera un'altra variabile tentativo (`ss`) e si continua così fino a quando tra un tentativo e l'altro non vi è miglioramento del risultato: ciò significa che abbiamo trovato la soluzione.
Ho inizializzato la variabile con il valore 0,1 che è molto adatto per la soluzione di equazioni semplici e per trovare una prima radice di quelle più complicate.

Seguono poi due funzioni, la prima, `risolvi()`, dedicata a trovare una prima soluzione, la seconda, `ripeti()`, richiamabile a volontà, dedicata a trovare altre soluzioni.
Nella funzione `risolvi()`, innanzi tutto inizializziamo le variabili che ci servono per le iterazioni che prevede il metodo di Newton: `ss` è la soluzione «tentativo» che, in alternanza con la soluzione tentativo `s` predefinita come variabile globale, alimenterà le iterazioni, test ci serve per valutare quando siamo arrivati con buona approssimazione ad una soluzione vera e contatore serve per contare le iterazioni e poter fermare il programma una volta raggiunto un numero di iterazioni tale da escludere che si possano trovare soluzioni ed evitare il loop per ricercarle inutilmente.
Indi avviamo il ciclo: dichiariamo il simbolo `x`, che rappresenta l'incognita, dichiariamo di voler utilizzare la variabile globale `s`, trasformiamo in espressione SymPy (`e`) la stringa Python che rappresenta l'equazione (passaggio necessario per poter utilizzare in seguito il metodo `subs()`) e calcoliamo la derivata (`de`) della funzione sottesa nell'espressione.
A questo punto comincia il ciclo vero e proprio, durante il quale si inseriscono nelle espressioni che rappresentano la funzione e la sua derivata, il cui rapporto è il cuore della formula di Newton, una via l'altra, le soluzioni «tentativo» e ci si ferma quando si trova una soluzione (`ss`) che non migliora più rispetto al test (che contiene la soluzione precedente `s`).
Assumiamo così `ss` come soluzione e la stampiamo.
Notare come il rapporto tra funzione e derivata sia effettuato sulle espressioni numeriche approssimate di queste attraverso la funzione `N()`, ad

evitare che, trattandosi di espressioni SymPy, la loro valorizzazione senza la forzatura della traduzione numerica conservi un simbolo numerico: il che manderebbe in loop il nostro programma in quanto non si arriverebbe mai ad una soluzione.
Ho infine inserito un paio di controlli per evitare loop quando non esistono soluzioni reali dell'equazione.

Alla fine del ciclo possiamo scegliere se ricercare altre soluzioni: nel qual caso ricorreremo alla funzione `ripeti()`, che semplicemente ci consente di modificare la variabile `s` che rappresenta la soluzione tentativo per avviare nuovamente la ricerca con la funzione `risolvi()` da un'altra posizione.

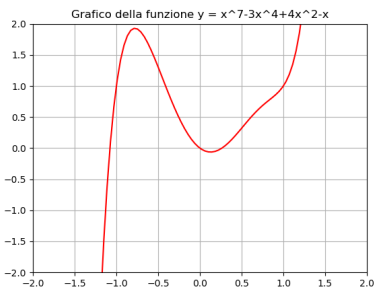
Con questo script possiamo calcolare le radici di una qualsiasi equazione, purché sia numerica e non contenga altri simboli oltre quello dell'incognita `x`.

Rispetto allo script che abbiamo visto prima, oltre a questa limitazione (l'altro script risolve anche equazioni simboliche), questo script ci indica una sola soluzione, la prima che trova con l'iterazione a partire dalla soluzione tentativo, e per trovare le altre occorre ritentare con un'altra soluzione tentativo.

Se passiamo l'equazione $x^7 - 3x^4 + 4x^2 - x$ con la soluzione tentativo di default (variabile s uguale a 0,1) otteniamo la radice 0.
Se ritentiamo modificando la soluzione tentativo in 1, otteniamo la radice 0.263663043733714.
Se ritentiamo modificando la soluzione tentativo in -1, otteniamo la radice -1.07697753213803.

La soluzione tentativo, come si vede, deve essere il più vicino possibile alla soluzione vera.

Il problema è sapere prima quante e pressapoco dove si trovano le radici di un'equazione ricorrendo a un grafico: in ciò ci possiamo aiutare, per esempio, con gnuplot.
Nel caso dell'esempio appena visto, il grafico



ci fa vedere che esistono tre radici reali, in quanto il grafico attraversa l'asse orizzontale 0 in tre punti, uno attorno all'ascissa -1, uno attorno all'ascissa zero e uno con ascissa tra 0 e 0,5.
Con il nostro script siamo arrivati a calcolare i valori numerici di queste radici.

Allo stesso modo possiamo calcolare quella radice che lo script precedente non evidenziava per l'equazione $e^x - 5x + 1 = 0$.
Se passiamo a questo nuovo script l'equazione con la soluzione tentativo di default 0,1 scoviamo la radice 0.544880440159981 che avevamo già trovato prima.
Se ritentiamo con la soluzione tentativo 3 scoviamo l'altra radice
2.39613850607607
che prima non era stata evidenziata.

Strumenti di analisi

Se vogliamo avere sottomano un agile strumento per calcolare derivate, integrali e limiti possiamo ricorrere a questo script:

```
from sympy import *
x = symbols('x')
def menu():
    print("Cosa ti interessa calcolare?")
    print(" digita 1 per la derivata")
    print(" digita 2 per l'integrale indefinito")
    print(" digita 3 per l'integrale definito")
    print(" digita 4 per il limite")
    global scelta
    scelta = int(input())
    if scelta == 0 or scelta > 4:
        print("Scelta sbagliata")
        menu()
    elif scelta == 1:
        derivata()
    elif scelta == 2:
        integrale_indefinito()
    elif scelta == 3:
        integrale_definito()
    elif scelta == 4:
        limite()
def derivata():
    print('Inserisci la funzione di x da derivare, con sintassi Python')
    espressione = input()
    f = sympify(espressione)
    d = diff(f, x)
    print('la derivata di:')
    pprint(f)
```

```
    print('è:')
    pprint(d)
    seguito()
def integrale_indefinito():
    print('Inserisci la funzione da integrare, con sintassi Python')
    espressione = input()
    f = sympify(espressione)
    i = integrate(f, x)
    print("l'integrale indefinito di:")
    pprint(f)
    print('è:')
    pprint(i)
    seguito()
def integrale_definito():
    print('Inserisci la funzione da integrare, con sintassi Python')
    espressione = input()
    print('Inserisci il limite inferiore (sinistro)')
    li = float(input())
    print('Inserisci il limite superiore (destro)')
    ls = float(input())
    f = sympify(espressione)
    i = integrate(f, (x, li, ls))
    print("l'integrale definito tra", li, 'e', ls, 'di:')
    pprint(f)
    print('è:')
    pprint(N(i))
    seguito()
def limite():
    print('Inserisci la funzione di cui trovare il limite, con sintassi Python')
    espressione = input()
    print('Inserisci a cosa deve tendere la variabile (oo per infinito)')
    t = input()
    f = sympify(espressione)
    l = limit(f, x, t)
    print('Il limite di:')
    pprint(f)
    print('per x tendente a', t, 'è:')
    pprint(N(l))
    seguito()
def seguito():
    print("")
    print("Vuoi fare altro?")
    print("Rispondi SI per proseguire")
    opzione = input()
    if opzione == "si" or opzione == "SI" or opzione == "Si":
        menu()
menu()
print("PREMI INVIO PER USCIRE DAL PROGRAMMA")
input()
```

In carattere corsivo abbiamo il codice per il modulo SymPy e in carattere diritto il codice Python generico.
E' un utile esercizio che si spiega da sé.

* * *

Giunto alla fine di questo capitolo illustrativo di SymPy sento il dovere di avvertire il lettore che SymPy non è solo quello che abbiamo visto qui.

Ciò che ho illustrato qui è solo una parte delle moltissime cose che possiamo fare con SymPy.

La parte che ho selezionato ritengo sia quella che riguarda la matematica più comunemente praticata: trattamento delle espressioni, delle equazioni, qualche strumento di analisi.

Ed anche nel trattare la parte selezionata ho tralasciato alcuni aspetti e modalità che ho ritenuto complicanti per un semplice dilettante, sia pure evoluto. Non ho accennato, per esempio alle equazioni alle differenze finite ed alle equazioni differenziali, ecc.

Per chi voglia di più è disponibile la documentazione alla cui esistenza ho accennato fin dalla premessa: oltre ad approfondimenti su ciò che è stato visto in questo capitolo si potranno trovare strumenti per la matematica combinatoria, per la geometria, per il calcolo numerico e tanto altro.

Capitolo 6

Python per la data science

Nelle varie fasi del metodo scientifico ne troviamo due che hanno a che fare con dati: quella delle osservazioni iniziali che supportano la formulazione di una ipotesi e quella dell'analisi dei risultati della sperimentazione dell'ipotesi in modo da decidere se l'ipotesi stessa sia da accettare o da scartare.

Tradizionalmente, sia i dati delle osservazioni iniziali sia i dati della sperimentazione sono disponibili in quantità limitata, in quelle entità che gli statistici chiamano campioni.

E la statistica ci offre strumenti per valutare quanto sia attendibile fondare congetture e derivare conoscenza da quantità limitate di dati e il sotto-modulo `stats` di SciPy, che abbiamo visto nel precedente Capitolo, ci dà modo di fruire di strumenti di questo tipo.

Da inizio secolo, da quando il progresso tecnologico e la digitalizzazione hanno reso facile e relativamente poco costoso raccogliere ed archiviare grandi moli di dati, si è consolidata una nuova disciplina, ormai ufficialmente battezzata come *data science*, che non ha tanto l'obiettivo di trarre dall'analisi di dati all'uopo raccolti conferme a ipotesi scientifiche quanto quello di trarre da dati comunque disponibili indicazioni di varia natura su azioni e comportamenti attinenti le materie cui i dati si riferiscono.

La statistica tradizionale opera prevalentemente al servizio di laboratori di ricerca (sociologica, medica, scientifica in genere) a supporto della formulazione di ipotesi scientifiche, la data science opera prevalentemente al servizio di aziende che intendano trarre dall'esame dei dati indicazioni sulla gestione del business.

Gli strumenti di cui si avvalgono le due discipline sono molto spesso gli stessi ma, in aggiunta a tutto ciò che abbiamo visto nel precedente

capitolo, la data science ha bisogno di
. strumenti per la manipolazione e il trattamento di grandi moli di dati,
. strumenti per il trattamento di dati qualitativi oltre che quantitativi,
. strumenti per rendere possibile trarre conoscenza dai dati anche in mancanza di algoritmi espliciti.

Alla prima esigenza Python risponde con il modulo **Pandas**, che offre strutture dati e operazioni per manipolare tabelle numeriche e serie di dati.

A fronte della seconda esigenza abbiamo il modulo **NLTK**, suite di strumenti per l'analisi simbolica e statistica nel campo dell'elaborazione del linguaggio naturale (parole anziché numeri).

A fronte della terza esigenza abbiamo il modulo **Scikit-learn**, libreria di apprendimento automatico.

6.1 Pandas

Pandas sta per Panel Data.

All'indirizzo *https://pandas.pydata.org/* troviamo la documentazione ufficiale in lingua inglese e cliccando sul pulsante INSTALL PANDAS NOW ci vengono descritti vari modi per installare Pandas, primo fra tutti quello che utilizza Anaconda.

Noi ci accontentiamo di farlo con pip.

Quanto alla documentazione, ciò che vedremo qui si riferisce ai primi rudimenti: il file PDF che troviamo sul sito all'indirizzo prima citato si sviluppa su 3509 pagine e questo la dice lunga circa ciò che si possa fare con Pandas.

Per usare Pandas dobbiamo importarlo nello script e, fedeli alla tradizione instaurata in questo manuale, lo facciamo con
`import pandas as pd`.

Con Pandas possiamo creare due oggetti che corrispondono a due strutture di dati: la serie e il frame.

6.1.1 Serie

La serie è un elenco etichettato di dati di qualsiasi tipo. Le etichette ne costituiscono l'indice.

La serie si costruisce con la funzione `Series` di Pandas, con la sintassi
`Series(<dati>, index = <lista>)`
dove <dati> può essere una lista, un ndarray, uno scalare o un dizionario e <lista> è una lista.

Se <dati> è un dizionario si deve omettere l'argomento index in quanto le etichette indice vengono assunte dal dizionario.

Se <dati> non è un dizionario e si omette l'argomento index le etichette vengono assegnate come numeri interi partendo da zero.

Esempi:
Avendo importato Pandas con `import pandas as pd`,

```
pd.Series([12, 45, 'a'], index = ['a', 'b', 'c'])
```
crea la serie
```
a     12
b     45
c     a
```

```
pd.Series({'x':45, 'y':15, 'z':22, 'w':12})
```
crea la serie:
```
x     45
y     15
z     22
w     12
```

```
pd.Series([15, 22, 13])
```
crea la serie
```
0     15
1     22
2     13
```

```
pd.Series([15, 22, 13], index = [1, 2, 3])
```
crea la serie
```
1     15
2     22
3     13
```

```
pd.Series(8, index = [10, 20, 30])
```
crea la serie
```
10     8
20     8
30     8
```

Con la funzione di Python di base `len` si ottiene il numero di elementi della serie con la sintassi
`len(`<serie>`)`.

Ad un elemento della serie si accede attraverso l'indice con la sintassi <serie>`[`<indice>`]`.

Il valore di un elemento può essere modificato accedendo ad esso ed assegnandogli un altro valore.

Si può aggiungere un elemento a una serie assegnandone il valore ad una nuova posizione di indice.

Esempi:
Costruita la serie s = pd.Series([12,8,22,15], index = [1,2,3,4])

`len(s)` ritorna `4`
`s[3]` ritorna `22`
`s[2] = 16` sostituisce nel secondo elemento il valore `16` al valore `8`
`s[5] = 18` aggiunge un quinto elemento con indice `5` e valore `18`.

Per togliere un elemento da una serie abbiamo la funzione
`drop([<indice_elemento>])`
che toglie l'elemento indicato senza tuttavia modificare la serie originaria.

Data la serie s di prima, comprensiva del quinto elemento aggiunto nell'esempio,
con `s.drop([5])` la leggiamo ignorando il quinto elemento,
con `s = s.drop([5])` la modifichiamo togliendo il quinto elemento.

Per certi versi, la serie di Pandas è simile a un ndarray di Numpy e, data la serie `s` degli esempi, se in una IDLE di Python scriviamo `s.` apriamo il lungo elenco delle funzioni dell'oggetto Serie di Pandas e vi troviamo le funzioni `min`, `max`, `mean`, `std`, `sum`, `prod`, `cumsum` che abbiamo visto nel Paragrafo 4.1.1, ovviamente operanti su serie numeriche.

Differente la funzione `sort()` che qui si presenta nella doppia veste di
`sort_index()` per eseguire l'ordinamento in base all'indice e
`sort_values()` per eseguire l'ordinamento in base al valore,
in ogni caso solo in lettura senza modificare la serie originaria.

Per quanto riguarda le operazioni matematiche con gli operatori `+ - * / **`, mentre nel caso degli ndarray di Numpy esse sono possibili solo tra array delle stesse dimensioni, nel caso delle serie di Pandas esse sono sempre possibili, fermo restando che ove non vi sia possibilità di instaurare l'operazione membro a membro a causa della differente dimensione delle serie il risultato sarà NaN (Not a Number).

Ovviamente è possibile moltiplicare una serie Pandas per uno scalare, ottenendo una nuova serie formata dai membri di quella di partenza moltiplicati per lo scalare.

Esempi:
Date le serie
```
a = pd.Series((1,2,4))
b = pd.Series((3,1))
```
`a + b` ritorna
```
0    4.0
1    3.0
2    NaN
```
`a ** b` ritorna
```
0    1.0
1    2.0
2    NaN
```

```
    a * 5 ritorna
0     5
1     10
2     20
```

Dal momento che Pandas nasce per trattare grandi moli di dati (i così detti big data), può accadere che una sua serie contenga centinaia o migliaia di elementi.

Per ispezionarne semplicemente l'inizio o la fine evitando di stampare tutta la serie abbiamo a disposizione le funzioni
`head()` che ci fa vedere i primi cinque elementi,
`tail()` che ci fa vedere gli ultimi cinque elementi.
Come avviene per le altre funzioni, esse si richiamano facendole precedere dal nome della serie e da un punto.

6.1.2 Frame

Il frame è una tabella con righe e colonne etichettate.

Il frame si costruisce con la funzione `DataFrame` di Pandas, con la sintassi
`DataFrame(`<dati>`, columns =` <lista>`, index =` <lista>`)`
dove
<dati> può essere una lista di tuple, un dizionario o un ndarray bidimensionale.
<lista> è una lista con la quale si danno nomi alle colonne (`columns`) e alle righe (`index`) della tabella.

Esempi:
Avendo importato Pandas con `import pandas as pd`,
possiamo ottenere il seguente frame

	altezza	peso
Vittorio	1,81	92
Luigi	1.78	77

con
```
pd.DataFrame([(1.81,92),(1.78,77)], columns=['altezza','peso'],
                                        index=['Vittorio','Luigi'])
```
oppure con
```
pd.DataFrame({'altezza':(1.81,1.78),'peso':(92,77)},
                                        index=['Vittorio','Luigi'])
```
oppure, disponendo di un ndarray

$$a = \begin{matrix} 1.81 & 92 \\ 1.78 & 77 \end{matrix}$$

con
`pd.DataFrame(a,columns=['altezza','peso'], index=['Vittorio','Luigi'])`

Se non si indicano gli argomenti `columns` e/o `index`, essi vengono automaticamente inseriti con numeri interi a partire da zero.

Con la sintassi
<frame>`[`<nome_colonna>`]`
si accede ad una singola colonna del frame, la quale è, in tutto e per tutto, una serie di Pandas ed è dotata delle relative funzioni.

Esempi:
Posto che il frame degli esempi precedenti sia in una variabile nominata df,
`df['altezza'].mean()` ritorna `1.795`
`df['peso'].max()` ritorna `92`

Per accedere ad un singolo dato del frame abbiamo a disposizione le funzioni
`loc[`<nome_riga>`][`<nome_colonna>`]` seleziona per etichetta,
`iloc[`<numero_riga>`][`<numero_o_nome_colonna>`]` seleziona per posizione,
sapendo che righe e colonne sono intrinsecamente numerate partendo da zero.

Esempi:
Sempre posto che il frame degli esempi precedenti sia in una variabile nominata df,
`df.loc['Luigi']['peso']` ritorna `77`
`df.iloc[0][0]` ritorna `1.81`
`df.iloc[1]['altezza']` ritorna `1.78`

Con la funzione loc possiamo aggiungere righe al frame con la sintassi
`loc[`<numero_o_nome_riga>`] =` <lista_valori>
od anche colonne con la sintassi
`loc[:,`<numero_o_nome_colonna>`] =` <lista_valori>

`df.loc['Giuseppe']=[1.90,88]` aggiunge l'altezza e il peso di Giuseppe al nostro dataframe `df`
`df.loc[:,'massa']=df['peso']/(df['altezza']*df['altezza'])` aggiunge al dataframe df la nuova colonna contenente la massa corporea calcolata secondo la formula peso/altezza^2

Ora il frame df è così

	altezza	peso	massa
Vittorio	1.81	92	28.082171
Luigi	1.78	77	24.302487
Giuseppe	1.90	88	24.376731

Possiamo leggere il frame cancellando righe e colonne con la funzione `drop()` e la seguente sintassi
`drop([<numero_o_nome_riga>])` per cancellare una riga,
`drop([<numero_o_nome_colonna>], axis = 1)` per cancellare una colonna.
Entrambe le funzioni operano solo in lettura e ciò che ritornano può essere inserito in un'altra variabile o, per modificare il frame di partenza, nella stessa variabile che lo conteneva.

Possiamo anche leggere il frame filtrando i dati con la sintassi
<frame>`[`<condizione>`]`.

Con riferimento al nostro frame df possiamo estrarre i sovrappeso (massa corporea maggiore di 28) con
`df[df['massa'] > 28]`

Possiamo calcolare i valori medi di chi è più alto di 1.80 con
`df[df['altezza'] > 1.8].mean()`

Molto spesso i frame vengono alimentati da tabelle esterne contenute in fogli di calcolo, database, ecc.

Pandas contiene molte funzioni ad hoc, tutte con la radice `read_`, ma, posto che da qualsiasi fonte di dati è possibile generare tabelle in formato `.csv` (comma separated value), da principianti ci possiamo accontentare della funzione
`read_csv()`
che ha come primo argomento obbligato una stringa con il percorso al file `.csv` contenente i dati e, come successivo argomento opzionale, separato da virgola, `delimiter = '`<simbolo_separatore>`'` utile quando il separatore non sia una virgola (,) ma un punto e virgola (;) o una barra (|), ecc.
La prima riga del file viene automaticamente utilizzata per l'intestazione delle colonne.
Se si vuole evitare questo occorre usare l'argomento opzionale `header = None`. In tal caso le colonne vengono numerate con interi a partire da zero e si possono rinominare con <frame>`.columns =` <lista>.
Le righe vengono numerate con interi a partire da zero. Se si vuole utilizzare una colonna per indicizzare le righe lo si può fare con l'argomento opzionale `index_col =` <numero_o_nome_colonna>.

6.2 NLTK

NLTK sta per Natural Language ToolKit ed è una libreria che permette l'analisi di testi.

Troviamo tutto su NLTK all'indirizzo *http://www.nltk.org/*, in particolare troviamo il manuale completo all'indirizzo *https://www.nltk.org/book/*.

Come al solito installiamo NLTK con pip.

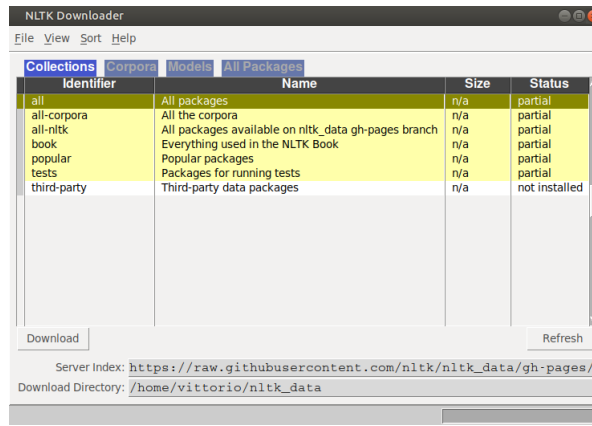
Per usarlo lo importiamo con la solita sintassi

`import nltk as nl`.

Quello che installiamo con pip è il modulo base e può essere arricchito richiamandone la funzione download con il seguente comando nella shell di Python o nell'IDLE

`nl.download()`

che apre la seguente finestra



Si tratta dell'accesso ad un repository che contiene vari tools e testi predisposti per esercitarsi con NLTK.

Per ciò che faremo qui non utilizziamo questo downloader ma, per ciò che dobbiamo caricare, ci basta usare il comando `nl.download()` mettendo tra le parentesi la stringa con il nome di ciò che ci serve.

6.2.1 Analisi del testo

Per analizzare un testo con NLTK dobbiamo predisporlo con la funzione `Text()`.

Un testo analizzabile si crea passando a questa funzione, come argomento, non già una semplice stringa ma un testo sotto forma di lista di token, cioè di entità linguistiche separate (parole o frasi).

Abbiamo a disposizione due funzioni per ottenere questo

`word_tokenize()` trasforma un testo in una lista di parole,

`sent_tokenize()` trasforma un testo in una lista di frasi.

Prima di usare con profitto queste funzioni dobbiamo però installare un package aggiuntivo al nostro NLTK di base e lo facciamo, previa importazione di NLTK in una shell Python o nella IDLE con
`import nltk as nl`,
proseguendo con
`nl.download('punkt')`.

Da qui in poi il nostro NLTK di base è arricchito con punkt e potremo fare le nostre tokenizzazioni come si deve.

Seguono un paio di esempi.
`nl.word_tokenize('Oggi è una bella giornata.  Ieri era peggio')` ritorna
`['Oggi', 'è', 'una', 'bella', 'giornata', '.', 'Ieri', 'era', 'peggio']`,
`nl.sent_tokenize('Oggi è una bella giornata.  Ieri era peggio')` ritorna
`['Oggi è una bella giornata.', 'Ieri era peggio']`.

Un testo analizzabile con NLTK partendo da una stringa di testo si crea con `word_tokenize`, così:
`testo = nl.Text(nl.word_tokenize(<stringa>)`.

L'oggetto, qui denominato testo, così creato possiede i metodi che ci consentono di analizzarlo e ne troviamo l'elenco scrivendo, nella IDLE, il nome dell'oggetto seguito da un punto.

Cito i principali:
. `concordance(<parola>)` ricerca una parola nel testo e ne mostra il contesto,
. `collocations()` indica le sequenze di parole che nel testo appaiono molto spesso assieme,
. `dispersion_plot(<lista_di_parole>)` mostra graficamente la distribuzione di parole nel testo.

Inoltre possiamo utilizzare i normali metodi del linguaggio Python applicabili alle liste (`len` per ottenere il numero dei token, ricerca di parole utilizzando indici, ecc.).

Infine NLTK ha una funzione che ci fornisce la distribuzione di frequenza delle parole presenti nel testo, con la sintassi
`FreqDist(<nome_testo>)`
potendo indicare con `n` la quantità da mostrare con la sintassi
`FreqDist(<nome_testo>).most_common(n)`.

Esempio:
Ho un file di testo che contiene l'Addio di Lucia ai monti dal Capitolo VIII dei Promessi Sposi e lo voglio esaminare.
Innanzi tutto lo apro con
`f = open('/home/vittorio/Documenti/addio.txt')`
poi dopo `import nltk as nl`, lo rendo adatto all'analisi con NLTK con
`testo = nl.Text(nl.word_tokenize(f.read().lower()))`

(il richiamo della funzione `lower()` serve se si vuole che tutte le parole inizino con la minuscola in quanto alcune funzioni, come la `dispersion_plot()`, distinguono tra maiuscole e minuscole e possono fornire risultati parziali se non se ne tiene conto).
Ora cominciamo l'analisi.

`testo.concordance('addio')`
ritorna la seguente schermata

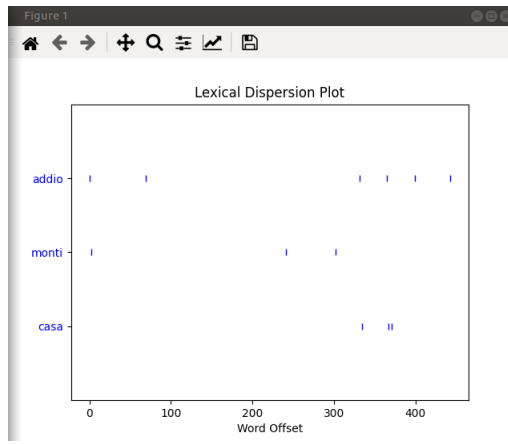
```
Displaying 6 of 6 matches:
 addio , monti sorgenti dall'acque , ed ele
 , come branchi di pecore pascenti ; addio ! quanto è tristo il passo di chi ,
n momento stabilito per il ritorno ! addio , casa natìa , dove , sedendo , con
aspettato con un misterioso timore . addio , casa ancora straniera , casa soggu
rno tranquillo e perpetuo di sposa . addio , chiesa , dove l'animo tornò tante
enir comandato , e chiamarsi santo ; addio ! chi dava a voi tanta giocondità è
```

che mostra le 6 ricorrenze della parola addio nel loro contesto.

`testo.collocations()`
ritorna la seguente schermata

```
cresciuto tra; tante volte; tra voi; più care
```

testo.dispersion_plot(['addio', 'monti', 'casa'])
ritorna questo grafico



con indicate le posizioni delle parole indicate dalla lista nel testo.

`len(testo)`
ritorna 478, che è il numero di token.
Se contiamo le parole del testo sono 406. I token sono di più perché sono token, oltre alle parole, anche i segni di punteggiatura, gli apostrofi, ecc.

`nl.FreqDist(testo)`
ritorna la seguente schermata

```
FreqDist({',': 54, 'e': 17, ';': 14, 'di': 9, 'a': 8, 'non': 8, 'un': 8, 'che':
7, "'": 7, 'il': 7, ...})
```

che elenca le prime dieci maggiori ricorrenze a dizionario.
`nl.FreqDist(testo).most_common(20)`
ritorna la seguente schermata

```
[(',', 54), ('e', 17), (';', 14), ('di', 9), ('a', 8), ('non', 8), ('un', 8), ('
che', 7), ("'", 7), ('il', 7), ('addio', 6), ('chi', 6), ('più', 6), ('!', 5), (
'si', 5), ('.', 5), ('è', 4), ('de', 4), ('se', 4), ('con', 4)]
```

che elenca le prime venti, questa volta come lista di tuple.

6.2.2 Pulizia del testo

Abbiamo visto nel precedente paragrafo che un testo, se lo suddividiamo nei token che lo compongono, contiene una miriade di simboli (punteggiatura) e ricorrenze (articoli, congiunzioni, ecc.) che nulla aggiungono all'essenza del suo contenuto, servono solo a noi umani per leggerlo meglio e per gustarne le sfumature e, spesso, la musicalità.

Se lo leggiamo con il computer, al fine, per esempio, di stabilire se si tratta di un testo allegro o di un testo triste, come vedremo si possa fare nel prossimo paragrafo, possiamo benissimo lavorare solo sulle parole essenziali ed eliminare tutto ciò che non serve, alleggerendo così il materiale da esaminare.

Per ripulire un testo a questi fini, oltre che attrezzarci per la sua tokenizzazione secondo quanto visto nel paragrafo precedente, dobbiamo procurarci un archivio che contiene i simboli e i termini che possiamo eliminare da un testo senza perderne l'essenza. Nel mondo Python questo archivio si trova in un file denominato `stopwords`.

Lo possiamo installare sul nostro computer in maniera stabile in modo da averlo sempre a disposizione, previa importazione di NLTK in una shell Python o nella IDLE con
`import nltk as nl`,
proseguendo con
`nl.download('stopwords')`.

L'archivio può essere utilizzato per lavorare su testi scritti in una delle oltre venti lingue riconosciute, tanti sono i file delle parole "inutili" che ci vengono proposti.
Bene sapere che il file è un semplice file di testo, con una parola per riga, e lo possiamo modificare con un editor di testo arricchendolo di altre parole o simboli: per esempio questi file non contengono i simboli di punteggiatura (, . : ; ! ? ecc.) e possiamo utilmente aggiungerli indicando ciascuno di essi su altrettante righe.
Il file per la lingua italiana si chiama `italian` e si trova all'indirizzo

/nltk_data/corpora/stopwords/,
essendo la directory nltk_data nella nostra home (se usiamo Linux o Mac) o in AppData\Roaming del nostro percorso utente se usiamo Windows.

Il procedimento di ripulitura consiste nello scorrere i token di un file di testo ricopiando in un nuovo file solo i token non appartenenti alla lista delle stopwords. In questo modo avremo un testo derivato dal precedente e ripulito dei termini che non servono per interpretarne l'essenza.

La lista delle stopwords che ci interessa, per esempio quella della lingua italiana, la rendiamo disponibile per il procedimento con i comandi

```
from nltk.corpus import stopwords
stop_words = set(stopwords.words('italian'))
```

Esempio:
Il seguente script ripulisce il file di testo contenente l'Addio di Lucia che abbiamo utilizzato nel precedente paragrafo.

```
import nltk as nl
from nltk.corpus import stopwords
stop_words = set(stopwords.words('italian'))
f = open('/home/vittorio/Documenti/addio.txt')
testo = nl.Text(nl.word_tokenize(f.read()))
for parola in testo:
    if not parola in stop_words:
        ff = open('/home/vittorio/Documenti/addio_pulito.txt', 'a')
        ff.write(" " + parola)
        ff.close()
```

In questo modo abbiamo creato il file addio_pulito.txt che contiene l'addio di Lucia ridotto a 238 parole, quando il file originario ne contiene 406.

6.2.3 Sentiment analysis

La sentiment analysis è un'analisi del testo finalizzata a stabilire se il testo stesso esprime un sentimento positivo, negativo o neutrale.

Viene utilizzata con profitto per classificare i giudizi espressi dai consumatori su determinati prodotti senza che per questo sia necessario occupare un operatore alla lettura dei giudizi stessi: questi vengono letti e classificati dal computer.

Bell'esempio di intelligenza artificiale. Il trucco, come per ciò che abbiamo visto nel paragrafo precedente, dove il computer eliminava le parole inutili in quanto noi gli abbiamo detto quali sono, sta nell'istruire il computer su quali parole hanno connotazione positiva, quali hanno connotazione negativa e quali hanno connotazione neutrale.

Per fare questo esiste parecchio materiale già predisposto, sia nel mondo Python, sia in altri contesti.

Qui propongo la versione multi-lingua di VADER (Valence Aware Dictionary and sEntiment Reasoner) che funziona solo con collegamento Internet attivo in quanto si avvale delle Google Translate API attraverso la libreria Python `translatte`.

Installiamo una tantum quanto ci serve con pip
`pip` (o `pip3`) `install vader-multi`.

Per compiere l'analisi dobbiamo importare l'analizzatore con
`from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer`
e poi costruire il nostro analizzatore con
`analizzatore = SentimentIntensityAnalyzer()`.

Sottoponendo al metodo `polarity_scores` dell'analizzatore una stringa di testo otteniamo un dizionario che contiene
. in corrispondenza alla chiave `'neg'` un valore indicante la valenza negativa,
. in corrispondenza alla chiave `'neu'` un valore indicante la valenza neutra,
. in corrispondenza alla chiave `'pos'` un valore indicante la valenza positiva,
. in corrispondenza alla chiave `'compound'` un valore di sintesi complessiva.
I primi tre valori sono compresi tra 0 e 1 ed esprimono il peso delle varie valenze.
Il quarto valore, di sintesi complessiva, varia tra -1 e 1 ed esprime una valenza complessiva tanto più positiva quanto più il valore si avvicina a 1, tanto più negativa quanto più il valore si avvicina a -1 e tendente alla neutralità per valori attorno allo zero.

Esempio di sentiment analysis su frasi:

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analizzatore = SentimentIntensityAnalyzer()
analizzatore.polarity_scores('Ho trovato il libro molto noioso')
```
restituisce il dizionario
```
{'neg': 0.341, 'neu': 0.659, 'pos': 0.0, 'compound': -0.3804}
analizzatore.polarity_scores('Oggi è una bellissima giornata')
```
restituisce il dizionario
```
{'neg': 0.0, 'neu': 0.506, 'pos': 0.494, 'compound': 0.5994}
```
Possiamo divertirci anche con altre lingue con lo stesso analizzatore:
```
analizzatore.polarity_scores('VADER is very smart and funny')
```
restituisce il dizionario
```
{'neg': 0.0, 'neu': 0.394, 'pos': 0.606, 'compound': 0.7316}
analizzatore.polarity_scores('Je suis très heureux de vous connaître')
```

restituisce il dizionario
{'neg': 0.0, 'neu': 0.6, 'pos': 0.4, 'compound': 0.6115}

Il testo da analizzare deve essere in forma di stringa e non necessariamente è necessario passare attraverso la tokenizzazione e la ripulitura di cui abbiamo parlato nei paragrafi precedenti.
Nel seguente esempio vediamo come sia possibile determinare il sentiment dell'addio di Lucia che abbiamo in un file di testo e del relativo derivato ripulito che abbiamo in un altro file di testo:

```
f = open('/home/vittorio/Documenti/addio.txt', 'r')
ff = open('/home/vittorio/Documenti/addio_pulito.txt', 'r')
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analizzatore = SentimentIntensityAnalyzer()
analizzatore.polarity_scores(f.read())
{'neg': 0.068, 'neu': 0.761, 'pos': 0.171, 'compound': 0.9954}
analizzatore.polarity_scores(ff.read())
{'neg': 0.101, 'neu': 0.681, 'pos': 0.218, 'compound': 0.9922}
```

La caratterizzazione altamente positiva del testo è confermata sia dall'analisi del testo originario sia dall'analisi del testo ripulito. Nel secondo caso ha lavorato meno l'analizzatore in quanto le parole da analizzare erano in quantità minore.

Ovviamente, in presenza di grandi moli di frasi da analizzare occorre organizzare un set di dati riconducibile ad uno dei tipi contenitori di Python, ad esempio una lista, ed agire su quella.

Esempio:

Poniamo di avere in una lista Python i seguenti giudizi espressi dai lettori di un libro:

```
giudizi = ["Ho letto il libro e l'ho trovato molto noioso", "L'autore ha
confermato la sua maestria nel presentare personaggi e situazioni", "Ho
trovato il libro molto interessante", "Mi verrebbe da dire che ho letto
una schifezza"].
```

Con il seguente script troviamo e stampiamo il risultato dell'analisi:

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analizzatore = SentimentIntensityAnalyzer()
positivi = 0; negativi = 0; neutrali = 0
for x in giudizi:
    sentiment = analizzatore.polarity_scores(x)
    if sentiment['compound'] > 0:
        positivi = positivi + 1
    elif sentiment['compound'] < 0:
        negativi = negativi + 1
    else:
        neutrali = neutrali + 1
print("giudizi positivi = ", positivi)
print("giudizi negativi = ", negativi)
print("giudizi neutrali = ", neutrali)
```

Il risultato è:

```
giudizi positivi = 1
giudizi negativi = 2
giudizi neutrali = 1
```

Evidentemente il risultato positivo è attribuito al terzo giudizio, i due risultati negativi sono attribuiti al primo e al quarto giudizio e risulta neutrale il secondo giudizio.

Nulla vieta, infine, che l'analisi del sentiment possa essere applicata in tempo reale a frasi scritte su twitter e c'è chi lo fa anche a nostra insaputa per tastare il polso su certe situazioni.

Addirittura frasi dette al telefono possono essere analizzate.

Lo script seguente, derivato da quanto descritto nel Capitolo 4, mostra come sia possibile organizzare un dialogo con il computer formulando certe risposte in base al sentiment di certe altre.

```
import pyttsx3
import speech_recognition as sr
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
recitante = pyttsx3.init()
recitante.setProperty('voice', 'italian')
recitante.say('Come ti chiami?')
recitante.runAndWait()
riconoscitore = sr.Recognizer()
with sr.Microphone() as source:
    audio = riconoscitore.listen(source)
testo = riconoscitore.recognize_google(audio, language = "it")
f = open('/home/vittorio/Documenti/saluto.txt', 'w')
f.write('Ciao' + testo + 'come stai?')
f.close()
f = open('/home/vittorio/Documenti/saluto.txt', 'r')
recitante.say(f.read())
recitante.runAndWait()
riconoscitore = sr.Recognizer()
with sr.Microphone() as source:
    audio = riconoscitore.listen(source)
testo = riconoscitore.recognize_google(audio, language = "it")
analizzatore = SentimentIntensityAnalyzer()
giudizio = analizzatore.polarity_scores(testo)
if giudizio['compound'] > 0:
    recitante.say("Mi fa piacere")
    recitante.runAndWait()
else:
    recitante.say("Oh.  Mi dispiace!")
    recitante.runAndWait()
```

Nel colloquio prodotto con questo script, se alla domanda «come stai?» rispondiamo con una frase con sentiment positivo, tipo «Molto bene», il computer risponde, a sua volta, con «Mi fa piacere». Se invece rispondiamo con una frase con sentiment negativo, tipo «Male, ho un

gran dolore ai denti», il computer risponde, a sua volta, con «Oh. Mi dispiace».

6.3 Scikit-learn

Scikit-learn è il modulo Python dedicato al machine learning, la nuova frontiera della scienza dei dati; in italiano si dice apprendimento automatico.

Il termine è stato coniato nel lontano 1959 dallo scienziato americano Arthur Lee Samuel, ma la definizione di ciò che si intende oggi per machine learning è più recente e ce l'ha data un altro americano, Tom M. Mitchell, nel 1997: «Si dice che un programma apprende dall'esperienza E con riferimento ad alcune classi di compiti T e con misurazione della performance P, se le sue performance nel compito T, come misurato da P, migliorano con l'esperienza E.»

Ci si può chiedere come mai una disciplina della quale si parla da oltre cinquant'anni sia diventata attuale solo da poco più di una decina d'anni.

La risposta è semplice: il machine learning ha senso solo in presenza di una grande quantità di dati e necessita di grande capacità di calcolo, cose che cinquant'anni fa non c'erano.

All'indirizzo *https://scikit-learn.org/stable/* troviamo tutto ciò che c'è da sapere su Scikit-learn e dalla home page apprendiamo che questo modulo Python è basato sui moduli NumPy, SciPy e Matplotlib e ce ne vengono indicati i tre gruppi di applicazioni:
. classificazione,
. regressione,
. clustering.

Il primo ci propone metodi per determinare la categoria di appartenenza di un oggetto (ad esempio per identificare messaggi SPAM nella posta elettronica o per il riconoscimento di immagini).

Il secondo ci propone metodi per prevedere valori associabili ad un oggetto (ad esempio ricerca di legami funzionali tra variabili). Anche in SciPy abbiamo la regressione lineare, che è il principale di questi metodi, ma qui c'è molto altro e la stessa regressione lineare è migliore (per esempio funziona meglio in presenza di variabili non standardizzate).

Il terzo ci propone metodi per il raggruppamento di oggetti simili tra loro (ad esempio per segmentare la clientela). Anche in SciPy abbiamo uno di questi metodi, K-means, ma qui abbiamo molto di più e lo stesso

K-means è arricchito dalla possibilità di identificare preventivamente il numero dei gruppi.

Possiamo installare Scikit-learn con pip con il comando
`pip` (o `pip3`) `install scikit-learn`

Il modulo viene importato negli script con il nome abbreviato `sklearn`.

Nella home page di Scikit-learn abbiamo i link alla User Guide e alle API. Basta dare un'occhiata e ci accorgiamo che abbiamo a che fare con qualche cosa che esula dagli interessi e dalle capacità del principiante cui è dedicato questo libro. E già il principiante a cui penso è uno che sa usare un computer ed ha una preparazione di base di matematica generale e statistica.

In questa sede mi limito pertanto ad un paio di esempi, tanto per vedere come funziona.

6.3.1 Esempio di machine learning supervisionato: regressione multipla

Partiamo da un piccolo dataset, di dimensioni sufficienti per capirci, contenuto in un file .csv, nel quale abbiamo una variabile y accostata al manifestarsi di sei diverse terne di variabili x.

y	x1	x2	x3
34	12	11	21
72	26	20	45
87	44	12	58
28	32	14	37
16	21	11	24
97	76	7	75

Ci proponiamo di verificare quale sia il più probabile valore della variabile y in presenza della terna inedita in cui x1 valga 11, x2 valga 22 e x3 valga 33.

Con questo script

```
import pandas as pd
dati = pd.read_csv('/home/vittorio/Documenti/dati.csv')
X = dati[['x1','x2','x3']]
y = dati['y']
from sklearn.linear_model import LinearRegression
regressore = LinearRegression()
regressore.fit(X, y)
print('intercetta', regressore.intercept_)
coefficienti = pd.DataFrame(regressore.coef_,X.columns,columns=['coefficienti'])
print(coefficienti)
```

```
y_calcolati = regressore.predict(X)
from sklearn import metrics
print('R2', metrics.r2_score(y, y_calcolati))
```

otteniamo i seguenti risultati

```
intercetta 2.879968788655681
coefficienti
x1 -2.648351
x2 -2.616547
x3 4.122166
R2 0.972625953228526
```

dove abbiamo l'intercetta (il punto dell'asse ortogonale verticale da cui parte la retta di regressione), i coefficienti delle variabili x e il coefficiente di determinazione, altrimenti chiamato R quadro: se questo indicatore è prossimo all'unità, come accade nel nostro caso, significa che il modello ottenuto interpreta con attendibilità molto elevata le relazioni esistenti tra le variabili.

A questo punto siamo abbastanza sicuri che, in presenza dei valori di x1, x2, x3, rispettivamente 11, 22 e 33, il valore di y sia:

```
2.879968788 - 2.648351 x 11 - 2.616547 x 22 + 4.122166 x 33 = 52.215551788
```

6.3.2 Esempio di machine learning non supervisionato: clusterting

Stiamo parlando di metodi di analisi applicabili a big data ma ancora esemplifico su dimensioni ridotte, per fare prima e capirci meglio.

Abbiamo fatto un'indagine sui nostri clienti - nel caso sono solo 12 ma potrebbero anche essere 120.000 - e siamo venuti a conoscere ciò che vediamo in questa tabella, memorizzata nel file clienti.csv:

cliente	cambio	prezzo	marca	frequenza
Luigi	5	5	5	3
Maria	7	7	4	2
Vittorio	6	5	5	3
Giovanni	6	6	4	2
Beatrice	5	5	4	3
Giuseppe	9	8	2	4
Elena	8	8	2	5
Antonio	9	7	3	4
Paola	10	7	3	4
Mario	9	8	2	4
Carlo	9	10	2	5
Cecilia	10	8	3	3

Le grandezze numeriche variano da 1 (per niente) a 10 (moltissimo).
La colonna «cambio» indica la propensione del cliente a cambiare negozio.
La colonna «prezzo» indica la sensibilità del cliente al prezzo della merce.
La colonna «marca» indica la preferenza del cliente per articoli di marca.
La colonna «frequenza» indica il numero medio di acquisti in un anno.

Aiutati dalla piccola dimensione del data set e dall'ordine con cui ho esposto i dati notiamo subito a occhio che abbiamo un gruppo di clienti, i primi cinque, tendenzialmente quelli che fanno meno acquisti, che hanno una moderata propensione a cambiare negozio, e sono moderatamente sensibili al prezzo e alla marca; questo gruppo si contrappone al gruppo degli altri sette, con molto più elevata propensione a cambiare negozio, molto più sensibili al prezzo, meno sensibili alla marca e mediamente migliori clienti sul piano della frequenza di acquisto.

Tutte cose utili per le nostre strategie di marketing o semplicemente per qualche intervento tendente a fidelizzare meglio i sette clienti più «volatili».

Ma vediamo come anche la macchina si accorge di ciò che abbiamo adocchiato noi utilizzando il più diffuso algoritmo di clustering, K-means, noto anche per la sua leggerezza a sollievo del lavoro della CPU del computer.

Con questo script

```
import pandas as pd
dati = pd.read_csv('/home/vittorio/Documenti/clienti.csv', usecols = ['cambio',
                                'prezzo','marca','frequenza'])
from sklearn.cluster import KMeans
gruppi = KMeans(n_clusters = 2, init = 'k-means++', tol = 0.0001)
gruppi_c = gruppi.fit_predict(dati)
print('raggruppamento clienti:  ', gruppi_c)
print('inerzia:  ', gruppi.inertia_)
```

otteniamo il seguente risultato

```
raggruppamento clienti:  [0 0 0 0 0 1 1 1 1 1 1 1]
inerzia:  21.82857142857143
```

da cui deriviamo che i primi cinque clienti appartengono al cluster 0 e gli altri sette al cluster 1.
L'indicatore chiamato «inerzia» è una somma di errori quadratici medi derivanti dal raggruppamento proposto e, per avere una segmentazione accurata deve essere il più basso possibile. Purtroppo il suo valore assoluto che vediamo dice poco e andrebbe confrontato con altri. Di per sé, comunque, avendo la nostra tabella di partenza dati compresi tra 1 e 10, mi pare che un valore di quasi 22 non possa essere considerato basso: ne deriva che la nostra segmentazione non è gran che, almeno dal punto di

vista scientifico, anche se sul piano pratico pare funzionare abbastanza bene.

* * *

Con questi due esempi notiamo che anche la scrittura del codice con il modulo Scikit-learn si complica abbastanza.

E' per questo motivo che qualcuno ha pensato di regalare ai principianti, e non solo, quanto vedremo nel prossimo capitolo.

Capitolo 7

Python per il deep learning

Nel paragrafo 6.3, parlando del modulo Scikit-learn, abbiamo introdotto il machine learning.

Nel machine learning (apprendimento automatico) un computer, analizzando le caratteristiche di un set di dati o le interrelazioni presenti in un set di dati, elabora autonomamente modelli in grado di riprodurre o prevedere le variazioni dei dati stessi al variare dello scenario di riferimento.

Oltre che di uno strumento per arricchire l'analisi di dati, si tratta di una importante branca dell'intelligenza artificiale, strettamente collegata alla statistica.

Possiamo capire meglio di che cosa stiamo parlando se ripercorriamo le definizioni di talune sfaccettature del machine learning dal punto di vista del tipo di analisi.

Possiamo avere un set di dati costituito da dati relazionati tra di loro e l'apprendimento consiste nel capire quale sia questa relazione in modo da poterla applicare a dati che non fanno parte del set stesso. In questo caso si parla di apprendimento supervisionato.

E' il caso, per esempio, della regressione, dove si stabilisce il più probabile legame funzionale tra una o più variabili indipendenti e una variabile dipendente presenti in un data set, in modo da trovare un modello che ci consenta di prevedere quale sia il più probabile valore della variabile dipendente in corrispondenza a variabili indipendenti non contemplate nel data set.

Altro esempio quello della classificazione, dove, data una certa etichettatura attribuita ad una certa conformazione di dati nel data set, se ne ricavano elementi per etichettare casistiche non contemplate nel data set stesso. Per esempio per classificare come truffaldino un messaggio di

posta elettronica dopo aver assimilato quali siano le caratteristiche di una certa serie di messaggi classificati truffaldini in un data set.

A volte il data set contiene semplicemente dei dati, non relazionati tra loro e non etichettati e, in questo caso, l'apprendimento consiste nel capire talune caratteristiche di questi dati. In questo caso si parla di apprendimento non supervisionato.
E', per esempio, il caso del clustering, dove si cerca di capire come i dati del data set possano essere raggruppati in sotto-insiemi di dati che presentino tra di loro determinate omogeneità.

Per estensione, infine, rientra nel machine learning anche l'apprendimento che avviene senza che vi sia alcun dato per il condizionamento. E' il caso dell'apprendimento con rinforzo, dove l'apprendimento avviene attraverso un meccanismo di ricompensa (per incentivare il compimento di azioni virtuose) e penalità (per scoraggiare azioni indesiderate) in un contesto di interazione con l'ambiente esterno.
E' per esempio ciò che avviene per far funzionare gli assistenti al parcheggio presenti su certe vetture. In questo caso l'apprendimento avviene attraverso i sensori montati sulla vettura che individuano la presenza di ostacoli al parcheggio e attraverso la penalizzazione delle azioni tendenti alla collisione con questi o la ricompensa per le azioni tendenti a schivarli.

Dal punto di vista del metodo con cui vengono effettuate analisi e apprendimento possiamo utilizzare altre definizioni.
Normalmente, fino a che lavoriamo su dataset composti da numeri o da stringhe di caratteri, le metodologie con cui abbiamo a che fare sono quelle dell'algebra lineare, che sfrutta proprietà e operazioni su vettori e matrici, dell'analisi matematica, con lo studio di funzione e di ottimizzazione, e del calcolo delle probabilità.
Se il dataset è composto da immagini, suoni o video dobbiamo ricorrere a metodologie diverse, dove in luogo dei soliti vettori a una o due dimensioni abbiamo vettori di dimensione superiore, detti tensori, e dove la modellizzazione di funzioni complesse può convenientemente avvenire utilizzando reti neurali artificiali.
Ed è qui che il machine learning diventa deep learning.

7.1 Reti neurali artificiali

Nel Capitolo 6, Python per la Data Science, ho presentato alcuni moduli del linguaggio Python concepiti per trattare e fare machine learning con dati numerici o testuali.

Vi troviamo un esempio di apprendimento supervisionato, un esempio di apprendimento non supervisionato e un esempio di sentiment analysis su dati testuali utilizzata per instaurare un piccolo dialogo con il computer in cui il computer si esprime tenendo conto di ciò che gli si dice.

Piccole cose che possiamo fare con un computer qualsiasi di media potenza. Ma già in queste piccole cose c'è il germe dell'intelligenza artificiale, anche se l'apprendimento del computer è piuttosto superficiale e non profondo.

Con il deep learning i metodi di analisi dei dati e di apprendimento vanno più in profondità e si avvalgono dello strumento della rete neurale artificiale, accorgimento informatico che imita il funzionamento del cervello umano.

Una rete neurale artificiale, come le applicazioni tradizionali per computer, è un software. Il relativo codice, anche se scritto con gli stessi linguaggi usati per sviluppare le applicazioni tradizionali, non è un insieme di istruzioni fisse che permettono ai programmi di elaborare richieste specifiche in modo standard. Esso è costituito da algoritmi che permettono alle reti di imparare quali output fornire a fronte di input che possono essere molto grandi, complessi e inattesi.

Come il neurone è l'unità cellulare del sistema nervoso in grado di ricevere, elaborare e trasmettere impulsi nervosi, il neurone artificiale è l'unità cellulare di una rete neurale artificiale.

L'elaborazione attraverso neuroni artificiali consiste nella "somma pesata" di segnali in ingresso: se il risultato supera un certo valore di soglia il neurone si attiva e genera un potenziale di azione, se il risultato rimane sotto un certo valore di soglia il neurone rimane in uno stato di riposo.

Come l'informatica sia riuscita ad imitare in questo modo il cervello umano e come tutto ciò si possa implementare utilizzando il linguaggio di programmazione Python comporta uno notevole sforzo di comprensione. A chi si voglia cimentare suggerisco i Capitoli 2 e 12 del testo di Sebastian Raschka, Machine Learning con Python, edito da Apogeo.

Le reti neurali artificiali vengono utilizzate in applicazioni nei campi più disparati, dalla finanza alla medicina, dalla sicurezza alla multimedialità. Soprattutto in applicazioni ove vi sia necessità di riconoscere ed elaborare immagini, riconoscere voci e interpretare il parlato.

Una delle ultime applicazione di questo strumento è il trasformatore generativo pre-addestrato, il modello GPT alla base delle spettacolari chat che hanno incuriosito il mondo nel corso del 2023.

Ma è da tempo che esistono cose di questo tipo. Nell'allegato «impro_visor.pdf» al mio articolo «Intelligenza artificiale per l'improvvisazione musicale» pubblicato nel settembre 2018 sul mio blog all'indirizzo *www.vittal.it* si trova la presentazione di un software musicale che utilizza il deep learning per generare improvvisazioni jazzistiche secondo lo stile di un certo interprete.

Le reti neurali artificiali non sono da considerare sempre un'alternativa alle applicazioni tradizionali.

Ha senso usarle quando la complessità dei dati e le difficoltà di elaborazione renderebbero inefficace o impossibile ricorrere a metodi di analisi tradizionale, che sono comunque sempre efficaci quando i dati da elaborare sono ben chiari e di natura numerica, anche se la mole dei dati da elaborare è molto consistente.

La grande comunità di Python ha prodotto e produce moduli di questo software innovativo e ne propongo una carrellata nel prossimo paragrafo.

Si tratta di moduli che possiamo caricare sul nostro computer con il solito `pip`.

Dal momento, tuttavia, che il funzionamento di questo tipo di software richiede spesso una notevole capacità elaborativa e normalmente diventa difficile o impossibile utilizzarlo su computer casalinghi da dilettanti principianti, suggerisco un primo avvicinamento ad esso attraverso Google Colaboratory.

Chi non conosca questo formidabile strumento può utilmente consultare l'Appendice B.

7.2 Strumenti

La comunità Python ha elaborato molti strumenti per il deep learning e ne propongo una rassegna con utili indicazioni per chi voglia approfondire la conoscenza e fare qualche prova.

7.2.1 Scikit-learn

Rilasciato nel 2007, è il modulo Python più anziano e completo per il machine learning.

Ne abbiamo parlato nel Paragrafo 6.3

All'indirizzo *https://scikit-learn.org/stable/*, scegliendo la pagina USER GUIDE possiamo consultare un'ottima guida in lingua inglese, ricca di esemplificazioni, nella quale sono dedicati al deep learning i capitoli

NEURAL NETWORK MODELS delle sezioni SUPERVISED LEARNING e UNSUPERVISED LEARNING.

Veniamo preavvisati sul fatto che l'implementazione deep learning di Scikit-learn non è pensata per applicazioni su larga scala, in quanto Scikit-learn non offre supporto per elaborazioni GPU o TPU.

7.2.2 Aesara

A questa debolezza di Scikit-learn si è ovviato con la produzione del modulo Theano, rilasciato nel 2008 e autodefinito come il nonno dei moduli Python dedicati al deep learning.

Attualmente il relativo codice è confluito nel modulo Aesara, che troviamo descritto all'indirizzo
https://aesara.readthedocs.io/en/latest/.

Si tratta di una libreria Python che consente di definire, ottimizzare/riscrivere e valutare in modo efficiente espressioni matematiche che coinvolgono array multidimensionali (tensori).

Siamo sempre nel campo degli strumenti di machine learning che fanno anche deep learning.

7.2.3 TensorFlow

E' la prima libreria open source pensata e prodotta per il deep learning, all'inizio limitata ad applicazioni di calcolo numerico ma presto resa atta ad elaborare algoritmi per diversi tipi di compiti percettivi e di comprensione del linguaggio.

Creata e rilasciata nel 2015 da Google è utilizzabile con i linguaggi Python e Javascript, attraverso i quali si creano applicazioni che vengono eseguite in C++ ad alta prestazione.

Il suo utilizzo con il linguaggio Python è agevolato dal frontend **Keras**, rilasciato nel 2015 dalla comunità Python.

All'indirizzo *https://www.tensorflow.org/overview?hl=it* troviamo tutto ciò che c'è da sapere su TensorFlow e non solo.

Seguendo quanto ci viene proposto nelle pagine TUTORIAL e GUIDA siamo infatti introdotti al deep learning in maniera che più facile non si può, con molti esempi di ciò che si può fare con TensorFlow e Keras.

Tutti questi esempi, di cui si può copiare il codice negli appunti, possono essere convenientemente eseguiti utilizzando Google Colab.

Anche con tutte queste agevolazioni imparare a lavorare con TensorFlow è molto difficile.

7.2.4 PyTorch

Ottima alternativa a TensorFlow, rispetto al quale è un tantino più facile, rilasciata dalla comunità Python nel 2016.

E' una libreria Python basata su Torch, a sua volta scritta con i linguaggi Lua, C e C++, dotata di due moduli: `torch` per trattare tensori (array multidimensionali) e `nn` per creare reti neurali.

Per lavorare con dati audio e video possiamo disporre dei moduli `torchaudio` e `torchvideo`.

Lavorando con Google Colab abbiamo accesso a tutto quanto: basta importare quel che serve con `import`.

Troviamo tutto ciò che riguarda PyTorch all'indirizzo *https://pytorch.org/*

Nella pagina TUTORIAL abbiamo interessanti esercitazioni che possiamo far funzionare utilizzando Google Colab e nella pagina DOCS abbiamo la documentazione ufficiale.

Ai principianti suggerisco il corso accelerato in lingua italiana che si trova all'indirizzo
https://www.intelligenzaartificialeitalia.net/post/deep-learning-con-python-e-pytorch-la-guida-pratica

Capitolo 8

Orange

Nel 1996, presso l'Università di Lubiana, venne lanciato un progetto per un frame di machine learning, denominato appunto ML, da realizzarsi in linguaggio C++.

L'anno dopo al linguaggio C++ si associò il linguaggio Python.

Nel 2003 venne ridisegnata l'interfaccia grafica utilizzando la libreria PyQt, il binding Python del framework Qt.

Nel frattempo il progetto venne ribattezzato con il nome Orange e nel 2009 furono superati i 100 strumenti di elaborazione messi a disposizione, chiamati widgets.

Orange è una libreria Python che può essere importata per programmare script di elaborazione dati in linguaggio python. La libreria è concepita per semplificare lo scripting e potenziare alcuni comandi ma si appoggia alle solite librerie del mondo Python per il data mining (numpy, scipy, matplotlib, pandas, sklearn) nulla aggiungendo alla capacità elaborativa di queste.

I suoi comandi semplificati o potenziati sono stati organizzati in modo da poter essere anche richiamati visualmente attraverso icone in un canvas dove, opportunamente collegati tra loro attraverso il mouse, vadano a svolgere elaborazioni anche complesse, senza che all'utente sia richiesto di scrivere una sola riga di codice.

Troviamo Orange all'indirizzo *https://orangedatamining.com/*.

Su questo sito è presente, in lingua inglese, tutta la documentazione esistente su Orange, anche in forma di simpatici video dimostrativi, sempre in lingua inglese, ma facile e sorretta da didascalie. Visitando le varie pagine possiamo leggere e vedere molte cose.

Anche se il modo più sicuro di installare Orange è farlo con Anaconda, fedeli al nostro proposito iniziale da principianti lo facciamo con pip.

Dobbiamo solo fare attenzione a che non vi sia troppo divario temporale tra l'epoca di installazione di Python e dei moduli per il data mining e quella di installazione di Orange. Il comando da utilizzare è
`pip` (o `pip3`) `install orange3`

Dalla pagina DOWNLOAD del sito internet di Orange possiamo comunque scaricare gli installer per Windows e macOS.

Per Windows è disponibile anche una versione portable in formato zip. Basta estrarla su disco o su pennetta USB ed è pronta all'uso.

Per Linux è disponibile il tarball del source su GitHub.

Con la prima installazione abbiamo disponibili praticamente tutti gli strumenti per il data mining numerico, salvo alcune finezze, come, per esempio, la regressione polinomiale, e non vengono installati strumenti per l'analisi del testo (text mining).

Componenti aggiuntive possono essere installate una volta avviato Orange nei modi che vedremo.

8.1 Orange come modulo Python

Secondo lo stile sin qui praticato importiamo Orange con un alias abbreviante con

```
import Orange as o
```

(non possiamo utilizzare l'alias `or`, come verrebbe istintivo secondo usanza, in quanto `or` è una parola riservata del linguaggio Python).

Se, dopo l'importazione, nella IDLE scriviamo `o` seguito da un punto vediamo l'elenco dei sottomoduli e, procedendo per selezioni e ulteriori punti, possiamo esplorare tutto il contenuto della libreria.

Ci accorgiamo così che per utilizzare Orange come modulo in uno script Python dobbiamo praticamente imparare un altro linguaggio.

Per esempio dobbiamo imparare che, per calcolare la media di una successione di numeri utilizzando la libreria Orange, dobbiamo scrivere:

```
import Orange as o
dati = [2, 4, 6.7, 15, 18]
o.statistics.util.mean(dati)
```

anziché scrivere, in maniera probabilmente più semplice:

```
import numpy as np
dati = np.array((2, 4, 6.7, 15, 18))
dati.mean()
```

come faremmo con gli strumenti che abbiamo visto prima.

Pertanto non ci interessiamo oltre a questo modo di utilizzare Orange.

8.2 Orange come strumento visuale

La vera potenza di Orange sta nel fatto che tutte le funzioni della libreria sono rappresentate graficamente come oggetti richiamabili, chiamati widgets, collegabili tra loro in un canvas a costituire un workflow in cui si compiono elaborazioni, anche sofisticate, senza scrivere una sola riga di codice.

Il tutto in un ambiente abbastanza intuitivo e che ci consente di raggiungere risultati in maniera molto probabilmente meno affaticante rispetto al doverlo fare scrivendo codice.

Una volta installato Orange ne avviamo l'interfaccia grafica con il comando a terminale

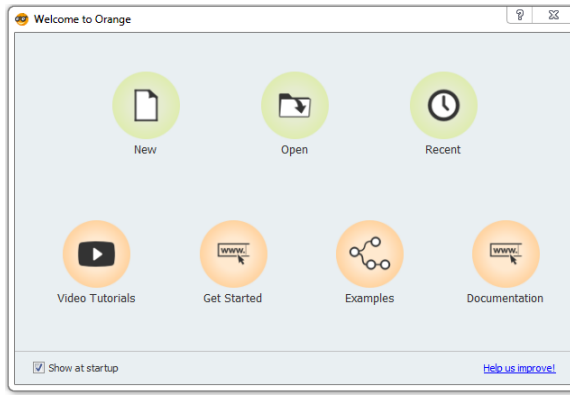
```
orange-canvas
```

oppure

`python -m Orange.canvas` (o `python3 -m Orange.canvas` se sul sistema è installato anche Python 2).

Se utilizziamo la versione portable su Windows possiamo lanciare l'interfaccia grafica con il lanciatore `Orange` che si trova nella directory che ospita i file del programma.

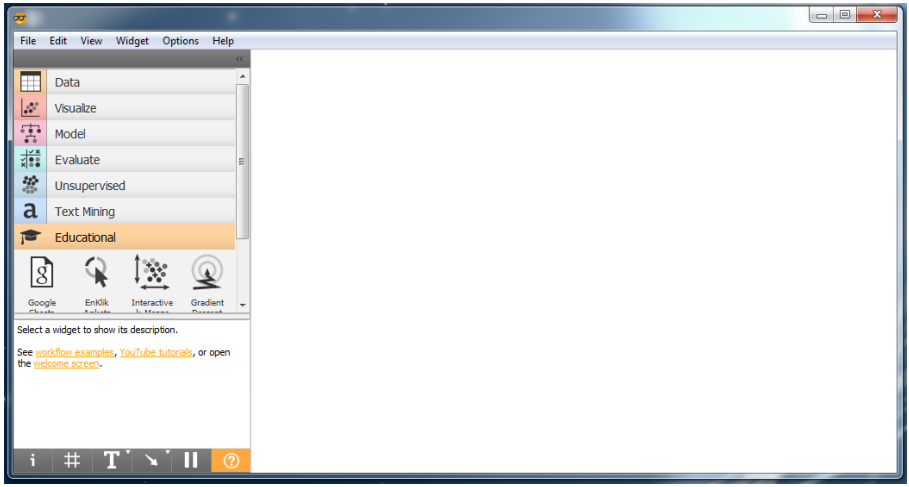
La prima finestra che si presenta è una finestra di benvenuto



Questa finestra, che possiamo evitare si ripresenti deselezionando l'opzione SHOW AT STARTUP in fondo a sinistra, è utile per il neofita in quanto, con le icone in basso, dà accesso a tutta una serie di documentazioni di utile consultazione.

Le tre icone in alto aprono il canvas per lavorare su un nuovo progetto, per aprire un progetto già salvato o per aprire un progetto operato di recente.

Il canvas per un nuovo progetto si presenta così



Sulla sinistra abbiamo la finestra dove si trovano i raggruppamenti dei widget disponibili.

I primi cinque raggruppamenti (Data, Visualize, Model, Evaluate e Unsupervised) sono quelli che si rendono disponibili alla prima installazione.

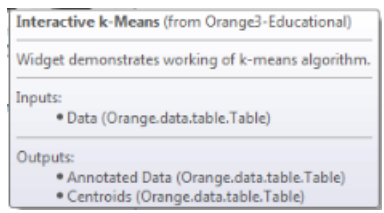
Gli altri due (Text mining e Educational) ritengo opportuno siano installati in aggiunta.

Per farlo, con collegamento internet attivo, si sceglie da menu OPTIONS ▷ ADD-ONS... e, nella finestra successiva, si selezionano le voci ORANGE3-EDUCATIONAL e ORANGE3-TEXT e si preme OK in fondo alla finestra.

Cliccando sulla voce che indica il raggruppamento si apre l'elenco dei widget che contiene. Nella figura è aperto l'elenco dei widget del raggruppamento EDUCATIONAL e se ne vedono i primi quattro; i successivi si possono vedere agendo sulla barra di scorrimento sulla destra della finestrella.

Fermando il puntatore del mouse sull'icona di un widget apriamo una finestrella che illustra il widget stesso.

Se, per esempio, fermiamo il puntatore del mouse sull'icona del widget INTERACTIVE K-MEANS, il terzo che vediamo tra quelli del raggruppamento EDUCATION nella figura, apriamo questa finestrella



nella quale ci viene detto che il widget dimostra il modo di lavorare dell'algoritmo K-Means, uno dei più utilizzati per il clustering, accetta in input dati da una tabella e fornisce due output leggibili su altrettante tabelle.

Per avere spiegazioni complete sul funzionamento dei widget è disponibile, con collegamento internet attivo, il Widget catalog. Lo possiamo aprire cliccando sull'icona DOCUMENTATION nella pagina di benvenuto prima illustrata oppure aprendo la pagina DOCS sul sito di Orange, scegliendo poi VISUAL PROGRAMMING ▷ WIDGET CATALOG. In questo catalogo sono illustrati tutti i widget di Orange, anche quelli non installati: in tal modo possiamo valutare l'utilità di ciò che non abbiamo installato ed eventualmente provvedere all'installazione nel modo illustrato prima.

La zona bianca a destra della finestra dei widget è l'area di lavoro, dove andremo a costruire il nostro workflow, fatto di widget collegati tra loro.

Inseriamo gli widgets che ci servono nell'area di lavoro scegliendo ciascuno di essi nella finestra di sinistra, dopo aver aperto il raggruppamento dove si trova, trascinando la relativa icona nell'area di lavoro o semplicemente cliccando sulla relativa icona.

L'icona inserita appare con un arco tratteggiato sulla destra e, se è richiesto un input, con un arco tratteggiato sulla sinistra.

L'workflow si costruisce collegando con il mouse archi di output e archi di input dei vari widgets.

Con doppio click su un widget nell'area di lavoro, se si tratta di un widget destinato a compiere una elaborazione apriamo una finestra in cui possiamo inserire opzioni per l'elaborazione stessa, se si tratta di un widget destinato a mostrare risultati apriamo una finestra in cui vediamo questi risultati.

Collegando ad uno di questi widget contenenti risultati il widget per la memorizzazione, che troviamo nel raggruppamento DATA, possiamo salvare i risultati stessi in un file.

Alcuni esempi

Senza altra pretesa se non quella di mostrare come concretamente funziona Orange propongo qualche semplice esempio.

Relazioni tra dati

Poniamo di essere di fronte a questa serie di dati accostati

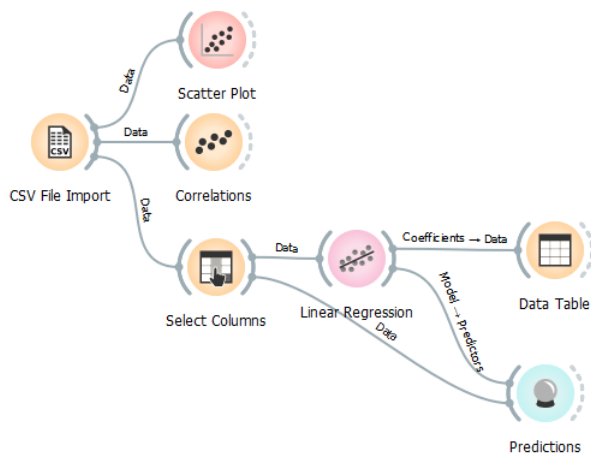
72	124
83	138
45	87
65	105
13	21
54	96

memorizzata in un file .csv (file di testo con dati separati da virgola) e di voler indagare sulle relazioni esistenti tra loro.

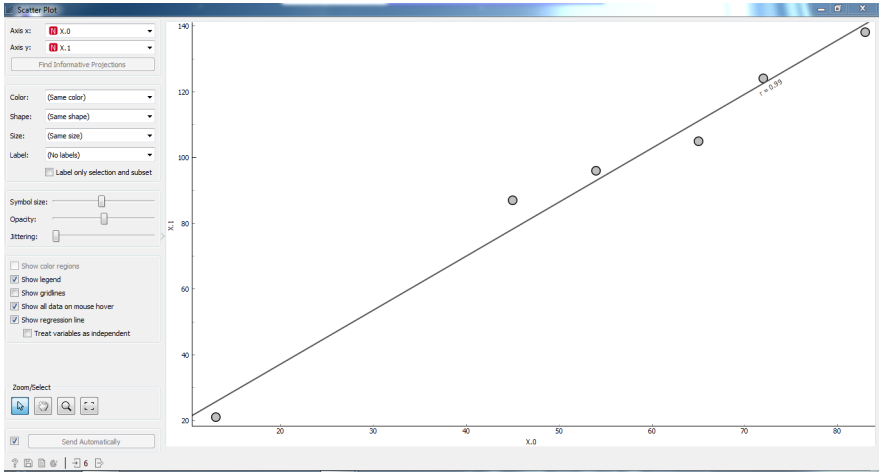
Con il workflow riprodotto qui sotto ci possiamo rendere conto di molte cose.

Innanzi tutto importiamo i dati e, visto che sono memorizzati in un file .csv, lo facciamo avvalendoci del widget CSV FILE IMPORT, che troviamo nel raggruppamento DATA. Una volta inserito il widget sul workflow ne apriamo la finestra di configurazione con doppio click e inseriamo percorso e nome del file da importare agendo nelle due finestrelle FILE:.

La prima cosa che possiamo fare è visualizzare come si relazionano i nostri dati attraverso un grafico a dispersione (scatter plot) allocando i dati su assi cartesiane ortogonali.



Lo facciamo ricorrendo al widget SCATTER PLOT che troviamo nel raggruppamento VISUALIZE. Collegati i due widget nel modo che abbiamo visto nel precedente esempio, con doppio click sul widget SCATTER PLOT apriamo questa finestra



nella quale vediamo il grafico a dispersione e, avendo attivato l'opzione SHOW REGRESSION LINE, la retta interpolante accompagnata dall'indice di correlazione di Pearson `r` (del valore di 0,99, ad indicare elevatissima correlazione tra i dati).

Possiamo salvare il grafico agendo su una di queste icone in basso a sinistra . Con quella di sinistra salviamo il grafico in un formato grafico a scelta e con quella di destra lo salviamo su una pagina html.

Possiamo misurare la correlazione esistente tra i dati anche ricorrendo al widget dedicato CORRELATIONS, che troviamo nel raggruppamento DATA. Collegato questo widget a quello dell'importazione dei dati, con doppio click su di esso possiamo scegliere tra il calcolo dell'indice di correlazione di Pearson e l'indice di correlazione di Spearman.

Se poi riteniamo che tra le due serie di dati possa esistere un legame funzionale che faccia dipendere le variazioni dell'una dalle variazioni dell'altra possiamo scoprirlo e descriverlo con un modello matematico applicabile in sede previsiva.

Per fare questo dobbiamo innanzi tutto dichiarare quale, nella nostra coppia di dati, sia da considerarsi la variabile dipendente che varia in funzione dell'altra. Lo facciamo utilizzando il widget SELECT COLUMNS che troviamo nel raggruppamento DATA. Collegato questo widget a quello dell'importazione dei dati, con doppio click su di esso apriamo la finestra di configurazione e vediamo che le colonne dei dati letti sono chiamate `X0` e `X1` e sono inserite entrambe nella sotto-finestra FEATURES; se riteniamo, contrariamente a quanto fatto per default nel disegno del grafico, che la variabile dipendente sia quella della colonna `X0`, la colon-

na di sinistra della nostra tabella, con il mouse ne trasciniamo il simbolo nella sotto-finestra TARGET VARIABLE.
Ora ricorriamo al widget LINEAR REGRESSION, che troviamo nel raggruppamento MODEL, per descrivere in termini matematici la relazione funzionale tra le due variabili. Tipicamente la regressione lineare descrive la relazione funzionale con l'equazione di una retta del tipo $y = a + bx$ dove a è l'intercetta sull'asse verticale e b è il coefficiente angolare. Collegato questo widget al precedente che abbiamo inserito prima, con doppio click ne apriamo la finestra di configurazione e, per i nostri usi dilettanteschi, lasciamo tutto come si presenta di defalt; solo accertiamoci che sia selezionata l'opzione FIT INTERCEPT: in questo modo verrà calcolato anche il valore a dell'equazione della retta e non solo il coefficiente b.
Attraverso il widget DATA TABLE che troviamo nel raggruppamento DATA, una volta inserito nel workflow e collegato al LINEAR REGRESSION, con doppio click su di esso possiamo vedere gli elementi del nostro modello matematico:
. intercetta: -1,59722
. coefficiente della variabile indipendente `x1`: 0,598219

Visto che siamo nel machine learning la macchina ha imparato e, in previsione del manifestarsi futuro del valore, per esempio, 152 della variabile indipendente, ci dirà che, con ogni probabilità, il corrispondente valore della variabile dipendente sarà 89 (`-1,59722+0,598219x152`).

Se vogliamo avere un'idea dell'affidabilità di questa stima possiamo ricorrere al widget PREDICTIONS, che troviamo nel raggruppamento EVALUATE. Inserito questo widget nel workflow, lo colleghiamo al widget LINEAR REGRESSION e al widget SELECT COLUMNS (in tal modo applichiamo il modello regressivo ai dati della colonna di destra della nostra tabella di partenza e determiniamo i valori corrispondenti teorici della colonna di sinistra). Con doppio click sul widget apriamo una tabella in cui vediamo i nostri dati, effettivi e teorici e, soprattutto vediamo quattro indicatori di affidabilità del modello: lasciamo i primi tre agli statistici esperti e, da dilettanti, accontentiamoci del quarto, R2, detto anche R quadro o coefficiente di determinazione: nel nostro caso è pari a 0,982, valore molto elevato che ci tranquillizza sulla validità interpretativa del nostro modello.

Non tutto è lineare, però...

Supponiamo ora di trovarci di fronte a questa serie di dati accostati

12	24
45	125
56	223
84	512
92	798
116	985

Se misuriamo la correlazione sicuramente troviamo che esiste: anche a occhio vediamo che al crescere dei dati sulla sinistra crescono i dati sulla destra.

Sempre a occhio vediamo tuttavia che, al crescere dei dati sulla sinistra, i dati sulla destra crescono in maniera vistosamente più che proporzionale: da qui il sospetto che un modello lineare come quello visto nel paragrafo precedente non interpreti bene la relazione esistente tra i dati.

Orange ci offre uno strumento molto utile per ragionare su casi come questo, il widget POLYNOMIAL REGRESSION, che troviamo nel raggruppamento EDUCATION.

Lo troviamo in questo raggruppamento anziché, come il widget LINEAR REGRESSION, nel raggruppamento MODEL, probabilmente perché si tratta innanzi tutto di uno strumento di studio, che può diventare strumento di generazione di un modello dopo averci meditato molto bene.

In statistica abbiamo due tipi di interpolazione dei dati: l'interpolazione per punti noti e l'interpolazione tra punti noti.

Nel primo caso si tratta di trovare un modello matematico che disegni una linea interpolante che passi per tutti i punti dati. L'equazione di questa linea sarà un polinomio di grado pari al numero di coppie di dati disponibili meno uno. Nel caso delle nostre sei coppie di dati un polinomio di quinto grado, del tipo

$$y = a + bx + cx^2 + dx^3 + ex^4 + fx^5$$

dove y (la variabile dipendente) è il dato nella colonna di destra e x (la variabile indipendente) è il dato nella colonna di sinistra.

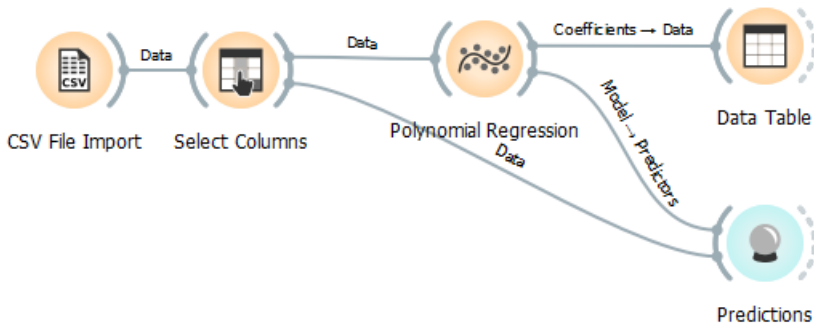
Nel secondo caso si tratta di trovare una linea interpolante che passi attraverso i punti noti e che si avvicini il più possibile ad essi: il migliore avvicinamento si ha quando la somma dei quadrati degli scarti tra i valori effettivi e i valori stimati dalla linea è minima.

Il primo tipo di interpolazione è sicuramente valido per trovare i più probabili valori della variabile dipendente corrispondenti a valori ignoti della variabile indipendente all'interno della serie di dati noti di questa.

Nel nostro caso, per esempio, per trovare il più probabile valore del dato di destra corrispondente ad un valore del dato di sinistra di 50.

Se il nostro obiettivo è quello di trovare il più probabile valore del dato di destra corrispondente ad un valore del dato di sinistra di 150, cioè quando al termine interpolazione è bene sostituire il termine estrapolazione, come a dire previsione per uno sviluppo al di fuori dai dati noti, è opinione comune che sia preferibile ragionare in termini di interpolazione tra punti noti, con preferenza per il metodo della regressione lineare, cioè con il polinomio di primo grado, la retta.

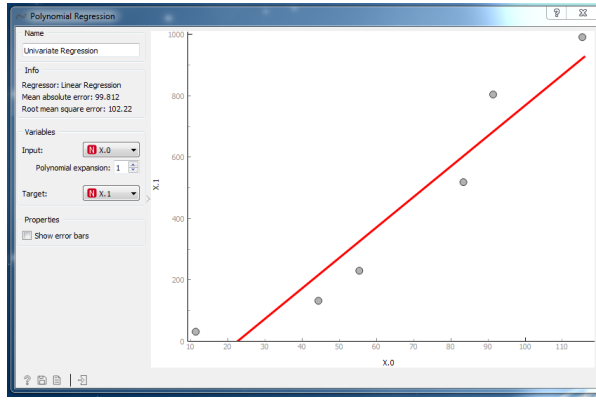
Vediamo come il widget POLYNOMIAL REGRESSION ci aiuti a ragionare su questi casi, costruendo il seguente workshop.



Importiamo i nostri dati, che ipotizzo si trovino su un file .csv, con il widget CSV FILE IMPORT; con il widget SELECT COLUMNS stabiliamo che la variabile dipendente (target) sia X.1 (nell'importazione dei dati viene dato il nome X.0 alla prima colonna e il nome X.1 alla seconda) e colleghiamo a questo widget il widget POLYNOMIAL REGRESSION. Inseriamo poi e colleghiamo i widget DATA TABLE e PREDICTIONS come abbiamo visto fare nell'esempio del precedente paragrafo.

Con doppio click sul widget POLYNOMIAL REGRESSION ne apriamo la finestra di configurazione e, nella finestrella POLYNOMIAL EXPANSION, inseriamo il grado del polinomio che intendiamo utilizzare per l'interpolazione dei dati.

Inserendo il valore 1 utilizziamo un polinomio di primo grado, che corrisponde alla linea retta. Pertanto se inseriamo il valore 1 è come se avessimo utilizzato il widget LINEAR REGRESSION e ci troviamo di fronte la seguente finestra, in cui vediamo disegnata la retta interpolante



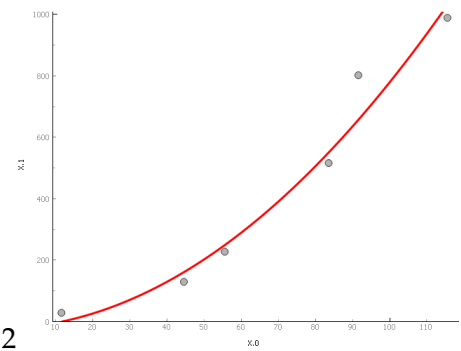
Agendo sui widget DATA TABLE e PREDICTIONS individuiamo la seguente equazione della retta interpolante

$$y = -226,879 + 9,94635x$$

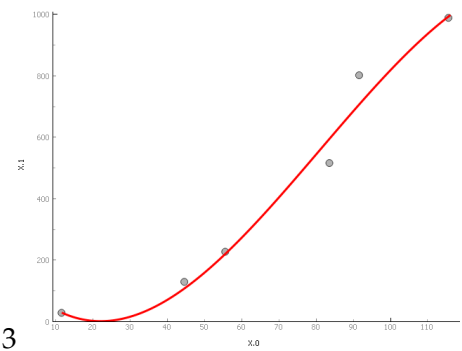
e coefficiente di determinazione 0,916, elevato ma non esaltante.
Possiamo pertanto prevedere che, con buona probabilità, al valore di 150 della variabile indipendente corrisponda un valore di 1265 della variabile dipendente.

Rimane il fatto che i dati rivelano un andamento esponenziale della relazione tra di loro: lo si vede anche dal grafico se teniamo conto della differenza di scala tra gli assi delle ascisse e delle ordinate.

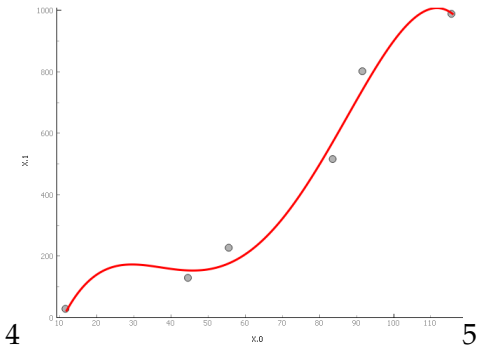
Agendo sulla finestrella POLYNOMIAL EXPANSION della finestra di configurazione del widget POLYNOMIAL REGRESSION possiamo vedere in tempo reale cosa succede inserendo il valore 2, alla ricerca di una parabola interpolante, il valore 3, alla ricerca di una curva flessuosa, del valore 4, fino al valore 5, corrispondente all'espressione analitica di una curva che tocchi tutti i punti del grafico (interpolazione per punti noti).



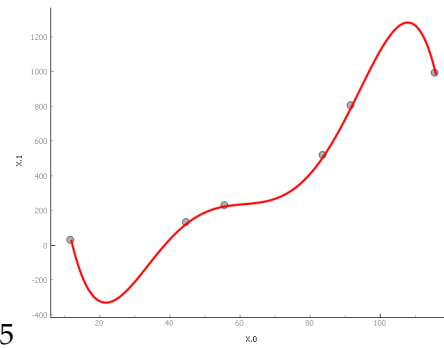
2



3



4



5

L'equazione della parabola di secondo grado risulta essere

$$y = -20,9841 + 0,901479x + 0,0708207x^2$$

con coefficiente di determinazione 0,965.
Possiamo pertanto prevedere che, con probabilità superiore a quella di prima, al valore di 150 della variabile indipendente corrisponda un valore di 1707 della variabile dipendente. Il valore previsto è più elevato di quello previsto con la retta in quanto qui si tiene maggiormente conto dell'andamento esponenziale.

La curva che rappresenta l'equazione di terzo grado rileva un andamento a flesso e, con un coefficiente di determinazione 0,979, lascia prevedere un valore di 1093 in corrispondenza al solito valore ipotetico di 150 della variabile indipendente. Il valore previsto è inferiore sia a quello previsto con la retta sia a quello previsto con la parabola in quanto si tiene conto della flessione del ritmo di crescita che si verifica a metà del grafico.

Con l'equazione di quarto grado la curva si avvicina sempre più ai punti noti, tanto che il coefficiente di determinazione sale a 0,987 e rileva una inversione di tendenza verso la fine del grafico.

Con l'equazione di quinto grado abbiamo una curva che tocca tutti i punti noti, pertanto il coefficiente di determinazione sale a 1, ma accentua la rilevazione dell'inversione di tendenza a fine grafico.

Le previsioni fatte applicando queste due equazioni fanno addirittura corrispondere valori negativi alla variabile dipendente in corrispondenza al valore di 150 della variabile indipendente.

Con questo esempio si dimostra come sia difficile fare previsioni, anche se si è aiutati dal rigore della matematica.

Soprattutto si dimostra che la ricerca di coefficienti di determinazione sempre più elevati può servirci per descrivere sempre meglio ciò che è

successo ma può portarci parecchio fuori strada quando vogliamo capire ciò che può succedere.

Nel nostro caso specifico, comunque, ritengo valide le prime tre previsioni: quella ottimistica della parabola, quella media della retta e quella pessimistica del modello di terzo grado. Aiutino altre conoscenze a fare la scelta definitiva.

Ecco anche spiegato perché si tende comunque a preferire previsioni fatte attraverso la retta e la regressione lineare, ovviamente a patto che il coefficiente di determinazione sia abbastanza prossimo all'unità.

A proposito di coefficiente di determinazione

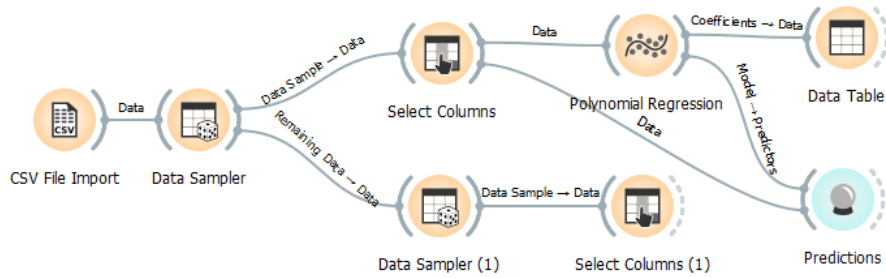
Il problema che abbiamo esaminato nel precedente paragrafo è esasperato dal fatto che abbiamo lavorato su una serie di dati molto limitata, come spesso avviene nella statistica tradizionalmente applicata a piccoli gruppi campionari o a dati di difficoltosa produzione.

Se abbiamo a che fare con dati disponibili in quantità superiore, addirittura con le grandi masse di dati che oggi sono facilmente prodotti dalla digitalizzazione di tutto ciò che facciamo, quelli che si usano chiamare big data, diventa più facile e sicuro per gli strumenti che stiamo vedendo individuare trend utili a scopo previsionale e possiamo anche meglio misurare l'affidabilità di ciò che scopriamo.

Nel caso visto nel precedente paragrafo, per esempio, data la scarsa numerosità dei dati da cui siamo partiti, il coefficiente di determinazione è stato calcolato, in modo autoreferenziale, utilizzando gli stessi dati che sono serviti per scoprire il modello.

Quando si lavora sui big data, per essere più sicuri, si scopre il modello avvalendosi di una parte dei dati a disposizione e poi lo si verifica anche su un'altra parte dei dati stessi e solo se entrambi i coefficienti di determinazione (quello calcolato sui dati utilizzati per scoprire il modello e quello calcolato sugli altri dati) sono accettabili si può essere tranquilli sulla validità del modello trovato.

Orange ci mette a disposizione un widget per estrarre da grandi masse di dati campioni più piccoli e lo possiamo utilizzare con profitto nel modo qui descritto.



Si tratta del widget DATA SAMPLER.

Importati i dati, con il widget CSV FILE IMPORT, sempre nell'ipotesi che i nostri dati siano su file .csv, inseriamo nel workflow il widget DATA SAMPLER e lo colleghiamo al precedente. Con doppio click apriamo la finestra di configurazione nella quale dobbiamo scegliere in che modo estrarre il campione di dati, avendo a disposizione due opzioni:
. FIXED PROPORTION OF DATA, scegliendo la quale indichiamo la percentuale di dati da estrarre per costruire il campione,
. FIXED SAMPLE SIZE, scegliendo la quale indichiamo il numero di dati da estrarre per costruire il campione.
A scelta effettuata premiamo il pulsante SAMPLE DATA.
A questo punto il widget ha pronto un doppio output: il DATA SAMPLE, cioè il campione di dati estratto, e i REMAINING DATA, cioè i dati che non sono entrati a far parte del campione.

Utilizziamo il DATA SAMPLE per applicare un modello regressivo (nell'illustrazione abbiamo il POLYNOMIAL REGRESSION ma potremmo avere il LINEAR REGRESSION). Preventivamente dobbiamo passare attraverso il widget SELECT COLUMNS per indicare la colonna che contiene il target.

Utilizziamo i REMAINING DATA per testare il modello regressivo. Se abbiamo a che fare con moltissimi dati conviene forse che estraiamo da questi dati un campione inserendo allo scopo un altro DATA SAMPLER, come si vede nell'illustrazione. A seguire il solito SELECT COLUMNS per indicare la colonna che contiene il target.

Per scegliere l'output giusto, visto che, per default, quando tracciamo con il mouse il collegamento, viene scelto l'output DATA SAMPLE, dobbiamo fare doppio click sulla linea del collegamento ed agire sulla finestrella del link editor che si apre.

Inserito il widget PREDICTIONS e collegatolo al widget del modello regressivo, attraverso il collegamento tra il widget PREDICTIONS e il widget SELECT COLUMNS del DATA SAMPLE, con doppio click sul wid-

get PREDICTIONS vedremo il coefficiente di determinazione autoreferenziale calcolato sui dati da cui abbiamo derivato il modello. Se questo secondo collegamento lo effettuiamo tra il widget PREDICTIONS e il widget SELECT COLUMNS del REMANINIG DATA vedremo il coefficiente di determinazione calcolato su dati che non hanno contribuito all'individuazione del modello. Se entrambi i coefficienti di determinazione ci confortano per la loro vicinanza al numero 1 possiamo con maggiore tranquillità accettare per buono il nostro modello.

Sentiment analysis

Il widget di Orange attribuisce valori positivi a testi di sentimento positivo (allegri) e valori negativi a testi di sentimento negativo (tristi), valore zero a testi normali.

Proponiamoci di analizzare le seguenti tre frasi:
. Quando passo per questa strada mi sento allegro.
. Quando incrocio i tuoi occhi tristi mi viene da piangere.
. Non riesco a scegliere un programma televisivo che mi piaccia.
che si trovano in altrettanti file di testo, denominati testo_1, testo_2 e testo_3, memorizzati in una directory chiamata testi.

Possiamo attribuire un indicatore di sentiment a ciascuna di esse utilizzando questo semplice workflow

Il primo widget (IMPORT DOCUMENTS) serve per importare i testi da esaminare. Lo troviamo nel raggruppamento TEXT MINING. Una volta trascinato nell'area di lavoro, con doppio click su di esso apriamo una finestrella in cui indichiamo la directory da cui importare i file di testo da esaminare.

Il secondo widget (SENTIMENT ANALYSIS) analizza il testo ed attribuisce l'indicatore di sentiment. Lo troviamo nel raggruppamento TEXT MINING. Una volta trascinato nell'area di lavoro lo collochiamo sulla destra del precedente e lo colleghiamo ad esso trascinando il mouse, con premuto il tasto sinistro, tra l'arco sulla destra del primo widget e l'arco sulla sinistra del secondo. A collegamento avvenuto apriamo la finestra di configurazione del widget con doppio click e scegliamo l'opzione MULTILINGUAL SENTIMENT: nella finestrella sulla destra della dicitura

scorriamo e scegliamo ITALIAN (che è la lingua in cui sono scritti i testi da esaminare).

Il terzo widget (DATA TABLE) serve per leggere i risultati dell'elaborazione. Lo troviamo nel raggruppamento DATA. Una volta trascinato nell'area di lavoro lo collochiamo sulla destra del widget precedente e lo colleghiamo ad esso nel modo visto prima. Con doppio click su questo widget vediamo che al primo testo è stato attribuito sentiment positivo (11.1111), al secondo è stato attribuito sentiment negativo (-9,09091) ed il terzo testo è stato ritenuto neutrale, con sentiment zero.

Se vogliamo salvare questi risultati su file possiamo farlo cliccando sulla piccola icona (REPORT), la seconda che si trova nella barra di icone in fondo a sinistra della tabella che mostra i risultati, oppure ricorrendo al widget SAVE DATA che troviamo nel raggruppamento DATA (inserendo il widget nell'area di lavoro, collegandolo al terzo widget e configurandolo nella finestra che si apre con doppio click indicando dove e in che formato effettuare il salvataggio).

Regressione multipla

Ripropongo, con qualche arricchimento, l'esempio che abbiamo visto nel paragrafo 6.3.1

Abbiamo questo piccolo dataset, di dimensioni sufficienti per capirci,

y	x1	x2	x3
34	12	11	21
72	26	20	45
87	44	12	58
28	32	14	37
16	21	11	24
97	76	7	75

dove c'è una variabile dipendente y accostata al manifestarsi di sei diverse terne di variabili x.

Ci proponiamo di trovare il più probabile valore della y in corrispondenza delle terne

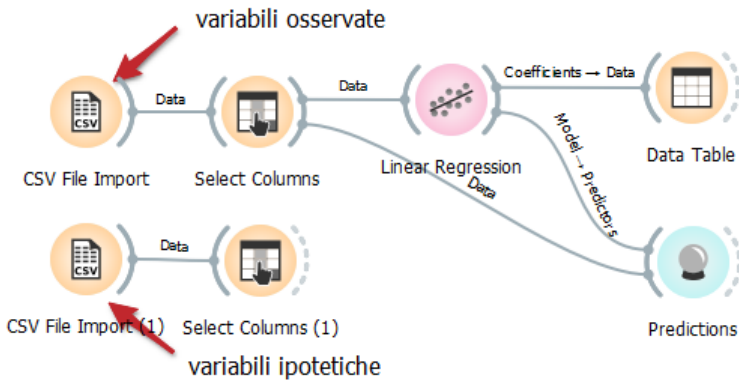
11	22	33
32	45	64
62	28	45

valori che dobbiamo introdurre in un file .csv in questo modo

```
,11,22,33
,32,45,64
,62,28,45
```
(iniziamo con una virgola in quanto il file deve avere la stessa formattazione di quello contenente i dati e il primo campo, quello della y, deve essere vuoto).

In questo modo abbiamo due file .csv: quello contenente le variabili osservate e quello contenente le variabili ipotetiche.

Risolviamo il nostro problema in questo modo.



Carichiamo il file .csv delle variabili osservate, nel relativo widget SELECT COLUMNS spostiamo la colonna `X0` nella finestrella TARGET e poi proseguiamo come abbiamo fatto nel Paragrafo 7.2.3 con i widget LINEAR REGRESSION, DATA TABLE e PREDICTIONS.
Con doppio click sul widget DATA TABLE troviamo gli elementi per scrivere il modello regressivo trovato:

$$y = 2,87997 - 2,64835x_1 - 2,61655x_2 + 4,12217x_3$$

Collegati i widget SELECT COLUMNS e PREDICTIONS, con doppio click sul widget PREDICTIONS constatiamo il coefficiente di determinazione di ottimo livello 0,973.

Ora importiamo il file .csv delle variabili ipotetiche, lo colleghiamo a un nuovo widget SELECT COLUMNS e lasciamo tutte le colonne importate nelle FEATURES senza indicare alcun TARGET.
Collegato questo nuovo SELECT COLUMNS a PREDICTIONS, con doppio click sul widget PREDICTIONS vediamo che il valore calcolato in corrispondenza alla terna 11 22 33 è 52, quello calcolato in corrispondenza

della terna 32 45 64 è 64 e quello calcolato in corrispondenza alla terna 62 28 45 è -49.

Ovviamente il risultato per la terna 11 22 33 è uguale a quello ottenuto nel paragrafo 6.3.1.

Clustering

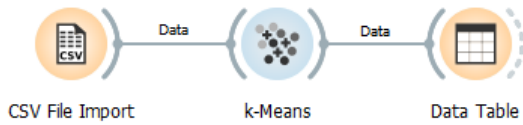
Ripropongo ora l'esempio già esaminato nel paragrafo 6.3.2, dove abbiamo il risultato di una ricerca sui nostri clienti che ne mostra le caratteristiche in termini di propensione al cambio di negozio, di sensibilità al prezzo del prodotto, di sensibilità alla marca e di frequenza degli acquisti.

cliente	cambio	prezzo	marca	frequenza
Luigi	5	5	5	3
Maria	7	7	4	2
Vittorio	6	5	5	3
Giovanni	6	6	4	2
Beatrice	5	5	4	3
Giuseppe	9	8	2	4
Elena	8	8	2	5
Antonio	9	7	3	4
Paola	10	7	3	4
Mario	9	8	2	4
Carlo	9	10	2	5
Cecilia	10	8	3	3

Le grandezze numeriche variano da 1 (per niente) a 10 (moltissimo).

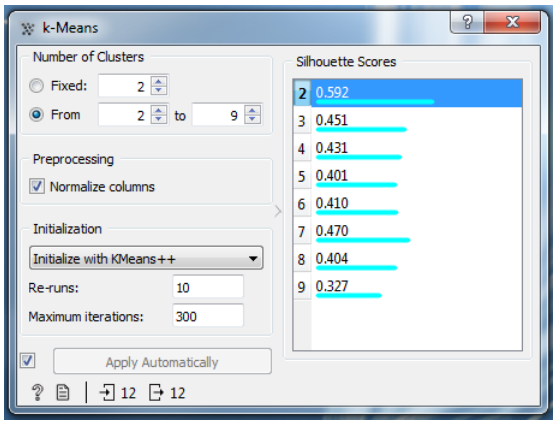
Già abbiamo detto che, aiutati dalla piccola dimensione del data set e dall'ordine in cui sono esposti i dati si nota subito a occhio che abbiamo un gruppo di clienti, i primi cinque, tendenzialmente quelli che fanno meno acquisti, che hanno una moderata propensione a cambiare negozio, e sono moderatamente sensibili al prezzo e alla marca; questo gruppo si contrappone al gruppo degli altri sette, con molto più elevata propensione a cambiare negozio, molto più sensibili al prezzo, meno sensibili alla marca e mediamente migliori clienti sul piano della frequenza di acquisto.

Per compiere questa analisi non più a occhio ma in maniera rigorosa, anche quando i dati sono migliaia e non solo 12 come in questo esempio, Orange ci mette a disposizione il widget K-MEANS.



Al solito, importiamo innanzitutto i dati con il widget CSV FILE IMPORT e poi colleghiamo a questo widget il widget K-MEANS, che troviamo nel raggruppamento UNSUPERVISED.

Con doppio click su questo widget ne apriamo la finestra di configurazione



Quella che si apre di default è solo la parte di sinistra della finestra, senza la parte di destra intitolata SILHOUETTE SCORES.

Se abbiamo già le idee chiare sul numero di cluster che intendiamo individuare, scegliamo l'opzione FIXED, nella finestrella indichiamo il numero dei cluster e chiudiamo.

Se vogliamo essere preventivamente orientati sul numero dei cluster più attendibili che possiamo individuare nel nostro dataset scegliamo l'opzione FROM... TO... indicando le relative quantità ipotetiche. Per il nostro piccolo dataset la figura mostra la scelta di mettere a prova un numero di cluster compreso tra 2 e 9.

Un cluster è attendibile quando gli oggetti che contiene sono sufficientemente simili tra loro e sufficientemente diversi dagli oggetti appartenenti ad altri gruppi.

Tra i vari modi per rappresentare con un indicatore l'attendibilità, Orange, per il suo k-Means, ha scelto l'indice di silhouette. Esso varia tra -1 e 1 e l'esperienza insegna che

. se è compreso tra 0,70 e 1 il partizionamento è estremamente attendibile,

. se è compreso tra 0,50 e 0,70 il partizionamento è attendibile,
. se è compreso tra 0,25 e 0,50 il partizionamento è poco attendibile,
. se è compreso tra -1 e 0,25 il partizionamento non è per niente attendibile.

Scelta e configurata l'opzione con l'indicazione dei cluster da sottoporre al test della silhouette, nella parte destra della finestra compare la lista degli indici di silhouette corrispondenti al range di possibili quantità di cluster.

Nel nostro caso vediamo che il valore massimo dell'indicatore corrisponde ad un partizionamento su due cluster con indice di silhouette 0,592, ad indicare che anche il partizionamento su due cluster è appena attendibile. Per tutti gli altri possibili partizionamenti non si raggiunge la soglia di attendibilità.

Selezioniamo pertanto, come fatto in figura, la riga corrispondente a 2 cluster e chiudiamo.

Colleghiamo un widget DATA TABLE in cui far confluire il risultato del partizionamento, che vediamo corrispondere esattamente all'impressione che avevamo percepito a occhio.

Se vogliamo salvare i risultati in formato html possiamo utilizzare il pulsante REPORT del widget DATA TABLE. Per un salvataggio in altri formati dobbiamo ricorrere al widget SAVE DATA, preventivamente selezionando, nella DATA TABLE, quanto vogliamo salvare.

Ovviamente il risultato coincide con quello che avevamo ottenuto nel paragrafo 6.3.2.

* * *

Mi spiace concludere un libro che ha l'obiettivo di invogliare alla programmazione con l'illustrazione di uno strumento che ci porta a risultati strabilianti senza programmare.

Del resto mi sono rivolto a principianti da poco più che principiante e io stesso non avrei avuto la capacità di divulgare, a portata di principiante, l'uso di uno strumento di programmazione come Scikit-learn, a parte lo spazio che ciò avrebbe richiesto.

Parlare di Orange è servito a chiarire la portata dei problemi che la scienza dei dati comporta e a far capire come sarebbe bello affrontarli programmando in Python.

Dal momento, poi, che tutti i libri che ho visto affrontare programmazione a questo livello partono dando per scontate molte conoscenze di base, mi auguro che questo libro serva almeno ad acquisire queste e a

rendere facile andare avanti, con altri libri, per uscire dalla categoria dei principianti.

Capitolo 9

Appendici

9.1 Appendice A - Jupyter

Jupyter è un editor di testo che ci consente di inserire, oltre ai soliti arricchimenti consistenti in illustrazioni e altri contenuti multimediali, codice interpretabile nello stesso editor per ottenere risultati di elaborazioni che pure entrano a far parte del testo.

Il nome è l'acronimo di JUlia, PYThon R, ad indicare i tre linguaggi di programmazione nativamente supportati, con in più una E eufonica ben collocata.

Dico nativamente supportati in quanto sono quelli che più facilmente possiamo utilizzare con Jupyter: infatti, per utilizzare il linguaggio Python non dobbiamo addirittura fare nulla in quanto Jupyter nasce in casa Python, per utilizzare il linguaggio Julia dobbiamo fare quasi nulla e per utilizzare il linguaggio R poco più di quasi nulla. Tutto a portata di dilettante.

Con qualche lavorio più da professionisti che da dilettanti possiamo fare in modo che Jupyter riconosca un'altra cinquantina di linguaggi: da C++ a Java, da Javascript a PHP, da Fortran a Ruby.

Le origini di Jupyter risalgono a quando due ricercatori dell'Università di Berkeley, Fernando Pérez e Robert Kern, danno avvio ad un progetto per creare un interprete interattivo per Python, IPython, che tuttora è utilizzabile come tale. Sono poi gli sviluppatori del team di questo progetto che, a partire dal 2010, creano una interfaccia arricchita, basata su IPython, che si chiama Jupyter Notebook.

Jupyter Notebook è la base di Jupyter.

Nel tempo sono intervenuti arricchimenti ed oggi il vecchio Jupyter Notebook sta passando il testimone a JupyterLab, che altro non è che un'interfaccia che ci consente di lavorare con più Notebook.

Abbiamo, infine, JupyterHub, che trasferisce JupyterLab su un server multiutente, in modo che sullo stesso progetto possano lavorare più persone distanziate nello spazio.

Qui presento Jupyter Notebook, esponendo quanto basta per il neofita e quanto comunque necessario conoscere per poter utilizzare anche gli altri due strumenti citati.

Installazione

Jupyter Notebook è un prodotto del mondo Python.

Per essere utilizzato occorre sia installato l'interprete Python e quello di eventuali altri linguaggi si vogliano utilizzare.

Chi ha installato Python Anaconda si trova Jupyter Notebook in Anaconda navigator senza dover fare nulla.

Possiamo installarlo con pip con il comando a terminale

```
pip3 install jupyter
```

Con questo abbiamo a disposizione Jupyter Notebook abilitato per il linguaggio Python3 e per l'esportazione dei documenti in formato `.html`.

Per poter esportare documenti nel formato PDF attraverso il formato HTML occorre che sul computer sia installato il browser Chromium.

Per poter esportare documenti in formato LaTeX (`.tex`), o in formato PDF attraverso il formato LaTeX, oltre ad avere installato una distribuzione LaTeX tipo Tex live (avendo cura che, tra gli altri, sia installato il package `texlive-xetex`), dobbiamo installare il software Pandoc, che troviamo all'indirizzo *https://pandoc.org/installing.html* nelle versioni per Linux, Mac e Windows. Chi usa Linux trova sicuramente Tex live e Pandoc nel repository della sua distro e può installare quanto serve con il gestore dei programmi.

Visto che l'installazione di questi software occupa parecchio spazio su disco, teniamo presente che i documenti in formato `.html` prodotti da Jupyter Notebook sono perfetti e fanno la loro bella figura anche loro e, per esportare in formato PDF ci basta aver installato Chromium.

Come funziona

Jupyter Notebook viene lanciato con il comando a terminale
`jupyter notebook`

A seconda del sistema operativo che usiamo, con l'installazione si può essere creato un lanciatore o una voce di menu che ci evita la fatica.

Possiamo anche creare noi un lanciatore in cui inserire il comando per il lancio.

Con il lancio si crea una nuova finestra del nostro browser web predefinito in quanto Jupyter Notebook è un'applicazione web basata su una struttura server-client.

Per default il server si attesta su localhost all'indirizzo 127.0.0.1:8888.

Se succede qualche cosa di strano e il server non si attiva provare a lanciare con il comando
`jupyter notebook --NotebookApp.use_redirect_file=False`

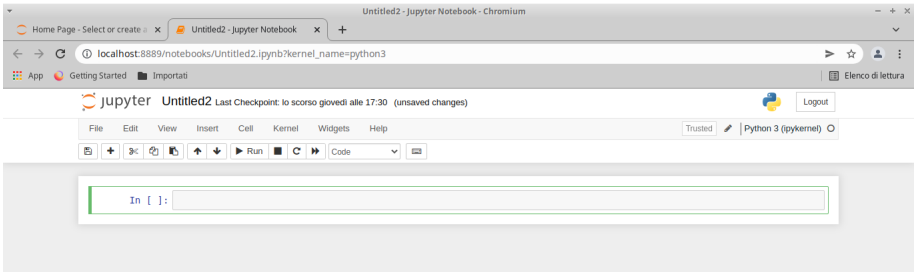
La finestra che si apre al lancio è la seguente

Essa non è altro che un gestore di file aperto sulla nostra home directory e ci invita, come sta scritto nella prima riga visibile in alto, a selezionare o a creare un notebook.

Se vogliamo lavorare su un notebook già avviato lo possiamo cercare scorrendo la parte inferiore della finestra, dove abbiamo l'elenco delle sottodirectory e dei file contenuti nella nostra home directory, e caricarlo cliccandoci sopra.

Se vogliamo creare un nuovo notebook apriamo il menu contenuto nella voce NEW sulla destra e scegliamo il linguaggio di programmazione che intendiamo utilizzare per il codice che vi inseriremo.
Come si vede dal menu che compare, potremmo anche scegliere di fare altro, come un semplice file di testo. Ma direi che per fare queste altre cose è inutile disturbare Jupyter Notebook.

Sia che abbiamo caricato un notebook esistente sia che ne abbiamo creato uno nuovo, lo ritroviamo in una nuova finestra del nostro browser predefinito che si aggiunge alla home page vista prima.



L'illustrazione mostra un notebook nuovo.

Abbiamo una barra di menu esplorando la quale, sia pure in lingua inglese, veniamo informati di cosa possiamo fare.

Appena sotto abbiamo una barra di strumenti iconizzati che velocizzano alcune cose, le più ricorrenti, che potremmo fare con le voci di menu. Passando il mouse sulle varie icone siamo informati, in lingua inglese, di ciò che otteniamo cliccando su di esse.

Il tutto è abbastanza intuitivo.

La prima cosa utile che possiamo fare è dare un nome al notebook e lo facciamo cliccando sulla dizione `UntitledX` che compare sopra la barra di menu ed inserendo il nome voluto nella finestra di dialogo che si apre.

Ora la cosa più importante da capire è come si lavora nell'editor sottostante che, come si vede nell'illustrazione, presenta una finestra di immissione dati da tastiera, chiamata cell.

Tutto ciò che vogliamo far comparire nel notebook da scambiare con altri o da utilizzare per produrre un documento deve essere immesso utilizzando finestre come questa.

Nell'illustrazione la finestra è predisposta, ed è la situazione di default, per l'immissione di codice: lo si vede dalla scritta `In [ ]:` sulla sinistra e dalla scritta `Code` che compare nel penultimo widget della barra degli strumenti.

Nell'ultima finestrella della barra di menu vediamo che il codice deve essere in linguaggio Python3, secondo la scelta effettuata nel momento in cui abbiamo creato il notebook da NEW. Importante ricordare che il linguaggio scelto non è modificabile una volta creato il notebook e, nello stesso notebook, non possiamo alternarlo con altri linguaggi.

La finestra che vediamo nell'illustrazione è selezionata e lo si vede dal fatto che è intelaiata in una cornice con una barretta verticale di colore blu sulla sinistra.

Per inserire il codice dobbiamo cliccare all'interno dello spazio di inserimento in modo che compaia il cursore lampeggiante.

In presenza di più finestre, che si possono aggiungere cliccando sul tasto con l'icona + nella barra degli strumenti, per selezionare la finestra che interessa basta cliccarci sopra.

Per inserire del normale testo con eventuali arricchimenti (illustrazioni e altri contenuti multimediali) dobbiamo commutare la finestra di immissione dati selezionata in modo che si predisponga all'immissione di testo.

Lo possiamo fare aprendo la finestrella in cui è scritto `Code` nella barra degli strumenti e scegliendo `Markdown`.

La finestra predisposta per l'immissione di testo si riconosce per la mancanza della scritta `In [ ]:` sulla sinistra e dalla scritta `Markdown` che compare nel penultimo widget della barra degli strumenti.
Questa scritta richiama il fatto che anche il così detto testo che inseriamo in questa finestra, in realtà è un codice, precisamente è un testo scritto con un linguaggio di markup che si chiama Markdown.

Un modo veloce per commutare la finestra selezionata è il seguente:
. premendo il tasto ESC e poi il tasto `y` si ottiene una finestra per il codice,
. premendo il tasto ESC e poi il tasto `m` si ottiene una finestra per il markdown.
Se la finestra non è ancora pronta per l'inserimento (non è presente il cursore lampeggiante) si può evitare di premere il tasto ESC e basta premere i tasti `y` o `m`.

Lavorando nella finestra di immissione, con la pressione del tasto INVIO passiamo ad una riga successiva all'interno della finestra e con la pressione del tasto MAIUSCOLO (SHIFT) insieme al tasto INVIO provochiamo l'esecuzione di quanto abbiamo immesso nella finestra.

Appena fatto questo si crea una nuova finestra, per default predisposta per l'inserimento di codice.

Tutto ciò che abbiamo fatto nel notebook lo possiamo salvare in un file con estensione `.ipynb`.

Basta che clicchiamo sul pulsante  , il primo nella barra degli strumenti.

Troveremo così il file con il nome che abbiamo dato al notebook e l'estensione `.ipynb` nella home directory e un suo backup, che in ter-

minologia Jupyter si chiama Checkpoint, nella sottodirectory nascosta `.ipynb_checkpoints`.

Questo file può essere ricaricato in un Jupyter Notebook, anche su un computer diverso, che abbia le stesse attrezzature software di quello su cui è stato creato, per proseguire il lavoro, per rieseguire il codice, modificarlo, migliorarlo, ecc.

Il file deve essere accompagnato dagli eventuali contenuti multimediali linkati nel testo e dai file di dati esterni eventualmente utilizzati dal codice.

L'esportazione in altri formati avviene da menu FILE ▷ DOWNLOAD AS scegliendo il formato nel menu che si apre.

Anche le esportazioni in formato `.html` e `.tex`, se destinate ad essere utilizzate su computer diversi, vanno accompagnate dagli eventuali contenuti multimediali linkati nel testo.

L'unico formato da opera definitivamente compiuta è il PDF: il file prodotto in questo formato viaggia da solo ma non è modificabile.

Per produrlo attraverso il formato HTML, avendo installato sul computer il browser Chromium, basta scrivere a terminale, posizionati nella home directory, il comando

```
jupyter nbconvert --to webpdf --allow-chromium-download <nome_file>.ipynb
```

La chiusura in modo ordinato di Jupyter Notebook dovrebbe avvenire secondo la seguente procedura.

. chiudere il tab del notebook nel browser cliccando sul simbolo `x` sulla destra nella tacca,

. nella home page che rimane aperta cliccare sulla scheda RUNNING,

. cliccare sul pulsante SHUT DOWN corrispondente al notebook da chiudere,

. non appena arriva la conferma che il notebook è stato chiuso, chiudere il browser,

. cliccare sulla finestra terminale che si era aperto al momento del lancio di Jupyter Notebook,

. chiudere il terminale con due `Ctrl-C` ravvicinati.

Editing del codice

Il codice può essere inserito ed eseguito comando per comando, come se lavorassimo in una shell, o per blocchi di comandi fino a formare un programma completo.

Editing del contorno al codice

Per gli elementi diversi dal codice in linguaggio di programmazione utilizziamo la finestra predisposta per il linguaggio Markdown.

Qui di seguito espongo le marcature Markdown riconosciute da Jupyter Notebook per creare documenti anche di un certo impegno e varietà di contenuto.

Testo

Il normale testo non formattato va semplicemente scritto.

Titoli e intestazioni vanno scritti premettendo al testo da uno a cinque cancelletti (#) e lasciando uno spazio tra i cancelletti e il testo. Con un solo cancelletto si realizza il titolo di dimensione maggiore e tanti più cancelletti inseriamo, tanto più piccolo sarà il titolo.

Per andare a capo prima della fine riga occorre inserire la marca `<br>`.

Per iniziare un nuovo capoverso andando a capo occorre inserire una riga bianca.

Ciò che si vuole scrivere in corsivo va incluso tra due asterischi (*testo*).

Ciò che si vuole scrivere in grassetto va incluso tra due doppi asterischi (**testo**).

Ciò che si vuole scrivere in colore diverso dal nero di default va incluso tra le marche `<font color = "nome_colore">` e `</font>` senza lasciare spazi bianchi tra marche e testo.
I nomi colore riconosciuti sono: gray, silver, teal, yellow, green, lime, olive, red, fuchsia, maroon, aqua, blue, navy, purple (oltre a black e white, inutili in questa sede).

Un testo da indentare va preceduto dalla marca > senza lasciare spazio bianco tra marca e testo.

Elenchi non numerati vanno inseriti facendo precedere le voci da asterisco e lasciando uno spazio tra l'asterisco e la voce.

Elenchi numerati vanno inseriti facendo precedere le voci da `1.` e lasciando uno spazio tra il punto e la voce. La numerazione progressiva delle voci avviene automaticamente.

Formule ed espressioni matematiche

Vanno inserite secondo la sintassi LaTeX
. scrivendole tra due marche $ se devono essere riprodotte sulla riga stessa,

. scrivendole tra due marche $$ se devono essere riprodotte centrate in riga dedicata.

Esempi:

Con questo inserimento

```
Questa è una formula $y=\frac{\sin(x)}{x+1}$
```

si produce questo

Questa è una formula $y = \frac{\sin(x)}{x+1}$

Con questo inserimento

```
Questa è una formula $$\frac{\sin(x)}{x+1}$$
```

si produce questo

Questa è una formula

$$\frac{\sin(x)}{x+1}$$

Con questo inserimento

```
$$\lim_{n \to \infty}\sum_{k=1}^n \frac{1}{k^2}=\frac{\pi^2}{6}$$
```

si produce questo

$$\lim_{n\to\infty}\sum_{k=1}^{n}\frac{1}{k^2}=\frac{\pi^2}{6}$$

Per la sintassi LaTeX relativa alla creazione di formule ed espressioni matematiche rimando al Capitolo riguardante la matematica in Lorenzo Pantieri o Lorenzo Pantieri & Tommaso Gordini - L'arte di scrivere con LaTeX, che si trova all'indirizzo
http://www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf.

Fin dove può servire vale anche la sintassi Plain TeX, per la quale rimando al Capitolo 4 del manualetto plain_tex.pdf allegato al mio articolo del giugno 2022 sul blog *www.vittal.it* intitolato «Plain Tex».

Il rendering delle formule sulle pagine in formato HTML avviene senza che sia installato il software per LaTeX.

Link

Possiamo creare link esterni, verso un url esterno al documento, oppure interni, verso un punto all'interno del documento.

Il link esterno si crea con la sintassi

```
[nome_link](nome_url)
```

Per esempio, per creare un link verso l'indirizzo dove si trova la guida LaTeX richiamata alla fine del paragrafo precedente, scriviamo

```
[guida_latex](http://www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf)
```

Il link interno si crea innanzi tutto inserendo nel punto verso il quale si vuole linkare

```
<a id="nome_link"></a>
```

e inserendo nel punto dal quale si vuole linkare

```
[nome_link](#nome_link]
```

Contenuti multimediali

Possiamo inserire nel documento anche immagini, audio e video.

Per le immagini usiamo la sintassi

```
![titolo_immagine](path_al_file)
```

Per esempio, con

```
![grafico](/home/vittorio/Immagini/logaritmo.png)
```

inserisco nel documento, con la descrizione «grafico» l'immagine del grafico di una funzione che si trova nella directory Immagini della mia home.

In alternativa possiamo usare la sintassi del linguaggio HTML, attraverso la quale abbiamo il vantaggio di poter regolare le dimensioni dell'immagine:

```
<img src="path" alt="titolo_immagine" height="pixel" width="pixel"/>
```

Per esempio, con

```
<img src="/home/vittorio/Immagini/logaritmo.png" alt="grafico" height="200" width="200"/>
```

inserisco nel documento la stessa immagine di prima ridotta a 200 pixel per 200.

Per l'audio usiamo la sintassi del linguaggio HTML

```
<audio src="path_al_file" controls/>
```

Per il video usiamo pure la sintassi del linguaggio HTML

```
<video width="pixel"_height="pixel" src="path_al_file" controls/>
```

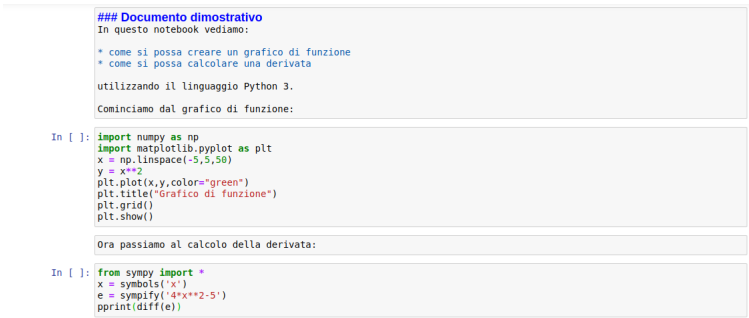
Le immagini inserite nel documento le troviamo presenti in tutte le esportazioni del documento stesso, siano esse in formato HTML, siano esse in formato PDF.

I file audio e video compariranno con una inutile traccia nelle esportazioni in PDF via HTML e questa traccia nemmeno si vedrà nelle esportazioni in PDF via LaTeX. Saranno invece ben presenti e potranno funzionare facendoci udire suono o facendoci vedere immagini soltanto dal formato HTML, a patto che i relativi file siano raggiungibili. Un modo per assicurarci questo è di aprire il file `.html` esportato da Jupyter Notebook e salvarlo con SALVA PAGINA CON NOME...: ciò crea una copia

del file `.html`, alla quale possiamo dare un nuovo nome, e una directory, con lo stesso nome, contenente i file multimediali inseriti nel documento. Trasferendo file e directory, insieme, possiamo riprodurre i nostri contenuti multimediali dove vogliamo.

Esempio

Questo è un notebook con evidenziato quanto inserito nelle varie finestre per ottenere il testo e i codici eseguibili



```
### Documento dimostrativo
In questo notebook vediamo:

* come si possa creare un grafico di funzione
* come si possa calcolare una derivata

utilizzando il linguaggio Python 3.

Cominciamo dal grafico di funzione:
```

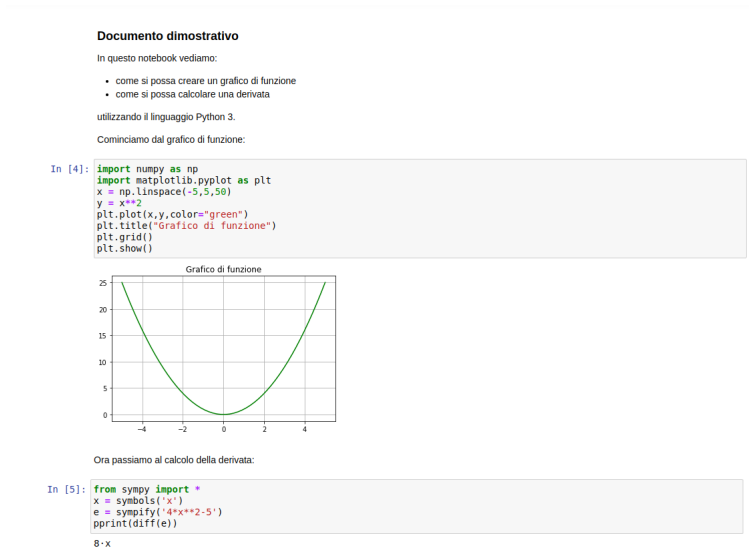
```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-5,5,50)
y = x**2
plt.plot(x,y,color="green")
plt.title("Grafico di funzione")
plt.grid()
plt.show()
```

```
Ora passiamo al calcolo della derivata:
```

```
In [ ]: from sympy import *
x = symbols('x')
e = sympify('4*x**2-5')
pprint(diff(e))
```

Vi si alternano due caselle di testo e due di codice.

Selezionando ciascuna casella ed eseguendone il contenuto attraverso la pressione dei tasti Shift+Invio il notebook assume questa forma



Documento dimostrativo

In questo notebook vediamo:

- come si possa creare un grafico di funzione
- come si possa calcolare una derivata

utilizzando il linguaggio Python 3.

Cominciamo dal grafico di funzione:

```
In [4]: import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-5,5,50)
y = x**2
plt.plot(x,y,color="green")
plt.title("Grafico di funzione")
plt.grid()
plt.show()
```


Ora passiamo al calcolo della derivata:

```
In [5]: from sympy import *
x = symbols('x')
e = sympify('4*x**2-5')
pprint(diff(e))
```

```
8·x
```

Qui vediamo che il testo è diventato testo formattato e sotto alle due finestre di codice vediamo il risultato dell'esecuzione del codice stesso: il grafico della funzione x^2 indicata nel codice `In [4]` e il valore $(8x)$ della derivata della funzione $4x^2 - 5$ indicata nel codice `In [5]`.

Questo è ciò che appare nel file `.ipynb` dove abbiamo memorizzato il notebook, file che è ancora uno strumento di lavoro: per noi stessi sul computer dove l'abbiamo creato e per chiunque, su un altro computer, abbia il software Python 3 arricchito dei moduli matplotlib e sympy importati nei due spezzoni di codice eseguibile, è possibile intervenire per modificare il testo, previa selezione della zona in cui è contenuto e doppio click sulla relativa finestra, o per modificare il codice eseguibile, per migliorarlo o per applicarlo diversamente.

Se, per esempio vogliamo che il grafico si riferisca alla funzione $y = sin(x)$ nell'intervallo tra 0 e 2π, basta che nella finestra `In [4]`:
. tra le importazioni indichiamo anche `from math import *`,
. sostituiamo la riga x = np.linspace(0,2*pi,50)
alla riga `x = np.linspace(-5,5,50)`
. sostituiamo la riga `y = np.sin(x)`
alla riga `y = x**2`.

Allo stesso modo possiamo calcolare derivate di altre funzioni modificando, nella finestra `In [5]`, l'espressione della funzione da derivare.

Ovviamente tutto questo non si può fare sui rendering del notebook dopo la sua esportazione in formato `.html`, `.tex` o `.pdf` nei quali è semplicemente visibile il contenuto statico del notebook.

9.2 Appendice B - Google Colaboratory

Google Colaboratory ci consente di scrivere codice in linguaggio Python su un browser web e di ottenerne l'elaborazione da parte di Google, con potenze di molto superiori a quelle normalmente disponibili sui nostri personal computer, tablet o addirittura smartphone, senza dover affrontare particolari configurazioni o predisposizioni.

Fino a certi livelli e pur con potenza variabile nel tempo in maniera imprevedibile possiamo fruire gratuitamente di questo servizio.

Per avere certezze sulla disponibilità dei livelli di servizio occorre ovviamente sottoscrivere versioni a pagamento.

Se le nostre esigenze non necessitano di particolari pianificazioni e, soprattutto, se da inesperti non abbiamo contezza di ciò che veramente ci serve abbiamo comunque la possibilità di provare ad «imbucarci» gratuitamente e vedere cosa succede.

Senza, peraltro, aver provato a risolvere i problemi con la nostra attrezzatura casalinga.

Ritengo, infatti, che l'avere a disposizione il linguaggio Python nel cloud non debba farci perdere l'interesse ad avere a disposizione lo stesso linguaggio a casa nostra, visto che non costa nulla in termini economici e costa poco in termini di occupazione del nostro disco fisso.

In altri termini ritengo che la vera utilità di Google Colaboratory sia semplicemente quella di fornirci una potenza di calcolo superiore a quella che ci può dare il nostro computer, se e quando serve.

Preliminare

Visto che Google Colab (da qui in poi uso il termine abbreviato) è un servizio Google, per potervi accedere dobbiamo avere un account Google, quello che ci può servire anche per la posta elettronica e che, appunto, si identifica con un formato da indirizzo di posta elettronica del tipo

```
nome@gmail.com
```

Molto probabilmente già l'abbiamo e l'abbiamo sicuramente se possediamo uno smartphone equipaggiato Android.

Se ancora non l'abbiamo possiamo facilmente ottenerlo andando all'indirizzo

https://support.google.com/accounts/answer/27441?hl=it#

dove scopriremo, se non ci interessa avere il servizio di posta elettronica gmail, di poter utilizzare anche il nostro di altra natura.

Attrezzatura

Per accedere a Google Colab ci serve semplicemente un browser web e un collegamento a Internet.

I browser su cui è testato il funzionamento ottimale del servizio sono Chrome, Firefox e Safari.

Probabilmente funziona bene anche con altri browser: posso personalmente garantire che ciò avviene per Chromium.

Non ha nessuna importanza il sistema operativo e la macchina su cui lavoriamo, basta che abbiamo la possibilità di collegarci a Internet e di utilizzare un browser web adatto al servizio.

Accesso al servizio

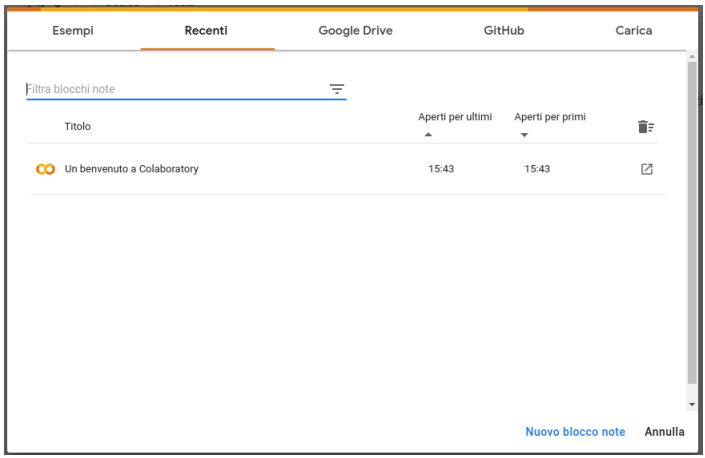
Il modo normale per accedere a Google Colab è quello di andare all'indirizzo

https://colab.research.google.com/?hl=it

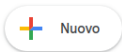
Se abbiamo il browser con l'accesso a Google già predisposto il collegamento al servizio avverrà immediatamente.

In caso contrario ci verrà richiesto di accedere a Google tramite l'account di cui abbiamo parlato prima, inserendo nome dell'account e password.

Per questa via, ad accesso avvenuto, ci troviamo di fronte questa finestra



Avendo il browser con l'accesso a Google già predisposto, possiamo accedere a Google Colab anche attraverso Google Drive.
In questo caso, dal momento che le finestre di lavoro di Google Colab vengono salvate nel Drive in file con estensione `.ipynb`, possiamo aprire una finestra di lavoro già presente con doppio click su di essa.
Oppure possiamo aprire una nuova finestra in cui lavorare cliccando sul pulsante



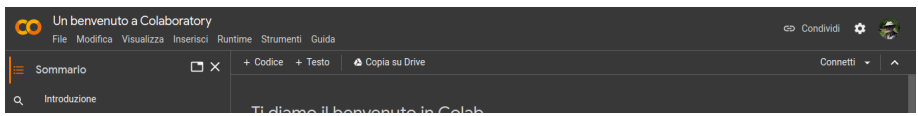
nella finestra di Google Drive e scegliere dal menu che si presenta ALTRO ▷ GOOGLE COLABORATORY.

Con l'accesso normale, se è la prima volta che utilizziamo Google Colab, ci si presenta la finestra vista prima.

Essa si apre, per default, sulla scheda RECENTI, dove sono elencate le ultime cose che abbiamo fatto in modo da poterle richiamare.

Cliccando sulla voce visualizzata, UN BENVENUTO A COLABORATORY, oppure cliccando sul tasto ANNULLA apriamo una finestra di benvenuto in cui ci viene descritto il servizio:

Se clicchiamo sul pulsante in alto a destra attiviamo la visualizzazione di una intestazione



utile in quanto contiene un menu che ci dà modo di avviare i nostri lavori.

Sempre in alto a destra abbiamo il pulsante con cui apriamo una finestra di dialogo autoesplicativa per alcune impostazioni.

La finestra con sfondo scuro appena riprodotta deriva dalla scelta della modalità `dark` come tema del sito, in luogo della modalità `adaptive` di default che caratterizza la finestra di apertura precedentemente riprodotta.

Oltre alla scheda RECENTI, la finestra di apertura contiene le seguenti altre schede:

. ESEMPI: che contiene file di esempio,

. GOOGLE DRIVE: che visualizza e ci consente di aprire file salvati su Google Drive,

. GITHUB: che consente di collegarsi ad un account GitHub,

. CARICA: che consente di caricare un file da una directory locale.

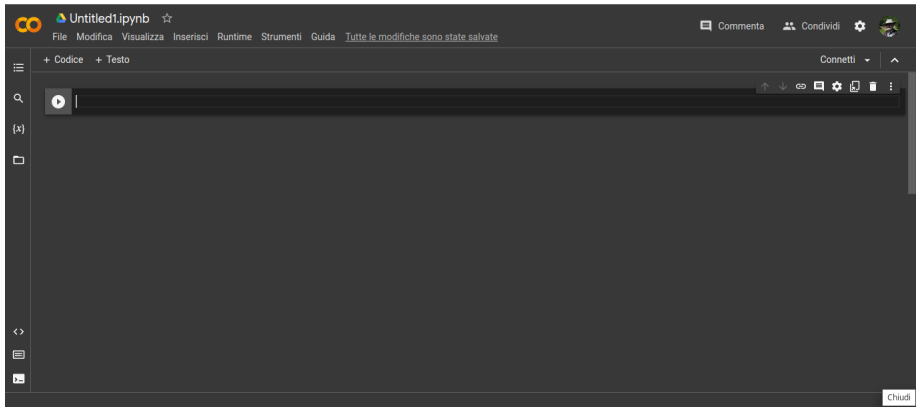
Come funziona

Dalla finestra di apertura possiamo avviare un nuovo lavoro cliccando sulla voce NUOVO BLOCCO NOTE in fondo a destra.

Per proseguire o modificare un lavoro già avviato possiamo avvalerci della scheda RECENTI o GOOGLE DRIVE oppure, se il file è stato salvato in una directory locale, CARICA.

Possiamo fare le stesse cose, se siamo nella finestra di benvenuto, agendo da menu FILE ▷ NUOVO BLOCCO NOTE, oppure FILE ▷ APRI BLOCCO NOTE e FILE ▷ CARICA BLOCCO NOTE.

Una nuova finestra di lavoro si presenta così



e scopriamo che si tratta di un notebook Jupyter: da qui il ricorrente appellativo di blocco note o, in inglese, notebook con cui si identificano le finestre di lavoro di Google Colab.

Per cominciare a lavorare dobbiamo connetterci e lo facciamo cliccando sulla voce CONNETTI, in alto, sulla destra, sotto la barra di intestazione. Dopo qualche istante saremo connessi e, in luogo della voce CONNETTI, avremo una doppia finestrella su cui possiamo posizionare il mouse ottenendo

conferma di essere connessi ad un motore Python3, con quanto ci è inizialmente assegnato di una RAM di complessivi 12,68 GB e quanto spazio ci è inizialmente assegnato su un disco di 107,72 GB.

La prima cosa che facciamo è assegnare un nome al nostro notebook, selezionandone con il mouse, nella barra dell'intestazione, quello assegnato per default (`Untitled1`) e modificandolo a nostro piacimento, conservando l'estensione `.ipynb`.

All'apertura il notebook presenta la finestra per l'immissione di codice

A cella selezionata abbiamo, in alto a destra, una piccola barra di strumenti: passando il mouse sulle icone troviamo le spiegazioni.

Il codice inserito si esegue cliccando sul triangolino sulla sinistra oppure premendo insieme i tasti MAIUSCOLO e INVIO della tastiera.

Per aprire una finestra che serva per immettere testo clicchiamo sulla voce + TESTO nella barra degli strumenti ed otteniamo l'inserimento nel nostro note book della riga

Con la finestra aperta per inserire testo o modifiche, oltre alla solita piccola barra di strumenti in alto a destra, abbiamo una barra di strumenti con icone che facilitano l'immissione di tag nel linguaggio Markdown, utilizzato in Jupyter Notebook per scrivere testi formattati.

Per fissare nel notebook il testo chiudiamo la finestra di inserimento premendo insieme i tasti MAIUSCOLO e INVIO della tastiera.

Per tornare all'inserimento di codice, posizionati nel punto del notebook dove vogliamo fare l'inserimento, clicchiamo sulla voce + CODICE e così via.

Tutto ciò che facciamo viene automaticamente salvato nel nostro Google Drive, in una directory Colab Notebooks. A fine lavoro abbiamo comunque la possibilità di salvare utilizzando le voci del menu, una delle quali ci consente di salvare il nostro notebook su GitHub.

Possiamo stampare il nostro lavoro da menu FILE ▷ STAMPA, avendo modo di scegliere tra stampante collegata oppure salvataggio in formato PDF.

Con menu FILE ▷ SCARICA possiamo salvare il file del notebook in locale sul nostro computer, dove possiamo lavorarlo avvalendoci dello Jupyter Notebook che abbiamo sul nostro computer ed eventualmente ricaricarlo poi su Google Colab da menu FILE ▷ CARICA BLOCCO NOTE.

Cosa c'è dietro

Già abbiamo visto, esplorando la modalità di connessione, di essere connessi ad un motore Python3 (Python2 non è più supportato).

Potremmo volerne conoscere la versione e possiamo farlo inserendo in una finestrella del codice il seguente codice

```
import platform
print(platform.python_version())
```

che, eseguito, fornisce il risultato

Altra curiosità quella di conoscere il sistema operativo su cui lavoriamo nel cloud e possiamo appagarla inserendo in una finestrella del codice il seguente codice

```
import platform
print(platform.system())
```

che, eseguito, fornisce il risultato

Possiamo anche conoscere la versione di singoli moduli che importiamo per l'uso. Per esempio, per conoscere la versione di numpy che ci viene fornita, inseriamo il codice

```
import numpy as np
print (np.__version__)
```

ottenendo

Risorse hardware e loro disponibiltà

Da quanto visto finora sappiamo che la risorsa software che ci offre Google Colab è un interprete Python 3 utilizzabile attraverso Jupyter Notebook su sistema operativo Linux.

Sul piano dell'hardware, nella versione gratuita e per default, ci vengono messi a disposizione una RAM fino a 12,68 GB, un disco di memoria fino a 107,72 GB e una CPU senza acceleratore.

Penso che il problema minore sia quello del disco di memoria, che ha una capacità che mi sembra veramente elevata, almeno per contenere dati numerici.

Quanto alla RAM possiamo avere un'idea di cosa significa avere a disposizione 12,68 GB valutando che in questa dimensione può essere contenuta una lista di poco oltre 1.500.000 elementi.
Lo possiamo sperimentare eseguendo in un notebook Google Colab il seguente codice

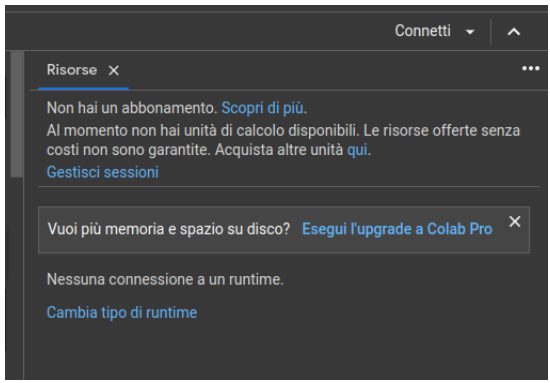
```
l = []
while(True):
    l.append('ciao')
```

che va in crash per esaurimento della RAM quando nella lista abbiamo la parola «ciao» ripetuta per circa 1.520.000 volte.

La CPU senza acceleratore è praticamente una cosa simile a quella che probabilmente abbiamo sul nostro computer casalingo.

Sempre nella versione gratuita possiamo utilizzare il nostro notebook continuativamente per 12 ore e dopo 30 minuti di inattività veniamo disconnessi.

Se clicchiamo, in alto a destra, sul triangolino di fianco alla voce CONNETTI e, dal menu che si apre, scegliamo la voce RISORSE apriamo questa finestrella



nella quale ci viene ricordato che siamo degli «imbucati» senza abbonamento, che non possediamo unità di calcolo e che nulla ci è garantito circa la disponibilità di risorse.

In realtà questa mancata garanzia si riferisce unicamente alle risorse di CPU.

Per le risorse in termini di memoria e spazio su disco abbiamo quelle viste sopra e per averne di più dobbiamo sottoscrivere l'abbonamento a Colab Pro (11,28 € al mese) o Colab Pro+ (51,54 € al mese).

Quanto, invece, alla CPU, il non avere unità di calcolo disponibili significa semplicemente che non ci è garantito di poter rafforzare la CPU, non che non lo possiamo fare.

Ci possiamo pertanto provare.

Se nella finestrella appena sopra riprodotta clicchiamo sulla dizione CAMBIA TIPO DI RUNTIME (oppure scegliamo da menu RUNTIME ▷ CAMBIA TIPO DI RUNTIME) possiamo accedere ad una finestrella nella quale aprire un menu a tendina con cui possiamo scegliere un acceleratore hardware tra Nessuno, GPU e TPU.

A scelta effettuata, se non compaiono messaggi che lo negano, possiamo dedurne di essere riusciti a rafforzare la CPU con l'acceleratore scelto.

Con GPU (Graphics Process Unit) possiamo arrivare ad una velocità di elaborazione dalle 30 alle 80 volte superiore a quella della CPU senza acceleratore.

Con TPU (Tensor Process Unit) possiamo arrivare ad una velocità dalle 15 alle 30 volte superiore a quella della GPU.

Per avere la certezza di ottenere queste accelerazioni e di averle a disposizione per elaborazioni di una certa durata dobbiamo acquistare unità di calcolo.

Possiamo farlo senza avere abbonamenti, pagando per quello che usiamo, acquistando 100 unità per € 11,28 oppure 500 unità per € 51,54.

Sottoscrivendo l'abbonamento a Colab Pro abbiamo 100 unità al mese incluse nell'abbonamento, che sono 500 nel caso di abbonamento a Colab Pro+.

Come caricare i dati in Google Colab

Google Colab ha una directory radice che si chiama `content` nella quale deve essere presente il file dei dati da elaborare.

Il formato di questo file deve essere della specie riconosciuta dal modulo Pandas di Python. Tra i tanti cito i più noti: `.csv`, `.excel`, `.json`, `.html`, `.sas`, ecc.

Per il caricamento dei dati in questa directory abbiamo procedure diverse, a seconda di dove è collocato il file di dati da caricare.

Dati presenti nel filesystem locale

Se il file dei dati è archiviato sul computer su cui stiamo lavorando lo carichiamo eseguendo le seguenti istruzioni da scrivere come codice:

```
from google.colab import files
uploaded = files.upload()
```

che, all'esecuzione, mostra quanto segue

Cliccando su SCEGLI FILE apriamo una finestra di navigazione attraverso la quale raggiungiamo il file da caricare e il file, una volta caricato, è raggiungibile nella directory `/content` di Google Colab.

Questo è un esempio di elaborazione su un file `dati.csv` presente sul computer su cui lavoriamo e caricato in Google Colab secondo la procedura appena vista:

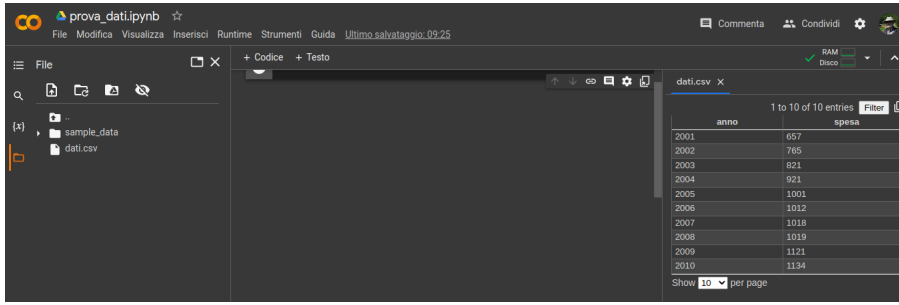
Possiamo caricare il file di dati presente sul computer su cui stiamo lavorando anche in quest'altro modo.

Se clicchiamo sull'icona  presente nella barra verticale sulla sinistra della finestra del notebook abbiamo accesso alla directory `/content`, nella quale, se non abbiamo ancora caricato dati, è presente una sottodirectory `sample_data`, contenente alcuni file utili per esercitazione.

Per caricare il nostro file locale possiamo cliccare sull'icona  e andare a scegliere il file attraverso la finestra di navigazione che si presenta e caricarlo.

Il file caricato è ora visibile nella directory `/content`.

Con doppio click sul suo nome possiamo anche vederne il contenuto.



Dati presenti in Google Drive

Se il file dei dati su cui lavorare si trova in Google Drive, dobbiamo montare il Drive in Google Colab.

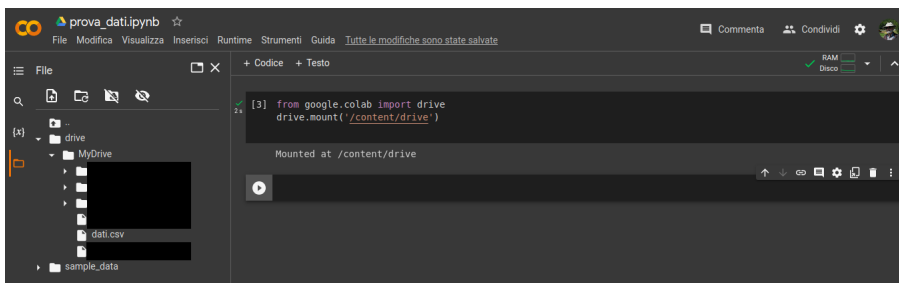
Lo possiamo fare eseguendo il seguente codice

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

e seguendo i passaggi che ci vengono proposti in una serie di finestre di dialogo.

In maniera più rapida otteniamo lo stesso risultato cliccando sull'icona  presente nella barra verticale sulla sinistra della finestra del notebook e successivamente sull'icona  .

A drive montato la directory `/content` appare così



e il path per raggiungere il file di dati con Pandas diventa `'/content/drive/MyDrive/dati.csv'`.

Dati da Internet

La rete è una ricchissima fonte di dati di ogni tipo.

I siti che rendono disponibili dati molte volte offrono all'utente la possibilità di scaricare i dati selezionati semplicemente premendo determinati pulsanti o addirittura cliccando sul nome del dataset. E' il

caso, per esempio, del nostro Istituto Nazionale di Statistica, all'indirizzo *https://www4.istat.it/it/istituto-nazionale-di-statistica*, dove basta cliccare sul nome del dataset e viene scaricato il relativo file in formato Excel.

Altre volte dobbiamo provvedere noi a scaricare il dataset e lo possiamo fare su Google Colab eseguendo la seguente istruzione

`!wget <indirizzo_url> -P '/content'`

per avere il dataset disponibile nella directory radice di Google Colab, da dove lo possiamo utilizzare, potendolo anche scaricare se vogliamo salvarlo nel file system del computer.

Teniamo comunque presente che il modulo Pandas, una volta che conosciamo l'indirizzo dove si trova il file del dataset e il formato del file che lo contiene, può collegarsi direttamente ad esso senza bisogno di scaricare nulla.

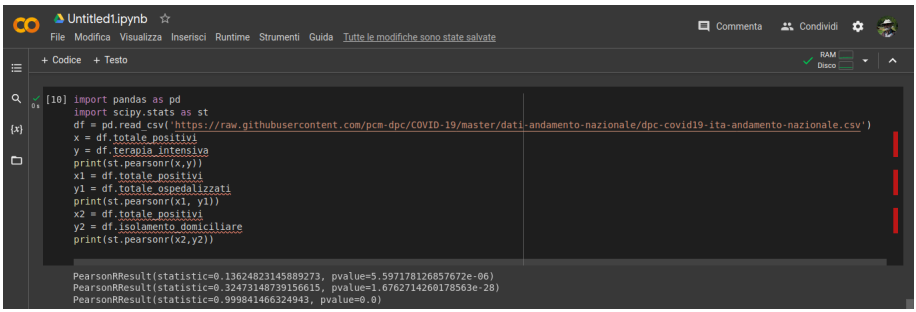
Esiste, per esempio, su GitHub un dataset che contiene tutto ciò che è avvenuto nella famigerata esperienza della recente epidemia Covid.
Il file del dataset è disponibile all'indirizzo

https://github.com/pcm-dpc/COVID-19/blob/master/dati-andamento-nazionale/dpc-covid19-ita-andamento-nazionale.csv

Per avere ciò che ci serve dobbiamo optare per il formato Raw, cliccando sul relativo pulsante nella barra degli strumenti che si trova nell'intestazione della tabella dei dati.
In tal modo l'indirizzo che ci interessa diventa

https://raw.githubusercontent.com/pcm-dpc/COVID-19/master/dati-andamento-nazionale/dpc-covid19-ita-andamento-nazionale.csv

Con l'esercizio qui riportato, collegando direttamente Pandas all'URL, ho verificato quale sia stata la correlazione tra la serie giornaliera dei positivi e i ricoverati in terapia intensiva, i normali ospedalizzati e gli isolati domiciliari:



I risultati evidenziano come tra positivi e ricoverati in terapia intensiva non vi sia stata alcuna correlazione (indice 0,136), come la correlazione tra positivi e ospedalizzati normali sia stata molto bassa (indice

0,325) e come invece vi sia stata correlazione perfetta tra positivi e isolati domiciliari (indice 0,999).
Come dire che il ricovero in ospedale raramente si è rivelato necessario per combattere il morbo e meno ancora si è rivelata necessaria la terapia intensiva.
Pertanto è stato brutto ma poteva anche andare peggio.