Grafica interattiva con Python (autore: Vittorio Albertoni)

Premessa

In questo manualetto, rilasciato nell'ottobre 2018, non tratto della grafica funzionale alla programmazione di software dotato della così detta Graphical User Interface (GUI): per questo rimando al manualetto su tkinter di qualche mese fa.

Tratto, invece, della grafica fine a sé stessa, cioè semplicemente destinata a creare figure di vario tipo oppure al servizio della geometria o della matematica. In quest'ultimo caso evitando il massimo che Python mette a disposizione di matematici e scienziati, cioè la libreria Matplotlib, di utilizzo tutt'altro che banale e fuori portata per i dilettanti cui generalmente rivolgo i miei lavori.

Tratto, cioè, cose alla portata di tutti, soprattutto utili per imparare e svagarsi.

Proprio a questo fine illustrerò come usare ciò che propongo utilizzando la shell di Python, in modo da poter vedere passo dopo passo i risultati della nostra creatività e rendersi consapevoli in tempo reale dell'effetto che fanno le nostre istruzioni in linguaggio Python.

Le istruzioni che inseriamo nella shell sono ovviamente le stesse che, scritte in un file di testo salvato con estensione .py, formano uno script eseguibile con l'interprete Python.

Indice

1 La shell di Python **2**

2 Tkinter **3**

3 Graphics **4**

4 Pygraph **8**

4.1 Pycart 9
4.2 Pyturtle 11
4.3 Pyplot 12
4.4 Pyig 13

1 La shell di Python

In informatica una shell è l'involucro (la conchiglia, appunto), la parte visibile di un sistema operativo, attraverso la quale è possibile interagire con il sistema stesso scrivendo comandi con la tastiera del computer. I sistemi Linux e Mac OS X hanno una shell, praticamente la stessa di derivazione Unix, chiamata terminale e il sistema Windows ha una sua shell, chiamata prompt dei comandi.

Python ha anche lui una shell, la shell di Python.

Essa si apre scrivendo nella shell del sistema operativo (terminale in Linux e Mac OS X, prompt dei comandi in Windows) il comando `python` (nel caso sul sistema operativo siano disponibili entrambe le versioni di Python, la 2 e la 3, la shell di Python 3 si apre con il comando `python3`).

Si chiude premendo insieme i tasti CTRL e D.

La seguente figura 1 mostra una shell di Python 3 su sistema Linux.

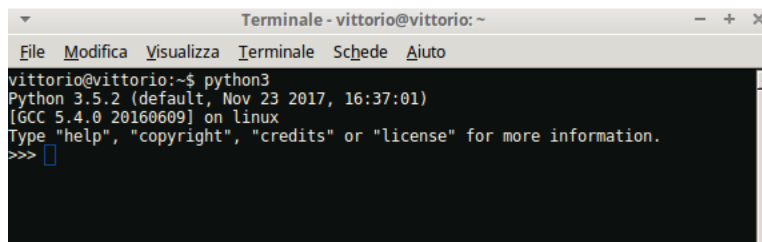

Figura 1: Shell di Python 3 su Linux

Nella prima riga abbiamo il comando immesso nella shell di sistema e nelle righe successive abbiamo l'intestazione della shell di Python con il suo prompt, indicato dai tre caratteri >>>.
Se sulla riga del prompt scriviamo un'istruzione in linguaggio Python 3 e diamo INVIO, l'istruzione viene immediatamente eseguita.
Per esempio, se scriviamo l'istruzione `print(3**2)`, che in linguaggio Python 3 significa «scrivi 3 elevato al quadrato» e premiamo INVIO, immediatamente nella riga successiva compare il risultato 9 e si ricrea una nuova riga di prompt, pronta (da qui, appunto, il termine prompt) a ricevere un'altra istruzione.

Questa capacità della shell di eseguire le istruzioni al volo e di presentarcene immediatamente i risultati ci aiuta molto a studiare l'effetto delle varie istruzioni del linguaggio ed ha un enorme potenziale didattico.

Ma possiamo disporre di una shell di Python molto più bella di quella che abbiamo appena visto: quella contenuta nella IDLE.
Quando installiamo Python su Windows o su Mac OS X con gli installatori scaricabili dal sito *https://www.python.org/* automaticamente installiamo anche la IDLE.
Nei sistemi operativi Linux, dove Python è sempre preinstallato, non è preinstallata la IDLE. Per installarla basta ricorrere all'installatore dei programmi in quanto si trova sicuramente nel repository.

La grande utilità della IDLE, acronimo di Integrated Development and Learning Environment, sta nei continui suggerimenti che ci fornisce quando scriviamo le istruzioni, suggerimenti che possono essere relativi ai parametri che dobbiamo inserire nelle chiamate di funzione, ai metodi disponibili nelle classi dei vari pacchetti, ecc.

La figura 2 nella pagina seguente mostra una shell da IDLE con la descrizione dei parametri necessari alla funzione `print` di Python 3.
Praticamente nulla cambia rispetto alla shell normale, salvo il fatto che, non appena scritta la parentesi aperta dopo l'istruzione print compare la finestrella dove sono riepilogati i parametri da indicare, da accettare o da modificare.
Nello specifico il valore o i valori separati da virgola da scrivere (`value, ....`), il separatore da mettere tra i valori (`sep=' '` che per default è uno spazio vuoto), il carattere di fine riga

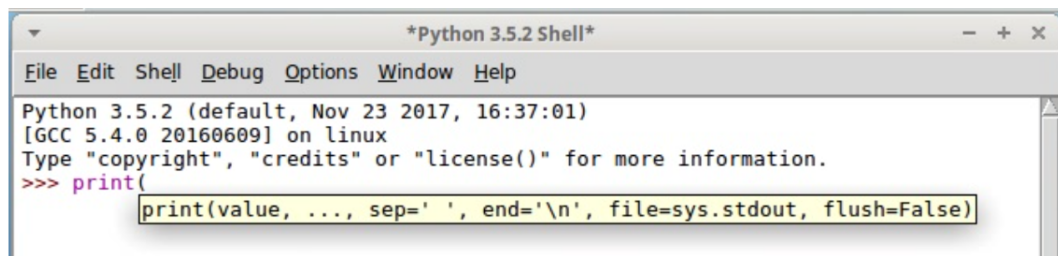

Figura 2: Shell di Python 3 su Linux nella IDLE

(`end='\n'` che per default è accapo), il file su cui scrivere (`file=sys.stdout` che per default è lo standard output, cioè il terminale in cui stiamo lavorando) e il comportamento con il buffer (`flush=False` che per default vuol dire senza flush).

Se ci vanno bene le impostazioni di default dobbiamo semplicemente scrivere i valori da stampare (tra apici se si tratta di stringhe).

Il valore può anche essere indicato come formula, come abbiamo fatto nel precedente esempio per stampare il quadrato di 3.

Tanto basta per capire perché quella di Python è una IDLE, con la L di Learning, e non una semplice IDE.

Ricordare sempre che l'istruzione scritta nella riga del prompt della shell va lanciata premendo INVIO.

Nel caso di istruzioni complesse (cicli, definizioni di funzioni, ecc.), quelle la cui prima riga si chiude con i due punti (:), premendo INVIO ci si posiziona nella riga successiva, indentata, su cui proseguire. Terminate le istruzioni, si chiude con un doppio INVIO.

2 Tkinter

Il modulo Turtle e il widget Canvas di Tkinter possono utilmente essere fruiti nella shell di Python con il vantaggio di vedere costruite le nostre creazioni grafiche man mano che inseriamo le istruzioni.

Per le istruzioni da utilizzare rimando al manualetto su Tkinter allegato all'articolo «Grafica con Python» del maggio 2018, archiviato in Programmazione sul mio blog *www.vittal.it*.

Visto che stiamo parlando di creazioni grafiche che magari ci interessa conservare su file, rammento che l'oggetto Canvas ha il metodo

```
postscript(file = '<path_e_nome_file>.eps', colormode = <valore>)
```

grazie al quale salviamo il nostro lavoro su file postscript, trasformabile e manipolabile con un software grafico come GIMP o ImageMagick.

Il valore da attribuire al parametro colormode può essere '`color`' per l'uso del colore, '`gray`' per l'uso di una scala di grigi e '`mono`' per il bianco e nero.

Per quanto riguarda figure create con Turtle, l'oggetto Turtle ha un metodo `getscreen()` che cattura la figura e la salva in un oggetto grafico:

```
<oggetto_grafico> = <identificativo_turtle>.getscreen()
```

Avendo importato, oltre al modulo turtle anche tkinter, possiamo applicare al nostro oggetto grafico il metodo postscript di Canvas con l'istruzione

```
<oggetto_grafico>.getcanvas().postscript(file ='<path_e_nome_file>.eps', colormode =<valore>).
```

In nessun caso verrà riprodotto il colore di fondo del canvas o del piano della tartaruga, che rimane bianco.

Tutto sommato si fa prima a salvare la nostra figura con una delle tante utilità di cattura dello schermo.

3 Graphics

Il modulo graphics.py è stato scritto da John Zelle come sussidio al testo «Python programming: an introduction to computer science» ed è finalizzato all'introduzione dei concetti della programmazione orientata agli oggetti. Pertanto, oltre che consentirci di giocare con la grafica, è un ottimo strumento educativo per assuefarci alla programmazione per oggetti.

Lo troviamo all'indirizzo

http://mcsp.wartburg.edu/zelle/python/

insieme ad un'ottima guida in lingua inglese, fornita in formato html e in formato PDF.
A questa rimando coloro che vogliano approfondire la conoscenza del modulo, in quanto, in questa sede, mi limito a descriverne le funzionalità principali.

Una volta scaricato il file **graphics.py** lo dobbiamo collocare nel posto giusto, e cioè:
. se usiamo Linux in `/usr/local/lib/pythonX.X/dist-packages`,
. se usiamo Windows in `C:\PythonXX\Lib\site-packages`,
. se usiamo Mac in `/Library/Frameworks/Python.framework/Versions/X.X/lib/pythonX.X/site-packages/`,
dove `X.X` sta per i numeri che identificano la versione Python installata sul computer (esempio 3.5 o 3.7, ecc.).

Trattandosi di un modulo ospitato dal repository di Python, un modo rapido e sicuro per l'installazione, ovviamente avendo installato il programma pip, è quello di digitare il seguente comando a terminale
`pip3 install graphics.py`[1].

Il modulo si appoggia alla grafica di Tkinter, pertanto avere installato Tkinter è prerequisito per il funzionamento di Graphics[2].

Per utilizzare il modulo lo dobbiamo importare scrivendo nella shell l'istruzione
`from graphics import *`

Oggetto Finestra

Il metodo costruttore dell'oggetto finestra consiste nell'istruzione
`<nome_finestra> = GraphWin(<titolo>, <larghezza>, <altezza>)`
Sicché, se scriviamo l'istruzione `f = GraphWin('PROVA', 300, 400)` nella shell, immediatamente ci si presenta sullo schermo una finestrella larga 300 pixel e alta 400 pixel, intitolata PROVA.
Se non indichiamo i parametri tra parentesi viene costruita una finestra di default intitolata Graphics Window di 200 per 200 pixel.
L'identificativo di questo oggetto finestra è la lettera f minuscola (possiamo usare una lettera o un nome qualsiasi) ed è con questo identificativo che possiamo indicarla e richiamarne i metodi.
La finestra è destinata ad ospitare gli oggetti grafici che andremo a costruire.

I principali metodi propri della finestra sono

`setBackground(`<colore>`)`
dove <colore> deve essere il nome inglese del colore, scritto tra apici, con cui desideriamo colorare la finestra[3].
Se scriviamo `f.setBackground('yellow')`, la nostra finestra f si colora immediatamente di giallo.

---

[1] Per saperne di più sul repository Python e su pip rimando al manualetto mondo_python allegato al mio articolo Python per tutti del febbraio 2017 archiviato nella categoria Programmazione sul mio blog all'indirizzo *www.vittal.it*

[2] Rammento che Tkinter è sicuramente installato se abbiamo installato la IDLE di cui si parla nel Capitolo 1.

[3] I colori normali sono generalmente indicabili con i loro nomi inglesi white, black, red, green, cyan, purple, blue, yellow, ecc. Per colorazioni intermedie sin può usare l'indicazione `color_rgb(`r, g, `b)` dove r, g e b sono i numeri tra 0 e 255 che indicano rispettivamente le quantità di rosso, giallo e blu nel mix. Così color_rgb(255, 0, 0) indicherà il rosso vivo, color_rgb(130, 0, 130) indicherà un magenta medio, color_rgb(150, 150, 150) indicherà un grigio, ecc.

plot(x, y, <colore>)
dove x e y sono le coordinate di un punto (pixel) da disegnare nella finestra con il colore indicato.
L'origine del piano di disegno, di coordinate 0 e 0 è l'angolo in alto a sinistra della finestra; le x si sviluppano verso destra e le y si sviluppano verso il basso.
Se scriviamo `f.plot(50, 50, 'red')` immediatamente nella nostra finestra f compare un puntino rosso a 50 pixel dall'alto e a 50 pixel dal lato sinistro.
Con un po' di linguaggio Python possiamo anche cominciare da qui a fare qualche disegnino.
Per esempio, se scriviamo nella shell quanto segue

```
for i in range(100, 200):
   f.plot(i, 100, 'blue')
```

immediatamente nella nostra finestra f compare una linea blu orizzontale a 100 pixel dall'alto della finestra, tracciata a partire dal centesimo pixel dalla sinistra e lunga 100 pixel.
Se scriviamo nella shell

```
for i in range(30):
   for j in range(30):
       f.plot(i, j, 'green')
```

immediatamente nella nostra finestra f compare un quadratino verde di 30 pixel in alto a sinistra.

getMouse()
crea l'attesa che si clicchi in un punto della finestra e registra le coordinate del punto su cui si è cliccato.
Se scriviamo nella shell `p = f.getMouse()` e, dato INVIO, clicchiamo su un punto della finestra, nell'oggetto grafico p vengono memorizzate le coordinate di quel punto.

close()
chiude la finestra.
L'istruzione `f.close()` nella shell provoca la scomparsa della nostra finestra f.

Oggetti grafici

Il modulo Graphics contiene le classi per costruire gli oggetti grafici punto (Point), linea (Line), cerchio (Circle), ellisse (Oval), rettangolo (Rectangle), poligono (Polygon) e testo (Text).

I principali metodi comuni a tutti questi oggetti grafici sono i seguenti.

draw(<identificativo_finestra>)
disegna l'oggetto nella finestra indicata.

undraw()
toglie l'oggetto precedentemente disegnato dalla finestra (l'identificativo della finestra non va indicato).

setWidth(<pixels>)
stabilisce la pesantezza, in pixel, della linea che delimita l'oggetto. Non ha effetto sull'oggetto Point.

setOutline(<colore>)
stabilisce il colore della linea che delimita l'oggetto.

setFill(<colore>)
stabilisce il colore interno di un oggetto.

Vediamo ora i principali metodi specifici di ciascun oggetto.

Point

`Point(x, y)`
è il metodo costruttore.
Scrivendo `p1 = Point(50, 60)` costruiamo l'oggetto punto, denominato p1, di coordinate 50 e 60.
Per disegnarlo nella finestra f ne richiamiamo il metodo `draw()` con l'istruzione `p1.draw(f)`.
Il colore di default è il nero. Se lo vogliamo rosso dobbiamo scrivere `p1.setOutline('red')`.
Ricordiamo questi meccanismi perché essi sono gli stessi con cui andremo a costruire, a disegnare ed a modificare tutti gli oggetti grafici che vedremo.

`getX()` e `getY()`
ritornano, rispettivamente, la coordinata x e la coordinata y del punto.
Con riferimento al punto p1 che abbiamo costruito prima, scrivendo `a = p1.getX()` inseriamo nella variabile `a` il valore della coordinata x (50) e scrivendo `b = p1.getY()` inseriamo nella variabile `b` il valore della coordinata y (60).
Ovviamente questi due metodi si applicano anche a oggetti punto costruiti senza ricorrere al costruttore che abbiamo visto sopra.
Per esempio, quando prima abbiamo visto il metodo getMouse() dell'oggetto finestra, con quel metodo abbiamo costruito un oggetto punto p cliccando nella finestra f. Se avessimo voluto disegnarlo in verde nella finestra stessa con il metodo plot() avremmo potuto scrivere `f.plot(p.getX(), p.getY(), 'green')`.

Line

Line(<primo_punto>, <secondo_punto>)
è il metodo costruttore.
I due parametri da indicare sono il punto di partenza e il punto di arrivo della linea.
Essi possono essere costruiti nel costruttore stesso con il metodo Point(x, y) che abbiamo appena visto.
Ad esempio con l'istruzione `l1 = Line(Point(50, 50), Point(100, 50))` costruiamo una linea, che abbiamo chiamato l1, che va dal punto di coordinate 50, 50 al punto di coordinate 100,50. Ricordo che, per vederla, dobbiamo disegnarla con l'istruzione l1.draw(<identificativo_finestra>).
Se avessimo già a disposizione gli oggetti punto, per esempio chiamati `p1` e `p2`, costruiti prima con il metodo Point(x, y) o con il metodo getMouse() della finestra, nel costruttore della linea potremmo richiamare quelli con l'istruzione `l1 = Line(p1, p2)`.
Ovviamente la nostra linea l1 che abbiamo costruito ha a disposizione i metodi comuni agli oggetti grafici che ho elencato prima.
Sicché può essere colorata di rosso con l'istruzione `l1.setOutline('red')`, resa più pesante con l'istruzione `l1.setWidth(3)`.

Circle

Circle(<punto_centrale>, <raggio>)
è il metodo costruttore.
I due parametri richiesti per costruire il cerchio sono il centro, per l'indicazione del quale vale quanto ho detto nel paragrafo precedente per l'indicazione del primo o del secondo punto della linea, e il raggio, che va indicato come numero, intero o decimale (parte decimale separata dal punto).
Con l'istruzione `c = Circle (Point(100, 90), 22.75)` costruiamo il cerchio `c` con centro nel punto di coordinate 100 e 90 e raggio di 22,75 pixel.
Essendo una figura chiusa, questo oggetto ha a disposizione il metodo comune per colorarne l'interno. Se vogliamo che il nostro cerchio c sia tutto verde diamo l'istruzione `c.setFill('green')`.
Attenzione: con questa istruzione coloriamo solo la parte interna del cerchio. Se vogliamo

verde anche la linea che disegna il cerchio (circonferenza) dobbiamo anche dare l'istruzione `c.setOutline('green')`.

Rectangle

`Rectangle(`<primo_punto>, <secondo_punto>`)`
è il costruttore.
I parametri indicano le coordinate del punto vertice in alto a sinistra di un rettangolo e le coordinate del punto vertice in basso a destra.
Inutile, da qui in poi, ripetere come si indicano i punti, con quali metodi si possano colorare i lati del rettangolo e/o l'area interna: dopo aver letto i precedenti paragrafi siamo ormai degli esperti.

Oval

`Oval(`<primo_punto>, <secondo_punto>`)`
è il costruttore.
L'oggetto grafico che costruiamo è l'ellisse inscritta nel rettangolo con vertice in alto a sinistra indicato dal primo punto e vertice in basso a destra indicato dal secondo punto.

Polygon

`Polygon(`<punto_1>, <punto_2>, <punto_3>, `.....)`
è il costruttore del poligono con i vertici nei punti indicati.

Text

Questo oggetto grafico è una scritta collocata in un certo punto della finestra. La collocazione avviene indicando un punto di ancoraggio che verrà assunto come punto centrale della scritta indicata, come stringa tra apici, nel costruttore
`Text(`<punto_ancoraggio>, <stringa_testo>`)`.
Le istruzioni `t = Text(Point(50,20), 'CIAO')` e `t.draw(f)` collocano la scritta CIAO centrata sul punto di coordinate 50 e 20 di una finestra f.

`setText(`<stringa_testo>`)`
modifica una scritta precedentemente costruita.
L'istruzione `t.setText('ARRIVEDERCI')` sostituisce la parola ARRIVEDERCI alla parola CIAO dell'oggetto t che abbiamo costruito prima.

`setTextColor(`<colore>`)`
indica il colore con cui scrivere il testo, per default nero.
L'istruzione `t.setTextColor('pink')` rende immediatamente rosa la nostra scritta t.

`setFace(`<famiglia>`)`
indica la famiglia del font, per default helvetica.
I valori possibili sono 'helvetica', 'courier', 'times roman' e 'arial'.

`setSize(`<punti_grafici>`)`
indica la dimensione del carattere in punti grafici, per default 12.
I valori accettati vanno da 5 a 36.

`setStyle(`<stile>`)`
indica lo stile del font, per default normal.
I valori accettati sono 'normal', 'bold', 'italic' e 'bold italic'.

* * *

I metodi che ho indicato per i vari oggetti sono quelli che ritengo i più utili. Per una rassegna completa dei metodi rimando alla guida in lingua inglese Graphics Reference che si trova sul sito all'indirizzo *http://mcsp.wartburg.edu/zelle/python/*.

Animazione

Ogniqualvolta aggiungiamo o modifichiamo un oggetto grafico, la finestra si aggiorna automaticamente in tempo reale.

Se desideriamo che questi aggiornamenti avvengano con una gradualità tale da creare un effetto di animazione possiamo ricorrere alla funzione `update()`.

Essa funziona nell'ambito di un ciclo che produce una serie di finestre modificate cadenzandone la visualizzazione aggiornata.
Sicché se scriviamo

```
for i in range(100):
   <istruzioni per la produzione o la modifica di oggetti grafici>
   update(25)
```

otteniamo che le variazioni grafiche derivanti dalle istruzioni siano 100 e siano visualizzate 25 per secondo.
Come dire che avremo un'animazione della durata di 4 secondi.
Ovviamente più è elevato il parametro dato alla funzione update più veloce sarà l'animazione (20 o 30 sono valori buoni per un'animazione che non sfarfalli e che abbia un certo effetto di gradualità).

Esempi

Dopo aver creato una finestra, che chiamiamo f, con le istruzioni

```
from graphics import *
f = GraphWin('PROVE', 300,300)
```

scriviamo quest'altra

```
for i in range (150):
   c.undraw()
   c = Circle(Point(30+i, 30), 20)
   c.draw(f)
   c.setFill(color_rgb(200-i, i, 100+i))
   update(20)
```

Dato doppio Invio, trattandosi di istruzione complessa, vediamo rotolare una pallina nella nostra finestra che, rotolando, modifica il proprio colore da un iniziale granata intenso ad un finale azzurro.

Se invece scriviamo quest'altra

```
for i in range(100):
   c.undraw()
   c = Circle(Point(150, 150), i)
   c.draw(f)
   c.setFill(color_rgb(250-i, 100+i, i))
   update(30)
```

vediamo crescere al centro della nostra finestra un cerchio che dal nulla arriva ad occupare quasi tutta la finestra e che, crescendo, modifica il proprio colore da arancione a verde chiaro.

4 Pygraph

Nel repository di Python troviamo un package chiamato pygraph che si occupa di grafi. Non è di questo che ci occupiamo qui e quello di cui ci occupiamo, ovviamente, data l'omonimia, non si trova nel repository PyPI e non è installabile con l'utility pip.

Il nostro Pygraph, al pari del modulo Graphics che abbiamo visto nel precedente Capitolo, è un prodotto dalle finalità didattiche e ce lo regala l'ottimo insegnante Daniele Zambelli.

Il manuale, questa volta scritto in buon italiano, è scaricabile all'indirizzo
https://media.readthedocs.org/pdf/pygraph/latest/pygraph.pdf
ed è stato aggiornato il 22 giugno 2018.
Esso contiene tutte le istruzioni per lavorare con pygraph ma anche passi la cui lettura è molto istruttiva per chi voglia apprendere o consolidare la tecnica di programmazione a oggetti con Python.

Il luogo da cui possiamo scaricare il software è

https://bitbucket.org/zambu/pygraph

Aperta la scheda del dowload (ultimo dei link elencati sulla destra della home page) scarichiamo il file **pygraph33_RC01.zip**, che contiene la versione 3.3 del 18 gennaio 2018, nella directory dei download.

Estratto l'archivio, ciò che maggiormente interessa sono il file `pygraph.pth` e la directory `pygraph` col relativo contenuto: questo è il vero e proprio pacchetto. Trasferiamo il file e la directory nel posto giusto, cioè:
. se usiamo Linux in `/usr/local/lib/pythonX.X/dist-packages`,
. se usiamo Windows in `C:\PythonXX\Lib\site-packages`,
. se usiamo Mac in `/Library/Frameworks/Python.framework/Versions/X.X/lib/pythonX.X/site-packages/`,
dove `X.X` sta per i numeri che identificano la versione Python installata sul computer (esempio 3.5 o 3.7, ecc.).

Tutto il resto è documentazione e lo possiamo mettere dove più ci fa comodo.

Anche questo pacchetto funziona solo se abbiamo installato Tkinter, la madre di tutta la grafica di Python.

Visto che possiamo disporre del manuale in italiano scritto dallo stesso autore del software non starò certo a ripetere ciò che comodamente vi si può trovare scritto e mi limiterò a una breve descrizione del pacchetto e ad alcune esemplificazioni.

Il pacchetto contiene quattro librerie: pycart.py, pyturtle.py, pyplot.py e pyig.py.

4.1 Pycart

Implementa un piano cartesiano e si basa essenzialmente su due classi: Plane, che ci dà modo di generare un oggetto piano cartesiano e Pen, che ci dà modo di generare un oggetto penna per disegnarci sopra.

Un modo spiccio per lavorare con questa libreria è di importarla con l'istruzione nel prompt della shell
`from pycart import *`
e di generare il piano con l'istruzione costruttrice
<identificativo_del_piano> `= Plane()`
Se usiamo la shell della IDLE, appena scritta la parola Plane e aperta la prima parentesi tonda, si apre una finestrella in cui vediamo tutti i parametri richiesti con indicati i relativi valori di default. Se non indichiamo parametri nell'istruzione costruttrice avremo un piano caratterizzato da quei valori. Eventuali valori diversi vanno indicati.
Ad esempio, se non ci va bene il titolo di default, che vogliamo indicare come PROVA e vogliamo gli assi ortogonali disegnati in rosso, la costruzione dovrà avvenire con l'istruzione
`mio_piano = Plane(name='PROVA', axescolor ='red').`
La penna per disegnare si costruisce con l'istruzione
<identificativo_della_penna> `= Pen()`
Anche in questo caso dopo aperta la prima parentesi tonda compaiono i parametri richiesti, con indicati quelli di default e vale il discorso di prima se vogliamo cambiarli.
Teniamo comunque presente che possiamo cambiare successivamente alla sua costruzione le due principali caratteristiche della penna, cioè il colore e la dimensione del tratto.

Sicché, se abbiamo costruito la penna generica di default con l'istruzione
`mia_penna = Pen()`
che ci ha fornito una penna che scrive in nero con il tratto di un punto grafico e desideriamo che scriva con un tratto più pesante, per esempio di tre punti grafici, possiamo utilizzare l'istruzione
`mia_penna.width = 3`
e se vogliamo che la penna scriva in rosso possiamo utilizzare l'istruzione
`mia_penna.color = 'red'`.

Nessuno ci proibisce di costruire più penne da tenere a disposizione, tipo una biro che scriva in blu sottile, con l'istruzione
`biro = Pen(color = 'blue')`
un pennarello che scriva in verde con l'istruzione
`pennarello = Pen(color = 'green', width = 3)`
e un pennello che pitturi in rosso con l'istruzione
`pennello = Pen(width = 7, color = 'red')`.

Per questo e altro è stata inventata la programmazione a oggetti.

A proposito di programmazione a oggetti, se qualcuno consulta il manuale del prof. Zambelli che ho citato prima, noterà che l'importazione del modulo pycart avviene con l'istruzione
`import pygraph.pycart as cg`
e che l'istruzione costruttrice del piano è
`piano = cg.Plane()`
ecc., tutto con una scrittura un po' diversa da quella che ho utilizzato io, definendola spiccia. Con questa scrittura, più corretta, si fa costante riferimento alle classi che si utilizzano ed alla loro collocazione. Si dice, infatti, «importa il modulo pycart che fa parte del modulo pygraph e ciò che importi chiamalo cg, poi costruisci un oggetto, chiamato piano, utilizzando la classe Plane che trovi in cg, ecc.». Oltre che stilisticamente più corretta, questa scrittura ci mette al coperto da possibili incidenti dovuti ad omonimie di classi e di metodi che ci possono essere nel pacchetto complessivo. E', tuttavia, una scrittura più appesantita.

Ora che abbiamo un piano e una o più penne vediamo come e che cosa possiamo disegnare.

Se scriviamo nella shell della IDLE il nome identificativo di una delle nostre penne e lo facciamo seguire da un punto, compare un menu a discesa che elenca tutti i metodi dell'oggetto indicato. La seguente figura 3 mostra ciò che appare se, creato, per esempio, l'oggetto pennarello come visto prima, scriviamo
`pennarello.`

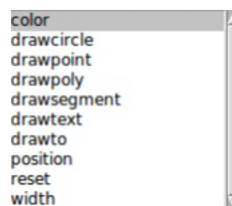

Figura 3: Metodi dell'oggetto costruito con Pen()

Vediamo che, oltre a metodi relativi allo status dell'oggetto, come quelli visti prima di color e width, ne abbiamo cinque che disegnano un cerchio (drawcircle), un punto (drawpoint), un poligono (drawpoly), un segmento (drawsegment) e una scritta (drawtext).

La scelta del metodo che vogliamo utilizzare può comodamente avvenire selezionandone il nome scorrendo con le freccette della tastiera e, una volta posizionati sul nome del metodo scelto, aprendo la parentesi tonda: immediatamente compare la lista dei parametri che serve inserire affinché il metodo produca ciò che vogliamo.

Propongo un esempio riepilogativo, utile anche per comprendere come indicare i parametri nei vari casi.

Se inseriamo le seguenti istruzioni nella shell otteniamo via via le figure che vediamo concentrate nella figura 4, che rappresenta il risultato finale.

```
from pycart import *
p = Plane(name = 'PROVE GRAFICHE', color = 'cyan')
biro = Pen()
pennarello = Pen(width = 3, color = 'red')
pennello = Pen(width = 6, color = 'yellow')
pennello.drawpoint(position = (-6, 6))
pennarello.drawcircle(radius = 3, center = (5,5), incolor = 'yellow')
pennarello.drawcircle(radius = 3, center = (-5,-5), color = 'blue')
pennello.drawsegment(point0 = (-5,-5), point1 = (5,5), color = 'red')
biro.drawpoly(vertices = ((3,-3), (6,-3), (3,-6.5), (6,-6.5)))
pennarello.drawtext(text = 'CIAO', position = (8,-5))
biro.drawpoly(vertices = ((11,-3), (14,-3), (14,-6.5), (11,-6.5)))
```

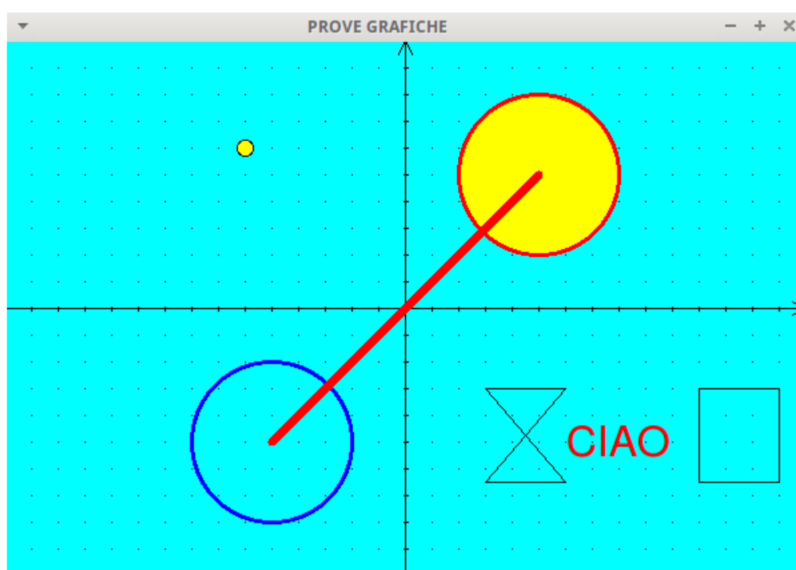

Figura 4: Risultato delle prove grafiche

Due piccole annotazioni:
. i lati di un poligono vengono disegnati secondo l'ordine dei vertici indicati e, per creare poligoni come il rettangolo, il quadrato, ecc. occorre fare attenzione ad indicare i vertici in senso orario;
. se si indica il colore come parametro della figura da creare la penna produce quel colore anche se era inizializzata per produrne un altro.

Se vogliamo salvare il nostro lavoro, lo possiamo fare con la seguente istruzione
<identificatore_del_piano>`.save('`<path e nome file>`')`.
Per esempio, scrivendo

```
p.save('/home/vittorio/Immagini/prove_grafiche')
```

salvo il lavoro appena fatto nella cartella Immagini nei dati personali del mio sistema Linux.
Dal momento che sul mio sistema ho caricato ImageMagick, il salvataggio avviene automaticamente in formato grafico .png. In caso contrario avverrebbe in formato postscript e occorrerebbe convertirlo altrimenti.
Purtroppo si trascina anche qui il difetto del metodo di salvataggio del modulo Tkinter, che non salva il colore di sfondo del piano, lasciandolo bianco.

4.2 Pyturtle

Implementa la grafica della tartaruga con un modulo che è lo stesso contenuto in Tkinter con qualche modifica.

Si basa su due classi, TurtlePlane e Turtle ed è impostato come Pycart che abbiamo appena visto, con TurtlePlane al posto di Plane e Turtle al posto di Pen: il piano della tartaruga si presenta pulito senza assi e la penna, per default visibile, si presenta come un triangolino.

Con il mio solito modo spiccio il modulo si importa con l'istruzione
`from pyturtle import *`
Il piano si genera con l'istruzione costruttrice
<identificativo_del_piano> `= TurtlePlane()`
e la tartaruga con l'istruzione costruttrice
<identificativo_della_tartaruga> `= Turtle().`

I comandi, con qualche piccola differenza, sono gli stessi che si usano per la tartaruga di Tkinter e si possono trovare nel mio già citato manualetto su Tkinter. Al solito, se usiamo la shell di IDLE, l'elenco dei comandi disponibili compare digitando l'identificativo della tartaruga e un punto sulla riga del prompt della shell e si attiva quello desiderato scorrendo l'elenco con le freccine della tastiera e fermandosi su di esso.

Tra le piccole differenze segnalo la presenza di un metodo `ccircle()`, che disegna un cerchio centrato sulla posizione della tartaruga (il metodo `circle()`, comune a Tkinter e a Pyturtle, disegna il cerchio partendo dalla posizione della tartaruga).
Inoltre, tutti i metodi che in Tkinter possono essere richiamati con identificativi con o senza il prefisso pen, in Pyturtle non hanno mai questo prefisso (ad esempio, per alzare la penna/tartaruga, in Tkinter possiamo scrivere `penup()` oppure `up()` mentre in Pyturtle possiamo scrivere solo `up()`, così come per stabilire il colore del disegno della tartaruga in Tkinter possiamo scrivere `pencolor()` oppure `color()` mentre in Pyturtle possiamo scrivere solo `color()`, ecc.).

Anche il piano della tartaruga ha un metodo `save()` per salvare il nostro lavoro.

4.3 Pyplot

Questa libreria serve per tracciare grafici di funzioni matematiche.

Si basa su due classi: PlotPlane, che genera un oggetto piano cartesiano su cui tracciare il grafico e Plot, che genera un oggetto plotter per disegnare il grafico della funzione.

Possiamo importare il modulo con l'istruzione
`from pyplot import *`
Il piano si genera con l'istruzione costruttrice
<identificativo_del_piano> `= PlotPlane()`
dove tra le parentesi, se non ci va bene l'impostazione di default che compare appena battuta la prima parentesi se usiamo la shell di IDLE, possiamo indicare le varianti,
e il plotter con l'istruzione costruttrice
<identificativo_del_plotter> `= Plot()`
dove tra le parentesi possiamo indicare il colore (con `color =` '<colore>') e la pesantezza di tratto (con `width =` <punti_grafici>).

Il grafico della funzione si traccia con l'istruzione
<identificativo_del_plotter>`.`<tipo_plot>`()`
dove <tipo_plot> può essere `xy` per tracciare il grafico della funzione *y = f(x)*, `yx` per tracciare il grafico della funzione *x=f(y)*, `polar` per tracciare il grafico di una funzione polare nella forma: *ro=f(th)*.

I parametri da passare tra le parentesi tonde possono essere
. una funzione `fun` previamente definita con
```
def fun(x):
    return <espressione>
```
. una funzione definita con l'espressione
```
lambda x: <espressione>
```
Esempi:
Importiamo la libreria con
`from pyplot import *`

Creiamo il piano con
```
piano = PlotPlane('GRAFICI DI FUNZIONE')
```
Creiamo un primo plotter con l'impostazione di default (colore nero, tratto sottile da 1 punto grafico) con
```
p = Plot()
```
Se con questo plotter vogliamo tracciare il grafico della funzione $y = x^3$, innanzi tutto definiamo la funzione con
```
def cubo(x):
   return x**3
```
e poi diamo l'istruzione
```
p.xy(cubo)
```
Se ora vogliamo tracciare il grafico della funzione $y = \frac{1}{x}$ in rosso con un tratto un po' più marcato possiamo creare un nuovo plotter con queste caratteristiche con l'istruzione
```
p = Plot(color='red', width=3)
```
e possiamo dare l'istruzione per il tracciamento questa volta utilizzando la notazione lambda con
```
p.xy(lambda x:  1/x).
```
Quando abbiamo costruito il plotter rosso abbiamo usato lo stesso identificatore p e abbiamo sovrascritto il precedente plotter; se avessimo voluto conservare il vecchio plotter nero sottile di default avremmo dovuto dare un altro identificativo a quello rosso, per esempio p1 al posto di p.

4.4 Pyig

Questa libreria implementa un piano cartesiano di geometria interattiva.

L'interattività deriva dal fatto che nel piano inseriamo innanzi tutto i punti su cui costruiamo figure geometriche e, successivamente, spostando con il mouse questi punti possiamo modificare le figure stesse. Dal momento che di queste figure possiamo calcolare perimetro e superficie, questi dati si aggiornano in tempo reale con la modifica.

Purtroppo dalla shell possiamo solo costruire il piano ma non gestirne l'interattività in quanto, per rendere interattivo il piano, dobbiamo lanciare dalla shell il comando `mainloop()` di Tkinter e, da questo momento, non possiamo più lavorare con la shell ma possiamo solo manovrare con il mouse sul piano diventato interattivo.

Il modulo Pyig è il più ricco di funzioni e metodi e non starò certo a ripetere qui, copiando, ciò che ha già scritto in ottima lingua italiana l'autore del software nel manuale che ho già citato.

Presenterò solo un piccolo esempio di media complessità che ritengo istruttivo.

Utilizzando la mia impostazione spiccia importiamo il modulo con l'istruzione
```
from pyig import *
```
e quindi costruiamo un piano, che chiamiamo piano, con l'istruzione
```
piano = InteractivePlane()
```
al solito, all'apertura della prima parentesi, se usiamo la shell di IDLE, compare l'elenco dei parametri con indicati i valori di default, che io ho accettato in toto non indicando modifiche tra le parentesi.
Il piano compare di fianco alla IDLE e si presenta con le assi cartesiane e una griglia.

Ora costruiamo sul piano due punti, `p1` e `p2`, con coordinate qualsiasi, per esempio
p1 = Point(1, 1)
p2 = Point(3, 3)
dopo ciascuna istruzione vediamo comparire sul piano questi punti, nelle posizioni volute.

Assumiamo che il punto p1 sia il centro di un cerchio e il segmento che unisce p1 a p2 sia il raggio di questo cerchio.
Costruiamo allora un cerchio `c` con l'istruzione
```
c = Circle(p1, p2)
```
e immediatamente lo vedremo tracciato sul nostro piano.

Dal momento che sarebbe interessante tener nota del valore del raggio di questo cerchio, anche senza vederlo disegnato, possiamo utilmente dare l'istruzione

```
r = Segment(p1, p2, visible = False)
```

con la quale costruiamo un oggetto segmento `r` senza farlo vedere.

A questo punto facciamo un po' di scena e mostriamo sul piano alcuni valori di ciò che abbiamo disegnato. Dal momento che vogliamo essere interattivi dobbiamo scrivere questi valori in maniera tale che si possano aggiornare automaticamente: viene buono, a questo scopo, il metodo `VarText`.

Cominciamo a scrivere il valore del raggio del cerchio con l'istruzione

```
VarText(-10, -12, 'raggio:  {0}', r.length(), color = 'red')
```

e vedremo comparire, in rosso, in basso a sinistra del nostro piano il valore del raggio del nostro cerchio.

Il simbolo `{0}` è il segnaposto, nella stringa `raggio:` per il valore che segue dopo la virgola.

Con analoga istruzione scriviamo, una riga sotto, il valore della circonferenza

```
VarText(-10, -13, 'circonferenza:  {0}', c.perimeter(), color = 'red')
```

e, infine, il valore dell'area del cerchio

```
VarText(-10, -14, 'area:  {0}', c.surface(), color = 'red').
```

Il nostro piano compare come nella seguente figura 5.

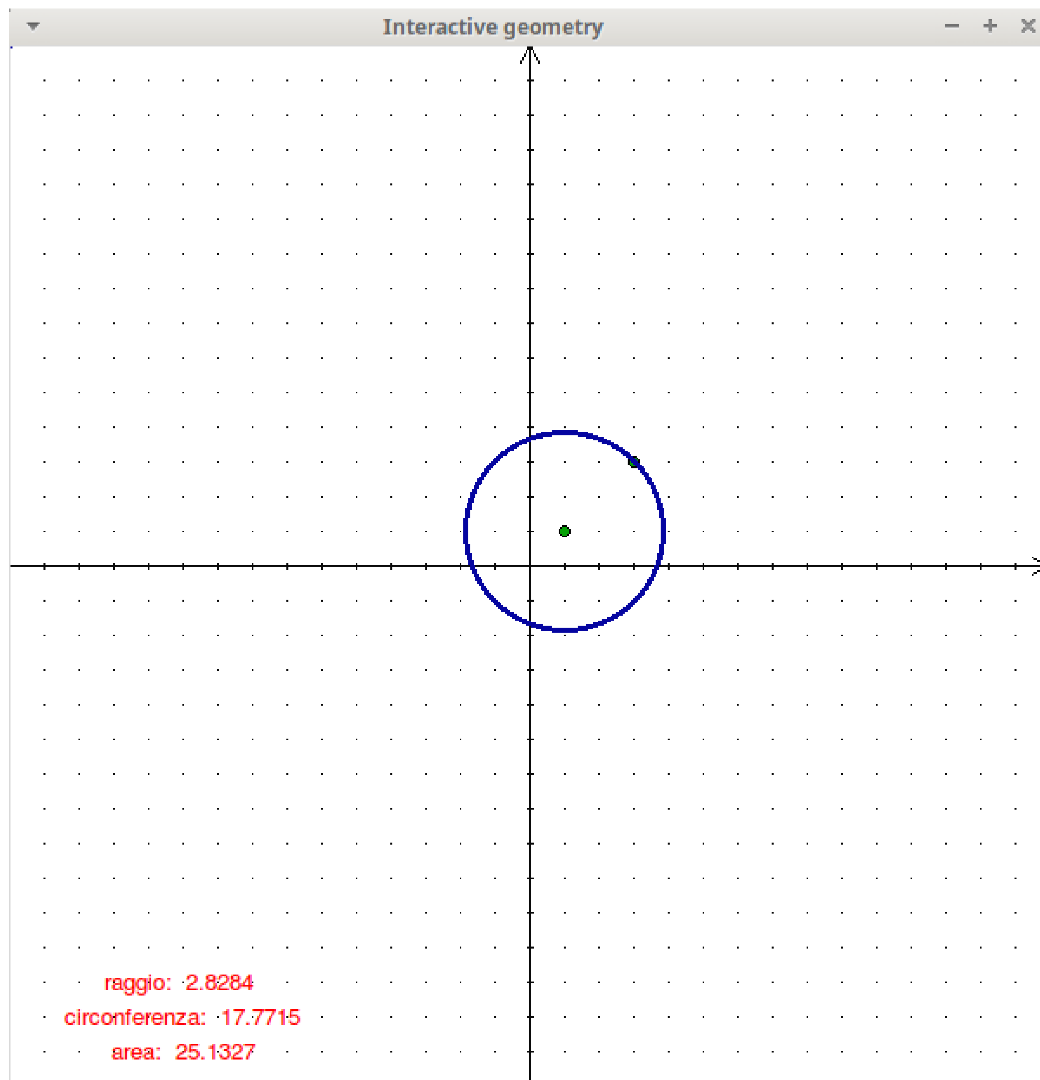

Figura 5: Piano di geometria interattiva

Con l'istruzione

```
piano.mainloop()
```

rendiamo interattivo il piano e, da questo momento la shell passa il controllo al piano.

Con il mouse possiamo spostare uno o entrambi i punti e, di conseguenza, il cerchio si ridisegna e si modificano i relativi valori.

La seguente figura 6 mostra il risultato di uno spostamento di entrambi i punti.

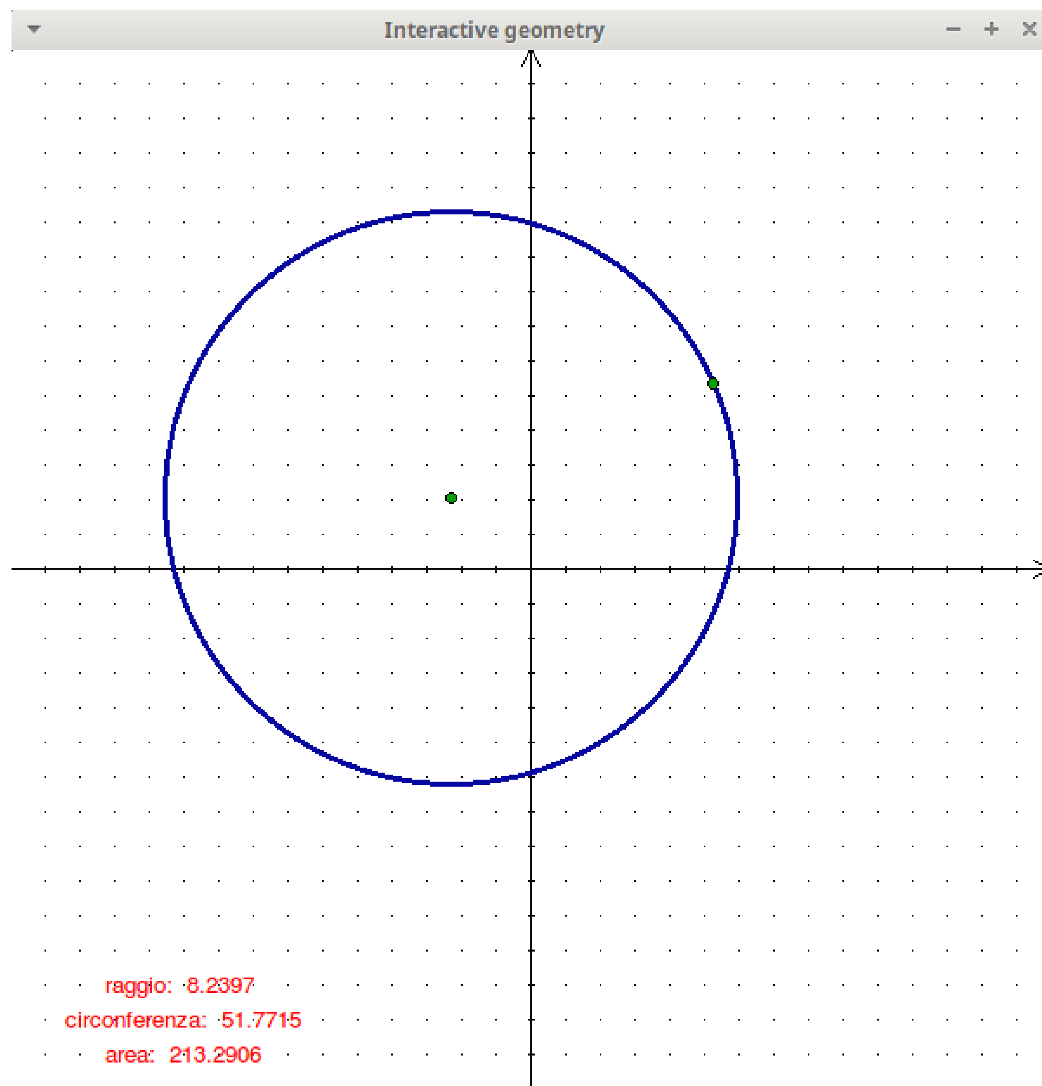

Figura 6: Variazione introdotta nel piano di geometria interattiva con lo spostamento dei due punti.

* * *

Mi rendo conto che tutto quanto abbiamo visto probabilmente non è di grandissima utilità sul piano pratico ma ritengo sia molto istruttivo sia per riscontrare la grande utilità della shell di Python, specialmente se usiamo quella della IDLE, sia per prendere dimestichezza con la logica della programmazione a oggetti.