Manuale di Pygraph

Release 2.9.01

Daniele Zambelli

06 gen 2018

Indice

Contents:

CAPITOLO 1

Introduzione

Dove si cerca di giustificare il tempo perso per questo lavoro e dove viene data qualche idea su come installare l'interprete Python e la libreria grafica.

1.1 Scopo di Pygraph e di questo manuale

Mi sono imbattuto in Python mentre cercavo un linguaggio di alto livello, multi-piattaforma, con una sintassi semplice. Oltre ad uso personale volevo utilizzarlo anche a scuola per insegnare qualche elemento di informatica ai miei alunni. Ritenendo che la programmazione di elementi grafici possa coinvolgere di più dei semplici comandi di testo, ho cercato le librerie grafiche messe a disposizione da questo linguaggio. Con sorpresa e grande gioia ho scoperto che una delle librerie ufficiali implementa la grafica della tartaruga. Dopo averla utilizzata per un po" ho sentito la necessità di modificarla e darle una connotazione più Object-Orienteed. Ho incominciato a pasticciarla, modificarla, adattarla alle mie esigenze (grazie software libero!). L'ho suddivisa in due parti: l'implementazione di un piano cartesiano e della grafica della tartaruga. Nel frattempo, sempre per esigenze didattiche ho aggiunto il modulo per tracciare funzioni matematiche integrandolo con il piano cartesiano. Altro strumento didattico interessante è costituito dai programmi di geometria interattiva (Cabri Géomètre, Kig, Dr. Geo, GeoGebra, CaR, ...). Un forte limite di questi programmi è l'assenza, o il difficile uso, di un linguaggio al loro interno, ho pensato quindi di aggiungere a pygraph una libreria per realizzare figure geometriche interattive. Il risultato finale è abbastanza diverso dalla libreria turtle.py da cui sono partito.

Dopo i primi anni di uso in diverse situazioni - uso personale, con alunni di scuola media e di scuola superiore - queste librerie mi sono sembrate abbastanza robuste da poter essere proposte ad altre persone interessate a Python e alla grafica. Così ho riunito un po" di esempi di uso e ho scritto questo manualetto per documentarle. Ora, dopo altre esperienze, ho rivisto un po" tutto il materiale per rilasciare una nuova versione.

I primi capitoli di questo testo sono pensati per guidare passo passo alla scoperta della programmazione in Python. Non vogliono essere una guida completa, per questo ci sono altri lavori ben più importanti del mio, ma vorrebbero essere uno stimolo e un'indicazione per iniziare l'esplorazione del mondo dei linguaggi di programmazione.

1.2 Installazione

Per installare un qualunque software bisogna essere un po" pratici di computer. Conviene eventualmente farsi aiutare. L'autore non può essere ritenuto responsabile della perdita di informazioni conseguenti ad errori di installazione.

1.2.1 Installare Python

1. Linux: il programma deve essere installato dall'amministratore del sistema, lui sa come fare. Se si utilizza una distribuzione la cosa più semplice è caricare Python al momento dell'installazione del sistema. O comunque installarlo a partire dai *repository* della distribuzione stessa. Sotto Linux a volte Python è distribuito separatamente da IDLE. In questo caso, per utilizzare l'ambiente di sviluppo IDLE (vedi capitolo 1), bisogna installarlo separatamente.
2. Windows: doppio clic sul pacchetto da installare.
3. La versione più aggiornata di Python può essere scaricata da http://www.python.org.
4. Python è un software libero ed è distribuito sotto una licenza compatibile con la licenza GPL quindi è possibile usarlo, copiarlo e distribuirlo senza restrizioni.

1.2.2 Installare pygraph

1. Procurarsi il file «pygraph_x.xx» (dove al posto dei vari «x» ci saranno delle cifre). Lo si può scaricare a partire dal sito http://bitbucket.org/zambu/pygraph/downloads
2. Scompattare il file all'interno di una directory di lavoro.
3. La directory pygraph contiene le quattro directory seguenti:
   - **doc**: documentazione varia,
   - **examples**: esempi d'uso,
   - **test**: test delle varie funzioni della libreria,
   - **pygraph**: le librerie del progetto:
     - **pycart.py**: un piano cartesiano
     - **pyturtle.py**: la grafica della tartaruga
     - **pyplot.py**: grafico di funzioni nel piano, cartesiane e polari
     - **pyig**: geometria interattiva
4. Spostare le directory doc, examples e test in una propria cartella facilmente raggiungibile (ad esempio: .../mieidocumenti/python/pygraph)
5. Spostare la directory pygraph e il file pygraph.pth all'interno di: .../pythonx.x/Lib/site-packages/ o .../pythonx.x/dist-packages/

Nota:

- per fare questo, probabilmente, bisogna avere i privilegi di amministratore;
- a seconda della versione di Python installata, pythonx.x potrebbe essere python3.5 o python3.6...;
- nella mia versione, Python3.5 sotto Debian, la directory in cui spostare la cartella `pygraph` e il file `pygraph.pth` si chiama: **dist-packages**.

1.2.3 Trovare documentazione su Python

1. http://www.python.org
2. http://www.python.it
3. http://www.python-it.org
4. Un'ottima introduzione all'informatica usando questo linguaggio di programmazione è il testo: «Pensare da informatico: Imparare con Python» di Downey, Allen tradotto magnificamente in italiano (si trova su internet partendo dai link precedenti).
5. Le dispense di un laboratorio di matematica con Python e pygraph si possono trovare a partire dal sito http://www.fugamatematica.blogspot.com.

1.2.4 Riassumendo

Per eseguire i programmi scritti in Python bisogna installare Python (ma dai!). Per utilizzare la libreria grafica pyturtle (e le altre) bisogna aver copiato la cartella pygraph e anche il file pygraph.pth in una directory visibile da Python (site-packages o dist-packages).

1.3 Licenza

1.4 Feedback

Spero che qualcuno trovi interessante questo lavoro e lo usi per imparare o per insegnare l'informatica e la geometria con Python. Utilizzandolo, senz'altro verranno alla luce molti errori, difetti o carenze sia nel software sia nella documentazione; invito chi troverà qualche motivo di interesse in questo lavoro a inviarmi:

1. Commenti,
2. Critiche,
3. Suggerimenti.

Ringrazio chiunque si prenderà la briga di mandarmi qualche riscontro.

Daniele Zambelli email: daniele.zambelli at gmail.com

CAPITOLO 2

I primi comandi

Dove vengono presentati: il comando print, gli oggetti numero e stringa e le variabili. E dove si vede anche come si fa a ripetere più volte un comando.

2.1 Comandi ed errori

Quando avviamo `IDLE` si apre una *shell*, una finestra dove si possono eseguire dei comandi. La `shell di IDLE`, è caratterizzata dalla presenza di 3 simboli di maggiore all'inizio della riga:

```
>>>
```

Questi simboli sono il *prompt* di `Python` e stanno per la domanda:

Cosa devo fare?

Se scriviamo qualcosa e premiamo il tasto `<invio>` `Python` interpreta quello che abbiamo scritto come un comando e cerca di eseguirlo. Non sempre ci riesce! In questo caso ci avvisa con un messaggio (chissà perché, scritto in inglese). Ad esempio:

```
>>> salta
Traceback (most recent call last):
File "<pyshell#47>", line 1, in ?
    salta
NameError: name 'salta' is not defined
```

In pratica ci comunica che non sa cosa voglia dire «salta». Nel messaggio di errore, le prime righe indicano dove è avvenuta l'incomprensione, possono essere utili quando si lavora ad un programma lungo, la riga significativa per noi è l'ultima.

Nota: a seconda della versione di `Python` utilizzata ci sono delle differenze; se sono trascurabili le trascurerò, altrimenti riporterò sia il comportamento di `Python 2.x` sia quello di `Python 3.x` (anche se mi auguro che stiate usando `Python 3.x`).

Si può provare qualche altro esempio:

```
>>> dammi 10 euri
SyntaxError: invalid syntax
```

Questa riga non riesce proprio ad interpretarla. Peccato! Scrivendo programmi ci si imbatte spesso in messaggi di questo genere. Anche se all'inizio è un po" pesante, bisogna abituarsi a leggerli per capire come mai l'interprete non fa quello che noi volevamo. Con il tempo si impara a individuare rapidamente all'interno dei messaggi le informazioni interessanti. Per riuscire ad ottenere qualcosa di significativo dovremmo operare qualche cambiamento al nostro modo di dare i comandi. Per prima cosa teniamo conto che, come i messaggi di errore, anche i comandi di `Python` sono tutti in inglese, poi bisogna conoscere quali sono i comandi che `Python` conosce.

Vediamone alcuni.

2.2 print

Un primo comando che è utile conoscere è: `print <qualcosa>`. Esempio:

```
>>> print(45)  #versione 3.x
45

>>> print 45  #versione 2.x
45
```

`<qualcosa>` può essere un numero. Se al posto di un numero però scrivo un nome, ottengo un errore:

Nota: in `Python 3.x`, `print` è una funzione che vuole come argomento, tra parentesi, l'oggetto da visualizzare; in `Python 2.x` `print` è un comando che deve essere seguito dall'oggetto che vogliamo venga visualizzato sullo schermo.

```
#versione 3.x
>>> print(Mario)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print(Mario)
NameError: name 'Mario' is not defined

#versione 2.x
>>> print Mario
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in ?
    print Mario
NameError: name 'Mario' is not defined
```

In pratica l'interprete ha cercato il nome «Mario» nel suo vocabolario e non l'ha trovato. Per stampare un nome o una frase qualsiasi devo utilizzare le virgolette:

```
>>> print("Mario")  #versione 3.x
Mario

>>> print "Mario"  #versione 2.x
Mario
```

Così il comando funziona! Una sequenza di caratteri racchiusa tra virgolette (semplici o doppie) si chiama «stringa».

Nota: IDLE ci mette a disposizione un semplice meccanismo per riscrivere, ed eventualmente modificare, dei comandi dati in precedenza: basta portare il cursore sul comando e premere il tasto <Invio>. Quindi, se volessimo modificare il comando precedente, possiamo portare il cursore (con il mouse o con i tasti freccia) sul comando da modificare, premere <Invio> e fare i cambiamenti che vogliamo, poi farlo eseguire premendo di nuovo <Invio>:

```
>>> print("Mario ha 45 anni")     #versione 3.x
Mario ha 45 anni

>>> print "Mario ha 45 anni"     #versione 2.x
Mario ha 45 anni
```

Il comando `print` è il più semplice modo che un programma ha per comunicarci qualcosa.

2.3 Variabili

Un elemento fondamentale della programmazione è costituito dalle variabili. Una variabile è una *parola* (in termini tecnici: *identificatore*) a cui è collegato un *oggetto*. Questo «oggetto» può, ad esempio, essere un numero o una stringa. Per associare un *oggetto* ad un *identificatore* si utilizza l'operatore di assegnazione: «=». Per tirar fuori qualcosa basta scrivere il nome della variabile. Esempio:

```
>>> nome = "Mario"
```

Ora all'identificatore «nome» è associata la stringa «Mario».:

```
>>> print(nome)  #versione 3.x
Mario

>>> print nome  #versione 2.x
Mario
```

Il comando `print` può accettare più oggetti separati da virgole, li stamperà uno di seguito all'altro:

```
>>> eta = 45
```

Ora all'identificatore «eta» è associata la stringa il numero 45.:

```
>>> print(nome, "ha", eta, "anni.")  #versione 3.x
Mario ha 45 anni.

>>> print nome, "ha", eta, "anni."  #versione 2.x
Mario ha 45 anni.
```

La funzione `print`, quando viene utilizzata con più oggetti, mette automaticamente uno spazio tra un oggetto e l'altro. Non sempre questo comportamento ci va bene. A volte desideriamo stampare due stringhe una di seguito all'altra senza spazi in mezzo. Ad esempio:

```
>>> primo_pezzo="mario"
>>> secondo_pezzo="netta"

>>> print(primo_pezzo, secondo_pezzo)  #versione 3.x
mario netta

>>> print primo_pezzo, secondo_pezzo  #versione 2.x
mario netta
```

Se voglio che le stringhe vengano stampate senza spazi in mezzo, dovrò concatenarle. L'operatore di concatenazione di stringhe è il simbolo: «+». Esempio:

```
>>> print(primo_pezzo + secondo_pezzo)  #versione 3.x
marionetta

>>> print primo_pezzo + secondo_pezzo  #versione 2.x
marionetta
```

2.4 Iterazione

Spesso i programmi devono ripetere più volte le stesse istruzioni. Ad esempio se dovessi far stampare tre volte "Ciao" potrei dare il comando:

```
>>> print("Ciao"); print("Ciao"); print("Ciao")  #versione 3.x
Ciao
Ciao
Ciao

>>> print "Ciao"; print "Ciao"; print "Ciao"  #versione 2.x
Ciao
Ciao
Ciao
```

Notare che diverse istruzioni devono essere separate o da un `<Invio>` o da un `;`. Questo metodo però non risulta affatto comodo se il numero di ripetizioni di un comando è elevato. Tutti i linguaggi di programmazione mettono a disposizione diversi comandi per ripetere un gruppo di istruzioni. Un modo per far scrivere 3 volte «Ciao» a `Python` è:

```
>>> for cont in range(3):  #versione 3.x
        print("Ciao")

>>> for cont in range(3):  #versione 2.x
        print "Ciao"
```

..note:: Attenzione ai *due punti* che terminano la prima riga. Incontrando il simbolo `:`, l'interprete non esegue il comando ma va a capo spostando il cursore verso destra in modo da permettere di scrivere il blocco di istruzioni da ripetere.

Ora, premendo una prima volta il tasto `<Invio>` non succede niente, perché venga eseguito il comando, si deve premere una seconda volta `<Invio>`. Analizziamo il comando `for` che è piuttosto complesso:

```
for <variabile> in range(<numero>):
    <comandi da ripetere>
```

- `for` è il nome del comando;
- `cont` è il nome di una variabile, al posto di `cont` potrei scrivere qualunque altro nome. In questa variabile vengono messi, uno alla volta i valori restituiti dalla funzione range. In questo caso `cont` non viene usata, ma spesso è utile avere una variabile che contiene, ad ogni ciclo, un valore diverso;
- `in` fa parte del comando `for`;
- `range` fornisce l'elenco dei numeri interi tra 0 compreso e il numero tra parentesi escluso;
- `:` alla fine di questa riga c'è il carattere due punti, indica che di seguito è scritta la porzione di codice che verrà ripetuta;

- `print("Ciao")` (o `print "Ciao"` ``) è la parte di codice che viene ripetuta, deve essere *indentata* cioè deve essere scritta più a destra rispetto al comando ``for``. Può essere formata da più righe, in questo caso devono avere tutte lo stesso rientro (trovarsi allo stesso livello di indentazione).

Da questo esempio può non apparire in modo molto chiaro quanto sia comodo il comando `for`, ma se il numero di ripetizioni fosse molto più elevato, o il codice che deve essere ripetuto più complesso, si vedrebbe immediatamente la differenza. Consideriamo ad esempio il seguente codice:

```
>>> for cont in range(3000):
        print("Ciao")
```

nessuna persona normale si metterebbe a scrivere a mano i singoli comandi per ottenere lo stesso risultato delle due righe precedenti.

Per vedere il contenuto della variabile `cont` durante l'esecuzione del ciclo possiamo dare il comando:

```
>>> for cont in xrange(8):        #versione 3.x
        print("cont vale", cont)

>>> for cont in xrange(8):        #versione 2.x
        print "cont vale", cont

cont vale 0
cont vale 1
cont vale 2
cont vale 3
cont vale 4
cont vale 5
cont vale 6
cont vale 7
>>>
```

Se volessi un elenco dei primi otto interi e dei loro quadrati e dei loro cubi:

```
>>> for n in range(8):                    #versione 3.x
        print(n, "\t", n*n, "\t", n*n*n)

>>> for n in range(8):                    #versione 2.x
        print n, "\t", n*n, "\t", n*n*n

0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
>>>
```

Nota: alcune stringhe non vengono visualizzate come le abbiamo scritte, ma vengono interpretate ad esempio al posto di `\t` verrà visualizzato un salto di tabulazione, al posto di `\n` viene eseguito un *a capo*.

2.5 Riassunto

- Nell'ambiente IDLE si dà un comando a `Python` scrivendolo dopo il prompt e terminandolo con il tasto `<Invio>`
- Se `Python` non è in grado di eseguire un nostro comando ci avvisa con un messaggio.
- I più semplici oggetti che possiamo trovare in `Python` sono: i numeri e le stringhe.
- Le stringhe sono formate da una sequenza di caratteri racchiusa tra apici doppi o singoli.
- Una variabile, in `Python` è l'associazione tra un `identificatore`, un nome, e un `oggetto`.
- Si può assegnare un oggetto ad un `identificatore` utilizzando l'operatore di assegnazione: `=`.
- Si ottiene l'oggetto associato ad un identificatore scrivendo semplicemente l'identificatore.
- La funzione `print` scrive uno o più oggetti sullo schermo.
- Si possono concatenare delle stringhe utilizzando l'operatore di concatenazione: `+`.
- Nella shell di `IDLE` posso portare il cursore sulla riga

che contiene un comando già scritto, premendo una volta `<Invio>` il comando viene ricopiato in fondo, posso quindi modificarlo e rieseguirlo.

- Per ripetere più volte le stesse istruzioni posso utilizzare il comando:

```
for <identificatore> in range(<numero>):
    <istruzioni>
```

CAPITOLO 3

La grafica della tartaruga

Dove si parla di come generare tartarughe virtuali e mandarle in giro per lo schermo a disegnare.

3.1 Creare una *tartaruga*

Per giocare con la grafica della tartaruga bisogna procurarci... una tartaruga. Ci sono molti modi per farlo, verranno illustrati più avanti, qui ne vediamo uno, il più comune. I passaggi necessari, in sintesi, sono:

1. caricare la libreria che contiene gli oggetti della grafica della tartaruga,
2. creare una finestra dove far vivere delle tartarughe,
3. creare una tartaruga,
4. scrivere i comandi che la tartaruga deve eseguire,
5. rendere attiva la finestra creata.

Di seguito sono illustrati i precedenti passaggi.

Per utilizzare la libreria della grafica della Tartaruga dobbiamo dire a `Python` di caricarla in memoria. In realtà, di tutta la libreria ci interessa, per ora, solo l'ambiente dove vivono le tartarughe. Per caricare questo ambiente dobbiamo scrivere:

```
>>> import pygraph.pyturtle as tg
```

Nota: Io ho usato la parola `tg` perché mi ricorda: *turtle geometry*, ma è possibile usare un qualunque altro nome, basta essere coerenti nel resto del programma.

Questa istruzione legge la libreria `pyturtle.py` e la associa al nome `tg`. Se non sono apparsi messaggi di errore vuol dire che `Python` ha trovato e caricato la libreria.

Ora dobbiamo creare un piano in cui far vivere la tartaruga. Il comando che realizza questo è: `TurtlePlane()`, ma questa è una funzione che si trova nella libreria che abbiamo associato al nome `tg` quindi il comando completo sarà: `tg.TurtlePlane()` che significa: *esegui la funzione TurtlePlane() della libreria tg*. Ma manca ancora un

particolare: vogliamo associare il piano prodotto dalla funzione ad un nome in modo da potervi fare riferimento in seguito. Il comando da dare è dunque:

```
>>> piano = tg.TurtlePlane()
```

Nota: Bisogna prestare molta attenzione alle maiuscole e minuscole e alle parentesi. Mentre il nome `piano` può essere sostituito da un qualunque altro nome, basta essere coerenti nel seguito.

Ora che abbiamo una superficie adatta possiamo passare al punto 3: dobbiamo creare una tartaruga. Ci sono diversi modi per farlo, verranno illustrati nel capitolo relativo alla libreria `pyturtle`, qui propongo il più semplice. Come prima, dobbiamo intanto pensare un nome da dare alla tartaruga che stiamo per creare, ad esempio `tina`, e dare il comando:

```
>>> tina = tg.Turtle()
```

Riassumendo:

- `tp` è un piano dove vivono tartarughe;
- `tina` è una tartaruga che vive nel piano `tp`.

Nota: Questa versione della grafica della tartaruga è piuttosto spartana, la nostra tartaruga `tina` è quel triangolino che si vede al centro del piano delle tartarughe, non assomiglia affatto ad una tartaruga!

A questo punto si possono esplorare i principali comandi della geometria della tartaruga. La tartaruga `tina` è ora a nostra disposizione per gli esperimenti. Possiamo provare a darle qualche comando:

```
>>> tina.forward(100)
```

`tina` si sposta avanti di 100 passi lasciando una traccia.

```
>>> tina.right(90)
```

Non si *sposta* verso destra, ma *ruota* verso destra di 90 gradi. E per ruotare verso sinistra?

```
>>> tina.left(90)
```

Esatto! Ripetiamo ora uno spostamento e una rotazione per un po" di volte:

```
>>> for i in range(4):
        tina.forward(100)
        tina.right(90)
```

`tina` disegna un quadrato.

```
>>> tina.up()
>>> tina.back(200)
>>> tina.down()
```

Solleva la penna si sposta indietro di `200` passi e riappoggia la penna, in questo modo si può far muovere Tartaruga senza disegnare.

```
>>> tina.color = "blue"
```

Cambia il colore della penna.

```
>>> for i in range(180):
        tina.forward(80)
        tina.back(78)
        tina.left(2)
```

Altro disegno, questa volt blu... Spostiamo ancora `tina`:

```
>>> tina.up()
>>> tina.back(100)
>>> tina.down()
```

Poi cambiamo il colore della penna:

```
>>> tina.color = (1, 0, 0)
```

Si può assegnare il colore della penna in vari modi:

- scrivendo tra virgolette il nome di colore in inglese:
- fornendo tre numeri compresi tra 0 e 1 ragruppati in una parentesi tonda. In questo caso il primo numero indica la componente rossa, il secondo quella verde, il terzo quella blu
- (o in un terzo modo che è spiegato più avanti).

E ora proviamo a fare una stella con 100 raggi:

```
>>> for i in range(100):
        tina.forward(80)
        tina.back(80)
        tina.left(360/100)
```

I raggi sono proprio `100` (li ho contati!), ma la stella non appare affatto regolare, ne manca un pezzo!

Nota: Se stai lavorando con `Python 2` può darsi che il risultato non sia proprio quello che ti aspetti, perché in questo caso la divisione tra due numeri interi dà un risultato troncato all'intero:

```
>>> print 360/100
3
```

Se voglio un risultato più preciso almeno uno degli operandi deve essere un numero decimale:

```
>>> print 360.0/100
3.6
```

Oppure semplicemente:

```
>>> print 360./100
3.6
```

Dopo un po" che si fanno dei tentativi con i comandi grafici può succedere di avere la finestra grafica così piena di scarabocchi da non capirci più niente. Un altro comando fondamentale è `reset`. Questo è un metodo del piano della tartaruga, ripulisce la finestra grafica e rimette a posto Tartaruga:

```
>>> tp.reset()
```

Per quanto riguarda l'ultimo punto, rendere attivo il piano della tartaruga, verrà illustrato più avanti.

Gli oggetti della classe Turtle, oltre ad avere dei metodi (come `forward`, `right`, ...) che corrispondono a delle azioni hanno anche degli attributi che corrispondono a delle proprietà, delle caratteristiche. Ad esempio ogni tartaruga ha un colore e uno spessore della sua penna. Ad esempio:

```
>>> tp.reset()
>>> tina.color = "green"
>>> tina.width = 4
>>> for i in range(20):
        tina.forward(80)
        tina.back(80)
        tina.left(360.0/20)
```

3.2 Riassumendo

- Per lavorare con la grafica della tartaruga posso scrivere:

```
>>> import pygraph.pyturtle as tg
>>> tp = tg.TurtlePlane()
>>> tina = tg.Turtle()
```

  ottengo così due oggetti: un piano delle tartarughe (`tp`) e una tartaruga (`tina`).

- I comandi fondamentali per far muovere la tartaruga creata e disegnare qualcosa sono:

```
<tartaruga>.forward(<numero>)
<tartaruga>.back(<numero>)
<tartaruga>.left(<numero>)
<tartaruga>.right(<numero>)
<tartaruga>.up()
<tartaruga>.down()
<piano>.reset()
```

- Gli oggetti della classe `Turtle`, hanno anche diversi attributi, due di questi sono: `width` e `color`, la sintassi per modificare questi attributi è:

```
<tartaruga>.width = <numero>
<tartaruga>.color = <colore>
```

CAPITOLO 4

Un po" di vocabolario

Dove ci si allontana un po" dalla tastiera per riflettere e capire meglio.

4.1 Computer

Ora vediamo alcune parole che possono essere utili per capire meglio questo linguaggio di programmazione.

Possiamo pensare il computer formato da:

- Il microprocessore: un circuito in grado di leggere e modificare il contenuto della memoria RAM e di eseguire alcune operazioni logiche o matematiche.
- La memoria RAM (Random Acces Memory, Memoria ad Accesso Casuale): contiene le istruzioni che devono essere eseguite dal microprocessore e i dati necessari al programma. La memoria RAM è realizzata con circuiti elettronici, ha una velocità paragonabile a quella del microprocessore, è volatile cioè quando si spegne il computer, perde irrimediabilmente il suo contenuto.
- La memoria di massa: generalmente costituita da dischi magnetici (hard disk), ottici (CDRom) o memorie flash (USB Disk), mantiene le informazioni memorizzate anche quando viene spento il computer, ma è piuttosto lenta.
- Le periferiche: dispositivi che permettono al computer di comunicare con l'esterno: tastiera, monitor, mouse, stampante, modem, ...

4.2 Linguaggi di programmazione

Le istruzioni che il microprocessore deve eseguire sono scritte nella memoria RAM.

Quando un computer funziona, il microprocessore legge una ad una le istruzioni scritte nella memoria RAM e le esegue. Scrivere un programma vuol dire mettere nella memoria RAM le istruzioni giuste per far fare al microprocessore quello che vogliamo noi. Purtroppo il microprocessore non capisce comandi sensati, ma solo il linguaggio macchina che è composto da numeri binari cioè scritti usando solo le due cifre: zero e uno.

Perciò non potremmo scrivere nella RAM:

```
Somma 3 a 5
```

ma dovremmo scrivere qualcosa di questo genere:

```
1000110011101010
0000000000000011
0000000000000101
```

Ovviamente scrivere un programma, magari fatto da migliaia o milioni di istruzioni, utilizzando numeri binari è un'impresa da super eroi! I programmatori hanno ben presto scritto dei programmi che leggono comandi sensati e li traducono in numeri binari. In questo modo si possono scrivere programmi utilizzando linguaggi più vicini al linguaggio dell'uomo che a quello della macchina: linguaggi di alto livello.

Ci sono due modi per trasformare un programma scritto con linguaggio di alto livello in un programma scritto in linguaggio macchina: tradurlo tutto e poi eseguirlo oppure tradurre ogni singolo comando ed eseguirlo immediatamente. Nel primo caso si parla di *compilatori* che traducono tutto un programma scritto con linguaggio di alto livello trasformandolo in un programma scritto in linguaggio macchina e perciò eseguibile da un dato computer senza più la necessità della presenza del compilatore stesso. Nel secondo caso si parla di *interpreti*: per eseguire un programma interpretato, nel computer deve essere presente l'interprete del linguaggio che legge, traduce e fa eseguire istruzione per istruzione.

4.3 Python

`Python` è un linguaggio interpretato (in realtà le cose sono un po'più complicate di così): non si può eseguire un programma scritto in `Python` se sul computer non è presente l'interprete. Il nucleo del linguaggio `Python` è piuttosto limitato: può capire e far eseguire pochi (si fa per dire!) comandi. Ma è dotato di un meccanismo per cui può essere ampliato all'infinito: si possono scrivere librerie che contengono nuove funzioni e che allargano le possibilità del linguaggio. In effetti `Python` viene fornito con un gran numero di librerie già scritte e notevolmente sicure perché già utilizzate e messe alla prova da programmatori di tutto il mondo. Molte di queste librerie sono scritte in `Python` stesso e perciò si possono facilmente studiare e, magari, modificare. Altre, per ragioni di efficienza e di velocità, sono scritte usando linguaggi compilati.

Le librerie sono dunque dei programmi che ampliano le possibilità del linguaggio mettendo a disposizione del programmatore delle nuove funzioni non presenti nel nucleo del linguaggio.

Per utilizzare il contenuto di una libreria si deve dire all'interprete che la si vuole usare, il comando per includere nell'interprete le funzioni di una libreria è:

```
>>> import <nome della libreria> [as <abbreviazione>]
```

Ad esempio, il comando:

```
>>> import pygraph.pyturtle as tg
```

significa: carica tutta la libreria `pyturtle` chiamandola `tg`.

A questo punto possiamo creare tutti gli oggetti forniti dalla libreria `pyturtle`. Per incominciare ce ne servono due che vogliamo associare a due nomi:

```
>>> piano = tg.TurtlePlane()
>>> tina = tg.Turtle()
```

Bisogna prestare attenzione alle maiuscole e alle parentesi. Se tutto fila liscio a questo punto appare una nuova finestra sullo schermo è l'oggetto *piano della tartaruga* collegato all'identificatore `piano`.

Con, al centro un triangolino che rappresenta la tartaruga `tina`.

La classe `TurtlePlane` ha un metodo che crea e dà come risultato una nuova tartaruga. Per creare una tartaruga si può anche scrivere:

```
>>> tina = piano.newTurtle()
```

Gli oggetti possiedono degli *attributi* e dei *metodi*. Gli *attributi* sono delle caratteristiche che possono variare da un oggetto ad un altro come il colore, la posizione, lo spessore della penna, ... i *metodi* sono i comandi che l'oggetto è in grado di eseguire.

Gli attributi fondamentali della classe `Turtle` sono:

```
<tartaruga>.color
<tartaruga>.width
<tartaruga>.position
<tartaruga>.direction
```

e i metodi:

```
<tartaruga>.forward(<numero>)
<tartaruga>.back(<numero>)
<tartaruga>.left(<numero>)
<tartaruga>.right(<numero>)
<tartaruga>.up()
<tartaruga>.down()
```

Della classe `TurtlePlane` abbiamo visto i metodi:: <piano>.reset() <piano>.newTurtle()

4.4 Riassumendo

- `Python` è un linguaggio (compilato e) interpretato.
- È arricchito da numerose librerie. Una libreria che contiene la grafica della tartaruga è `pyturtle`.
- Si può caricare una libreria con il comando:

  ```
  import <nome libreria> [as <abbreviazione>]
  ```

  Ad esempio:

  ```
  import pygraph.pyturtle as tg
  ```

- Nella libreria `pyturtle` è descritta la classe `TurtlePlane`.
- È possibile creare un oggetto scrivendo un nome, il simbolo di uguale e il nome della classe seguito da una coppia di parentesi tonde. Ad esempio:

  ```
  mondoditina = tg.TurtlePlane()
  tina = tg.Turtle()
  ```

- È possibile fare eseguire ad un oggetto un suo metodo scrivendo il nome dell'oggetto seguito dal punto e dal nome del metodo completo di parentesi contenenti, se necessario, uno o più valori (argomenti). Ad esempio:

  ```
  tina.forward(100)
  ```

CAPITOLO 5

Le funzioni

Dove si impara come insegnare a ''Python'' a eseguire nuovi comandi.

5.1 Definire una funzione

Riprendiamo il lavoro con la grafica della Tartaruga. Se abbiamo appena avviato `IDLE` dobbiamo procurarci una tartaruga con le istruzioni:

```
>>> import pygraph.pyturtle as tg
>>> tp = tg.Turtleplane()
>>> tina = tp.newTurtle()
```

Riscriviamo il comando per disegnare un quadrato

```
>>> for i in range(4):
        tina.forward(30)
        tina.left(90)
```

Ora, in un programma può darsi che ci sia bisogno di disegnare molti quadrati, è scomodo e poco chiaro ripetere le tre righe scritte sopra ogni volta che serve un quadrato. Sarebbe molto più semplice avere un comando, magari di nome `quadrato`, che disegni un quadrato. In pratica sarebbe bello poter scrivere:

```
>>> quadrato()
```

e ottenere il disegno del quadrato al posto del messaggio:

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in ?
    quadrato()
NameError: name 'quadrato' is not defined
```

Gli informatici hanno pensato ad un modo per esaudire questo desiderio che, evidentemente, è di tutti quelli che si mettono a programmare. Hanno realizzato un meccanismo per cui si può ampliare un linguaggio insegnandogli a eseguire comandi nuovi di nostra invenzione. Questi nuovi comandi si chiamano procedure o funzioni.

Per insegnare a Python una nuova funzione bisogna scrivere la parola riservata def seguita dal nome che vogliamo dare alla funzione, da una coppia di parentesi e dal simbolo ":" (due punti). Dopo questa riga, con un opportuno rientro (indentazione), dobbiamo scrivere tutte le istruzioni che vogliamo vengano eseguite quando si chiamerà la funzione.

Per insegnare a Python a disegnare quadrati con la tartaruga dovremo scrivere:

```
>>> def quadrato():
        for i in range(4):
            tina.forward(30)
            tina.left(90)
```

Dobbiamo terminare la scrittura della funzione premendo due volte il tasto <Invio>. Ma non succede assolutamente niente! Infatti abbiamo insegnato a disegnare un quadrato, ma non abbiamo detto a Python di disegnarlo!

5.2 Eseguire una funzione

Proviamo a dare il comando:

```
>>> quadrato
<function quadrato at 0x836596c>
```

Strano messaggio... Non è un messaggio di errore, semplicemente Python ci avvisa che quadrato è una funzione. Per dire a Python che deve eseguire la funzione quadrato dobbiamo far seguire al nome le parentesi:

```
>>> quadrato()
```

Ora va! tina ha disegnato per noi il quadrato! In modo analogo possiamo insegnare altre funzioni ad esempio per disegnare un triangolo equilatero:

```
>>> def triangolo():
        for i in range(3):
            tina.forward(30)
            tina.left(...)
```

Ovviamente al posto dei puntini devo mettere un numero: il numero di gradi dell'angolo del triangolo (quale angolo?). E poi provarlo:

```
>>> triangolo()
```

Una volta insegnata una nuova funzione possiamo utilizzarla esattamente come le funzioni primitive del linguaggio, in particolare possiamo richiamarla dall'interno di un'altra funzione:

```
>>> def bandierina():
        tina.forward(100)
        quadrato()
        tina.back(100)
```

Ripuliamo lo schermo e disegniamo la bandierina:

```
>>> tp.reset()
>>> bandierina()
```

Ovviamente possiamo scrivere una funzione che chiama una funzione che chiama una funzione, che... Possiamo, ad esempio, mettere insieme tante bandierine per costruire una girandola:

```
>>> def girandola():
    for i in range(36):
```

```
        bandierina()
        tina.left(10)
```

E poi provarla:

```
>>> girandola()
```

Troppe bandierine: modifichiamo la funzione per diminuirne il numero. Possiamo riscrivere interamente la funzione, oppure portare il cursore sopra la prima riga e premere <Invio>. La funzione viene riscritta e abbiamo la possibilità di modificarla.

```
>>> def girandola():
        for i in range(24):
            bandierina()
            tina.left(360./24)
```

Qui ho usato un trucchetto, invece di fare le divisioni a mente, le facciamo fare a Python che ha tutta la potenza del computer a disposizione. Se usiamo Python 2 dobbiamo ricordagli però di usare l'aritmetica decimale, non quella intera (360.0/24). Ripuliamo lo schermo e proviamola:

```
>>> tp.reset()
>>> girandola()
```

Ancora troppe, diminuiamole:

```
>>> def girandola():
    for i in range(20):
        bandierina()
        tina.left(360./20)
```

E proviamola:

```
>>> tina.reset()
>>> girandola()
```

Ohohoh! Proprio come volevo... A dire il vero mi sembra un po" smorta come girandola. Un po" di colore non guasterebbe! Modifichiamo la funzione che disegna le bandierine in modo che le disegni con il contorno marron e l'interno verde. Utilizzeremo il metodo fill(flag), quando flag vale 1 la tartaruga incomincia a segnare gli oggetti da riempire di colore, quando flag vale 0 li riempie con il colore attuale della penna.

```
>>> def bandierina():
        tina.color = "brown"
        tina.forward(100)
        tina.fill(1)
        quadrato()
        tina.color = "green"
        tina.fill(0)
        tina.color = "brown"
        tina.back(100)
```

Proviamola...

```
>>> tp.reset()
>>> tina.width = 4
>>> girandola()
```

Adesso mi piace!

5.3 Riassumendo

- Possiamo insegnare a `Python` a interpretare ed eseguire comandi nuovi, la sintassi per fare ciò è:

```
def <nome della funzione>():
    <istruzioni>
```

- Una funzione scritta dal programmatore può essere utilizzata esattamente come quelle primitive del linguaggio.
- Per eseguire una funzione devo scriverne il nome seguito da una coppia di parentesi:

```
<nome della funzione>()
```

- Una nuova funzione può essere richiamata da un'altra funzione e così via ricorsivamente.
- `Python 2` ha un modo un po" suo di fare le divisioni: se dividendo e divisore sono numeri interi dà come risultato il quoziente intero troncando il risultato. Per ottenere il risultato razionale uno degli operandi deve essere un numero con la virgola (il punto in `Python`).

CAPITOLO 6

I parametri

Dove si impara a rendere più flessibili le funzioni definite dal programmatore.

6.1 Scrivere funzioni con parametri

La procedura quadrato ci permette di disegnare quanti quadrati vogliamo scrivendo un solo comando senza dover riscrivere ogni volta il ciclo `for`. È un risparmio di righe di codice e una bella semplificazione, pensiamo se dovessimo disegnare cento quadrati per disegnare un muro! Ma resta un problema. Se abbiamo bisogno di quadrati più grandi o più piccoli? Per ogni lunghezza del lato dobbiamo scrivere una procedura diversa. Prima di eseguire gli esempi di questo capitolo bisogna creare una tartaruga come visto nel capitolo precedente:

```
>>> import pygraph.pyturtle as tg
>>> tp = tg.TurtlePlane()
>>> tina = tg.Turtle()
```

Ad esempio se ci serve un quadrato piccolo e uno grande:

```
>>> def quadratino():
        for i in range(4):
            tina.forward(10)
            tina.left(90)

>>> def quadratone():
        for i in range(4):
            tina.forward(200)
            tina.left(90)

>>> quadratino()
>>> quadratone()
```

La soluzione potrebbe anche andare bene se mi bastassero quadrati con due lunghezze diverse di lati, ma se avessi bisogno di molti quadrati con lati diversi? L'interprete mette a disposizione un meccanismo simile a quello usato dal

linguaggio `Python` che permette di tracciare linee di lunghezze diverse semplicemente mettendo tra parentesi la sua lunghezza:

```
>>>tina.forward(50)
>>>tina.forward(57)
>>>tina.forward(72)
>>>tina.forward(163)
...
```

Cioè permette di scrivere:

```
>>>quadrato(50)
>>>quadrato(57)
>>>quadrato(72)
>>>quadrato(163)
...
```

Per ottenere quadrati di lato: 50, 57, 72, 163, ...

Tutti i linguaggi di programmazione moderni danno questa possibilità. Prima di vedere come fare ciò, riguardiamo le due procedure quadratino e quadratone. Sono praticamente uguali, cambia solo il numero che indica la lunghezza del lato. Al posto di questo numero noi possiamo mettere il nome di una variabile:

```
...
        tina.forward(lato)
...
```

Il nome della variabile ovviamente non ha nessun significato per il computer, ma è bene scegliere un nome che sia significativo per noi.

All'interno della funzione, quando `Python` trova il nome di una variabile lo sostituisce con il suo valore. Quindi se alla variabile di nome `lato` è collegato il numero 50, verrà disegnato un quadrato con il lato lungo 50, se è collegato il numero 57 la lunghezza del lato sarà di 57 unità, e così via.

Ma qual è il meccanismo per mettere un valore nella variabile lato? Quando viene definita una funzione, `Python` permette di inserire tra le parentesi che seguono il suo nome il nome di una o più variabili:

```
>>> def quadrato(lato):
...
```

Le variabili legate alle funzioni in questo modo, si chiamano *parametri*.

Quando viene chiamata una funzione, il valore che viene scritto tra parentesi, viene, dall'interprete, collegato a quella variabile:

```
>>> def quadrato(lato):
...
...
>>> quadrato(57)
```

Nell'esempio precedente `lato` è il parametro della funzione `quadrato` e 57 è l'argomento con cui viene chiamata la funzione `quadrato`. Dopo questo comando, la variabile `lato` contiene il numero 57. La funzione quadrato diventa dunque:

```
>>> def quadrato(lato):
        for i in range(4):
            tina.forward(lato)
            tina.left(90)
```

Proviamola:

```
>>> quadrato()
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in ?
    quadrato()
TypeError: quadrato() takes exactly 1 argument (0 given)
```

Accidenti, un errore! Già, ora non basta dire che voglio un quadrato, devo anche specificare la lunghezza del suo lato:

```
>>> quadrato(47)
>>> quadrato(58)
>>> quadrato(69)
```

Il parametro, cioè la variabile scritta nella definizione della funzione, è visibile solo all'interno della funzione stessa e non interferisce con altre variabili presenti nel programma.

6.2 Usare funzioni con parametri

La nuova funzione `quadrato` può essere utilizzata all'interno di altre funzioni esattamente come le funzioni primitive del linguaggio o le altre funzioni presenti nelle librerie. Possiamo quindi scrivere una nuova funzione che disegni molti quadrati:

```
>>> def quadrati():
        for dim in range(0, 200, 20):
            quadrato(dim)
```

Questa volta range(0, 200, 20) genera tutti i numeri da 0 (compreso) a 200 (escluso) andando di 20 in 20, ognuno di questi numeri viene collegato alla variabile di nome `dim` e, ogni volta, viene disegnato un quadrato con quel lato. Proviamo:

```
>>> tp.reset()
>>> quadrati()
```

Avrei ottenuto lo stesso risultato definendo quadrati in questo modo:

```
>>> def quadrati():
        for n in range(10):
            quadrato(n*20)
```

Quale delle due procedure è più chiara? È una questione di gusti personali... nel seguito teniamo questa seconda versione. Possiamo parametrizzare anche questa funzione: le assegniamo una variabile che controlli il numero di quadrati:

```
>>> def quadrati(numero):
        for n in range(numero):
            quadrato(n*20)
```

Può darsi che la distanza di 20 unità, tra un quadrato e l'altro, non sia di nostro gusto: possiamo aggiungere un parametro in modo da controllare l'incremento del lato:

```
>>> def quadrati(numero, incremento):
        for n in range(numero):
            quadrato(n*incremento)
```

Ora perché `quadrati` venga eseguito dobbiamo passargli 2 valori, possiamo fare diverse prove:

```
>>> tp.reset()
>>> quadrati(12, 5)
>>> tp.reset()
>>> quadrati(70, 2)
>>> tp.reset()
>>> quadrati(5, 50)
...
```

Ovviamente anche `quadrati(numero, incremento)` può essere utilizzato all'interno di altre funzioni, proviamo a disegnare una griglia di quadrati:

```
>>> def griglia():
        for i in range(4):
            quadrati(18, 10)
            tina.left(90)

>>> tp.reset()
>>> griglia()
```

Ma anche `griglia()` può essere parametrizzata...

6.3 Riassumendo

- Possiamo insegnare a `Python` a interpretare ed eseguire comandi nuovi dotati di parametri, la sintassi per fare ciò è:

```
def <nome della funzione>(<nome del parametro>):
    <istruzioni>
```

- Le funzioni con parametri sono molto più flessibili di quelle senza e risolvono intere classi di problemi invece che un solo problema (tutti i possibili quadrati invece che un solo quadrato).
- Nella definizione della procedura viene deciso il nome dei parametri.
- Il parametro è una variabile locale che viene creata quando viene chiamata la funzione e viene distrutta quando la funzione termina.
- Nella chiamata della funzione viene deciso il valore del parametro viene cioè passato alla funzione un argomento.
- Il valore viene assegnato al parametro ad ogni chiamata di funzione.

CAPITOLO 7

Un programma

Dove si scrive il primo programma.

7.1 Iniziare un programma

La possibilità di scrivere, rivedere, modificare, rieseguire linee di comandi permette di realizzare disegni anche piuttosto complicati utilizzando solo la shell di `IDLE`. Ma se vogliamo risolvere problemi più complessi dobbiamo realizzare tante funzioni e non è detto che riusciamo a farlo all'interno di un'unica sessione di lavoro. Dobbiamo scrivere un programma `Python` in una finestra di editor e salvarlo in un file in modo da poterlo riprendere, correggere, completare. Un programma è un testo che contiene una sequenza di funzioni, di comandi e di commenti. Il programma deve essere scritto in un testo *piano*, senza nessun tipo di formattazione per cui non va usato, per scriverlo, un normale elaboratore di testi, ma un «editor» che permette di scrivere e di salvare esattamente e solo i caratteri immessi dalla tastiera.

Oltre alla shell, che abbiamo utilizzato fin'ora, `IDLE` metta a disposizione anche un editor di testo e sarà quello che utilizzeremo. Per iniziare un nuovo programma, dal menu `File` scegliamo la voce `New File` o `New Window` e apparirà una nuova finestra completamente vuota con il cursore lampeggiante in alto a sinistra.

Prima ancora di incominciare a riempirla, pensiamo ad un nome da dare al programma, il nome deve essere significativo per noi. Nel mio caso, dato che questo è il primo programma che ho scritto per il manuale l'ho chiamato: «man1.py».

Nota: L'estensione «.py» indica che questo file contiene un programma scritto in `Python`. `IDLE`, fino ad una certa versione, non aggiunge l'estensione al file quindi bisogna ricordarsi di terminare sempre il nome del programma con le tre lettere: «`.py`» se vogliamo che il testo sia riconosciuto come un programma scritto in `Python`.

Pensato ad un nome adatto, salviamo il file prestando ben attenzione a dove viene salvato, in modo da riuscire a ritrovarlo in seguito: dal menu `File` scegliamo la voce `Salva`, spostiamoci nella cartella nella quale abbiamo deciso di mettere il programma, scriviamo il nome del programma e confermiamo con il tasto `<Invio>`.

7.2 I commenti

I comandi e le funzioni sono già stati visti dei capitoli precedenti. I commenti sono molto importanti per poter capire programmi scritti da altre persone ma anche programmi scritti da noi stessi, magari qualche tempo prima. Ci sono due tipi di commenti: commenti di linea e commenti su più linee. Il carattere cancelletto, `#`, indica l'inizio di un commento che si estende fino alla fine della riga. Quando l'interprete `Python` incontra il carattere `#` passa ad interpretare la riga seguente:

```
# questo è un commento
```

Se abbiamo bisogno di scrivere un commento che si estenda su più linee possiamo iniziare ogni linea con il carattere `#` oppure iniziare e terminare il commento con tre virgolette `"""` di seguito:

```
"""Questo è un

commento scritto su

più linee"""
```

7.3 Il primo mattone del programma

Rimbocchiamoci le maniche e iniziamo a scrivere il nostro primo programma.

Ogni programma deve essere ben commentato, in particolare deve avere un'intestazione che indichi l'interprete, il nome del file, l'autore, la data il tipo di licenza e, ovviamente, un titolo.

In questo esempio ci poniamo l'obiettivo di far disegnare alla tartaruga un muro di mattoni quadrati. L'intestazione del programma potrebbe essere:

```
# man1.py
# 2 gennaio 2018
# Daniele Zambelli

"""
Il programma deve disegnare un muro di mattoni quadrati.
"""
```

Le prime tre righe danno alcune informazioni sul programma: nome del file, data di creazione, autore, ... e con una *docstring* . Il commento scritto sotto forma di stringa, quello tra virgolette triple, si chiama *docstring* del programma: è un testo (una stringa) di documentazione che sintetizza ciò che fa, o deve fare, il programma stesso.

Scritta l'intestazione, se vogliamo, possiamo anche eseguire il programma. Possiamo farlo attraverso il menu:

```
Run - Run module
```

o premendo semplicemente il tasto:

```
<F5>
```

Per tutta risposta `IDLE` ci chiede se volgiamo salvare le modifiche apportate al programma. Noi, ovviamente, confermiamo.

Nota: dato che ogni volta che vogliamo eseguire il programma, vogliamo anche salvare le modifiche, possiamo andare su menu `Option-Configure IDLE` e nella scheda `General` selezionare `No Prompt`. D'ora in poi, dopo la pressione del tasto `F5` non ci verrà più richiesta la conferma del salvataggio del file.

Python lo interpreterà correttamente, non darà alcun messaggio di errore, ma, essendo composto solo da commenti non eseguirà assolutamente niente. Incominciamo allora ad aggiungere istruzioni. Poiché vogliamo utilizzare la grafica della tartaruga, come primo comando dobbiamo caricare la libreria, creare un piano e chiedere al piano di produrre una nuova tartaruga:

```
import pygraph.pyturtle as tg
tp = tg.TurtlePlane()
tina = tg.Turtle()
```

Ora, per costruire un muro di mattoni quadrati, abbiamo bisogno di una funzione che disegni quadrati:

```
def quadrato(lato):
    """Disegna un quadrato di dato lato."""
    for i in range(4)
        tina.forward(lato)
        tina.left(90)
```

Ormai siamo super esperti in quadrati... Ma questa volta aggiungiamo un commento come prima riga della funzione. Si chiama *docstring* e ogni funzione, oltre ad un nome significativo, dovrebbe averne una.

Ora dobbiamo eseguire il programma (Run - Run module o più velocemente tasto <F5>). A questo punto i casi sono due:

1. Non succede assolutamente niente.
2. Python scrive alcune strane righe nella shell di IDLE.

Nel primo caso, vuol dire che è andato tutto bene, Python non ha eseguito niente perché noi abbiamo definito una funzione, ma non abbiamo dato il comando di eseguirla e giustamente il computer non si prende la responsabilità di farlo di propria iniziativa.

Nel secondo caso, le strane scritte sono un messaggio di errore: Python non è riuscito a interpretare quello che abbiamo scritto e quindi non ha definito la procedura.

Tranello! Se la procedura è stata scritta esattamente come riportato sopra, Python ci risponde con:

```
File ".../sorgenti/graph/man1.py", line 10
  for i in range(4)
                   ^
SyntaxError: invalid syntax
```

Questo messaggio ci dà le seguenti informazioni:

- in quale file si trova l'errore: .../sorgenti/graph/man1.py
- la linea dove ha incontrato un errore: line 10,
- il punto in cui l'interprete si è bloccato,
- il tipo di errore: SyntaxError: invalid syntax

Già, ci siamo dimenticati i «:», correggiamola:

```
def quadrato(lato):
    """Disegna un quadrato di dato lato."""
    for i in range(4):
        tina.forward(lato)
        tina.left(90)
```

Ora dobbiamo eseguire di nuovo il programma tasto <F5> o menu: Run - Run module. Se tutto è andato bene, non dovrebbe succedere niente. Spostiamoci nella shell di IDLE e diamo il comando:

```
>>> quadrato(57)
```

Se non ci sono altri errori dovremmo ottenere un quadrato. Possiamo provare il funzionamento della funzione `quadrato(lato)` con diversi valori dell'argomento. È anche possibile aggiungere questo comando in fondo al programma. In questo modo, ogni volta che si esegue il programma verrà creata una tartaruga e disegnato un quadrato. Tutto il nostro programma è ora:

```
# man1.py
# 2 gennaio 2018
# Daniele Zambelli

"""
Il programma deve disegnare un muro di mattoni quadrati.
"""

# Lettura delle librerie
import pygraph.pyturtle as tg

# Definizione di funzioni

def quadrato(lato):
    """ Disegna un quadrato di dato lato """
    for cont in range(4):
        tina.forward(lato)
        tina.left(90)

# Programma principale
tp = tg.TurtlePlane()
tina = tg.Turtle()
quadrato(57)

# Rende attiva la finestra grafica
piano.mainloop()
```

L'ultima istruzione, come scritto nel commento, passa il comando alla finestra grafica rendendola attiva.

Eseguiamolo, se tutto funziona correttamente appare la finestra della grafica della tartaruga con dentro un quadrato. Bene, abbiamo scritto, salvato e provato il nostro primo programma funzionante, possiamo rilassarci un po" prima di procedere con il lavoro...

Nota: D'ora in poi non riporterò più l'intestazione e la docstring del programma.

7.4 Uno strato di mattoni

Disegnare un intero muro di quadrati può sembrare un'impresa piuttosto complicata. L'intero muro può essere pensato come composto da più file e ogni fila da più quadrati:

muro —(fatto da)—> **fila** —(fatta da)—> **quadrato**

La funzione che disegna un quadrato l'abbiamo realizzata, proviamo allora a realizzare la funziona che realizza una fila di quadrati tutti uguali. Questa funzione avrà bisogno di due informazioni: la dimensione dei quadrati e il numero di quadrati da mettere in una fila. Cioè vorremmo poter scrivere:

```
>>> fila(20, 15)
```

per ottenere una fila di `15` quadrati di lato `20`. Per il numero di volte che abbiamo deciso, la funzione `fila` deve disegnare un quadrato e spostare la tartaruga avanti di un tratto uguale al lato del quadrato. La funzione potrebbe essere:

```
def fila(lato, numero):
    """Disegna una fila di mattoni."""
    for cont_col in range(numero):
        quadrato(lato)
        tina.forward(lato)
```

`cont_col` avremmo anche potuto chiamarla: `contatore_delle_colonne` oppure semplicemente: `i` o `j`. In generale dobbiamo dare alle variabili nomi significativi, ma non troppo lunghi. Aggiungiamo al programma la funzione `fila(lato, numero)` sotto alla funzione `quadrato(lato)`, e, nel programma principale, al posto di disegnare un quadrato disegniamo una fila di quadrati:

```
# Lettura delle librerie
import pygraph.pyturtle as tg

# Definizione di funzioni

def quadrato(lato):
    """ Disegna un quadrato di dato lato """
    for cont in range(4):
        tina.forward(lato)
        tina.left(90)

def fila(lato, numero):
    """Disegna una fila di mattoni."""
    for cont_col in range(numero):
        quadrato(lato)
        tina.forward(lato)

# Programma principale
tp = tg.TurtlePlane()
tina = tg.Turtle()
fila(23, 7)

# Rende attiva la finestra grafica
piano.mainloop()
```

Eseguiamo il programma e proviamo la funzione modificando gli argomenti in modo da realizzare file più o meno lunghe di quadrati più o meno grandi.

La fila di quadrati viene disegnata, ma, la procedura `fila`, non è «trasparente»: oltre a disegnare una fila di quadretti, **sposta la tartaruga** senza rimetterla dove l'aveva trovata. È molto importante che ogni funzione faccia solo quello che dice di fare, senza effetti collaterali. Quindi la funzione `fila` deve preoccuparsi di rimettere a posto Tartaruga. Di quanto l'ha spostata? Per `numero` volte è andata avanti di `lato` passi, in totale `numero*lato`. Per rimettere a posto Tartaruga bisogna, finito il ciclo, farla indietreggiare della stessa quantità:

```
def fila(lato, numero):
    """Disegna una fila di mattoni."""
    for cont_col in range(numero):
        quadrato(lato)
        tina.forward(lato)
    tina.back(numero*lato)
```

La fila di quadrati che abbiamo ottenuto non richiama per niente dei mattoni, sono troppo appiccicati: un po" di malta tra uno e l'altro? Per distanziarli basta allungare lo spostamento dopo ogni quadrato e ovviamente anche al ritorno:

```
def fila(lato, numero):
    """Disegna una fila di mattoni."""
    for cont_col in range(numero):
        quadrato(lato)
        tina.forward(lato+5)
    tina.back(numero*(lato+5))
```

Funziona? Sì, ma non va bene. Esistono due tipi di errori quelli che bloccano il programma detti anche errori *di sintassi* e quelli che fanno fare al programma una cosa diversa da quella che volevamo, gli errori *di semantica*. Nel nostro caso la procedura funziona però non disegna quadrati staccati l'uno dall'altro, ma uniti da una linea non certo bella da vedere. Lo spostamento tra un quadrato e l'altro deve essere fatto senza lasciare segno:

```
def fila(lato, numero):
    """Disegna una fila di mattoni."""
    for cont_col in range(numero):
        quadrato(lato)
        tina.up()
        tina.forward(lato+5)
        tina.down()
    tina.up()
    tina.back(numero*(lato+5))
    tina.down()
```

7.5 L'intero muro

Possiamo essere soddisfatti del risultato ottenuto. Ora passiamo alla costruzione del muro. Cos'è un muro? È una `pila` di `file` di `mattoni`.

```
def muro():
    """Disegna un muro di quadrati."""
    for cont_rig in range(15):
        fila(20, 18)
        tina.up()
        tina.left(90)
        tina.forward(20+5)
        tina.right(90)
        tina.down()
    tina.up()
    tina.left(90)
    tina.forward(15*(20+5))
    tina.right(90)
    tina.down()
```

Aggiungiamo questa come terza funzione nel programma precedente e modifichiamo il programma principale sostituendo in modo che esegua: `muro()`. Il programma completo è:

```
# Lettura delle librerie
import pygraph.pyturtle as tg

# Definizione di funzioni

def quadrato(lato):
    """ Disegna un quadrato di dato lato """
    for cont in range(4):
        tina.forward(lato)
```

```
        tina.left(90)

def fila(lato, numero):
    """Disegna una fila di mattoni."""
    for cont_col in range(numero):
        quadrato(lato)
        tina.up()
        tina.forward(lato+5)
        tina.down()
    tina.up()
    tina.back(numero*(lato+5))
    tina.down()

def muro():
    """Disegna un muro di quadrati."""
    for cont_rig in range(15):
        fila(20, 18)
        tina.up()
        tina.left(90)
        tina.forward(20+5)
        tina.right(90)
        tina.down()
    tina.up()
    tina.left(90)
    tina.forward(15*(20+5))
    tina.right(90)
    tina.down()

# Programma principale
tp = tg.TurtlePlane()
tina = tg.Turtle()

tina.up()
tina.back(200)
tina.down()
muro()

# Rende attiva la finestra grafica
piano.mainloop()
```

Proviamola... funziona! ... Ma...

7.6 Riassumendo

- Un programma è un documento di testo che contiene istruzioni e funzioni.
- Scritto un programma bisogna salvarlo: menu: `File - Save` ed eseguirlo: menu: `Run - Run module`.
- Ogni programma *deve* iniziare con dei commenti che forniscono alcune informazioni generali sul programma stesso: nome del file, data di creazione, autore, ... e con una *docstring* che sintetizza ciò che fa, o deve fare, il programma stesso.
- Le funzioni definite in un programma possono essere eseguite anche dall'ambiente shell `IDLE`.
- Non è mai conveniente cercare di scrivere un intero programma e poi provarlo. Conviene, inizialmente, individuare obiettivi semplici, realizzarli e man mano completarlo provandone il funzionamento ad ogni aggiunta.

CAPITOLO 8

Refactoring

Dove si ristruttura un programma funzionante.

... Ma c'è una certa differenza tra un programma che funziona e un buon programma. Se vogliamo imparare a programmare non dobbiamo accontentarci di un programma che funziona, dobbiamo affinare una certa sensibilità anche all'aspetto estetico. Non dobbiamo affezionarci troppo al nostro prodotto, ma cercare di migliorarlo.

8.1 Parametri di default

Il risultato del programma è soddisfacente e anche la distanza tra i mattoni sembra adeguata, ma se cambiamo il lato dei quadrati, andrà sempre bene? Possiamo parametrizzare anche quella, ma lo facciamo dandole come valore predefinito 5:

```
def fila(lato, numero, spazio_colonne=5):
    """Disegna una fila di mattoni."""
    for cont_col in range(numero):
        quadrato(lato)
        tina.up()
        tina.forward(lato+spazio_colonne)
        tina.down()
    tina.up()
    tina.back(numero*(lato+spazio_colonne))
    tina.down()
```

In questo modo la funzione fila può essere chiamata con due o con tre argomenti:

- Se viene chiamata con due argomenti, il terzo viene automaticamente posto uguale a 5,
- Se viene chiamata con tre argomenti, il terzo parametro assumerà il terzo valore. :

```
>>> fila(30, 3)
```

Disegna una fila di 3 quadrati di lato 30 distanziati di 5 passi,

```
>>> fila(30, 3, 12)
```

Disegna una fila di 3 quadrati di lato 30 distanziati di 12 passi.

In questo modo la funzione `fila(lato, numero, spazio_colonne=5)` è diventata più flessibile.

Anche la funzione `muro()` va cambiata, ci sono un paio di cose che stonano in questa funzione:

- cinque righe si ripetono quasi identiche, e non va bene che in un programma si ripetano blocchi di istruzioni (quasi) identiche;
- ci sono troppi numeri;

Partiamo dal primo problema: la soluzione è costruire una funzione che le esegua con un solo comando.

Le tre righe che vanno da `tina.up()` a `tina.down()` producono uno spostamento di Tartaruga in avanti e a sinistra, senza lasciare traccia. Possiamo generalizzare questo comportamento aggiungendo oltre allo spostamento verticale uno orizzontale. Possiamo costruire una funzione `sposta` che riceve come argomenti lo spostamento nella direzione della Tartaruga e lo spostamento nella sua direzione perpendicolare. Questa funzione dovrà avere quindi due parametri:

```
def sposta(avanti=0, sinistra=0):
    """Effettua uno spostamento di Tartaruga
       in avanti e verso sinistra senza disegnare la traccia."""
    tina.up()
    tina.forward(avanti)
    tina.left(90)
    tina.forward(sinistra)
    tina.right(90)
    tina.down()
```

Scrivendo i parametri in questo modo, possiamo chiamare la funzione sposta in vari modi:

- `sposta()`, non fa niente;
- `sposta(47)`, sposta avanti Tartaruga di 47 unità senza tracciare segni;
- `sposta(47, 61)`, sposta Tartaruga avanti di 47 e a sinistra di 61 unità senza tracciare segni;
- `sposta(sinistra=61)`, sposta Tartaruga a sinistra di 61 unità senza tracciare segni;

Anche la funzione `fila` può essere migliorata usando questa funzione:

```
def fila(lato, numero, spazio_colonne=5):
    """Disegna una fila di mattoni quadrati."""
    for cont_col in range(numero):
        quadrato(lato)
        sposta(avanti=lato+spazio_colonne)
    sposta(avanti=-numero*(lato+spazio_colonne))
```

E muro diventa:

```
def muro():
    """Disegna un muro di quadrati."""
    for cont_rig in range(15):
        fila(20, 18,)
        sposta(sinistra=20+5)
    sposta(sinistra=-15*(20+5))
```

Nota: in queste funzioni ho utilizzato un terzo metodo per inserire un argomento in un parametro: il passaggio dell'argomento «per nome».

Confrontando la versione precedente di `muro` con questa si può notare una bella semplificazione!

Ora occupiamoci di eliminare un po" di numeri parametrizzando anche la procedura `muro`. I numeri presenti nella funzione riguardano: la lunghezza del lato, il numero di righe, il numero di colonne, lo spazio tra le righe, lo spazio tra le colonne. Questi due ultimi valori possono essere lasciati di default uguali a 5.

Trasformandoli tutti in parametri la funzione `muro` diventa:

```
def muro(lato, righe, colonne, spazio_colonne=5, spazio_righe=5):
    """Disegna un muro di mattoni."""
    for cont_rig in range(righe):
        fila(lato, colonne, spazio_colonne)
        sposta(sinistra=lato+spazio_righe)
    sposta(sinistra=-righe*(lato+spazio_righe))
```

E tutto il programma diventa:

```
# Lettura delle librerie
import pygraph.pyturtle as tg

# Definizione di funzioni

def sposta(avanti=0, sinistra=0):
    """Effettua uno spostamento orizzontale e verticale
       di Tartaruga senza disegnare la traccia."""
    tina.up()
    tina.forward(avanti)
    tina.left(90)
    tina.forward(sinistra)
    tina.right(90)
    tina.down()

def quadrato(lato):
    """Disegna un quadrato di dato lato vuoto o pieno."""
    for cont in range(4):
        tina.forward(lato)
        tina.left(90)

def fila(lato, numero, spazio_colonne=5):
    """Disegna una fila di mattoni quadrati."""
    for cont_col in range(numero):
        quadrato(lato)
        sposta(avanti=lato+spazio_colonne)
    sposta(avanti=-numero*(lato+spazio_colonne))

def muro(lato, righe, colonne, spazio_colonne=5, spazio_righe=5):
    """Disegna un muro di mattoni."""
    for cont_rig in range(righe):
        fila(lato, colonne, spazio_colonne)
        sposta(sinistra=lato+spazio_righe)
    sposta(sinistra=-righe*(lato+spazio_righe))

# Programma principale
piano = tg.TurtlePlane()
tina = tg.Turtle()

sposta(-250, -190)
muro(20, 15, 20)
```

```
# Rende attiva la finestra grafica
piano.mainloop()
```

8.2 Compattiamo il codice

Funziona tutto a meraviglia, o almeno dovrebbe funzionare se non ho introdotto qualche errore! Potremmo considerarci soddisfatti se l'istinto del programmatore non rodesse dentro... Perdendo un po" in chiarezza possiamo compattare di più il codice. Vale la pena? Dipende da ciò che si vuole ottenere, comunque, prima di dare un giudizio proviamo un'altra versione.

Prima avevamo staccato delle righe di codice per fare una funzione separata che realizzasse gli spostamenti data la componente orizzontale e verticale. Ora fondiamo due funzioni che hanno degli elementi in comune: la procedura `muro` e la procedura `fila`. L'intero muro si può ottenere annidando, uno dentro l'altro due cicli: il ciclo più esterno impila le file e quello più interno allinea quadrati. In pratica al posto della chiamata alla procedura `fila(...)`, trascriviamo tutte le sue istruzioni. Dobbiamo anche aggiustare i nomi e l'indentazione:

```
def muro(lato, righe, colonne, spazio_colonne=5, spazio_righe=5):
    """Disegna un muro di mattoni."""
    for cont_rig in range(righe):
        for cont_col in range(colonne):
            quadrato(lato)
            sposta(avanti=lato+spazio_colonne)
        sposta(avanti=-colonne*(lato+spazio_colonne))
        sposta(sinistra=lato+spazio_righe)
    sposta(sinistra=-righe*(lato+spazio_righe))
```

I due cicli annidati devono avere due variabili diverse noi abbiamo usato `cont_rig` e `cont_col`, ma spesso si usano per queste variabili di ciclo i nomi `i` e `j`.

Possiamo ancora eliminare una riga di codice! Come?

Alla fine del ciclo più interno ci sono due chiamate alla funzione `sposta` che possono essere sostituite da una sola.

Nota: Il fatto di ridurre le dimensioni di un programma non è solo una questione di spazio, ma è fondamentale per renderne più semplice la manutenzione.

E con questo il programma è terminato...

```
# Lettura delle librerie
import pygraph.pyturtle as tg

# Definizione di funzioni

def sposta(avanti=0, sinistra=0):
    """Effettua uno spostamento orizzontale e verticale
       di Tartaruga senza disegnare la traccia."""
    tina.up()
    tina.forward(avanti)
    tina.left(90)
    tina.forward(sinistra)
    tina.right(90)
    tina.down()

def quadrato(lato):
```

```
    """Disegna un quadrato di dato lato vuoto o pieno."""
    for cont in range(4):
        tina.forward(lato)
        tina.left(90)

def muro(lato, righe, colonne, spazio_righe=5, spazio_colonne=5):
    """Disegna un muro di mattoni."""
    for cont_rig in range(righe):
        for cont_col in range(colonne):
            quadrato(lato)
            sposta(avanti=lato+spazio_colonne)
        sposta(-colonne*(lato+spazio_colonne), lato+spazio_righe)
    sposta(sinistra=-righe*(lato+spazio_righe))

# Programma principale
piano = tg.TurtlePlane()
tina = tg.Turtle()

sposta(-250, -190)
muro(20, 15, 20)

# Rende attiva la finestra grafica
piano.mainloop()
```

. . . o quasi. . .

8.3 Riassumendo

- Quando è possibile è meglio sostituire i numeri e le costanti presenti in una funzione con parametri.
- Le procedure possono avere anche dei parametri con dei valori predefiniti (di *default*).
- Gli argomenti di una funzione possono essere passati anche *per nome*.
- Più linee di istruzioni che si ripetono all'interno di un programma possono essere raggruppate in un'unica funzione.
- All'interno di un ciclo possono essere annidati altri cicli, bisogna fare attenzione al nome delle variabili.
- Un programma più breve, più semplice e con nomi significativi è meglio di un programma più lungo, più complicato e con nomi insensati.

CAPITOLO 9

Una nuova classe

Dove definiamo la nostra prima classe

9.1 Classi e oggetti

Riassumiamo quello che avviene quando si esegue il programma `man1.py`:

1. viene caricata la libreria `pyturtle.py`;
2. viene creato un oggetto della classe `TurtlePlane()` della libreria `pyturtle.py`;
3. viene creata una tartaruga cioè un oggetto della classe `Turtle` della libreria `pyturtle.py`;
4. vengono definite tre funzioni (`sposta()`, `quadrato()`, `muro()`);
5. viene eseguito il programma principale che sposta la tartaruga e chiama la funzione `muro` che, a sua volta, chiama le funzioni `quadrato` e `sposta`.

Nel programma precedente abbiamo già creato e utilizzato un **oggetto** della **classe** `TurtlePlane` e un *oggetto* della *classe* `Turtle`. Abbiamo crearto una tartaruga con l'istruzione:

```
tina = tg.Turtle()
```

- `tina` è un nome (un *identificatore*) che ci siamo inventati noi;
- al nome `tina` è associato un oggetto della *classe* `Turtle` presente nella libreria `tg` (che sta per `pyturtle`).

Quindi `tina` è un riferimento ad un *oggetto* della *classe* `Turtle`. Nella *classe* vengono definite le caratteristiche e i comportamenti degli *oggetti* appartenenti a quella *classe*.

In questo capitolo vogliamo creare una nuova classe di oggetti.

La libreria `pyturtle` definisce le *classi* `TurtlePlane` e `Turtle`, concentriamoci sulla seconda e, per semplicità, chiamiamo *tartaruga* un *oggetto* della *classe* `Turtle`.

Una *classe* può essere utilizzata per:

1. costruire *oggetti* con le proprietà e i metodi di quella *classe*;

2. costruire altre *classi* che ampliano quella *classe*.

9.1.1 Metodi

Ogni *tartaruga* è in grado di eseguire alcuni comandi, sono i suoi **metodi**:

`forward(<numero>),back(<numero>),...`

I metodi sono funzioni che possono essere eseguite da tutti gli oggetti di quella classe. Per dire ad un oggetto di una classe di eseguire un suo metodo, devo scrivere il nome dell'oggetto seguito dal punto, dal nome del metodo e da una coppia di parentesi contenenti, eventualmente, gli argomenti necessari. In pratica se `tina` è un oggetto della classe `Turtle` l'istruzione:

```
tina.forward(97)
```

chiede all'*oggetto* collegato al nome `tina` di eseguire il suo *metodo* `forward` e `97` è l'*argomento* passato a questo metodo.

Quindi l'istruzione precedente comanda alla tartaruga `tina` di avanzare di 97 passi.

```
tp.reset()
```

comanda a `tp` di ripulire tutto il piano e di riportare la situazione allo stato iniziale. Perché ogni metodo deve essere preceduto dal nome dell'oggetto? Non sarebbe più semplice scrivere solo: `forward(97)` o `reset()`? Il fatto è che possiamo creare quanti oggetti vogliamo della classe Turtle quindi quando diamo un comando, dobbiamo specificare a quale oggetto quel comando è diretto:

```
tp = TurtlePlane()
tina = Turtle()
pina = Turtle()
gina = Turtle()
pina.left(120)
gina.left(240)
tina.forward(50)
pina.forward(100)
gina.forward(200)
```

In questo caso vengono create tre tartarughe, vengono sfasate di 120 gradi l'una dall'altra e infine vengono fatte avanzare di tre lunghezze diverse.

9.1.2 Attributi

Ogni *tartaruga* ha diverse caratteristiche proprie, sono i suoi **attributi**: `color`, `width`, `position`, `direction`, ...

Gli attributi definiscono lo stato di un oggetto. Ad esempio, per cambiare il colore della tartaruga e della sua penna si deve modificare il suo attributo `color`: `color = <colore>` dove `<colore>` è una stringa che contiene il nome di un colore, o `color = (<rosso>, <verde>, <blu>)` dove `<rosso>`, `<verde>`, `<blu>` sono tre numeri decimali compresi tra 0 e 1:

```
tina.color = "purple"
```

o

```
tina.color = (0.4, 0.4, 0.4) # grigio
```

Viceversa se voglio memorizzare in una variabile l'attuale colore di una tartaruga potrò assegnare ad una variabile il valore di `color`:

```
colore_attuale_di_tina = tina.color
```

Nota: per l'elenco completo di attributi e metodi si vedano i capitoli dedicati alle singole librerie.

9.2 Nuove classi

Una caratteristica importante delle classi è l'ereditarietà. Una classe può venir derivata da un'altra classe essere cioè figlia di un'altra classe; la classe figlia eredita dalla classe genitrice tutte gli attributi e i metodi e può:

- aggiungere altri attributi,
- aggiungere altri metodi,
- modificare i metodi della classe genitrice.

Buona parte della programmazione OOP consiste nel progettare e realizzare classi e gerarchie di classi di oggetti. Realizzare una nuova classe può essere un lavoro molto complicato ma potrebbe essere anche molto semplice quando la classe che realizziamo estende qualche altra classe già funzionante.

Come esercizio proviamo a costruire la classe di una tartaruga che sappia costruire un muro. Come al solito, prima di mettere mano a grandi opere, iniziamo a lavorare ad un problema abbastanza semplice e conosciuto. Voglio avere una tartaruga che oltre a saper fare tutto quello che sanno fare le altre tartarughe sappia anche disegnare mattoni quadrati: la nuova tartaruga deve quindi estendere le capacità di `Turtle`. La sintassi che `Python` mette a disposizione per estendere la gerarchia di una classe è:

```
class <nome di una nuova classe>(<nome di una classe esistente>):
```

Per iniziare gli esperimenti avviamo `IDLE`, aprimao un nuovo file: `menu-File-New File` esalviamo questo file nella cartella dove mettiamo tutti i nostri lavori: `menu-File-Save As`.

Come sempre scriviamo un po" di informazioni relative al programma e la *docstring* con una sintetica descrizione di cosa deve fare il programma stesso.

Poi:

- carichiamo la libreria `pyturtle`;
- definiamo una nuova classe derivata dalla classe `tg.Turtle`, una classe che, per ora, non fa niente (istruzione `pass`);
- infine scriviamo il programma principale che:
  - crea un piano,
  - crea un Ingegnere,
  - cambia qualche attributo a questo ingegnere,
  - gli chiede di tracciare una linea in avanti di 100 passi,
  - rende attiva la finestra grafica.

```
# man2.py
# 3 gennaio 2018
# Daniele Zambelli

"""
Il programma deve creare una classe di oggetti
capaci di disegnare un muro di mattoni quadrati.
"""

# Lettura delle librerie
```

```
import pygraph.pyturtle as tg

# Definizione di classi
class Ingegnere(tg.Turtle):
    """Una tartaruga che sa costruire muri."""
    pass

# Programma principale
piano = tg.TurtlePlane()
leonardo = Ingegnere()
leonardo.color = 'red'
leonardo.width = 6
leonardo.forward(100)

# Rende attiva la finestra grafica
piano.mainloop()
```

Corretti eventuali errori di battitura possiamo osservare che la classe `Ingegnere` derivata dalla classe `tg.Turtle` e che contiene solo l'istruzione `pass`, si comporta esattamente come una tartaruga. Infatti `leonardo` è un oggetto della classe `Ingegnere` e `Ingegnere` è un discendente di `tg.Turtle` e quindi, già alla nascita, sa fare tutto quello che sa fare `tg.Turtle`.

Ora estendiamo le capacità di `Ingegnere` in modo che sappia disegnare mattoni quadrati.

Modifichiamo il programma sostituendo `pass` con la definizione del metodo `quadrato` e modificando anche il programma principale:

```
# Definizione di classi
class Ingegnere(tg.Turtle):
    """Una tartaruga che sa costruire muri."""
    def quadrato(lato):
        """Disegna un quadrato di dato lato."""
        for i in range(4):
            forward(lato)
            left(90)

# Programma principale
piano = tg.TurtlePlane()
leonardo = Ingegnere()
leonardo.quadrato(80)
```

ed eseguiamo il programma:

```
Traceback (most recent call last):
  File "/dati/.../prove/man2.py", line 25, in <module>
  leonardo.quadrato(80)
TypeError: quadrato() takes 1 positional argument but 2 were given
```

Accidenti, non va!!! E non solo non funziona, ma ci dà un errore decisamente assurdo: `Python` si lamenta che quadrato vuole un argomento e noi gliene avremmo passati due!? È strabico? Non sa contare?? È stupido??? Boh, mah, forse... Chi ha programmato questo linguaggio ha deciso che l'oggetto che deve eseguire un metodo viene passato come primo parametro del metodo stesso.

Nota: Noi scriviamo:

```
leonardo.quadrato(80)
```

in realtà viene eseguito:

```
Ingegnere.quadrato(leonardo, 80)
```

provare per credere.

Quindi se `quadrato` è un metodo di una classe deve avere un primo parametro dentro il quale viene messo il riferimento all'oggetto che deve eseguire il metodo stesso. Per convenzione questo parametro viene chiamato `self` e noi seguiamo questa convenzione. Modifichiamo il metodo `quadrato` mettendo `self` come primo parametro:

```
class Ingegnere(Turtle):
    """Una tartaruga che sa costruire muri."""
    def quadrato(self, lato):
        """Disegna un quadrato di dato lato."""
        for i in range(4):
            forward(lato)
            left(90)
```

Ora `quadrato` ha i due parametri: uno per contenere l'oggetto che deve eseguire il metodo e uno per contenere la lunghezza del lato. Fatti questi cambiamenti eseguiamo il programma ottenendo:

```
Traceback (most recent call last):
File
  File "/dati/.../prove/man2.py", in <module>
    leonardo.quadrato(80)
  File "/dati/.../prove/man2.py", line 18, in quadrato
    forward(lato)
NameError: name 'forward' is not defined
```

Ancora qualcosa che non va... Eppure questa volta quadrato ha i due parametri richiesti! Infatti l'errore è cambiato: ci dice che non esiste un nome globale `forward`. Infatti `forward` è un metodo della classe `Turtle` e quindi può essere eseguito solo da un oggetto di questa classe. Ma dove lo trovo un oggetto della classe `Turtle` mentre sto definendo la mia nuova classe? Se osserviamo bene, la soluzione al precedente errore ce l'ha messo a disposizione, è proprio l'oggetto contenuto in `self`. Dobbiamo modificare la chiamata a `forward` scrivendo: `self.forward(lato)`. Terzo tentativo:

```
class Ingegnere(Turtle):
    """Una tartaruga che sa costruire muri."""
    def quadrato(self, lato):
        """Disegna un quadrato di dato lato."""
        for i in range(4):
            self.forward(lato)
            self.left(90)
```

Ovviamente quello che facciamo per `forward` lo dobbiamo fare anche per `left`. Modifichiamo l'*Ingegnere* e proviamo ancora il metodo ricorretto.

Va!!! Ora abbiamo una classe `Ingegnere` che oltre a saper fare tutto quello che sanno fare tutte le Tartarughe sa anche disegnare quadrati.

Ora possiamo prendere le altre funzioni del programma man1.py e trasformarle in metodi aggiungendo il parametro `self` dove serve modificando opportunamente il programma principale. Di seguito riporto l'intero programma

```
# man2.py
# 3 gennaio 2018
# Daniele Zambelli

"""
Il programma deve creare una classe di oggetti
capaci di disegnare un muro di mattoni quadrati.
```

```
"""

# Lettura delle librerie
import pygraph.pyturtle as tg

# Definizione di classi
class Ingegnere(tg.Turtle):
    """Una tartaruga che sa costruire muri."""

    def sposta(self, o=0, v=0):
        """Effettua uno spostamento orizzontale e verticale di
        Tartaruga senza disegnare la traccia."""
        self.up()
        self.forward(o); self.left(90)
        self.forward(v); self.right(90)
        self.down()

    def quadrato(self, lato):
        """Disegna un quadrato di dato lato."""
        for i in range(4):
            self.forward(lato)
            self.left(90)

    def muro(self, lato, righe, colonne,
             spazio_righe=5, spazio_colonne=5):
        """Disegna un muro di mattoni quadrati."""
        for i in range(righe):
            for j in range(colonne):
                self.quadrato(lato)
                self.sposta(o=lato+spazio_colonne)
            self.sposta(o=-colonne*(lato+spazio_colonne),
                        v=lato+spazio_righe)
        self.sposta(v=-righe*(lato+spazio_righe))

# Programma principale
piano = tg.TurtlePlane()
leonardo = Ingegnere()
leonardo.sposta(-250, -190)
leonardo.muro(20, 15, 20)

# Rende attiva la finestra grafica
piano.mainloop()
```

Salviamo, eseguiamo, . . . correggiamo gli errori che inevitabilmente sono stati fatti, . . . , rieseguiamo. . .

A parte la complicazione del parametro `self`, possiamo vedere come `Python` ci permetta di definire nuove classi in modo estremamente semplice. Utilizzando l'ereditarietà, cioè scrivendo classi derivate da altre già realizzate da altri, possiamo ottenere, con poche righe di programma classi:

- potenti, perché estendono altre classi;
- sicure, perché le classi genitrici sono utilizzate da molti altri programmatori ed eventuali errori sono sicuramente già stati trovati e corretti;
- ulteriormente estendibili, ma questo lo vedremo nel prossimo capitolo.

9.3 Riassumendo

- Per definire una classe si usa il comando `class <nome della classe>():`
- Per definire una classe discendente da un'altra si utilizza il comando:

  ```
  class <nome della classe figlia>(<nome della classe genitrice>):
  ```

- Quando si scrive una classe discendente da un'altra basta scrivere i metodi che si aggiungono ai metodi della classe genitrice o che li sostituiscono.
- Quando si definisce un metodo di una classe si deve mettere come primo parametro il nome di una variabile che conterrà l'oggetto stesso, di solito `self`.
- All'interno di una classe, il parametro `self` permette di richiamare i metodi e le proprietà dell'oggetto stesso.

CAPITOLO 10

Come modificare il comportamento di una classe

Dove rompiamo un po" gli schemi e costruiamo un metodo che oscura un metodo del genitore.

10.1 Estendere una classe: spostamento casuale

Nel capitolo precedente abbiamo realizzato un muro di quadrati. Tutti in ordine ben allineati, tutti uguali... Un po" troppo in ordine, un po" troppo uguali... Proviamo a mettere un po" di disordine nello schema. Invece che disegnare un quadrato con il primo lato in direzione della tartaruga, possiamo fare in maniera che il quadrato sia spostato casualmente. Sorge subito un problema: `Python` non ha un comando per ottenere dei valori casuali. Niente paura, c'è una libreria che ci fornisce la funzione adatta. La libreria è `random` e la funzione che ci interessa è `randrange(<numero>)` che restituisce un numero intero compreso tra zero incluso e `<numero>` escluso.

Possiamo ripensare la procedura quadrato in questo modo:

```
definisco quadrato(lato) così:
  metto nella variabile angolo un numero casuale tra 0 e 30
  metto nella variabile spostamento un numero casuale tra 0 e 30
  ruoto tartaruga di angolo e la sposto di spostamento
  disegno il quadrato
  rimetto a posto tartaruga
```

Ora se utilizzassimo la programmazione classica dovremmo prendere il programma scritto precedentemente e modificare la procedura quadrato. La programmazione ad oggetti ci permette un meccanismo diverso:

- si crea una classe discendente della classe `Ingegnere`,
- si ridefinisce solo il metodo `quadrato(...)`,

La nuova classe così costruita possiede tutte le caratteristiche della vecchia classe ma con il metodo quadrato diverso

Per poter realizzare questa nuova classe è necessario importare la libreria `random` e il `man2.py` che contiene la classe `Ingegnere`. Proviamola:

```
# man3.py
# 3 gennaio 2018
# Daniele Zambelli

"""
Il programma deve creare una classe di oggetti
capaci di disegnare un muro di mattoni quadrati
Disposti in posizione parzialmente casuale.
"""

# Lettura delle librerie
import pygraph.pyturtle as tg
import man2
import random

# Definizione di classi
class Architetto(man2.Ingegnere):
    """Una tartaruga che sa costruire muri scombinati."""

    def quadrato(self, lato):
        """Disegna un mattone spostato rispetto alla posizione
        attuale di Tartaruga."""
        angolo = random.randrange(30)
        spostamento = random.randrange(30)
        self.up()
        self.right(angolo)
        self.forward(spostamento)
        self.down()
        man2.Ingegnere.quadrato(self, lato)
        self.up()
        self.back(spostamento); self.left(angolo)
        self.down()

# Programma principale
piano = tg.TurtlePlane()
michelangelo = Architetto()
michelangelo.sposta(-250, -190)
michelangelo.muro(20, 15, 20)

# Rende attiva la finestra grafica
piano.mainloop()
```

Funziona... e non funziona.

Funziona perché abbiamo ottenuto i quadrati scombinati, come volevamo. Ma perché in un'altra finestra vengono disegnati anche i quadrati perfettamente schierati? Il fatto è che quando viene letta la libreria `man2.py`, questa viene anche eseguita, quindi, in particolare, vengono eseguite le sue ultime tre righe che producono il disegno dei quadrati tutti diritti. `Python` mette a disposizione gli strumenti (semplici) per evitare questo meccanismo.

Apriamo il programma `man2.py` e cambiamo il programma principale nel seguente modo:

```
# Programma principale
if __name__ == '__main__':
    piano = tg.TurtlePlane()
    leonardo = Ingegnere()
    leonardo.sposta(-250, -190)
    leonardo.muro(20, 15, 20)
```

```
    # Rende attiva la finestra grafica
    piano.mainloop()
```

Che più o meno vuol dire:

> Solo se questo programma viene eseguito come programma principale esegui

le istruzioni del blocco che segue, atrimenti non fare niente.

L'effetto ottenuto pare abbastanza naturale, ma cosa avviene nell'interprete?

`michelangelo` è un oggetto della classe `Architetto`.

Il comando `michelangelo.sposta(-250, -180)` chiama il metodo `sposta` di `Ingegnere` il quale chiama il metodo `forward` di `Turtle`...

Il comando `michelangelo.muro(20, 15, 20)` chiama il metodo `muro` di `Ingegnere` e questo chiama il metodo `quadrato` di `Architetto` e il metodo `sposta` di `Ingegnere`.

Questo comportamento non è semplice per il computer, ma appare naturale per il programmatore. Questo meccanismo permette di estendere a piacere librerie senza doverle modificare. In questo modo si possono realizzare librerie stabili, solide perché condivise e provate da molti utilizzatori, ma adattabili alle proprie esigenze.

10.2 Estendere una classe: aggiungiamo i colori

E se mi fossi stancato del bianco e nero e volessi dei mattoni colorati? Probabilmente la cosa più semplice potrebbe essere quella di modificare il metodo `quadrato`, ma proviamo a utilizzare ancora l'ereditarietà. Chiudiamo questo programma e apriamo una nuova finestra di editor dove definiamo una nuova classe, chiamiamola `Artista`, discendente da `Architetto`.

Dato che vogliamo usare `man3.py` come libreria dobbiamo modificare anche il suo programma principale in questo modo:

```
if __name__ == '__main__':
    # Programma principale
    piano = tg.TurtlePlane()
    michelangelo = Architetto()
    michelangelo.sposta(-250, -190)
    michelangelo.muro(20, 15, 20)

    # Rende attiva la finestra grafica
    piano.mainloop()
```

Effettuata la modifica e assicuratici di non aver introdotto errori eseguendo il programma, chiudiamo questo file e apriamone un altro dove mettiamo le solite intestazioni e una docstring.

Prima di procedere dobbiamo spendere due parole su uno dei modi di gestione dei colori. L'attributo `color` di Tartaruga accetta diversi tipi di argomenti: Ci sono vari modi per definire il colore di una tartaruga:

```
<tartaruga>.color = <nome di un colore>
```

ad esempio:

```
tina.color = 'pink'
```

oppure:

```
<tartaruga>.color = (<red>, <green>, <blue>)
```

dove <red>, <green>, e <blue> sono dei numeri razionali compresi tra 0 e 1 che rappresentano l'intensità dei tre colori fondamentali rosso, verde e blu. Ad esempio:

```
tina.color = (0.5, 0, 0.5) # metà rosso e metà blu
```

Ora useremo quesrto secondo metodo. Per riempire di un colore una figura si usa il metodo `fill(0|1)`. Quando viene chiamato il metodo `fill(1)`, Tartaruga tiene nota delle parti da riempire, quando viene chiamato `fill(0)`, Tartaruga le riempie. Quindi, se si vuole colorare una figura, si deve chiamare il metodo `fill` con l'argomento uguale a `1` prima di iniziare a disegnarla e `fill` con l'argomento uguale a `0` alla fine. A questo punto la figura viene riempita con il colore attuale di Tartaruga.

Dovremo quindi creare una *classe* derivata da `Architetto` che ridefinisce il metodo `quadrato`. Questo metodo dovrà:

- scegliere un colore a caso;
- attivare il comando di riempimento;
- disegnare il quadrato;
- riempirlo con il colore scelto.

Ovviamente per la parte del disegno del quadrato potremo fare riferimento al metodo già definito in `Architetto`:

Dobbiamo quindi riscrivere il metodo `quadrato`:

```
# man4.py
# 3 gennaio 2018
# Daniele Zambelli

"""
Il programma deve creare una classe di oggetti
capaci di disegnare un muro di mattoni quadrati
Disposti in posizione parzialmente casuale e
con colori casuali.
"""

# Lettura delle librerie
import pygraph.pyturtle as tg
import man3
import random

# Definizione di classi
class Artista(man3.Architetto):
    """Una tartaruga che sa costruire muri scombinati e colorati."""

    def quadrato(self, lato):
        """Disegna un mattone scombinato e colorato."""
        self.color = (random.random(), random.random(), random.random())
        self.fill(1)
        man3.Architetto.quadrato(self, lato)
        self.fill(0)

if __name__ == '__main__':
    # Programma principale
    piano = tg.TurtlePlane()
    raffaello = Artista()
    raffaello.sposta(-250, -190)
    raffaello.muro(20, 15, 20)
```

```
    # Rende attiva la finestra grafica
    piano.mainloop()
```

Per vedere se hai capito bene questi meccanismi, prova a spegare a chi ai in fianco il significato del comando:

`random.random()`

e del comando:

`man3.Architetto.quadrato(self, lato)`

Non preoccuparti se non ci riesci, non sono concetti semplici e si chiariranno con l'uso.

Ma se ci riesci allora hai capito i concetti di base della Programmazione Orientata agli Oggetti (OOP)!

10.3 Riassumendo

- Una classe discendente di un'altra, può aggiungere dei metodi alla classe genitrice.
- Una classe discendente di un'altra, può anche modificarne il comportamento ridefinendo alcuni suoi metodi.
- Per ridefinire un metodo basta definirne uno con lo stesso nome.
- È l'interprete che si incarica di eseguire i metodi corretti tra tutti quelli che hanno lo stesso nome.
- Una classe può forzare l'interprete ad eseguire un metodo di una sua classe antenata anteponendo al nome del metodo il nome della classe. Ad esempio il metodo `quadrato` di `Artista` chiama il metodo `quadrato` della classe `Architetto` in questo modo:

    ```
    Architetto.quadrato(self, lato)
    ```

CAPITOLO 11

La ricorsione

Dove parliamo di procedure che si comportano come gatti che inseguono la loro coda

11.1 Definizioni ricorsive

Abbiamo visto che una funzione, una volta definita, può essere utilizzata al pari di ogni altro comando primitivo. È anche possibile che una funzione chiami sé stessa. In questo caso si dirà che la funzione è ricorsiva.

Nella matematica esistono diversi esempi di definizioni ricorsive ad esempio nel caso di potenze ad esponente naturale si può dire che:

$$a^n = \begin{cases} 1 & \text{se } n = 0 \\ a \cdot a^{n-1}, & \text{se } n > 0 \end{cases}$$

Un altro classico esempio di ricorsione è la ricerca di una parola in un dizionario:

```
per la ricerca di una parola:
  apro a caso e leggo una parola a caso,
  se è la parola cercata:
    ho finito,
  se la parola che cerco viene prima di quella letta:
    eseguo la ricerca della parola a sinistra di quella letta
  altrimenti:
    eseguo la ricerca della parola a destra di quella letta.
```

11.2 Condizione di terminazione

Vediamo un altro esempio di definizione ricorsiva:

```
la scalinata di Giacobbe è:
  Un gradino seguito da una scalinata di Giacobbe
```

Quanti gradini ha? . . .

E se volessimo una scalinata un po" meno impegnativa?

```
Una scalinata di enne gradini è:
  uno scalino seguito da una scalinata di enne-1 gradini.
```

Oppure in una forma direttamente traducibile in un linguaggio di programmazione:

```
una scalinata di enne gradini è:
  se enne è uguale a 0: (la scalinata è) finita
  (altrimenti è) un gradino seguito da
  una scalinata di enne-1 gradini
```

Quest'ultima funzione presenta nella prima riga la verifica di una condizione. È la *condizione di terminazione* ed è importantissima per rendere funzionante la procedura, senza di questa la procedura continua all'infinito, o meglio finché non termina lo spazio disponibile nella memoria RAM e termina quindi con un errore.

11.3 Ricorsione non terminale

Negli esempi precedenti la chiamata ricorsiva è l'ultima istruzione eseguita della funzione. Queste sono dette funzioni ricorsive *terminali* e possono essere tradotte facilmente in un ciclo.

Ci sono anche funzioni che prevedono dei comandi da eseguire dopo la chiamata ricorsiva, in questo caso funzioni anche molto brevi possono presentare un comportamento molto complesso. Vediamo due esempi di funzioni ricorsive non terminali. Per la prima realizziamo un'astrazione a partire da un elemento naturale: un albero.

Un albero è formato da un tronco seguito da alcuni rami. Se seghiamo un ramo e lo raddrizziamo possiamo notare che assomiglia molto ad un albero: ha un primo segmento, una specie di tronco, seguito da alcuni rami, ognuno di questi rami assomiglia a sua volta ad un albero. Potremmo dire che un albero è un tronco con sopra alcuni alberi. Decidiamo una condizione di terminazione: se l'albero è più piccolo di 2 allora è fatto. Decidiamo anche che nel nostro albero da tutte biforcazioni partono due rami uguali e che l'angolo tra questi due rami è di 90 gradi. La funzione in un linguaggio intermedio potrebbe essere:

```
un albero di una certa lunghezza è:
  se la lunghezza è minore di 2: finito... altrimenti:
  disegno il tronco con quella lunghezza,
  giro a sinistra di 45 gradi
  disegno un albero di metà lunghezza,
  giro a destra di 90 gradi
  disegno un albero di metà lunghezza,
  rimetto a posto tartaruga
```

Da notare che l'ultima riga è fondamentale: alla fine del disegno di un albero (di ogni sottoalbero) tartaruga deve essere riportata nella posizione iniziale.

La traduzione in Python diventa:

```
import pygraph.pyturtle as tg

def albero(lung):
    if lung < 2: return    # condizione di terminazione
    tina.forward(lung)
    tina.left(45)
    albero(lung/2)         # prima chiamata ricorsiva
    tina.right(90)
    albero(lung/2)         # seconda chiamata ricorsiva
```

```
    tina.left(45)             # tartaruga
    tina.back(lung)           # è rimessa a posto

piano = tg.TurtlePlane()
tina = tg.Turtle()
tina.left(90)
tina.up()
tina.back(100)
tina.down()
albero(100)
piano.mainloop()
```

Ogni albero ha un caratteristico angolo tra i rami, possiamo aggiungere questo parametro alla nostra funzione:

```
import pygraph.pyturtle as tg

def alberobin(lung, angolo):
    if lung < 2: return
    tina.forward(lung)
    tina.left(angolo)
    alberobin(lung/2, angolo)
    tina.right(2*angolo)
    alberobin(lung/2, angolo)
    tina.left(angolo)
    tina.back(lung)

piano = tg.TurtlePlane()
tina = tg.Turtle()
tina.left(90)
tina.up(); t.back(100); t.down()
albero(100, 60)
piano.mainloop()
```

E possiamo così disegnare alberi di specie diverse. Un albero ha anche la caratteristica di avere lo spessore dei rami che diminuisce man mano che ci si allontana dalle radici. Possiamo modificare l'attributo `width` per rendere più *realistici* i nostri alberi:

```
import pygraph.pyturtle as tg

def alberobin(lung, angolo):
    if lung < 2: return
    tina.setwidth = lung/5
    tina.forward(lung)
    tina.left(angolo)
    alberobin(lung/2, angolo)
    tina.right(2*angolo)
    alberobin(lung/2, angolo)
    tina.left(angolo)
    tina.back(lung)

piano = tg.TurtlePlane()
tina = tg.Turtle()
tina.left(90)
tina.up()
tina.back(100)
tina.down()
alberobin(100, 60)
piano.mainloop()
```

11.4 Diversi alberi

Nello scrivere la funzione albero sono state fatte molte scelte che rendono piuttosto rigida la funzione stessa, possiamo parametrizzare diversi elementi:

- il rapporto tra lunghezza e larghezza dei rami,
- il decremento del sottoalbero di sinistra,
- che può essere diverso dal decremento del sottoalbero di destra.

Ovviamente non basta aggiungere i parametri nell'intestazione della funzione, ma bisogna anche che tutte le chiamate ricorsive abbiano il giusto numero di parametri:

```
import pygraph.pyturtle as tg

def alberobin(lung, angolo, larg, decsx, decdx):
    if lung < 2: return
    tina.width = lung*larg
    tina.forward(lung)
    tina.left(angolo)
    alberobin(lung*decsx, angolo, larg, decsx, decdx)
    tina.right(2*angolo)
    alberobin(lung*decdx, angolo, larg, decsx, decdx)
    tina.left(angolo)
    tina.back(lung)

piano = tg.TurtlePlane()
tina = tg.Turtle()
tina.left(90)
tina.up()
tina.back(100)
tina.down()
alberobin(100, 60, 0.1, 0.8, 0.6)
piano.mainloop()
```

Modificando i parametri possiamo disegnare alberi di un gran numero di specie diverse... I disegni ottenuti risultano un po" innaturali. Nel mondo reale molte cause provocano delle differenze di accrescimento dei vari rami, possiamo aggiungere un elemento di casualità alla crescita dei vari rami, una variazione casuale rispetto alla lunghezza passata come parametro.

```
import pygraph.pyturtle as tg
import random

def alberobincas(lung, angolo, larg, decsx, decdx, caos):
    if lung < 2: return
    var = int(lung*caos)+1
    l = lung - var + random.randrange(2*var)
    tina.width = lung*larg
    tina.forward(lung)
    tina.left(angolo)
    alberobincas(l*decsx, angolo, larg, decsx, decdx, caos)
    tina.right(2*angolo)
    alberobincas(l*decdx, angolo, larg, decsx, decdx, caos)
    tina.left(angolo)
```

```
    tina.back(lung)

piano = tg.TurtlePlane()
tina = tg.Turtle()
tina.left(90)
tina.up()
tina.back(100)
tina.down()
alberobincas(100, 60, 0.1, 0.8, 0.6, 0.3)
piano.mainloop()
```

Degli altri alberi possono essere realizzati a partire da alberi ternari:

```
def alberoter(lung, angolo):
    if lung < 2: return
    tina.width = lung/5
    tina.forward(lung)
    tina.left(angolo)
    alberoter(lung/2, angolo)
    tina.right(angolo)
    alberoter(lung/2, angolo)
    tina.right(angolo)
    alberoter(lung/2, angolo)
    tina.left(angolo)
    tina.back(lung)
```

L'approfondimento è lasciato a chi è interessato.

11.5 Il fiocco di neve

Negli alberi la funzione è trasparente, non modifica la posizione di Tartaruga: Tartaruga viene riportata dove era stata presa. C'è un'altra famiglia di linee ricorsive che si comportano come segmenti: prendono Tartaruga in un posto e la spostano in un altro. Un famoso esempio di queste linee è la curva di Koch:

```
livello 0: _______________

                  /\
livello 1:       /  \
           ____/    \____

                __/\__
livello 2:      \    /
           _/\_/    \_/\_
```

A livello zero la curva di Koch è un segmento, ad ogni aumento di livello ogni segmento è sostituito da un segmento con la stessa lunghezza e con una protuberanza al centro.

```
import pygraph.pyturtle as tg

def koch(lung, liv):
    if liv == 0: t.forward(lung); return
    koch(lung / 3.0, liv - 1)
    tina.left(60)
    koch(lung / 3.0, liv - 1)
    tina.right(120)
```

```
    koch(lung / 3.0, liv - 1)
    tina.left(60)
    koch(lung / 3.0, liv - 1)

fron pyturtle import *

piano = tg.TurtlePlane()
tina = tg.Turtle()
tina.up()
tina.back(100)
tina.down()
koch(200, 4)
piano.mainloop()
```

Comportandosi come un segmento, possiamo utilizzare koch per realizzare un poligono, un triangolo con lato frattale:

```
...

def fiocco(lato, liv):
    for i in range(3):
        koch(lato, liv)
        tina.right(120)

...

fiocco(200, 4)
```

Molto regolare, ma se volessi introdurre un elemento di casualità? Ad esempio potremmo produrre la protuberanza a destra o a sinistra rispetto Tartaruga. E con questo realizzare un fiocco casuale.

```
import pygraph.pyturtle as tg
import random

def kochcas(lung, liv):
    if liv == 0:
        tina.forward(lung)
        return
    verso = random.randrange(-1, 2, 2)     # dà come risultato -1 o +1
    kochcas(lung / 3.0, liv - 1)
    tina.left(60 * verso)
    koch(lung / 3.0, liv - 1)
    tina.right(120 * verso)
    kochcas(lung / 3.0, liv - 1)
    tina.left(60 * verso)
    koch(lung / 3.0, liv - 1)

def fioccocas(lato, liv):
    for i in range(3):
        kochcas(lato, liv)
        tina.right(120)

piano = tg.TurtlePlane()
tina = tg.Turtle()
tina.up()
tina.back(100)
tina.down()
fioccocas(200, 4)
piano.mainloop()
```

11.6 Riassumendo

- Una funzione si dice ricorsiva quando chiama sé stessa.
- Una funzione ricorsiva, perché sia utilizzabile all'interno di un programma, deve avere una condizione di terminazione, di solito posta all'inizio.
- Una funzione ricorsiva si dice terminale se la chiamata ricorsiva non è seguita da altre istruzioni.
- Le funzioni ricorsive terminali possono essere sostituite facilmente da cicli.
- Le funzioni ricorsive non terminali hanno un comportamento che può apparire sorprendente.
- Le funzioni ricorsive avvicinano i linguaggi procedurali a quelli dichiarativi. Infatti la domanda che si pone il programmatore per realizzare una funzione ricorsiva non è: "come si fa a fare. . . ?", ma: "cosa è . . . ?".

CAPITOLO 12

Pycart

Dove viene presentata la libreria Pycart.

12.1 Introduzione

`Pycart` è una libreria che implementa la classe *Piano cartesiano* con alcuni metodi che permettono di:

- Modificare alcuni parametri come la posizione dell'origine e la scala di visualizzazione.
- Effettuare le trasformazioni di coordinate necessarie.
- Disegnare gli assi.

`Pycart` si appoggia su alcune altre librerie:

- `Tkinter` per l'output grafico,
- `os` per eseguire un programma di conversione del formato grafico nel metodo `save(self, filename)`,
- `colors` per la gestione dei colori,
- `pygrapherror` per i messaggi di errore

Nella libreria sono presenti:

- la funzione `version`, che riporta la versione;
- la classe `Plane`, che implementa un piano cartesiano;
- la classe `Pen`, che implementa un tracciatore grafico.

Di seguito vengono presentati questi tre oggetti. Per le due classi verranno descritti gli attributi e i metodi e per ognuno verrà indicato:

- lo scopo
- la sintassi
- alcune osservazioni

- un esempio di uso

12.2 version

Scopo

È una funzione che restituisce il numero di versione della libreria.

Sintassi

```
version()
```

Osservazioni

Non richiede parametri, restituisce la stringa che contiene la versione.

Esempio

Controllare la versione della libreria.

```
import pygraph.pycart as cg
if cg.version()<"2.9.00":
    print "versione un po' vecchiotta"
else:
    print "versione:", cg.version()
```

12.3 class Plane

`Plane` contiene alcuni *attributi* e *metodi* che permettono di:

- cambiare la posizione dell'origine rispetto alla finestra grafica,
- cambiare la scala di rappresentazione,
- tracciare gli assi o una griglia di punti,
- salvare il grafico in un file,
- ...

Nota: Per poter creare oggetti della classi `Plane` bisogna importarla dalla libreria. In tutti gli esempi seguenti si suppone che sia già stato eseguito il comando:

```
import pygraph.pycart as cg
```

Se nell'esegure un esempio ottenete un messaggio che termina con: `NameError: name 'cp' is not defined`, forse avete dimenticato di caricare la libreria `pycart` assegnandole il nome `cg` con il comando scritto sopra.

Se nell'eseguire gli esempi osservate dei comportamenti strani della finestra che contiene il piano cartesiano, portate pazienza, o guardate quanto scritto nella descrizione del metodo `mainloop` di `Plane` più avanti dove ho cercato di dare delle indicazioni in proposito.

Nota: Chi è curioso e va a guardare il sorgente della libreria, scoprirà che ci sono alcuni *attributi* e *metodi* che non vengono descritti in questo manuale. Il loro nome inizia con il carattere «_» e sono *privati*. L'utente non dovrebbe aver bisogno di usarli e se lo fa, lo fa a suo rischio e pericolo perché potrebbero venir modificati nelle prossime versioni.

12.3.1 origin

Scopo

Permette di definire la posizione dell'origine degli assi all'interno della finestra grafica.

Sintassi

```
<piano>.origin = <coppia di numeri>
```

Osservazioni

Il valore predefinito è al centro della finestra. Gli argomenti indicano lo spostamento in pixel rispetto all'angolo in alto a sinistra della finestra grafica.

Esempio

Crea un piano, sposta l'origine a sinistra, disegna gli assi.

```
piano = cg.Plane('ex_01')
piano.origin = (10, 300)
piano.axes()
```

12.3.2 scale

Scopo

Definisce la scala di rappresentazione sui due assi.

Sintassi

```
<piano>.scale = <coppia di numeri>
```

Osservazioni

I due valori indicano rispettivamente la scala relativa all'asse x e all'asse y. I due valori sono numeri naturali che indicano il numero di pixel corrispondenti all'unità di misura.

Esempio

Crea un piano, modifica la scala e l'origine e colore poi ridisegna assi e griglia.

```
piano = cg.Plane('modifiche', sx=15,
            axescolor='green', color='AntiqueWhite3')
piano.after(500)
x_o, y_o = piano.origin
sx, sy = piano.scale
piano.origin = (x_o-4*sx, y_o+6*sy)
piano.scale = (30, 20)
prevcolor = piano.color
piano.color='yellow'
piano.axes(color='blue')
piano.grid(color='red')
```

```
piano.after(1000)
piano.color = prevcolor
```

12.3.3 __init__

Scopo

Crea il piano cartesiano e inizializza gli attributi di `Plane`.

Sintassi

```
<nome_variabile> = cg.Plane(<parametri>)
```

Osservazioni

Questo metodo non viene chiamato esplicitamente, ma viene eseguito quando si crea un oggetto di questa classe. L'intestazione di questo metodo è:

```
def __init__(self, name="Cartesian Plane",
             w=600, h=400,
             sx=20, sy=None,
             ox=None, oy=None,
             axes=True, grid=True,
             axescolor='black', gridcolor=None,
             color='white',
             parent=None):
```

Si può vedere che presenta molti parametri tutti con un valore predefinito. Nel momento in cui si crea un piano cartesiano si possono quindi decidere le sue caratteristiche. Vediamole in dettaglio:

- titolo della finestra, valore predefinito: «Cartesian Plane»;
- dimensione, valori predefiniti: larghezza=600, altezza=400;
- scala di rappresentazione, valori predefiniti: una unità = 20 pixel;
- posizione dell'origine, valore predefinito: il centro della finestra;
- rappresentazione degli assi cartesiani, valore predefinito: `True`;
- rappresentazione di una griglia di punti, valore predefinito: `True`;
- colore degli assi valore predefinito: "black".
- colore della griglia valore predefinito: lo stesso degli assi.
- colore dello sfondo valore predefinito: "white".
- riferimento alla finestra che contiene il piano cartesiano, valore predefinito: `None`.

Poiché tutti i parametri hanno un valore predefinito, possiamo creare un oggetto della classe `Plane` senza specificare alcun argomento: verranno usati tutti i valori predefiniti. Oppure possiamo specificare per nome gli argomenti che vogliamo siano diversi dal comportamento predefinito, si vedano di seguito alcuni esempi.

Esempio

Creare 3 piani cartesiani, il primo: con tutti i valori di default, il secondo: quadrato, con un titolo e con gli assi, il terzo: con un titolo, con le dimensioni di 400 per 200, con la scala e l'origine cambiati, con assi, griglia e sfondo colorati.

```
p0 = cg.Plane()
p1 = cg.Plane(name="Secondo piano", w=400, h=400, axes=True)
p2 = cg.Plane(name="Terzo piano",
```

```
            w=400, h=200,
            sx=10, sy=30,
            ox=40, oy=None,
            axescolor='orange', gridcolor='red', color='green')
```

12.3.4 mainloop

Scopo

Rende attiva la finestra grafica.

Sintassi

```
<piano>.mainloop()
```

Osservazioni

Questa istruzione è fondamentale nella geometria interattiva, mentre nei programmi con la geometria cartesiana e della tartaruga, può anche non essere usata.

Nota: Se la libreria è usata dall'interno di `IDLE`, possono sorgere dei problemi a seconda di come è stato avviato `IDLE` stesso.

- Se `IDLE` è stato avviato **con** il parametro «-n» allora **non** si deve usare il metodo `mainloop()` altrimenti, alla chiusura del piano cartesiano `IDLE` non risponde più ai comandi.
- Se `IDLE` è stato avviato **senza** il parametro «-n» allora si deve usare il metodo `mainloop()` altrimenti la finestra con il piano cartesiano non è attiva.

Nel resto degli esempi di questo capitolo non riporto questa istruzione, ma se la finestra creata non risponde al mouse aggiungetela come ultima istruzione.

Esempio

Disegna un piano se `IDLE` è stato avviato senza sottoprocessi, con il parametro «-n».

```
piano = cg.Plane('ex_051:  un piano', axescolor='pink')
```

Disegna un piano se `IDLE` è stato avviato con sottoprocessi, senza il parametro «-n».

```
piano = cg.Plane('ex_052:  altro piano', axescolor='olive drab')
piano.mainloop()
```

12.3.5 newPen

Scopo

Crea un nuovo tracciatore grafico, una nuova *penna* per disegnare figure geometriche elementari.

Sintassi

```
<piano>.newPen([x=<num>, y=<num>, color=<colore>, width=<num>])
```

Osservazioni

Normalmente è utile assegare la *penna* creata ad un identificatore.

L'intestazione di questo metodo è:

```
def newPen(self, x=0, y=0, color='black', width=1)
```

Tutti i parametri hanno un valore predefinito, per creare una penna nell'origine degli assi di colore nero e larghezza 1 non occorre passare argomenti alla funzione:

```
penna = piano.newPen()
```

Per maggiori informazioni sulle *penne* vedi la documentazione della classe `Pen`.

Esempio

Disegna un quadrato blu con il contorno spesso 4 pixel.

```
piano = cg.Plane('ex_11: Quadrato', w=400, h=400)
biro = piano.newPen(width=4, color='blue')
biro.drawpoly(((-7, -7), (7, -7), (7, 7), (-7, 7)))
piano.mainloop()
```

12.3.6 after

Scopo

Permette di inserire un ritardo in un programma.

Sintassi

```
<piano>.after(<numero>)
```

Osservazioni

Il parametro rappresenta il numero di millisecondi di attesa.

Esempio

Un quadrato in movimento, (Vedi gli esempi di `clear`, `reset`, `delete`).

```
piano = cg.Plane('ex_06: Quadrato in movimento',
             sx=1, sy=1, axes=False, grid=False)
biro = piano.newPen(width=4)
for i in range(0, 500, 2):
    color = '#ff{:02x}00'.format(i//2)
    verts = ((-300+i, -30), (-220+i, -50), (-200+i, 30), (-280+i, 50))
    id = biro.drawpoly(verts, color)
    piano.after(10)
    piano.delete(id)
id = biro.drawpoly(verts, 'green')
piano.mainloop()
```

12.3.7 axes

Scopo

Disegna una coppia di assi cartesiani ortogonali.

Sintassi

```
<piano>.axes(color=None)
```

Osservazioni

Se non è specificato il parametro, gli assi vengono disegnati con il colore che ha il cursore grafico in quel momento, altrimenti con il colore specificato. L'intestazione di questo metodo è:

```
def axes(self, color=None, ax=True, ay=True)
```

Come si vede dai parametri è anche possibile disegnare solo l'asse *x* o l'asse *y*.

Esempio

Disegna due sistemi di riferimento con uguale scala ma origini diverse.

```
piano = cg.Plane('ex_07: Sistema di riferimento traslato')
piano.origin = (80, 60)
piano.axes('red')
piano.mainloop()
```

12.3.8 grid

Scopo

Disegna una griglia di punti.

Sintassi

```
<piano>.grid(color=None)
```

Osservazioni

Se non è specificato il parametro, i punti vengono disegnati con il colore predefinito o con il colore specificato come parametro.

Esempio

Disegna in sequenza un piano con assi e griglia, senza assi, senza griglia e completamente bianco.

```
piano = cg.Plane('ex_08: Assi e griglie',  sx=12,  sy=36,
             axescolor='green', gridcolor='red')
biro = cg.Pen(x=-10, y=3, color='red', width=2)
biro.drawtext('assi e griglia, attendere... 4')
ritardo = 1000
piano.after(ritardo)
piano.clean()
piano.axes('purple')
biro.drawtext('solo assi, attendere... 3')
piano.after(ritardo)
piano.clean()
piano.grid('#22aa55')
biro.drawtext('solo griglia, attendere... 2')
piano.after(ritardo)
piano.reset()
biro.drawtext('situazione iniziale, attendere... 1')
piano.after(ritardo)
piano.clean()
biro.drawtext('bianco, finito!')
piano.mainloop()
```

12.3.9 clean

Scopo

Cancella tutti gli elementi disegnati nella finestra grafica, ridisegnando, se presenti, assi e griglia, ma non modifica i puntatori grafici, le *penne*.

Sintassi

```
<piano>.clean()
```

Osservazioni

Non ha parametri.

Esempio

Disegna un quadrato che scoppia.

```
piano = cg.Plane('ex_13: Quadrato che scoppia', sx=1,
             axes=False, grid=False)
biro = piano.newPen(width=4, color='blue')
a = 3
b = 1
for i in range (162):
    verts = (((-a*i, -b*i), (b*i, -a*i), (a*i,  b*i), (-b*i, a*i)))
    biro.drawpoly(verts)
    piano.after(10)
    piano.clean()
    biro.drawpoly(verts)
piano.mainloop()
```

12.3.10 reset

Scopo

Cancella tutti gli elementi disegnati nella finestra grafica, ridisegnando, se presenti, assi e griglia, riporta nella condizione iniziale i puntatori grafici, le *penne*.

Sintassi

```
<plot>.reset()
```

Osservazioni

Non ha parametri.

Esempio

Disegna gli assi con la griglia, un poligono verde, attendere un po", poi cancellare tutto ridisegnando gli assi e ririsegna il poligono con un altro colore.:

```
piano = cg.Plane('ex_14: Poligono', axes=False, grid=False)
piano.axes()
vertici = ((-2, -3), (4, -1), (6, 4), (-5, 6))
biro = cg.Pen(width=20, color='blue')
biro.drawpoly(vertici)
piano.after(500)
piano.reset()
biro.color="green"
biro.width=10
biro.drawpoly(vertici)
piano.mainloop()
```

12.3.11 delete

Scopo

Permette di cancellare un elemento grafico disegnato.

Sintassi

```
<piano>.delete(<id>)
```

Osservazioni

Ogni elemento grafico che viene disegnato nel piano ha un suo numero identificativo. Facendo riferimento a questo numero lo si può cancellare.

Esempio

Disegna un segmento che scivola sugli assi.

```
piano = cg.Plane('ex_15: Segmento che scivola sugli assi', sx=1)
biro = cg.Pen(width=5, color='navy')
lung = 200
for y in range(200, 0, -1):
    x = (lung*lung-y*y)**.5
    s = biro.drawsegment((0, y), (x, 0))
    piano.after(4)
    piano.delete(s)
biro.drawsegment((0, 0), (lung, 0))
piano.mainloop()
```

12.3.12 save

Scopo

Salva in un file l'immagine della finestra grafica.

Sintassi

```
<piano>.save(<filename>)
```

Osservazioni

<filename> è una stringa che contiene il nome del file. L'mmagine è salvata nel formato Postscript o "png" e a <nomefile> viene aggiunta l'estensione «.ps» o «.png» a seconda se pygraph trova il programma per la conversione in «png».

Esempio

Produrre un file che contiene il disegno di un quadrato.

```
piano = cg.Plane('ex_16: Quadrato')
biro = cg.Pen(width=6, color='pink')
q = ((-5, -3),(3, -5),(5, 3),(-3, 5))
biro.drawpoly(q)
piano.save('quadrato')
piano.mainloop()
```

12.3.13 getcanvaswidth e getcanvasheight

Scopo

Restituiscono le dimensioni in pixel della finestra grafica.

Sintassi

```
<piano>.getcanvaswidth()
<piano>.getcanvasheight()
```

Osservazioni

Restituiscono un numero intero.

Esempio

Disegna un rettangolo che circonda la finestra grafica.

```
piano = cg.Plane('ex_17: Cornice', sx=1, axes=False, grid=False)
biro = piano.newPen(width=4, color='green')
bordo = 10
w = piano.getcanvaswidth()//2-bordo
h = piano.getcanvasheight()//2-bordo
biro.drawpoly(((-w, -h), (w, -h), (w, h), (-w, h)))
piano.mainloop()
```

12.3.14 getcanvas

Scopo

Restituisce un riferimento alla finestra grafica.

Sintassi

```
<piano>.getcanvas()
```

Osservazioni

Il riferimento restituito può essere utilizzato per operare direttamente sulla finestra grafica senza passare attraverso i metodi messi a disposizione da Plane. Per saper cosa farsene bisogna conoscere la classe Canvas della libreria Tkinter.

Esempio

Questo metodo prevede la conoscenza della libreria Tkinter, essendo ciò al di fuori della portata di questo manuale, non viene proposto nessun esempio.

Si può trovare un suo uso nel programma `viewfun.py` distribuito assieme alla libreria pygraph.

12.4 onkeypress, onpress1, onrelease1, ...

Scopo

Sono delle funzioni che vengono chiamate quando si presentano certi eventi:

- `onkeypress`: è stato premuto un tasto;
- `onkeyrelease`: è stato rilasciato un tasto;
- `onenter`: il mouse è entrato nel piano;
- `onleave`: il mouse è uscito dal piano;
- `onpress1`: è stato premuto il tasto sinistro del mouse nel piano;
- `onpress2`: è stato premuto il tasto centrale del mouse nel piano;
- `onpress3`: è stato premuto il tasto destro del mouse nel piano;
- `onmotion1`: è stato mosso il mouse con il tasto sinistro premuto;

- `onmotion2`: è stato mosso il mouse con il tasto centrale premuto;
- `onmotion3`: è stato mosso il mouse con il tasto destro premuto;
- `onrelease1`: è stato rilasciato il tasto sinistro del mouse nel piano;
- `onrelease2`: è stato rilasciato il tasto centrale del mouse nel piano;
- `onrelease3`: è stato rilasciato il tasto destro del mouse nel piano;

Sintassi

```
onlclick = <miafunzione>
```

Osservazioni

La funzione ha un parametro che viene legato ad un oggetto evento.

Esempio

Stampa le azioni del mouse.

```
def onkeypress(event):
    print("Key pressed:", event.char)
def onkeyrelease(event):
    print("Key released:", event.char)
def onenter(event):
    print("Enter at", event)
def onleave(event):
    print("Leave at", event)
def onpress1(event):
    print("Left clicked at", event.x, event.y)
def onpress2(event):
    print("Center clicked at", event.x, event.y)
def onpress3(event):
    print("Right clicked at", event.x, event.y)
def onmotion1(event):
    print("Left motion at", event.x, event.y)
def onmotion2(event):
    print("Center motion at", event.x, event.y)
def onmotion3(event):
    print("Right motion at", event.x, event.y)
def onrelease1(event):
    print("Left released at", event.x, event.y)
def onrelease2(event):
    print("Center released at", event.x, event.y)
def onrelease3(event):
    print("Right released at", event.x, event.y)
piano = cg.Plane('04. events', color='pink')
piano.onkeypress(onkeypress)
piano.onkeyrelease(onkeyrelease)
piano.onenter(onenter)
piano.onleave(onleave)
piano.onpress1(onpress1)
piano.onpress2(onpress2)
piano.onpress3(onpress3)
piano.onmotion1(onmotion1)
piano.onmotion2(onmotion2)
piano.onmotion3(onmotion3)
piano.onrelease1(onrelease1)
piano.onrelease2(onrelease2)
piano.onrelease3(onrelease3)
```

```
biro = cg.Pen()
biro.drawtext('Clicca in vari punti del piano', (0, 9), 'navy', 2)
biro.drawtext('con i tre tasti del mouse', (0, 7), 'navy', 2)
piano.mainloop()
```

12.5 class Pen

La classe `Pen` contiene alcuni *attributi* e *metodi* che permettono di:

- Modificare le caratteristiche della penna.
- Modificare la posizione.
- Disegnare segmenti.
- Disegnare punti.
- Disegnare cerchi.
- Disegnare poligoni.
- Scrivere del testo.

12.5.1 __init__

Scopo

Crea una penna nel piano cartesiano e inizializza i suoi attributi.

Sintassi

```
<nome_variabile>= cg.Pen(<parametri>)
```

Osservazioni

L'intestazione di questo metodo è:

```
def __init__(self, x=0, y=0, color='black', width=1, plane=None)
```

che esplicita i valori predefiniti dei diversi attributi di una penna. Quando si crea una nuova penna, questa viene inserita didefault nell'ultimo piano creato. Nel creare una penna si possono anche specificare alcune caratteristiche, vediamole in dettaglio:

- la posizione del punto di partenza, valori predefiniti: x=0, y=0;
- il colore, valore predefinito: color="black";
- lo spessore del tratto, valore predefinito: width=1;
- il piano su cui scrive, valore predefinito: l'ultimo creato.

Dato che i parametri hanno valori predefiniti, possiamo creare penne passando un diverso numero di argomenti. Gli argomenti non specificati saranno sostituiti con i valori di default.

Questo metodo non viene chiamato esplicitamente, ma viene eseguito quando si crea un oggetto di questa classe.

Nota: vedi il metodo `newPen` di `Plane` per un altro modo per creare *penne*. Le due istruzioni:

```
piano = cg.Plane(sx=100, sy=100, axes=True)
p0= cg.Pen(x=1,  y=1,  color='violet', width=9)
```

e:

```
piano = cg.Plane(sx=100, sy=100, axes=True)
p0 = piano.newPen(x=1,  y=1,  color='violet', width=9)
```

sono del tutto equivalenti.

Esempio

Creare 4 penne diverse e le fa convergere nell'origine:

```
piano = cg.Plane('ex_30: __init__', sx=100, sy=100, grid=False)
p_0 = cg.Pen(x=+1, y=+1, color='violet', width=9)
p_1 = cg.Pen(x=-1, y=+1, color='magenta', width=7)
p_2 = cg.Pen(x=-1, y=-1, color='gold', width=5)
p_3 = cg.Pen(x=+1, y=-1, color='navy', width=3)
for biro in (p_0, p_1, p_2, p_3):
    biro.drawto((0, 0))
piano.mainloop()
```

La libreria `pygraph` permette di aprire più finestre grafiche contemporaneamente, l'oggetto `Pen` viene creato all'interno dell'ultimo piano creato. È possibile creare una penna in un piano qualsiasi o utilizzando il metodo `newPen` o specificando l'argomento `plane` nel costruttore della penna. Ad esempio se vogliamo quattro piani con una penna in ciascuno possiamo costruire la penna in vari modi:

```
piano_0 = cg.Plane('ex_31: piano 0 violet', sx=100, sy=100, grid=False)
biro0 = cg.Pen(x=+1, y=+1, color='violet', width=20)
piano_1 = cg.Plane('ex_31: piano 1 green', sx=100, sy=100, grid=False)
biro1 = cg.Pen(x=-1, y=+1, color='green', width=20)
piano_2 = cg.Plane('ex_31: piano 2 gold', sx=100, sy=100, grid=False)
piano_3 = cg.Plane('ex_31: piano 3 navy', sx=100, sy=100, grid=False)
biro2 = cg.Pen(x=-1, y=-1, color='gold', width=20, plane=piano_2)
biro3 = piano_3.newPen(x=+1, y=-1, color='navy', width=20)
for biro in (biro0, biro1, biro2, biro3):
    biro.drawpoint()
piano_0.mainloop()
```

`biro0` viene costruita nel piano `piano_0`, `biro1` nel piano `piano_1`, come è abbastanza logico aspettarsi; `biro2` verrebbe costruita nel piano `p3` se non fosse specificato l'argomento `plane = piano_2`; `biro3` verrebbe realizzata comunque nel piano `piano_3` dato che è l'ultimo piano creato, ma comunque l'istruzione `piano_3.newPen()` non lascia dubbi.

12.5.2 position

Scopo

È un attributo collegato alla posizione del cursore grafico. Permette di ottenere o modificare la posizione della penna.

Sintassi

```
<penna>.position = (<numero>, <numero>)
```

Osservazioni

I due numeri rappresentano le coordinate della posizione.

Esempio

Disegna due quadrati in posizioni casuali.

```
import random
piano = cg.Plane('ex_32: position')
biro = cg.Pen()
lato = random.randrange(5)+3
biro.position = (random.randrange(-14, 10), random.randrange(-9, 5))
x, y = biro.position
vertici = ((x, y), (x+lato, y), (x+lato, y+lato), (x, y+lato))
biro.drawpoly(vertici, color='gold', width=6)
lato = random.randrange(5)+3
biro.position = (random.randrange(-14, 10), random.randrange(-9, 5))
x, y = biro.position
vertici = ((x, y), (x+lato, y), (x+lato, y+lato), (x, y+lato))
biro.drawpoly(vertici, color='pink', width=6)
piano.mainloop()
```

12.5.3 color

Scopo

È un attributo collegato al colore del cursore grafico. Permette di tracciare disegni con linee di differenti colori.

Sintassi

```
<penna>.color = <colore>
```

Osservazioni

Il colore può essere indicato in diversi modi:

- Usando il nome di un colore (in inglese). I colori attualmente disponibili si trovano nel file `rgb.txt`.
- Usando una stringa nel formato "#RRGGBB" dove RR, GG, BB sono le numeri di due cifre scritti in forma esadecimale che rappresentano le componenti rossa, verde e blu del colore.
- Una tupla di tre numeri compresi tra 0 e 1 che rappresentano il livello delle componenti Rossa, Verde e Blu del colore.

Esempio

Disegna linee di diverso colore.

```
piano = cg.Plane('ex_33: color')
colors = ['red', 'green', 'blue', 'pink', 'yellow',
          'navy', 'gold', 'magenta', '#a0a0a0', (0.7, 0.5, 0.1)]
biro = cg.Pen(width=10)
for i, color in enumerate(colors):
    biro.color = color
    print(biro.color)
    biro.drawsegment((-5, i-4), (5, i-4))
piano.mainloop()
```

12.5.4 width

Scopo

È un attributo collegato allo spessore del cursore grafico. Permette di tracciare disegni con linee di differenti spessori.

Sintassi

```
<penna>.width = <numero>
```

Osservazioni

Numero indica la larghezza in pixel della traccia lasciata dal cursore grafico.

Esempio

Disegna linee di diverso spessore.

```
piano = cg.Plane('ex_34: width')
biro = cg.Pen()
for i in range(10):
    biro.width = i*2
    biro.drawsegment((-5, i-4), (5, i-4))
piano.mainloop()
```

12.5.5 drawto

Scopo

Sposta il cursore grafico nella posizione indicata dal parametro tracciando una linea che congiunge la vecchia alla nuova posizione. Il parametro è una coppia di numeri.

Sintassi

```
<penna>.drawto(<punto>)
```

Osservazioni

<punto> è una coppia di numeri. Sposta il puntatore grafico dalla posizione attuale al nuovo <punto> tracciando un segmento. Restituisce un numero che è il riferimento all'oggetto disegnato.

Esempio

Tracciare una linea tratteggiata in diagonale sullo schermo.

```
piano = cg.Plane('ex_35: drawto', sx=1, axes=False, grid=False)
biro = cg.Pen(width=2, color='blue')
x = -200
y = -100
biro.position = (x, y)
for i in range(25):
    x += 10
    y += 5
    biro.drawto((x, y))
    x += 6
    y += 3
    biro.position = (x, y)
piano.mainloop()
```

12.5.6 drawsegment

Scopo

Disegna un segmento nel piano cartesiano.

Sintassi

```
<penna>.drawsegment(p0, p1=None)
```

Osservazioni

Può essere chiamato:

- con un argomento, formato da una coppia di numeri, disegna un segmento dalla posizione indicata dal cursore grafico al punto passato come argomento.
- con due argomenti, due coppie di numeri, traccia il segmento che congiunge i due punti.

Non sposta il cursore grafico. Restituisce un numero che è il riferimento all'oggetto disegnato.

Esempio

Disegna i lati e le diagonali di un pentagono.

```
piano = cg.Plane('ex_36: drawsegment', sx=30)
biro = cg.Pen(width=4, color='gold')
vertici = ((-3, -4), (+3, -4), (+5, +1), (0, +5), (-5, +1))
for i, v0 in enumerate(vertici):
    for v1 in vertici[i+1:]:
        biro.drawsegment(v0, v1)
piano.mainloop()
```

12.5.7 drawpoint

Scopo

Disegna un punto nel piano cartesiano.

Sintassi

```
<penna>.drawpoint(p=None, color=None, width=None)
```

Osservazioni

Può essere chiamato:

- senza argomenti, disegna un punto nell'attuale posizione del cursore grafico;
- con un argomento, formato da una coppia di numeri, disegna un punto nella posizione indicata dall'argomento.
- specificando anche il colore o lo spessore e la forma del punto.

Nota: Attualmente le forme sono: `cg.CIRC` e `cg.RECT`.

Non sposta il cursore grafico. Restituisce un numero che è il riferimento all'oggetto disegnato.

Esempio

Disegna 100 punti di colore e spessore e forma casuali.

```
import random

def randcolor():
    return "#{0:02x}{1:02x}{2:02x}".format(random.randrange(256),
                                           random.randrange(256),
                                           random.randrange(256))


  piano = cg.Plane('27. punti', w=420, h=420, sx=1,
                   axes=False, grid=False)
```

```
  biro = cg.Pen(color='red', width=200)
  biro.drawpoint(shape=cg.RECT)
  for _ in range(100):
      biro.drawpoint((random.randrange(-100, 100),
                      random.randrange(-100, 100)),
                      color=randcolor(),
                      width=random.randrange(30)+1,
                      shape=random.randrange(2))
piano.mainloop()
```

12.5.8 drawcircle

Scopo

Disegna un cerchio nel piano cartesiano.

Sintassi

`<penna>.drawcircle(radius, center=None, color=None, width=None, incolor='')`

Osservazioni

Può essere chiamato:

- con un argomento: il raggio del cerchio con centro nell'attuale posizione del cursore grafico;
- con due argomenti: un numero che rappresenta il raggio e una coppia di

numeri, che indicano il centro.

- specificando anche il colore o lo spessore della circonferenza.
- specificando anche il colore dell'interno.

Non sposta il cursore grafico. Restituisce un numero che è il riferimento all'oggetto disegnato.

Esempio

Disegna 100 circonferenze.

```
import random

def randcolor():
    return "#{0:02x}{1:02x}{2:02x}".format(random.randrange(256),
                                           random.randrange(256),
                                           random.randrange(256))

piano = cg.Plane('28. cerchi', w=420, h=420, sx=1,
                 axes=False, grid=False)
  biro = cg.Pen(color='blue', width=4)
  biro.drawcircle(200, incolor='pink')
  for _ in range(100):
      biro.drawcircle(radius=random.randrange(80),
                      center=(random.randrange(-100, 100),
                              random.randrange(-100, 100)),
                      width=random.randrange(10),
                      color=randcolor(), incolor=randcolor())
piano.mainloop()
```

12.5.9 drawpoly

Scopo

Disegna un poligono dati i vertici

Sintassi

```
<penna>.drawpoly(<vertici>)
```

Osservazioni

- Il primo argomento è una sequenza, (lista o tupla) che contiene coppie di numeri.
- Si può specificare anche il colore o lo spessore della poligonale.
- Si può specificare anche il colore dell'interno.
- Restituisce un numero che è il riferimento all'oggetto disegnato.

Esempio

Disegna un pentagono casuale.

```
import random

piano = cg.Plane('ex_39: drawpoly', sx=10, grid=False)
biro = cg.Pen(width=4, color='blue')
pentagono = []
for i in range(5):
    pentagono.append((10-random.randrange(20), 10-random.randrange(20)))
biro.drawpoly(pentagono, incolor='pink')
piano.mainloop()
```

12.5.10 drawtext

Scopo

Scrive un testo nel piano cartesiano.

Sintassi

```
<penna>.drawtext(text, p=None, color=None, width=None)
```

Osservazioni

- Il primo argomento è il testo da scrivere.
- Si può specificare la posizione in cui scrivere.
- Si può specificare anche il colore o lo spessore dei caratteri.
- Restituisce un numero che è il riferimento all'oggetto disegnato.

Esempio

Scrive alcune parole poco sensate.

```
piano = cg.Plane('ex_40: drawpoly')
biro = cg.Pen(x=-5, y=7, color='blue')
biro.drawtext('testino')
biro.position = (-5, 3)
biro.drawtext('testo', color='magenta', width=2)
biro.position = (-5, -2)
```

```
biro.drawtext('testone', (-5, -2), color='green', width=4)
piano.mainloop()
```

12.6 Libreria `pycart`

12.6.1 Funzioni

`version` Restituisce la versione della libreria.

12.6.2 Classi

`Plane` Piano cartesiano.

`Pen` Tracciatore grafico per il piano cartesiano.

12.6.3 Attributi

`<penna>.color` Colore della penna.

`<penna>.position` Restituisce l'attuale posizione della penna.

`<penna>.width` Restituisce l'attuale larghezza della penna.

`<piano>.origin` Coordinate, in pixel dell'origine del piano.

`<piano>.scale` Scala di rappresentazione.

12.6.4 Metodi

`<penna>.__init__` Crea una penna.

`<penna>.drawcircle` Disegna un cerchio.

`<penna>.drawpoint` Disegna un punto.

`<penna>.drawpoly` Disegna un poligono.

`<penna>.drawsegment` Traccia un segmento.

`<penna>.drawtext` Scrive un texto.

`<penna>.drawto` Traccia una linea dalla posizione iniziale alla nuova posizione e sposta la penna.

`<penna>.reset` Riporta lo stato della penna alla situazione iniziale.

`<piano>.__init__` Crea un piano cartesiano.

`<piano>.after` Permette di effettuare un ritardo.

`<piano>.axes` Traccia gli assi.

`<piano>.clean` Cancella la finestra grafica.

`<piano>.delete` Cancella dal piano l'elemento specificato.

`<piano>.getcanvasheight` Restituisce l'altezza del piano.

`<piano>.getcanvaswidth` Restituisce la larghezza del piano.

<piano>.grid Traccia una griglia di punti.

<piano>.mainloop Rende attive le finestre grafiche.

<piano>.newPen Crea una nuova penna nel piano.

<piano>.reset Riporta il piano alla situazione iniziale.

<piano>.save Crea un file grafico con il contenuto del piano.

CAPITOLO 13

Pyplot

Dove viene presentata la libreria Pyplot

13.1 Introduzione

`pyplot` è una libreria che permette di tracciare grafici di funzioni matematiche. Fornisce alcuni metodi che permettono di tracciare grafici di:

- funzioni del tipo: y=f(x) e x=f(y);
- funzioni polari: ro=f(th);
- funzioni parametriche: x=f(t), y=g(t);
- e varie successioni.

Nella libreria sono presenti:

- Una funzione che restituisce il numero di versione corrente.
- La classe `PlotPlane` con alcuni metodi che permettono di gestire gli elementi base di un piano cartesiano.
- La classe `Plot` con alcuni metodi che permettono di tracciare il grafico di successioni e di funzioni matematiche.

Di seguito vengono descritti questi elementi. Per ognuno verrà indicato:

- lo scopo
- la sintassi
- alcune osservazioni
- un esempio di uso

13.2 version

Scopo

È una funzione che restituisce il numero di versione della libreria.

Sintassi

```
version()
```

Osservazioni

Non richiede parametri, restituisce la stringa che contiene la versione.

Esempio

Controllare la versione della libreria.

```
import pygraph.pyplot as pp
if pp.version()<"3.1.00":
    print "versione un po' vecchiotta"
else:
    print "versione:", pp.version()
```

13.3 class pp.PlotPlane

`PlotPlane` Estende la classe `Plane`, quindi è in grado di fare tutto quello che fa `Plane`. Cambiano i valori predefiniti che vengono utilizzati quando viene creato un oggetto di questa classe e c'è un ulteriore metodo.

In questa sezione presento solo le differenze e aggiunte alla classe `Plane`. Per tutto ciò che rimane invariato si veda il capitolo relativo a `pycart`.

Nota: Per poter creare oggetti della libreria `pyplot` bisogna prima di tutto importarla . In tutti gli esempi seguenti si suppone che sia già stato eseguito il comando:

```
import pygraph.pyplot as pp
```

Se nell'eseguire un esempio ottenete un messaggio che termina con: `NameError: name 'pp' is not defined`, forse avete dimenticato di caricare la libreria con il comando scritto sopra.

Se nell'eseguire gli esempi osservate dei comportamenti strani della finestra che contiene il piano cartesiano, portate pazienza, o riguardate quanto scritto nella descrizione del metodo `mainloop` di `Plane` nel capitolo su `pycart` dove ho cercato di dare delle indicazioni in proposito.

`pyplot` si appoggia su altre due librerie:

- `math` per avere a disposizione tutte le funzioni matematiche.
- `pycart` per le funzioni del piano cartesiano.

13.3.1 __init__

Scopo

Crea il piano cartesiano e inizializza gli attributi di `PlotPlane`.

Sintassi

```
<nome_variabile> = pl.PlotPlane(<parametri>)
```

Osservazioni

Questo metodo non viene chiamato esplicitamente, ma viene eseguito quando si crea un oggetto di questa classe. L'intestazione di questo metodo è:

```
def __init__(self, name="Functions pp.Plotter",
             w=400, h=400,
             sx=20, sy=None,
             ox=None, oy=None,
             axes=True, grid=True,
             axescolor='black', gridcolor='black',
             parent=None):
```

Si può vedere che presenta molti parametri tutti con un valore predefinito. Nel momento in cui si crea un piano cartesiano si possono quindi decidere le sue caratteristiche. Vediamole in dettaglio:

- titolo della finestra, valore predefinito: «Functions pp.Plotter»;
- dimensione, valori predefiniti: larghezza=400, altezza=400;
- scala di rappresentazione, valori predefiniti: una unità = 20 pixel;
- posizione dell'origine, valore predefinito: il centro della finestra;
- rappresentazione degli assi cartesiani, valore predefinito: `True`;
- rappresentazione di una griglia di punti, valore predefinito: `True`;
- colore degli assi valore predefinito: "black".
- colore della griglia valore predefinito: "black".
- riferimento alla finestra che contiene il piano cartesiano, valore predefinito: `None`.

Poiché tutti i parametri hanno un valore predefinito, possiamo creare un oggetto della classe `PlotPlane` senza specificare alcun argomento: verranno usati tutti i valori predefiniti. Oppure possiamo specificare per nome gli argomenti che vogliamo siano diversi dal comportamento predefinito, si vedano di seguito alcuni esempi.

Esempio

Si vedano tutti gli esempi seguenti.

13.3.2 `newPlot`

Scopo

Per creare un nuovo *tracciatore di funzioni*.

Sintassi

```
<nome_variabile> = <piano>.newPlot([color=<colore>, width=<numero>])
```

Osservazioni

Normalmente è utile assegnare il *tracciatore di funzioni* creato ad un identificatore.

L'intestazione di questo metodo è:

```
def newPlot(self, color='black', width=1)
```

I parametri hanno un valore predefinito, per creare un tracciatore di funzioni di colore nero e larghezza 1 non occorre passare argomenti alla funzione:

```
plot = piano.newPlot()
```

Esempio

Disegna una parabola di equazione: $y = 2x^2 - x + 1$

```
piano = pp.PlotPlane("Piano 2")
plot = piano.newPlot(color='red', width=2)
plot.xy(lambda x: 0.2*x*x-x+1)
piano.mainloop()
```

13.4 class pp.Plot

`Plot` discende da `Pen` e mette a disposizione, quindi, tutti gli attributi e i metodi di questa classe. Oltre a questi, che sono descritti nel capitolo precedente, contiene alcuni *metodi* che permettono di tracciare, facilmente, svariati tipi di funzioni.

13.4.1 __init__

Scopo Crea il Tracciatore di funzioni.

Sintassi

```
<nome_variabile> = pp.Plot(<parametri>)
```

Osservazioni

L'intestazione di questo metodo è:

```
def __init__(self, color='black', width=1)
```

L'oggetto come per l'oggetto `Pen`, anche `Plot` viene inserito nell'ultimo piano creato, a meno che non si chieda esplicitamente un comportamento diverso. Come appare dall'intestazione del metodo, alla creazione di un oggetto `Plot`, si possono anche specificare:

- il colore, valore predefinito: color="black";
- lo spessore del tratto, valore predefinito: width=1.

Dato che i parametri hanno valori predefiniti, possiamo creare oggetti `Plot` passando un numero variabile di argomenti. Gli argomenti non specificati saranno sostituiti con i valori predefiniti.

Il metodo `__init__` non viene chiamato esplicitamente, ma viene eseguito quando si crea un oggetto di questa classe.

Nota: vedi il metodo `newPlot` di `PlotPlane` per un altro modo per creare tracciatori di funzioni. Le due istruzioni:

```
piano = pp.PlotPlane(sx=100, sy=100, axes=True)
plot = pp.Plot(color='violet', width=3)
```

e:

```
piano = Plane(sx=100, sy=100, axes=True)
plot = piano.newPlot(color='violet', width=3)
```

sono del tutto equivalenti.

Esempio

Costruire un oggetto della classe `Plot`, tracciare una parabola e disegnare un quadrilatero inscritto.:

```
def parabola(x):
    return x**2-7

piano = pp.PlotPlane('x^2-7')
plot = pp.Plot(color='red', width=2)
plot.xy(parabola)
plot.drawpoly(((-1, parabola(-1)), (1, parabola(1)),
               (3, parabola(3)), (-3, parabola(-3))), 'gray')
piano.mainloop()
```

13.4.2 xy

Scopo

Traccia il grafico di una funzione nella forma `y=f(x)`.

Sintassi

```
<plot>.xy(<funzione>[, color=<colore>][,width=<numero>])
```

Osservazioni

`<funzione>` è una funzione `Python` con un parametro.

`<funzione>` può anche essere creata con una funzione `lambda`.

Quando si esegue questo metodo si possono anche specificare il colore e la larghezza del tratto. Questi due parametri hanno dei valori predefiniti e quindi non sono obbligatori.

Queste osservazioni valgono anche per gli altri metodi.

Esempio

Disegnare la funzione: y=sin(1./x).

```
def fun(x):
    return math.sin(1./x)

piano = pp.PlotPlane('sin(1./x)', w=600, h=250, sx=100, sy=100)
plot = pp.Plot(color='green', width=2)
plot.xy(fun)
piano.mainloop()
```

Usando una funzione `lambda`, lo stesso programma può essere scritto in modo più semplice così:

```
piano = pp.PlotPlane('sin(1./x)', w=600, h=250, sx=100, sy=100)
plot = pp.Plot(color='brown', width=2)
plot.xy(lambda x: math.sin(1./x))
piano.mainloop()
```

..note: Nel primo caso viene creata una funzione `Python` che viene associata a un identificatore (in questo caso `fun`), poi la funzione viene passata come argomento al metodo `xy` dell'oggetto di nome `plot`. Nel secondo caso, la funzione viene creata e passata direttamente come argomento al metodo `xy`.

13.4.3 yx

Scopo

Traccia il grafico di una funzione nella forma `x=f(y)`.

Sintassi

```
<plot>.yx(<funzione>[, color=<colore>][,width=<numero>])
```

Osservazioni

vedi *xy*

Esempio

Disegnare una parabola con l'asse di simmetria parallelo all'asse y e una con l'asse di simmetria parallelo all'asse x.

```
def parabola(var):
    return .3*var**2+3*var+4

piano = pp.PlotPlane("ex_03: xy, yx", w=600, h=600)
plot = pp.Plot(width=2)
plot.xy(parabola, color='green')
plot.yx(parabola, color='magenta')
piano.mainloop()
```

13.4.4 polar

Scopo

Traccia il grafico di una funzione polare nella forma: ro=f(th).

Sintassi

```
<plot>.polar(<funzione>[, <ma>][, color=<colore>][,width=<numero>])
```

Osservazioni

L'intestazione di questo metodo è:

```
def polar(self, f, ma=360, color=None, width=None)
```

Oltre alle osservazioni fatte per `xy`, il parametro `ma` contiene il massimo valore per la variabile `th` (il minimo è `0`). il valore predefinito di `ma` è `360`.

Esempio

Disegna tre funzioni polari in tre finestre diverse.

```
piano_0 = pp.PlotPlane("quadrifoglio", w=300, h=300)
piano_1 = pp.PlotPlane("cardioide", w=400, h=400)
piano_2 = pp.PlotPlane("spirale", w=500, h=500)
plot_0 = piano_0.newPlot(color='navy', width=2)
plot_1 = pp.Plot(plane=piano_1, color='maroon', width=4)
plot_2 = pp.Plot(color='gold', width=6)
plot_0.polar(lambda th: 5*math.cos(2*th))
plot_1.polar(lambda th: 8*math.sin(th/2))
plot_2.polar(lambda th: th, 720)
piano_0.mainloop()
```

13.4.5 param

Scopo

Traccia il grafico di una funzione parametrica nella forma: x=fx(t) y=fy(t).

Sintassi

<plot>.param(<fx>, <fy>[, mi][, ma][, color=<colore>]
[,width=<numero>])

Osservazioni

L'intestazione di questo metodo è:

```
def param(self, fx, fy, mi=0, ma=100, color=None, width=None)
```

<fx> e <fy> sono funzioni Python con un parametro che possono essere create con una funzione lambda. Il parametro <mi> ha come valore predefinito 0. Il parametro <ma> ha come valore predefinito 100. Si possono specificare il colore e lo spessore del tratto.

Esempio

Disegnare una funzione parametrica.

```
piano = pp.PlotPlane("ex_05: param", w=500, h=500)
plot = pp.Plot(color='blue', width=3)
plot.param(lambda t: 7*math.cos(3*t/180.)+3*math.cos(3*t/180.+8*t/180.),
           lambda t: 7*math.sin(3*t/180.)-3*math.sin(3*t/180.+8*t/180.),
           0, math.pi*180*2)
piano.mainloop()
```

13.4.6 ny

Scopo

Traccia il grafico di una successione nella forma y=f(n) dove n è un numero naturale.

Sintassi

<plot>.ny(<funzione>[, trace=True][, values=True] [,
color=<colore>][,width=<numero>])

Osservazioni

Per <funzione> valgono le solite considerazioni tenendo presente che la variabile indipendente è un numero naturale il grafico sarà quindi costituito da punti isolati.

trace, se posto uguale a True, traccia una linea che collega i punti consecutivi.

value, se posto uguale a True, stampa i valori della successione.

Esempio

Disegnare alcune successioni.

```
piano = pp.PlotPlane("ex_06: ny", sx=10, sy=10, ox=10)
plot = pp.Plot(color='blue', width=3)
plot.ny(lambda n: n/2+3)
plot.ny(lambda n: 3*math.sin(3*n), trace=True, values=True, color='gold')
plot.ny(lambda n: (-10*n-6)/(n), trace=True, color='maroon')
piano.mainloop()
```

13.4.7 succ

Scopo

Traccia il grafico di una successione dato il primo elemento e la funzione che restituisce l'"«enne_più_unesimo» dato l'"«ennesimo».

Sintassi

```
<plot>.succ(<a0>, <fan> [, trace=True][, values=True] [,
    color=<colore>][,width=<numero>])
```

Osservazioni

<a0> è il primo valore della successione,

<fan1> è una funzione che può avere: * un solo parametro: an * due parametri: an e n * tre parametri: an, n e a0

trace, se posto uguale a True, traccia una linea che collega i punti consecutivi.

value, se posto uguale a True, stampa i valori della successione.

Esempio

Disegnare alcune successioni.

```
def grandine(an):
    if an % 2: return 3*an+1
    else:      return an//2

piano = pp.PlotPlane("succ", sx=10, sy=10, ox=10)
plot = pp.Plot(color='green', width=3)
plot.succ(7, grandine, trace=True, values=True)
plot.succ(2, lambda an, n: an/10-an, trace=True,
          values=True, color='blue')
plot.succ(-2, lambda an, n: an-(n*2-1))
plot.succ(18, lambda an, n: an**0.5-(-1)**(n+1)*2./3+3,
          trace=True, values=True, color='maroon')
plot.succ(50, lambda an, n, a0: (an+a0/an)/2, values=True, color='pink')
piano.mainloop()
```

13.5 Libreria pyplot

13.5.1 Funzioni

version Restituisce la versione della libreria.

13.5.2 Classi

PlotPlane Piano cartesiano che permette di tracciare grafici di funzioni piane. Ha tutti gli attributi e i metodi di Plane.

Plot Tracciatore di grafici di funzioni piane. Ha tutti gli attributi e i metodi di pp.Pen.

13.5.3 Attributi

<oggetto_visibile>.color Attributo degli oggetti geometrici: imposta il colore dell'oggetto;

13.5.4 Metodi

<plot>.__init__ Crea un tracciatore grafico.

<plot>.ny Traccia una funzione del tipo: y=f(n) con n numero naturale.

<plot>.param Traccia una funzione parametrica.

<plot>.polar Traccia una funzione in coordinate polari.

<plot>.succ Traccia una successione.

<plot>.xy Traccia una funzione del tipo: y=f(x).

<plot>.yx Traccia una funzione del tipo: x=f(y).

<plotplane>.__init__ Crea un piano.

<plotplane>.newPlot Crea un tracciatore di funzioni.

<plotplane>.delplotters Elimina tutti i tracciatori di funzioni presenti.

<plotplane>.plotters Restituisce una lista con tutti i tracciatori di funzioni presenti nel piano.

CAPITOLO 14

Pyturtle

Dove viene presentata la libreria Pyturtle

14.1 Introduzione

`pyturtle` è una libreria che implementa la grafica della tartaruga. Contiene la classe `Turtle` che ha alcuni metodi che permettono di:

- Muovere la tartaruga avanti o indietro e farla ruotare su sé stessa.
- Sollevare la penna o riappoggiarla sul piano di disegno.
- Cambiare alcuni parametri relativi alla penna e al disegno.
- `pyturtle` fornisce una funzione: `version` e due classi: `TurtlePlane` e `Turtle`.

Nota: Per poter creare tartarughe bisogna importare la libreria `pyturtle`. In tutti gli esempi seguenti si suppone che sia già stato eseguito il comando:

```
import pyTurtle as tg
```

Se nell'eseguire un esempio ottenete un messaggio che termina con: `NameError: name 'tg' is not defined`, forse avete dimenticato di caricare la libreria `pyturtle` con il comando scritto sopra.

Nell'eseguire gli esempi può darsi che ci siano dei comportamenti strani della finestra che contiene il piano delle tartarughe:

- La finestra stessa non risponde ai comandi: si può ignorare il fatto o eseguire il metodo `mainloop()` del piano.
- Il programma non termina neppure quando viene chiusa la finestra: bisogna **non** eseguire il metodo `mainloop()` del piano.

Per ulteriori informazioni riguardate quanto scritto nella descrizione del metodo `mainloop` di `Plane` nel capitolo su `pycart` dove ho cercato di dare delle indicazioni in proposito.

Se vengono creati più piani basta chiamare il `mainloop()` per un solo piano (si veda il primo esempio della classe `TurtlePlane` dove si può provare a *decommentare* (togliere il carattere cancelletto "#") all'istruzione:

```
#tp0.mainloop()
```

per vedere la differenza.

`pyturtle` si appoggia sulle librerie:

- `math` per alcune funzioni matematiche,
- `pycart` per il piano cartesiano,
- `pygrapherror` per i messaggi di errore.

Nella libreria `pyturtle` sono presenti: * la funzione `version`, * la classe `TurtlePlane` che fornisce lo spazio dove possono vivere le tartarughe, * la classe `Turtle` che mette a disposizione i metodi fondamentali per la grafica della tartaruga.

Di seguito vengono presentati questi tre oggetti. Per le due classi verranno descritti gli attributi e i metodi e per ognuno verrà indicato:

- lo Scopo
- la Sintassi
- alcune Osservazioni
- un Esempio di uso

14.2 version

Scopo

Restituisce il numero di versione della libreria.

Sintassi

```
version()
```

Osservazioni

Non richiede parametri.

Esempio

Controllare la versione della libreria.

```
import pygraph.pyturtle as tg
print tg.version()
```

14.3 class tg.TurtlePlane

La classe `TurtlePlane` rappresenta lo spazio dove creare e far muovere le tartarughe e fornisce alcuni metodi che permettono di pulire la superficie, tracciare assi e griglie e gestire le tartarughe.

`TurtlePlane` è derivata da `Plane`, ne eredita quindi tutti gli attributi e i metodi, per questi si veda il capitolo relativo a `pycart`. Di seguito sono riportati i metodi (ri)definiti in `TurtlePlane`.

14.3.1 __init__

Scopo

Crea un piano in cui far muovere tartarughe.

Sintassi

```
<piano> = tg.TurtlePlane(<parametri>)
```

Osservazioni

Il metodo __init__ non viene chiamato esplicitamente ma viene eseguito quando si crea un oggetto di questa classe. Ha esattamente gli stessi parametri di Plane, vedi Plane per una illustrazione di ognuno di essi. I valori predefiniti però sono diversi, l'intestazione di questo metodo è:

```
def __init__(self, name="Turtle geometry",
             w=600, h=400,
             sx=1, sy=None,
             ox=None, oy=None,
             axes=False, grid=False,
             axescolor='black', gridcolor='black',
             master=None)
```

Esempio

Creare quattro piani con caratteristiche diverse e disegnare in ciascuno di essi un quadrato.

```
piano_0 = tg.TurtlePlane()
piano_1 = tg.TurtlePlane("TurtlePlane con gli assi", sx=10, axes=True)
piano_2 = tg.TurtlePlane("TurtlePlane con gli griglia", sx=20, grid=True)
piano_3 = tg.TurtlePlane("TurtlePlane con assi e griglia", sx=30, sy=20,
                         axes=True, grid=True)
piani = [piano_0, piano_1, piano_2, piano_3]
for p in piani:
    tarta = p.Turtle()
    for cont in range(4):
        tarta.forward(7)
        tarta.left(90)
piano_0.mainloop()
```

Nota: Se vengono creati più piani basta chiamare il mainloop() per un solo piano (si veda il primo esempio della classe TurtlePlane.

14.3.2 newTurtle

Scopo

Crea una nuova tartaruga cioè un nuovo oggetto della classe Turtle in questo piano.

Sintassi

```
<tarta> = <piano>.newTurtle(<parametri>)
```

Osservazioni

L'intestazione di questo metodo è:

```
def newTurtle(self, x=0, y=0, d=0, color='black', width=1)
```

Si può vedere che presenta molti parametri tutti con un valore di default. Nel momento in cui si crea una nuova tartaruga si possono quindi decidere le sue caratteristiche. Vediamole in dettaglio:

- ascissa e ordinata della sua posizione iniziale: `x=0, y=0`;
- direzione iniziale verso cui è rivolta: `d=0`;
- colore iniziale: `color='black'`.
- larghezza iniziale: `width=1`.

Poiché tutti i parametri hanno un valore predefinito, possiamo creare un oggetto della classe `Turtle` senza specificare alcun argomento: verranno usati tutti i valori predefiniti. Oppure possiamo specificare per nome gli argomenti che vogliamo abbiano valori diversi da quelli predefiniti. Si vedano di seguito alcuni esempi.

Esempio

Creare un piano con tre tartarughe e le fa avanzare di 100 unità.

```
piano = tg.TurtlePlane()
tina = piano.newTurtle(x=-100, y=-100, d=90, color='red')
gina = piano.newTurtle(y=-100, d=90, color='green')
pina = piano.newTurtle(x=100, y=-100, d=90, color='blue')
tina.forward(100)
pina.forward(100)
gina.forward(100)
piano.mainloop()
```

14.3.3 `turtles`

Scopo

Restituisce una lista che contiene tutte le tartarughe presenti in un piano.

Sintassi

```
<piano>.turtles()
```

Osservazioni

Ogni volta che viene creata o viene cancellata una tartaruga viene anche aggiornata questa lista.

Esempio

Creare un piano con tre tartarughe anonime e le fa avanzare di 10 unità.

```
piano = tg.TurtlePlane("Piano con tre tartarughe",
                  sx=20, sy=20, grid=True, gridcolor="grey")
tg.Turtle(x=-10, y=-8, d=90, color='red')
tg.Turtle(y=-8, d=90, color='green')
tg.Turtle(x=10, y=-8, d=90, color='blue')
for tarta in piano.turtles():
    tarta.forward(10)
piano.mainloop()
```

14.3.4 `delturtles`

Scopo

Elimina tutte le tartarughe presenti in un piano.

Sintassi

```
<piano>.delturtles()
```

Osservazioni

Elimina tutte le tartarughe create.

Esempio

Crea 2 tartarughe e fa muovere tutte le tartarughe, dopo un secondo le elimina. Crea altre 2 tartarughe e fa muovere tutte le tartarughe.:

```
piano = tg.TurtlePlane(name='2+2 Tartarughe', w=400, h=400,
                 sx=20, sy=20, grid=True)
g.Turtle(x=-3, y=-3, d=225, color='orange red', width=10)
tg.Turtle(x=+3, y=-3, d=315, color='orange', width=10)
piano.after(500)
for tarta in piano.turtles():
    tarta.forward(5)
piano.after(500)
piano.delturtles()
tg.Turtle(x=+3, y=+3, d= 45, color='pale green', width=10)
tg.Turtle(x=-3, y=+3, d=135, color='pale turquoise', width=10)
piano.after(500)
for tarta in piano.turtles():
    tarta.forward(5)
piano.mainloop()
```

14.3.5 clean

Scopo

Cancella tutti i segni tracciati dalle tartarughe.

Sintassi

```
<piano>.clean()
```

Osservazioni

Non modifica la posizione e lo stato delle tartarughe e neppure eventuali assi e griglia.

Esempio

Disegna un pentagono, si sposta, disegna un altro pentagono, applica `clean` e ridisegna un pentagono:

```
import random

def pentagono(lato):
    for cont in range(5):
        tina.forward(lato)
        tina.left(72)

piano = TurtlePlane(name="Clean", sx=10, grid=True)
tina = tg.Turtle(color='pink', width=3)
tina.tracer(5)
pentagono(5)
tina.color = (random.random(), random.random(), random.random())
tina.width = randrange(2, 20, 2)
tina.right(randrange(360))
```

```
tina.forward(randrange(5)+5)
pentagono(5)
piano.after(700)
piano.clean()
pentagono(5)
piano.mainloop()
```

14.3.6 reset

Scopo

Ripulisce il piano riportando le tartarughe nella situazione iniziale.

Sintassi

```
<piano>.reset()
```

Osservazioni

Ogni tartaruga memorizza lo stato che ha quando viene creata, reset riporta tutte le tartarughe al loro stato iniziale.

Esempio

Disegna un pentagono, si sposta, disegna un altro pentagono, applica `reset` e ridisegna un pentagono.

```
from random import random, randrange

def pentagono(lato):
    for cont in range(5):
        tina.forward(lato)
        tina.left(72)

piano = TurtlePlane(name="reset", sx=10, axes=True)
tina = tg.Turtle(color='pink', width=3)
tina.tracer(5)
pentagono(5)
tina.color = (random(), random(), random())
tina.width = randrange(2, 20, 2)
tina.right(randrange(360))
tina.forward(randrange(5)+5)
pentagono(5)
piano.after(700)
piano.reset()
pentagono(5)
piano.mainloop()
```

14.4 class Turtle:

La classe `Turtle` permette di costruire oggetti tartaruga, possiede diversi metodi che permettono di esplorare la geometria della tartaruga. `Turtle` è derivata da `Pen`, quindi possiede anche tutti i metodi di `Pen`, per questi si veda il capitolo su `pycart`.

Nota: Gli esempi riportati di seguito sono, in generale, semplici, alcuni anche banali, ma verso la fine si possono trovare come esempi dei programmi piuttosto complessi ispirati da: Abelson, Di Sessa, *La geometria della tartaruga*,

Padova 1986. Possono essere utili per approfondire alcuni elementi del linguaggio, ma la primitiva di cui si parla può essere utilizzata anche in programmi più semplici di quelli proposti.

14.4.1 position

Scopo

È un attributo della tartaruga, permette di memorizzare o di cambiare la sua posizione relativa al sistema di riferimento assoluto.

Sintassi

```
<tarta>.position = (<ascissa>, <ordinata>)
```

Osservazioni

Le coordinate sono costituite da una tupla contenente due numeri, cioè da una coppia di numeri racchiusa tra parentesi tonde.

Esempio

Sposta casualmente la tartaruga.

```
def tirett(cateto1, cateto2):
    """Disegna un triangolo rettangolo dati i due cateti"""
    tina.forward(cateto1)
    a = tina.position
    tina.back(cateto1)
    tina.left(90)
    tina.forward(cateto2)
    tina.position = a
    tina.right(90)
    tina.back(cateto1)

def inviluppo(l1, l2, inc=10):
    """Disegna un inviluppo di triangoli rettangoli."""
    if l1 < 0: return
    tirett(l1, l2)
    inviluppo(l1-inc, l2+inc)

piano = tg.TurtlePlane(name="Inviluppo di triangoli rettangoli")
tina = tg.Turtle(color='turquoise')
for i in range(4):
    inviluppo(200, 0)
    tina.left(90)
piano.mainloop()
```

14.4.2 direction

Scopo

È un attributo della tartaruga, permette di memorizzare o di cambiare la sua direzione relativa al sistema di riferimento assoluto.

Sintassi

```
<tarta>.direction = <angolo>
```

Osservazioni

La direzione assoluta ha angolo 0 verso destra e considera come verso positivo di rotazione il verso antiorario.

Esempio

Disegna un cielo stellato.

```
from random import random, randrange

def spostaacaso(raggio):
    """Sceglie una direzione e sposta la tartaruga di una
    lunghezza casuale minore a raggio."""
    tina.up()
    tina.position = (0, 0)
    tina.direction = randrange(360)
    tina.forward(randrange(raggio))
    tina.down()

def stella(lung, n):
    """Disegna una stellina con n raggi lunghi lung."""
    for cont in range(n):
        tina.forward(lung)
        tina.back(lung)
        tina.left(360./n)

def cielo():
    """Disegna un cielo stellato"""
    tina.fill(1)
    tina.ccircle(200)
    tina.fill(0)
    for cont in range(100):
        spostaacaso(190)
        tina.color = (random(), random(), random())
        stella(10, 5)

piano = tg.TurtlePlane()
tina = tg.Turtle()
cielo()
piano.mainloop()
```

14.4.3 color

Scopo

È un attributo della tartaruga, permette di memorizzare o di cambiare il suo colore.

Sintassi

```
<tarta>.color = <colore>
```

Osservazioni

`<colore>` può essere definito in 3 modi diversi (vedi Pycart), può essere:

1. una stringa che contiene il nome di un colore,
2. una tupla che contiene tre numeri compresi tra 0 e 1 che indicano l'intensità delle componenti rossa, verde, blu,
3. Una stringa nel formato: `#RRGGBB`, dove `RR`, `GG`, `BB` sono numeri esadecimali, da `00` a `ff`, che indicano i valori delle `Red`, `Green`, `Blue`).

Esempio

Disegna tre tracce colorate su uno sfondo di altro colore.

```
piano = tg.TurtlePlane()
fina = tg.Turtle()
fina.color = 'light blue'         # colore passato per nome
fina.fill(True)
fina.ccircle(200)
fina.fill(False)
fina.delete()
tina = tg.Turtle(y=-200, d=100)
tina.color = (1, 0, 0  )          # tupla che contiene 3 numeri
gina = tg.Turtle(y=-200, d=120)
gina.color = (1, 1, 1)            # tupla che contiene 3 numeri
pina = tg.Turtle(y=-200, d=140)
pina.color = '#00ff00'            # stringa nel formato '#rrggbb'
for cont in range(80):
    for tarta in piano.turtles():
        tarta.width = tarta.width+1
        tarta.forward(4)
        tarta.right(1)
piano.mainloop()
```

14.4.4 `width`

Scopo

È un attributo della tartaruga, permette di memorizzare o di cambiare la sua dimensione e lo spessore della sua penna.

Sintassi

```
<tarta>.width = <larghezza>
```

Osservazioni

Il valore minimo della larghezza della traccia è 1. Modificando la larghezza della traccia viene modificata anche la dimensione del disegno di Tartaruga.

Esempio

Disegna una fila di 8 tra quadrati e cerchi alternati.

```
def quadrato(lato):
    """Traccia un quadrato di lato lato."""
    for cont in range(4):
        tina.forward(lato)
        tina.left(90)

piano = tg.TurtlePlane()
tina = tg.Turtle()
tina.up()
tina.back(290)
for i in range(8):
    tina.down()
    if i % 2:
        tina.width = 5
        tina.color = 'orange'
        tina.circle(20)
    else:
```

```
        tina.width = 10
        tina.color = 'maroon'
        quadrato(40)
    tina.up()
    tina.forward(80)
tina.back(350)
tina.down()
piano.mainloop()
```

14.4.5 __init__

Scopo

Crea una nuova Tartaruga.

Sintassi

```
<tarta> = tg.Turtle(self, plane=<piano>)
```

Osservazioni

Il metodo `__init__` non viene chiamato direttamente, ma è eseguito quando viene creato un oggetto di questa classe. Ha un parametro obbligatorio, `plane`, che indica il piano in cui deve essere creata la tartaruga. L'intestazione di questo metodo è:

```
def __init__(self, x=0, y=0, d=0, color='black', width=1, plane=None)
```

Se viene chiamato senza parametri, verranno usati i valori predefiniti come valori iniziali. Vedi anche `newTurtle`.

Esempio

Creare un piano con tre tartarughe e farle avanzare di 15 unità.

```
piano = tg.TurtlePlane(name='Tre tartarughe', sx=20, sy=20,
                       axes=True, grid=True)
tina = tg.Turtle(x=-10, y=-8, d=90, width=5, color='red')
gina = tg.Turtle(y=-8, d=90, width=5, color='green')
pina = tg.Turtle(x=10, y=-8, d=90, width=5, color='blue')
tina.forward(15)
pina.forward(15)
gina.forward(15)
piano.mainloop()
```

14.4.6 forward

Scopo

Muove in avanti il cursore grafico, la tartaruga, di un certo numero di passi.

Sintassi

```
<tarta>.forward(<passi>)
```

Osservazioni

Il movimento è relativo alla direzione di Tartaruga. Lo spostamento dipende dalla scala del piano su cui si muove Tartaruga. `forward`, assieme a `back`, `right` e `left`, costituiscono i comandi base della geometria della tartaruga. Una geometria basata su un riferimento intrinseco al cursore.

`forward` è una funzione che restituisce il riferimento al segmento tracciato, questo riferimento può essere utile per cancellare un singolo segmento (vedi il programma orologio.py presente nella directory `examples/pyturtle`).

Esempio

Disegna un quadrato con il lato lungo 47 unità.

```
def quadrato(lato):
    """Disegna un quadrato con un certo lato"""
    for count in range(4):
        tina.forward(lato)
        tina.left(90)

piano = tg.TurtlePlane()
tina = tg.Turtle()
quadrato(47)
piano.mainloop()
```

14.4.7 back

Scopo

Muove indietro il cursore grafico, la tartaruga, di un certo numero di passi.

Sintassi

```
<tarta>.back(<passi>)
```

Osservazioni

Il movimento è relativo alla direzione di Tartaruga. Lo spostamento dipende dalla scala del piano su cui si muove Tartaruga. Anche `back` è una funzione che restituisce un riferimento al segmento tracciato.

Esempio

Disegna una stella con colori fumati.

```
piano = tg.TurtlePlane()
tina = tg.Turtle()
n = 200
for i in range(n):
    tina.color = '#ff{0:02x}00'.format(i*256//n)
    tina.forward(150)
    tina.back(150)
    tina.right(360./n)
piano.mainloop()
```

14.4.8 left

Scopo

Ruota verso sinistra la tartaruga di un certo angolo. All'avvio l'angolo viene misurato in gradi, ma si può anche dire alla tartaruga di misurarlo in radianti (vedi i metodi `degree` e `radians`).

Sintassi

```
<tarta>.left(<angolo>)
```

Osservazioni

Tartaruga non si sposta, ma ruota semplicemente in senso antiorario. All'inizio Tartaruga viene creata rivolta verso destra, inclinazione di 0 gradi.

Esempio

Disegna una stellina dato il numero e la lunghezza dei raggi.

```
def stella(num, lung):
  """Disegna una stella con enne raggi lunghi elle"""
  for i in range(num):
      tina.forward(lung)
      tina.back(lung)
      tina.right(360./num)

piano = tg.TurtlePlane()
tina = tg.Turtle()
stella(20, 50)
piano.mainloop()
```

14.4.9 `right`

Scopo

Ruota verso destra la tartaruga di un certo angolo. All'avvio l'angolo viene misurato in gradi, ma si può anche dire alla tartaruga di misurarlo in radianti (vedi i metodi `degree` e `radians`).

Sintassi

```
<tarta>.right(<angolo>)
```

Osservazioni

Tartaruga non si sposta, ma ruota semplicemente in senso rario. All'inizio Tartaruga viene creata rivolta verso destra, inclinazione di 0 gradi.

Esempio

Disegna un albero binario con una procedura ricorsiva.

```
def albero(lung=100):
    """Albero binario"""
    if lung > 2:
        tina.forward(lung)
        tina.left(45)
        albero(lung*0.7)
        tina.right(90)
        albero(lung*0.6)
        tina.left(45)
        tina.back(lung)

piano = tg.TurtlePlane()
tina = tg.Turtle(y=-100, d=90)
albero()
piano.mainloop()
```

14.4.10 up

Scopo

Solleva la penna della tartaruga in modo che, muovendosi, non tracci alcun segno.

Sintassi

```
<tarta>.up()
```

Osservazioni

Non muove Tartaruga. Se si vuole mandare la tartaruga verso la parte alta dello schermo, bisogna ruotarla verso l'alto e poi mandarla avanti.

Esempio

Trasla la tartaruga delle componenti x e y relative alla propria direzione.

```
def sposta(x, y):
    """Trasla Tartaruga delle componenti x e y"""
    tina.up()
    tina.forward(x)
    tina.left(90)
    tina.forward(y)
    tina.right(90)
    tina.down()

piano = tg.TurtlePlane()
tina = tg.Turtle()
tina.left(45)
sposta(100, -50)
piano.mainloop()
```

14.4.11 down

Scopo

Abbassa la penna di Tartaruga in modo che muovendosi lasci un segno.

Sintassi

```
<tarta>.down()
```

Osservazioni

Esegue l'operazione opposta di `up` e come il metodo `up`, non muove la tartaruga.

Esempio

Crea una nuova classe di tartarughe che sa tracciare linee tratteggiate e disegna un poligono tratteggiato.

```
class MyTurtle(tg.Turtle):
    """Una nuova classe di tartarughe
    che traccia anche linee tratteggiate."""
    def tratteggia(self,  lung, pieno=5, vuoto=5):
        """Traccia una linea tratteggiata luga lung. """
        q, r = divmod(lung, pieno+vuoto)
        for cont in range(q):
            self.forward(pieno)
            self.up()
            self.forward(vuoto)
```

```
            self.down()
        self.forward(r)

piano = tg.TurtlePlane(name="Pentagono tratteggiato")
tina = MyTurtle(color='navy', width=3)
for cont in range(5):
    tina.tratteggia(87)
    tina.left(72)
piano.mainloop()
```

14.4.12 write

Scopo

Scrive un testo sullo schermo a partire dalla posizione di Tartaruga.

Sintassi

```
<tarta>.write(<messaggio>[, move=False|True])
```

Osservazioni

Se viene chiamato con un unico parametro, la tartaruga scrive il testo e resta nella posizione attuale. Se oltre al messaggio, viene passato anche un argomento `True`, la tartaruga viene spostata alla fine della scritta. I metodi `up` e `down` non hanno effetto su `write`.

Esempio

Scrivi alcune parole nel piano della tartaruga.

```
piano = tg.TurtlePlane(name='Testo nella finestra grafica', w=400, h=200)
tina = tg.Turtle()
position = tina.position
tina.up()
tina.position = (-140, 80)
tina.color = 'green'
tina.write('Parole')
tina.color = 'pink'
tina.write('sovrapposte!')
tina.position = (-140, -80)
tina.color = 'red'
tina.write('Parole ', move=True)
tina.color = 'orange'
tina.write('scritte di seguito!')
tina.position = position
tina.down()
piano.mainloop()
```

14.4.13 fill

Scopo

Permette di colorare l'interno delle figure realizzate dalla tartaruga.

Sintassi

```
<tarta>.fill(True|False)
```

Osservazioni

`fill(True)` inizia a memorizzare le figure da riempire, `fill(False)` le riempie con il colore attuale.

Esempio

Una funzione che disegna un quadrato con l'interno colorato.

```
def quadratopieno(lato, coloreinterno):
    """Disegna un quadrato dati il lato e il colore della sup. interna."""
    tina.fill(True)
    for count in range(4):
        tina.forward(lato)
        tina.left(90)
    oldcol = tina.color
    tina.color = coloreinterno
    tina.fill(False)
    tina.color = oldcol

piano = tg.TurtlePlane()
tina = tg.Turtle(color='blue', width=3)
quadratopieno(100, 'green')
piano.mainloop()
```

14.4.14 `tracer`

Scopo

Rallenta la velocità di Tartaruga.

Sintassi

```
<tarta>.tracer(<ritardo>)
```

Osservazioni

Più è elevato l'argomento, più lentamente si muove Tartaruga nei comandi `forward`, `back` e `setpos`. Se l'argomento è 0, Tartaruga traccia un intero segmento in un solo passo.

Esempio

Disegna una traccia circolare variando la velocita" e lo spessore.

```
from random import random

piano = tg.TurtlePlane(name="Giro della morte")
tina = tg.Turtle(y=-180)
tina.color = (random(), random(), random())
passi = 72
angolo = 180./72
for i in range(passi):
  altezza = (tina.position[1]+200)/20
  tina.width = altezza
  tina.tracer(altezza)
  tina.left(angolo)
  tina.forward(15)
  tina.left(angolo)
piano.mainloop()
```

14.4.15 circle

Scopo

Traccia una circonferenza o un arco di circonferenza.

Sintassi

```
<tarta>.circle(<raggio> [, <estensione>])
```

Osservazioni

Se è presente un solo argomento viene tracciata una circonferenza di dato raggio.

Se ci sono due argomenti, viene disegnato un arco di circonferenza e il secondo argomento indica l'estensione dell'arco.

Esempio

Disegna 25 archi e 25 segmenti circolari alternati.

```
from random import random, randrange

piano = tg.TurtlePlane(w=600, h=600)
tina = tg.Turtle(width=5)
for j in range(50):
    tina.color = (random(), random(), random())
    tina.up()
    tina.position = (randrange(-250, 250), randrange(-250, 250))
    tina.down()
    tina.fill(j % 2 == 1)
    tina.circle(randrange(25)+25, randrange(360))
    tina.color = (random(), random(), random())
    tina.fill(0)
piano.mainloop()
```

14.4.16 ccircle

Scopo

Disegna una circonferenza o un arco centrati nella posizione di Tartaruga.

Sintassi

```
<tarta>.ccircle(<raggio> [, <estensione>])
```

Osservazioni

Se ci sono due argomenti, viene disegnato un arco di circonferenza e il secondo argomento indica l'estensione dell'arco. In questo caso la tartaruga ruota dell'ampiezza indicata dal secondo argomento.

Esempio

Disegna 25 archi e 25 settori circolari alternati.

```
from random import random, randrange

piano = tg.TurtlePlane(w=600, h=600)
tina = tg.Turtle(width=5)
for j in range(50):
    tina.color = (random(), random(), random())
    tina.up()
```

```
    tina.position = (randrange(-250, 250), randrange(-250, 250))
    tina.down()
    tina.fill(j % 2 == 1)
    tina.ccircle(randrange(25)+25, randrange(360))
    tina.color = (random(), random(), random())
    tina.fill(0)
piano.mainloop()
```

14.4.17 radians

Scopo

Dopo aver invocato questo metodo, Tartaruga misura gli angoli in radianti.

Sintassi

```
<tarta>.radians()
```

Osservazioni

Ogni tartaruga, appena creata misura gli angoli in gradi sessagesimali, se si preferisce lavorare con i radianti basta dirglielo.

Esempio

Vedi l'esempio di `degree`.

14.4.18 degrees

Scopo

Dopo la chiamata a questo metodo, Tartaruga misura gli angoli in gradi.

Sintassi

```
<tarta>.degrees([<angolo giro>])
```

Osservazioni

Se usato senza argomenti, l'angolo giro sarà di 360°, l'argomento permette di specificare il numero di gradi di un angolo giro. Io non l'ho mai usato, ma si potrebbe decidere che l'angolo giro sia di 400 gradi o di un qualunque altro valore.

Esempio

Disegna un orologio analogico che riporta l'ora del sistema (mescolare in uno stesso programma gradi e radianti, non mi sembra una grande idea).

```
from time import time, localtime
from math import pi

def lancette(d, ore, minuti, secondi):
    """Disegna le lancette di un orologio che segna ore:minuti:secondi"""
    tina.degrees()
    aore = round(90 - ore*30 - minuti*0.5)        # Angoli in gradi
    aminuti = round(90 - minuti*6 - secondi*0.1)
    asecondi = round(90 - secondi*6)
    tina.direction = aore
    tina.width = d*0.1
    tina.forward(d*0.6)
```

```
    tina.back(d*0.6)
    tina.direction = aminuti
    tina.width = d*0.05
    tina.forward(d*0.8)
    tina.back(d*0.8)
    tina.direction = asecondi
    tina.width = d*0.02
    tina.forward(d*0.9)
    tina.back(d*0.9)

def quadrante(dimensione):
    """Il quadrante di un orologio"""
    tina.radians()
    tina.ccircle(dimensione)
    d1 = dimensione*0.9
    d2 = dimensione-d1
    for i in range(12):
        tina.up()
        tina.forward(d1)
        tina.down()
        tina.forward(d2)
        tina.up()
        tina.back(dimensione)
        tina.down()
        tina.left(pi/6)                          # Angolo in radianti
    tina.degrees()

def orologio(dimensione):
    """Disegna un orologio che indica l'ora corrente."""
    quadrante(dimensione)
    ore, minuti, secondi = localtime(time())[3:6]
    lancette(dimensione, ore, minuti, secondi)

piano = tg.TurtlePlane(name="Ora esatta")
tina = tg.Turtle()
orologio(100)
piano.mainloop()
```

14.4.19 distance

Scopo

Restituisce la distanza tra Tartaruga e il punto dato come argomento.

Sintassi

```
<tarta>.distance(<coordinate>)
```

Osservazioni

L'argomento deve essere una coppia di numeri racchiusa tra parentesi.

Esempio

Dirige Tartaruga verso un bersaglio emulando l'olfatto.

```
from random import randrange

def saltaacaso():
```

```
    """Sposta Tartaruga in una posizione casuale."""
    tina.up()
    tina.position = (randrange(-280, 280), randrange(-180, 180))
    tina.down()

def bersaglio():
    """Disegna un bersaglio"""
    saltaacaso()
    for cont in range(4):
        tina.forward(5)
        tina.back(5)
        tina.left(90)
    larghezza = tina.width
    tina.width = 2
    tina.ccircle(20)
    tina.width = larghezza
    tina.ccircle(10)
    return(tina.position)

def odore(p):
    """Riporta un numero inversamente proporzionale
    al quadrato della distanza da p"""
    return 1./(tina.distance(p)**2)

def ricerca_per_odore(bersaglio):
    """Muove Tartaruga in base ad una regola olfattiva"""
    tina.tracer(10)
    ricordo_odore = odore(bersaglio)
    while tina.distance(bersaglio)>10:
        tina.forward(randrange(10))
        nuovo_odore = odore(bersaglio)
        if nuovo_odore < ricordo_odore:
            tina.right(randrange(80))
        ricordo_odore = nuovo_odore

piano = tg.TurtlePlane("Ricerca in base all'odore")
tina = tg.Turtle()
b = bersaglio()
saltaacaso()
ricerca_per_odore(b)
piano.mainloop()
```

14.4.20 dirto

Scopo

Restituisce la direzione sotto cui Tartaruga vede un certo punto

Sintassi

```
<tarta>.dirto(<coordinate>)
```

Osservazioni

Le coordinate sono una coppia di numeri. Il valore restituito è relativo alla direzione di Tartaruga: se il valore è 0 significa che il punto è a ore 12 cioè esattamente davanti a Tartaruga, se è 180 si trova a ore 6, cioè esattamente dietro, se è 90 si trova a ore 9, se è 270 si trova a ore 3, ...

Esempio

Simula l'inseguimento tra un gatto e un topo.

```
from random import randrange

class Topo(tg.Turtle):
    """Classe che simula il comportamento di un topo"""
    def __init__(self, **args):
        Turtle.__init__(self, **args)
        piano = self._plane
        self.up()
        self.position = (piano._s2x(20), piano._s2y(20))
        self.write("inseguimento in base alla vista")
        self.position = (randrange(120)-60, randrange(80)-40)
        self.down()

    def scappa(self, da):
        """Fa muovere il topo di un passo cercando di scappare da da"""
        dir_gatto = self.dirto(da.position)
        if dir_gatto < 170:
            self.right(randrange(20)-10)
        elif dir_gatto > -170:
            self.left(randrange(20)-10)
        else:
            self.left(randrange(80)-40)
        self.forward(1)

class Gatto(tg.Turtle):
    """Classe che simula il comportamento di un gatto"""
    def __init__(self, **args):
        Turtle.__init__(self, **args)
        self.color = "red"
        self.width = 3

    def insegui(self, chi):
        """Fa muovere il gatto di un passo come se inseguisse chi
        ritorna 0 se ha raggiunto chi"""
        if self.distance(chi.position) < 3:
            self.position = chi.position
            return 0
        dir_topo = self.dirto(chi.position)
        if dir_topo < 180:
            self.left(randrange(5))
        else:
            self.right(randrange(5))
        self.forward(2)
        return 1

piano = tg.TurtlePlane()
topo = Topo()
gatto = Gatto()
while gatto.insegui(topo):
    topo.scappa(gatto)
piano.mainloop()
```

14.4.21 whereis

Scopo

Restituisce una coppia di numeri che indicano la distanza di un punto e l'angolo sotto cui Tartaruga lo vede.

Sintassi

```
<tarta>.whereis(<coordinate>)
```

Osservazioni

Il risultato di questo metodo è una coppia di numeri.

Esempio

Simula la ricerca in base all'udito.

```
from random import randrange
from math import pi, sin

def saltaacaso():
    """Sposta tartaruga in una posizione casuale."""
    tina.up()
    tina.position = (randrange(-280, 280), randrange(-180, 180))
    tina.down()

def bersaglio():
    """Disegna un bersaglio"""
    saltaacaso()
    for cont in range(4):
        tina.forward(5)
        tina.back(5)
        tina.left(90)
    larghezza = tina.width
    tina.width = 2
    tina.ccircle(20)
    tina.swidth = larghezza
    tina.ccircle(10)
    return(tina.position)

def intensita_destra(distanza, angolo):
    """Restituisce un numero che indica l'intensita' con cui
    l'orecchio destro ode un suono proveniente da un bersaglio
    posto ad una certa distanza e con un certo angolo"""
    senangolo = sin((angolo % 360)*pi/180)
    if senangolo < 0:
        return -10.*senangolo/(distanza**2)
    else:
        return 5.*senangolo/(distanza**2)

def intensita_sinistra(distanza, angolo):
    """Restituisce un numero che indica l'intensita' con cui
    l'orecchio sinistro ode un suono proveniente da un bersaglio
    posto ad una certa distanza e con un certo angolo"""
    senangolo = sin((angolo % 360)*pi/180)
    if senangolo > 0:
        return 10.*senangolo/(distanza**2)
    else:
        return -5.*senangolo/(distanza**2)
```

```
def ricerca_per_udito(bersaglio):
    """Simula la ricerca di un bersaglio utilizzando l'udito"""
    tina.tracer(10)
    while tina.distance(bersaglio)>10:
        d, a = tina.whereis(bersaglio)
        tina.forward(randrange(5))
        if intensita_destra(d, a) > intensita_sinistra(d, a):
            tina.right(randrange(30))
        else:
            tina.left(randrange(30))

piano = tg.TurtlePlane("Ricerca in base all'udito")
tina = tg.Turtle()
b = bersaglio()
saltaacaso()
ricerca_per_udito(b)
piano.mainloop()
```

14.4.22 lookat

Scopo

Ruota Tartaruga in modo che sia rivolta verso un certo punto.

Sintassi

```
<tarta>.lookat(<coordinate>)
```

Osservazioni

A differenza dei tre metodi precedenti, non dà risultato, ma provoca una rotazione di Tartaruga.

Esempio

Disegna una parabola per mezzo dell'inviluppo di rette.

```
def asse(punto):
    """Disegna l'asse del segmnento che ha per estremi Tartaruga e
       punto"""
    direzione = tina.direction
    position = tina.position
    tina.up()
    tina.lookat(punto)
    tina.forward(tina.distance(punto)/2.)
    tina.left(90)
    tina.back(1000)
    tina.down()
    tina.forward(2000)
    tina.up()
    tina.position = position
    tina.direction = direzione
    tina.down()

piano = tg.TurtlePlane(name="Parabola formata da un inviluppo di rette")
tina = tg.Turtle()
fuoco = (0, 50)
numpassi = 80
lato = 5
tina.up()
```

```
tina.back(200)
tina.down()
for i in range(numpassi):
    asse(fuoco)
    tina.forward(lato)
piano.mainloop()
```

14.4.23 hide

Scopo

Nasconde la tartaruga.

Sintassi

```
<tarta>.hide()
```

Osservazioni

Il metodo non richiede argomenti.

Esempio

Vedi il metodo `show`

14.4.24 show

Scopo

Mostra la tartaruga.

Sintassi

```
<tarta>.show()
```

Osservazioni

Il metodo non richiede argomenti.

Esempio

Traccia una linea mostrando e nascondendo Tartaruga.

```
piano = tg.TurtlePlane("Cucu")
tina = tg.Turtle()
tina.up()
tina.back(295)
tina.down()
tina.tracer(10)
for i in range(15):
    tina.forward(19)
    tina.hide()
    tina.forward(19)
    tina.show()
piano.mainloop()
```

14.4.25 clone

Scopo

Crea un clone della tartaruga.

Sintassi

```
<tarta>.clone()
```

Osservazioni

Non richiede argomenti.

Esempio

Fa collaborare alcune tartarughe per disegnare un albero.

```
piano = tg.TurtlePlane("Albero realizzato da una famiglia di tartarughe",
                  w=500, h=500)
tg.Turtle(y=-200, d=90, color='olive drab')
dim = 100
angl = 25
angr = 35
while dim > 5:
    for tarta in piano.turtles():
        tarta.width = dim//5
        tarta.forward(dim)
        t1 = tarta.clone()
        tarta.left(angl)
        t1.right(angr)
    piano.after(200)
    dim *= 0.7
    print(len(piano.turtles()))
for tarta in piano.turtles():
    tarta.color = 'lime green'
piano.mainloop()
```

14.4.26 delete

Scopo

Elimina la tartaruga dal piano.

Sintassi

```
<tarta>.delete()
```

Osservazioni

Il metodo non richiede argomenti. I segni già tracciati rimangono.

Esempio

Crea 36 tartarughe all'interno dello stesso piano mettendole in una lista, poi le fa muovere.

```
piano = tg.TurtlePlane("36 tartarughe che eseguono gli stessi comandi",
                  w=500, h=500, sx=20, sy=20, grid=True)
n = 36
l = 4.
for i in range(n):
    tg.Turtle(color='#ff{0:02x}00'.format(i*256//n), d=(360/n*i), width=5)
```

```
for tarta in piano.turtles(): tarta.forward(l)
for tarta in piano.turtles(): tarta.left(90)
for tarta in piano.turtles(): tarta.forward(l/4)
for tarta in piano.turtles(): tarta.right(45)
for tarta in piano.turtles(): tarta.forward(l)
for tarta in piano.turtles(): tarta.right(120)
for tarta in piano.turtles(): tarta.forward(l/2)
for tarta in piano.turtles(): tarta.left(60)
for tarta in piano.turtles(): tarta.forward(l/2)
piano.after(500)
piano.clean()
for tarta in piano.turtles(): tarta.forward(2)
piano.after(500)
piano.reset()
for tarta in piano.turtles(): tarta.forward(5)
for tarta in piano.turtles():
    piano.after(100)
    tarta.delete()
piano.after(500)
piano.clean()
piano.mainloop()
```

14.5 Libreria `pyturtle`

14.5.1 Funzioni

`version` Restituisce la versione della libreria.

14.5.2 Classi

`TurtlePlane` Spazio dove creare e far muovere le tartarughe e fornisce alcuni metodi che permettono di pulire la superficie, tracciare assi e griglie e gestire le tartarughe.

`Turtle` Puntatore grafico che permette di realizzare la geometria della tartaruga.

14.5.3 Attributi

`<tartaruga>.color` Colore della tartaruga.

<tartaruga>.'direction' Restituisce l'attuale direzione di Tartaruga.

`<tartaruga>.position` Restituisce l'attuale posizione di Tartaruga.

`<tartaruga>.width` Restituisce l'attuale larghezza della penna.

14.5.4 Metodi

`<piano_per_tartarughe>.__init__` Crea un piano in cui far muovere tartarughe.

`<piano_per_tartarughe>.clear` Cancella tutti i segni tracciati dalle tartarughe.

`<piano_per_tartarughe>.delturtles` Elimina tutte le tartarughe presenti in un piano.

<piano_per_tartarughe>.newTurtle Crea una nuova tartaruga cioè un nuovo oggetto della classe `Turtle` in questo piano.

<piano_per_tartarughe>.reset Ripulisce il piano riportando le tartarughe nella situazione iniziale.

<piano_per_tartarughe>.turtles Restituisce una lista che contiene tutte le tartarughe presenti in un piano.

<tartaruga>.__init__ Crea una nuova Tartaruga.

<tartaruga>.back Muove indietro il cursore grafico, Tartaruga, di un certo numero di passi.

<tartaruga>.ccircle Disegna una circonferenza o un arco centrati nella posizione della tartaruga.

<tartaruga>.circle Traccia una circonferenza o un arco di circonferenza.

<tartaruga>.clone Crea un clone della tartaruga.

<tartaruga>.degrees Dopo la chiamata a questo metodo, Tartaruga misura gli angoli in gradi.

<tartaruga>.delete Elimina la tartaruga dal piano.

<tartaruga>.dirto Restituisce la direzione sotto cui Tartaruga vede un certo punto

<tartaruga>.distance Restituisce la distanza tra Tartaruga e il punto dato come argomento.

<tartaruga>.down Abbassa la penna di Tartaruga in modo che muovendosi lasci un segno.

<tartaruga>.fill Permette di colorare l'interno delle figure realizzate dalla tartaruga.

<tartaruga>.forward Muove in avanti il cursore grafico, Tartaruga, di un certo numero di passi.

<tartaruga>.hide Nasconde la tartaruga.

<tartaruga>.left Ruota verso sinistra la tartaruga di un certo angolo. All'avvio l'angolo viene misurato in gradi, ma si può anche dire alla tartaruga di misurarlo in radianti (vedi i metodi `degree` e `radians`).

<tartaruga>.lookat Ruota Tartaruga in modo che sia rivolta verso un certo punto.

<tartaruga>.radians Dopo aver invocato questo metodo, Tartaruga misura gli angoli in radianti.

right Ruota verso destra la tartaruga di un certo angolo. All'avvio l'angolo viene misurato in gradi, ma si può anche dire alla tartaruga di misurarlo in radianti (vedi i metodi `degree` e `radians`).

<tartaruga>.show Mostra la tartaruga.

<tartaruga>.tracer Rallenta la velocità di Tartaruga.

<tartaruga>.up Solleva la penna della tartaruga in modo che, muovendosi, non tracci alcun segno.

<tartaruga>.whereis Restituisce una coppia di numeri che indicano la distanza di un punto e l'angolo sotto cui Tartaruga lo vede.

<tartaruga>.write Scrive un testo sullo schermo a partire dalla posizione di Tartaruga.

CAPITOLO 15

Pyig

Dove viene presentata la libreria Pyig

15.1 Introduzione

`pyig` è una libreria che implementa la geometria interattiva. `pyig` contiene una classe `InteractivePlane` che fornisce una finestra grafica dove realizzare le costruzioni geometriche e alcune altre classi che rappresentano gli elementi della geometria interattiva: punti, rette, circonferenze, ...

Gli elementi di base di questa geometria sono i punti liberi che possono venir trascinati con il mouse. Gli altri oggetti dipendono in modo diretto o indiretto dai punti base. Quando viene spostato un punto base, tutti gli oggetti che dipendono da lui vengono ridisegnati. Ad esempio se costruisco una retta passante per due punti, quando sposto uno di questi punti, anche la retta si muove. Il programma minimo che utilizza questa libreria è composto da tre istruzioni:

```
import pygraph.pyig as ig              # Carica la libreria
piano = ig.InteractivePlane()          # Crea un piano
ip.mainloop()                          # Rendi interattivo il piano
```

Questo programma non fa un gran che: disegna una finestra grafica con un piano cartesiano e la rende attiva. Se volessimo disegnare al suo interno anche una retta per due punti, potremmo scrivere il seguente programma:

```
import pygraph.pyig as ig                 # Carica la libreria
piano = ig.InteractivePlane('Secondo')    # Crea un piano
p_0 = ig.Point(-3, -5)                    # Disegna due punti
p_1 = ig.Point(2, 3)
ig.Line(p0, p1)                           # Disegna la retta
piano.mainloop()                          # Rendi interattivo il piano
```

A seconda dell'ambiente in cui viene eseguito il programma, l'ultima istruzione può essere necessaria o, in certi casi, essere di intralcio. È importante osservare il comportamento di questo programma con e senza l'istruzione:

```
ip.mainloop()
```

e, in base p_a come si comporta il nostro sistema decidere se inserirla o no nel proprio programma.

Nella libreria sono presenti:

- Una funzione che restituisce il numero di versione corrente.
- La classe *InteractivePlane* che è un piano cartesiano con in più alcuni attributi che contengono i valori predefiniti degli oggetti geometrici e alcuni metodi che permettono di creare gli oggetti liberi: punti e testo.
- Un certo numero di classi che permettono di creare oggetti geometrici.

Di seguito vengono riportati gli attributi e i metodi delle classi. Per ognuno verrà indicato:

- lo scopo
- la sintassi
- alcune osservazioni
- un esempio di uso

15.2 version

Scopo

È una funzione che restituisce il numero di versione della libreria.

Sintassi

```
version()
```

Osservazioni

Non richiede parametri, restituisce la stringa che contiene la versione.

Esempio

Controllare la versione della libreria.

```
import pygraph.pyig as ig
if ig.version()<"2.9.00":
    print("versione un po' vecchiotta")
else:
    print("versione:", ig.version())
```

15.3 InteractivePlane

`InteractivePlane` estende la classe `Plane`, quindi è in grado di fare tutto quello che fa `Plane`. Cambiano i valori predefiniti che vengono utilizzati quando viene creato un oggetto di questa classe e ci sono alcuni atri attributi e metodi.

In questa sezione presento solo le differenze e aggiunte alla classe `Plane`. Per tutto ciò che rimane invariato si veda il capitolo relativo p_a `pycart`.

Nota: Per poter usare gli oggetti messi p_a disposizione da `pyig` bisogna importare la libreria. In tutti gli esempi seguenti si suppone che sia già stato eseguito il comando:

```
import pygraph.pyig as ig
```

Nei prossimi esempi la precedente istruzione è sottintesa, va sempre scritta all'inizio del programma. Se nell'esegure un esempio ottenete un messaggio che termina con: `NameError: name 'ig' is not defined`, forse avete dimenticato di caricare la libreria con il comando scritto sopra.

Di seguito sono elencati alcuni attributi della classe InteractivePlane.

15.3.1 `__init__`

Scopo

Crea il piano cartesiano e inizializza gli attributi di `InteractivePlane`.

Sintassi

```
InteractivePlane(<parametri>)
```

Osservazioni

Questo metodo non viene chiamato esplicitamente, ma viene eseguito quando si crea un oggetto di questa classe. L'intestazione di questo metodo è:

```
def __init__(self, name="Interactive geometry",
             w=600, h=600,
             sx=20, sy=None,
             ox=None, oy=None,
             axes=True, grid=True,
             axescolor='#080808', gridcolor='#080808',
             parent=None):
```

Si può vedere che presenta molti parametri tutti con un valore predefinito. Nel momento in cui si crea un piano cartesiano si possono quindi decidere le sue caratteristiche. Vediamole in dettaglio:

- titolo della finestra, valore predefinito: «Interactive geometry»;
- dimensione, valori predefiniti: larghezza=600, altezza=600;
- scala di rappresentazione, valori predefiniti: una unità = 20 pixel;
- posizione dell'origine, valore predefinito: il centro della finestra;
- rappresentazione degli assi cartesiani, valore predefinito: `True`;
- rappresentazione di una griglia di punti, valore predefinito: `True`;
- colore degli assi valore predefinito: `#808080` (grigio).
- colore della griglia valore predefinito: `#808080`.
- riferimento alla finestra che contiene il piano cartesiano, valore predefinito: `None`.

Poiché tutti i parametri hanno un valore predefinito, possiamo creare un oggetto della classe `InteractivePlane` senza specificare alcun argomento: verranno usati tutti i valori predefiniti. Oppure possiamo specificare per nome gli argomenti che vogliamo siano diversi dal comportamento predefinito, si vedano di seguito alcuni esempi.

Esempio

Si vedano tutti gli esempi seguenti.

15.3.2 Attributi

`InteractivePlane` ha alcuni attributi che possono essere modificati dagli utenti.

Scopo

Questi attributi definiscono alcuni valori predefiniti con i quali saranno creati i nuovi oggetti.

- `defvisible` stabilisce se i nuovi oggetti creati saranno visibili o invisibili;
- `defwidth` imposta la larghezza predefinita;
- `defwidthtext` imposta la larghezza predefinita dei caratteri;
- `defcolor` imposta il colore predefinito degli oggetti;
- `defdragcolor` imposta il colore predefinito per gli oggetti che si possono trascinare con il mouse.
- `defintcolor` imposta il colore predefinito per la superficie di circonferenze e poligoni.

Sintassi

```
<piano interattivo>.defvisible = v
<piano interattivo>.defwidth = w
<piano interattivo>.defwidthtext = w
<piano interattivo>.defcolor = c
<piano interattivo>.defdragcolor = c
<piano interattivo>.defintern = c
```

Osservazioni

- `v` è un valore booleano, può essere `True` o `False`.
- `w` è un numero che indica la larghezza in pixel.
- `c` può essere:
  - una stringa nel formato: `#rrggbb` dove `rr`, `gg` e `bb` sono numeri esadecimali di due cifre che rappresentano rispettivamente le componenti rossa, verde, e blu del colore;
  - una stringa contenente il nome di un colore;
  - una terna di numeri nell'intervallo 0-1 rappresentanti le componenti rossa verde e blu.

Esempio

Disegna l'asse di un segmento usando linee di costruzioni sottili e grige.

```
ip = ig.InteractivePlane('default')
ip.defwidth = 5
p_a = ip.newPoint(2, 1, name='A')
p_b = ip.newPoint(5, 3, name='B')
ip.defcolor = 'turquoise'
ig.Segment(p_a, p_b)
ip.defwidth = 1
ip.defcolor = 'gray40'
c0 = ig.Circle(p_a, p_b)
c1 = ig.Circle(p_b, p_a)
i0 = ig.Intersection(c0, c1, -1)
i1 = ig.Intersection(c0, c1, 1)
asse = ig.Line(i0, i1, width=3, color='royal blue')
ip.mainloop()
```

15.3.3 newText

Scopo

La geometria interattiva ha alcuni oggetti di base sono oggetti che non dipendono da altri. `InteractivePlane` ha alcuni metodi che permettono di creare questi oggetti. `newText` serve per creare un nuovo *testo*.

Sintassi

```
<piano>.newText(x, y, testo[, visible][, color][, width][, name])
```

Osservazioni

L'intestazione di questo metodo è:

```
def newText(self, x, y, text,
            visible=None, color=None, width=None, name=None):
```

Molti parametri hanno un valore predefinito, per creare un testo, bisogna specificare la posizione dove metterlo e la stringa che verrà visualizzata:

```
ip.newText(-3, 12, 'Titolo')
```

Esempio

Crea un testo nel piano interattivo.

```
ip = ig.InteractivePlane('newText')
ip.newText(0, 13, 'Finestra (quasi) vuota',
           width=20, color='DarkOrchid3')
ip.mainloop()
```

15.3.4 newPoint

Scopo

La geometria interattiva ha alcuni oggetti di base, oggetti che non dipendono da altri. `InteractivePlane` ha alcuni metodi che permettono di creare questi oggetti. `newPoint` serve per creare un nuovo *punto*.

Sintassi

```
<piano interattivo>.newPoint(x, y[, visible][, color][, width] [, name])
```

Osservazioni

Normalmente è utile assegare il *punto* creato ad un identificatore.

L'intestazione di questo metodo è:

```
def newPoint(self, x, y,
             visible=None, color=None, width=None, name=None):
```

Molti parametri hanno un valore predefinito, per creare un punto, gli unici due argomenti necessari sono i valori delle coordinate:

```
p_a = ip.newPoint(3, -7)
```

Esempio

Crea un punto in una data posizione.

```
ip = ig.InteractivePlane('newPoint')
ip.newText(0, 13, 'Finestra con un punto',
           width=20, color='DarkOrchid3')
p_0 = ip.newPoint(-2, 7, width=8, name="P")
ip.mainloop()
```

15.3.5 newVarText

Scopo

Un testo variabile è costituito da una stringa con alcuni segnaposti e da altrettanti dati. I segnaposti sono indicati da una coppia di parentesi graffe che racchiudono un numero prograssivo: `{0}`. I dati si ricavano dagli oggetti geometrici presenti nel piano.

Sintassi

`<piano>.newVarText(x, y, testo, dati[, visible][, color][, width] [, name])`

Osservazioni

L'intestazione di questo metodo è:

```
def newVarText(self, x, y, text, variables,
               visible=None, color=None, width=None, name=None):
```

In questo caso è necessario specificare la posizione dove dovrà essere visualizzata la stringa, il testo, e la (o le) variabile(i):

```
ip.newVarText(-3, 12, 'posizione: {0}', p.coords())
```

Esempio

Visualizza alcune informazioni che dipendono da altri oggetti.

```
ip = ig.InteractivePlane('newVarText')
ip.newText(0, 13, 'Finestra con un punto e le sue coordinate',
           width=20, color='DarkOrchid3')
p_0 = ip.newPoint(-2, 7, width=8, name="P")
ip.newVarText(-5, -4, 'P = {0}', p0.coords())
ip.mainloop()
```

15.4 Point

Scopo

Crea un *punto libero* date le coordinate della sua posizione iniziale.

Questo oggetto è la base di ogni costruzione; dai punti liberi dipendono, direttamente o indirettamente, gli altri oggetti grafici.

Quando il puntatore del mouse si sovrappone ad un punto libero questo cambia colore. Trascinando un punto libero, con il mouse, tutti gli oggetti che dipendono da lui, verranno modificati.

`Point` essendo un oggetto che può essere trascinato con il mouse ha un colore predefinito diverso da quello degli altri oggetti.

Sintassi

```
Point(x, y[, visible][, color][, width][, name])
```

Nota: Spesso nella pratica è necessario assegnare l'oggetto creato ad un identificatore in modo da poter fare riferimento ad un oggetto nella costruzione di altri oggetti

```
<identificatore> = Point(x, y[, visible][, color][, width][, name])
```

Si vedano gli esempi seguenti.

Osservazioni

- `x` e `y` sono due numeri, `x` è l'ascissa e `y` l'ordinata del punto.
- Per quanto riguarda i parametri non obbligatori si veda quanto scritto nel paragrafo relativo agli attributi degli oggetti visibili.

Nota: Nel resto del manuale riporterò solo gli argomenti obbligatori, è sottinteso che tutti gli oggetti che possono essere visualizzati hanno anche i parametri: `visible`, `color`, `width`, `name`.

Esempio

Funzione definita in N ad andamento casuale.

```
import random
ip = ig.InteractivePlane('Point')
y = 0
for x in range(-14, 14):
    y += random.randrange(-1, 2)
    ig.Point(x, y, color='red')
ip.mainloop()
```

15.4.1 Attributi degli oggetti geometrici

Scopo

`Point`, come tutti gli oggetti geometrici ha degli attributi che possono essere determinati nel momento della creazione dell'oggetto stesso o in seguito. Questi attributi definiscono alcune caratteristiche degli oggetti che possono essere visualizzati.

- `visible` stabilisce se l'oggetto sarà visibili o invisibile;
- `color` imposta il colore dell'oggetto;
- `width` imposta la larghezza dell'oggetto.
- `name` imposta il nome dell'oggetto.

Sintassi

```
<oggetto>.visible = v
```

```
<oggetto>.color = c
```

```
<oggetto>.width = w
```

```
<oggetto>.name = s
```

Osservazioni

- `v` è un valore booleano, può essere True o False.

- `w` è un numero che indica la larghezza in pixel.
- `c` può essere:
  - una stringa nel formato: «#rrggbb» dove rr, gg e bb sono numeri esadecimali di due cifre che rappresentano rispettivamente le componenti rossa, verde, e blu del colore;
  - Una stringa contenente il nome di un colore;
  - Una terna di numeri nell'intervallo 0-1 rappresentanti le componenti rossa verde e blu.
- `s` è una stringa

Esempio

Disegna tre punti: uno con i valori di default, uno con colore dimensione e nome definiti quando viene creato, uno con valori cambiati dopo essere stato cerato.

```
ip = ig.InteractivePlane('attributi')
p_a = ig.Point(-5, 3)
p_b = ig.Point(2, 3, color='indian red', width=8, name='B')
p_c = ig.Point(9, 3)
p_c.color = 'dark orange'
p_c.width = 8
p_c.name = 'C'
ip.mainloop()
```

15.4.2 Metodi degli oggetti geometrici

Scopo

Tutti gli oggetti geometrici hanno anche dei metodi che danno come risultato alcune informazioni relative all'oggetto stesso.

- `xcoord` l'ascissa;
- `ycoord` l'ordinata;
- `coords` le coordinate.

Sintassi

```
<oggetto>.xcoord() <oggetto>.ycoord() <oggetto>.coords()
```

Osservazioni

Non richiedono argomenti e restituiscono un particolare oggetto che può essere utilizzato all'interno di un testo variabile.

Esempio

Scrivi ascissa, ordinata e posizione di un punto.

```
ip = ig.InteractivePlane('coords, xcoord, ycoord')
p_a = ig.Point(-5, 8, name='A')
ig.VarText(-5, -1, 'ascissa di A: {0}', p_a.xcoord())
ig.VarText(-5, -2, 'ordinata di A: {0}', p_a.ycoord())
ig.VarText(-5, -3, 'posizione di A: {0}', p_a.coords())
ip.mainloop()
```

15.4.3 Operazioni con i punti

Scopo

Sono definite alcune operazioni tra punti: l'addizione, la sottrazione, la moltiplicazione per uno scalare e la moltiplicazione tra due punti.

Sintassi

```
<punto> + <punto>
<punto> - <punto>
<punto> * <numero>
<punto> * <punto>
```

Osservazioni

- È possibile così scrivere espressioni tra punti.

Esempio

Crea due punti poi calcola la loro somma, differenza e prodotto.

```
ip = ig.InteractivePlane('Point operations')
p_a = ig.Point(2, 1)
p_b = ig.Point(4, 4)
s = p_a+p_b
p_d = p_a-p_b
t = p_a*3
p = p_a*p_b
ig.VarLabel(p_a, 10, -10, 'A{0}', p_a.coords())
ig.VarLabel(p_b, 10, -10, 'B{0}', p_b.coords())
ig.VarLabel(s, 10, -10, 'A+B{0}', s.coords())
ig.VarLabel(p_d, 10, -10, 'A-B{0}', p_d.coords())
ig.VarLabel(t, 10, -10, '3A{0}', t.coords())
ig.VarLabel(p, 10, -10, 'A*B{0}', p.coords())
ip.mainloop()
```

15.5 `Segment`

Scopo

Crea un segmento dati i due estremi, i due estremi sono *punti*.

Sintassi

```
Segment(point0, point1)
```

Osservazioni

`point0` e `point1` sono due punti.

Esempio

Disegna un triangolo con i lati colorati in modo differente.

```
ip = ig.InteractivePlane('Segment')
# creo i 3 vertici
v_0 = ig.Point(-4, -3, width=5)
v_1 = ig.Point( 5, -1, width=5)
```

```
v_2 = ig.Point( 2,  6, width=5)
# creo i 3 lati
l_0 = ig.Segment(v_0, v_1, color='steel blue')
l_1 = ig.Segment(v_1, v_2, color='sea green')
l_2 = ig.Segment(v_2, v_0, color='saddle brown')
ip.mainloop()
```

15.5.1 length

Scopo

È il metodo della classe `Segment` che restituisce un oggetto `data` contenete la lunghezza del segmento stesso.

Sintassi

```
<obj>.length()
```

Osservazioni

La lunghezza è la distanza tra `point0` e `point1`.

Esempio

Disegna un segmento e scrivi la sua lunghezza.

```
ip = ig.InteractivePlane('length')
p_0 = ig.Point(-4, 7, width=5, name='A')
p_1 = ig.Point(8, 10, width=5, name='B')
seg = ig.Segment(p0, p1)
ig.VarText(-5, -5, 'lunghezza di AB = {0}', seg.length())
ip.mainloop()
```

15.5.2 midpoint

Scopo

È il metodo della classe `Segment` che restituisce il punto medio del segmento.

Sintassi

```
<obj>.midpoint()
```

Osservazioni

L'oggetto restituito è un punto.

Esempio

Disegna un segmento e il suo punto medio.

```
ip = ig.InteractivePlane('35: midpoint')
seg = ig.Segment(ig.Point(-4, 7, width=5, name='A'),
                 ig.Point(8, 10, width=5, name='B'))
seg.midpoint(color='gold', width=10)
ip.mainloop()
```

`Line`, `Ray` e `Segment` hanno dei metodi che forniscono degli oggetti che contengono alcune informazioni relative alla linea stessa. Vengono elencati di seguito.

15.5.3 equation

Scopo

È il metodo presente in tutte le classi *linea* che restituisce l'equazione esplicita della retta o della retta che p_a cui appartiene l'oggetto.

Sintassi

```
<obj>.equation()
```

Osservazioni

L'equazione è data sotto forma di stringa.

Esempio

Disegna una retta e scrivi la sua equazione .

```
ip = ig.InteractivePlane('equation')
p_0 = ig.Point(-4, 7, width=5)
p_1 = ig.Point(8, 10, width=5)
retta = ig.Line(p_0, p_1, name='r')
ig.VarText(-5, -5, 'equazione di r: {0}', retta.equation())
ip.mainloop()
```

15.5.4 slope

Scopo

È il metodo presente in tutte le classi *linea* che restituisce la pendenza della retta o della retta che p_a cui appartiene l'oggetto.

Sintassi

```
<obj>.slope()
```

Osservazioni

Se la retta è verticale la pendenza restituita è `None`.

Esempio

Disegna una semiretta e scrivi la sua pendenza .

```
ip = ig.InteractivePlane('slope')
p_0 = ig.Point(-4, 7, width=5)
p_1 = ig.Point(8, 10, width=5)
semiretta = ig.Ray(p_0, p_1, name='r')
ig.VarText(-5, -5, 'pendenza di r: {0}', semiretta.slope())
ip.mainloop()
```

15.5.5 point0 e point1

Scopo

Sono i metodi presenti in tutte le classi *linea* che restituiscono rispettivamente il *punto0* e il *punto1* dell'oggetto.

- Il *punto0* è il primo punto della retta passante per due punti, il primo estremo del segmento, l'origine della semiretta, il vertice dell'angolo per la bisettrice, il punto per cui passa la parallela o il punto in cui la perpendicolare interseca la retta.

- Il *punto1* è il secondo punto della retta passante per due punti, il secondo estremo del segmento, il punto per cui passa la semiretta, un punto della bisettrice, un punto della parallela o il punto per cui passa la perpendicolare.

Sintassi

```
<obj>.point0()
```

```
<obj>.point1()
```

Esempio

Disegna un segmento e scrivi le coordinate dei suoi estremi.

```
ip = ig.InteractivePlane('point0 point1')
seg = ig.Segment(ig.Point(-4, 7, width=5, name='A'),
                 ig.Point(8, 10, width=5, name='B'))
ig.VarText(-5, -5, 'A{0}', seg.point0().coords())
ig.VarText(-5, -6, 'B{0}', seg.point1().coords())
ip.mainloop()
```

15.6 MidPoints

Scopo

Crea il punto medio tra due punti.

Sintassi

```
MidPoints(point0, point1)
```

Osservazioni

`point0` e `point1` sono due punti.

Esempio

Punto medio tra due punti.

```
ip = ig.InteractivePlane('MidPoints')
# creo due punti
p_0 = ig.Point(-2, -5)
p_1 = ig.Point(4, 7)
# cambio i loro attributi
p_0.color = "#00a600"
p_0.width = 5
p_1.color = "#006a00"
p_1.width = 5
# creo il punto medio tra p_0 e p_1
m = ig.MidPoints(p_0, p_1, name='M')
# cambio gli attributi di m
m.color = "#f0f000"
m.width = 10
ip.mainloop()
```

15.7 MidPoint

Scopo

Crea il punto medio di un segmento

Sintassi

```
MidPoint(segment)
```

Osservazioni

`segment` è un oggetto che ha un `point0` e un `point1`.

Esempio

Punto medio di un segmento.

```
ip = ig.InteractivePlane('MidPoint')
# creo un segmento
s = ig.Segment(ig.Point(-2, -1, color="#a60000", width=5),
               ig.Point(5, 7, color="#6a0000", width=5),
               color="#a0a0a0")
# creo il suo punto medio
ig.MidPoint(s, color="#6f6f00", width=10, name='M')
ip.mainloop()
```

15.8 Line

Scopo

Crea una retta per due punti.

Sintassi

```
Line(point0, point1)
```

Osservazioni

`point0` e `point1` sono, indovina un po", due punti.

Vedi anche i metodi delle classi *linea* presentati nella classe `Segment`.

Esempio

Triangolo delimitato da rette.

```
ip = ig.InteractivePlane('Line')
# creo i 3 punti
p_a = ig.Point(0, 0)
p_b = ig.Point(1, 5)
p_c = ig.Point(5, 1)
# creo i 3 lati
ig.Line(p_a, p_b, color="#dead34")
ig.Line(p_b, p_c, color="#dead34")
ig.Line(p_c, p_a, color="#dead34")
ip.mainloop()
```

15.9 Ray

Scopo

Traccia una semiretta con l'origine in un punto e passante per un altro punto.

Sintassi

```
Ray(point0, point1)
```

Osservazioni

`point0` è l'origine della semiretta che passa per `point1`.

Vedi anche i metodi delle classi *linea* presentati nella classe `Segment`.

Esempio

Triangolo delimitato da semirette.

```
ip = ig.InteractivePlane('Ray')
# creo i 3 punti
p_a = ig.Point(0, 0)
p_b = ig.Point(1, 5)
p_c = ig.Point(5, 1)
# creo i 3 lati
ig.Ray(p_a, p_b, color="#de34ad")
ig.Ray(p_b, p_c, color="#de34ad")
ig.Ray(p_c, p_a, color="#de34ad")
ip.mainloop()
```

15.10 Orthogonal

Scopo

Crea la retta perpendicolare ad una retta data passante per un punto.

Sintassi

```
Orthogonal(line, point)
```

Osservazioni

`line` è la retta alla quale si costruisce la perpendicolare passante per `point`.

Vedi anche i metodi delle classi *linea* presentati nella classe `Segment`.

Esempio

Disegna la perpendicolare ad una retta data passante per un punto.

```
ip = ig.InteractivePlane('Orthogonal')
retta = ig.Line(ig.Point(-4, -1, width=5),
                ig.Point(6, 2, width=5),
                width=3, color='DarkOrange1', name='r')
punto = ig.Point(-3, 5, width=5, name='P')
ig.Orthogonal(retta, punto)
ip.mainloop()
```

15.11 Parallel

Scopo

Crea la retta parallela ad una retta data passante per un punto.

Sintassi

```
Parallel(line, point)
```

Osservazioni

`line` è la retta alla quale si costruisce la parallela passante per `point`.

Vedi anche i metodi delle classi *linea* presentati nella classe `Segment`.

Esempio

Disegna la parallela ad una retta data passante per un punto.

```
ip = ig.InteractivePlane('Parallel')
retta = ig.Line(ig.Point(-4, -1, width=5),
                ig.Point(6, 2, width=5),
                width=3, color='DarkOrange1', name='r')
punto = ig.Point(-3, 5, width=5, name='P')
ig.Parallel(retta, punto)
ip.mainloop()
```

15.12 `Vector`

Scopo

Crea un segmento orientato dati i due estremi o il punto di applicazione e un altro vettore.

Sintassi

```
Vector(point0, point1)
```

```
Vector(point0, vector)
```

Osservazioni

`point0` e `point1` sono punti vector è un vettore.

È anche possibile operare con i vettori sono p_a disposizione la somma, la differenza, l'opposto, il prodotto e il quoziente per uno scalare.

Esempio

Disegna alcuni vettori e il risultato di alcune operazioni tra di essi.

```
ip = ig.InteractivePlane('Vector')
ig.Text(-7, 13, """Vector""", color='#408040', width=12)
p_a = ig.Point(-5, 10, name="A", width=6)
p_b = ig.Point(-1, 13, name="B", width=6)
p_c = ig.Point(-3, 5, name="C", width=6)
p_d = ig.Point(-10, 6, name="D", width=6)
vect0 = ig.Vector(p_a, p_b, width=6, color='navy', name='V0')
vect1 = ig.Vector(p_a, p_c, width=6, color='red', name='V1')
vect2 = ig.Vector(p_d, vect1, color='red', name='V2')
vect3 = vect0 + vect1
vect4 = vect0 - vect1
vect5 = vect0 * 3
vect6 = -vect0 / 3
vect6.color = 'red'
ig.VarText(7, -2, '{0}', vect0.components(), (.2, .4, .8))
ip.mainloop()
```

15.13 Polygon

Scopo

Crea un poligono data una sequenza di vertici.

Sintassi

```
Polygon(points)
```

Osservazioni

`points` è una sequenza di punti, può essere una lista (delimitata da parentesi quadre) o una tupla (delimitata da parentesi tonde).

Esempio

Disegna un poligono date le coordinate dei vertici.

```
ip = ig.InteractivePlane('24: Polygon')
# Lista di coordinate
coords = ((-8, -3), (-6, -2), (-5, -2), (-4, 2), (-2, 3), (0, 4),
          (2, 3), (4, 2), (5, -2), (6, -2), (8, -3))
# Costruzione di una lista di punti partendo da una lista di coordinate:
# listcompreension
ip.defwidth = 5
points = [ig.Point(ip, x, y) for x,y in coords]
ig.Polygon(points, color='HotPink3')
ip.mainloop()
```

15.13.1 perimeter e surface

Scopo

Sono metodi presenti in tutte le classi *figura*, restituiscono la lunghezza del contorno e l'area della superficie dell'oggetto.

Sintassi

```
<figure>.perimeter()
```

```
<figure>.surface()
```

Osservazioni

Sono metodi degli oggetti che sono *figure piane* e non richiede argomenti.

Esempio

Scrive alcune informazioni relative a un poligono.

```
ip = ig.InteractivePlane('24: Polygon')
poli = ig.Polygon((ig.Point(-7, -3, width=5, name="A"),
                   ig.Point(5, -5, width=5, name="B"),
                   ig.Point(-3, 8, width=5, name="C")),
                   width=4, color='magenta', intcolor='olive drab')
ig.VarText(-3, -6, "perimetro={0}", poli.perimeter(), color='magenta')
ig.VarText(-3, -7, "area={0}", poli.surface(), color='olive drab')
ip.mainloop()
```

15.14 Circle

Scopo

Circonferenza dato il centro e un punto o il centro e il raggio (un segmento).

Sintassi

```
Circle(center, point)
Circle(center, segment)
```

Osservazioni

`center` è il centro della circonferenza passante per `point` o di raggio `segment`.

Vedi anche i metodi delle classi *figure piane* presentati nella classe `Polygon`.

Esempio

Circonferenze con centro nell'origine.

```
ip = ig.InteractivePlane('Circle(Point, Point)')
origine = ig.Point(0, 0, visible=False, name="O")
p_0 = ig.Point(-7, -3, width=5, name="P")
ig.Circle(origine, p0, color="#c0c0de", width=4)
raggio = ig.Segment(ig.Point(-7, 9, width=5, name="A"),
                    ig.Point(-4, 9, width=5, name="B"))
ig.Circle(origine, raggio, color="#c0c0de", width=4)
ip.mainloop()
```

15.14.1 radius

Scopo

Restituisce un *oggetto dato* che contiene la lunghezza del raggio della circonferenza.

Sintassi

```
<circle>.radius()
```

Osservazioni

È un metodo degli oggetti circonferenza e non richiede argomenti.

Esempio

Visualizza il raggio di una circonferenza.

```
ip = ig.InteractivePlane('radius')
centro = ig.Point(3, 4, name="C")
p_0 = ig.Point(-5, 4, width=5, name="P")
p_c = ig.Circle(centro, p0, color="#c0c0de", width=4)
ig.VarText(-5, -1, 'raggio: {0}', p_c.radius())
ip.mainloop()
```

15.15 Intersection

Scopo

Crea il punto di intersezione tra due oggetti.

Sintassi

```
Intersection(obj0, obj1)
```

```
Intersection(obj0, obj1, which)
```

Osservazioni

`obj0` e `obj1` possono essere rette o circonferenze. Se uno dei due oggetti è una circonferenza è necessario specificare quale delle due intersezioni verrà restituita indicando come terzo parametro `+1` o `-1`.

Esempio

Disegna una circonferenza tangente a una retta.

```
ip = ig.InteractivePlane('Intersection line line')
# Disegno retta e punto
retta = ig.Line(ig.Point(-4, -1, width=5),
                ig.Point(6, 2, width=5),
                width=3, color='DarkOrange1', name='r')
punto = ig.Point(-3, 5, width=5, name='P')
# trovo il punto di tangenza
perpendicolare = ig.Orthogonal(retta, punto, width=1)
p_tang = ig.Intersection(retta, perpendicolare, width=5)
# disegno la circonferenza
ig.Circle(punto, p_tang, width=4, color='IndianRed')
ip.mainloop()
```

Disegna il simmetrico di un punto rispetto ad una retta.

```
ip = ig.InteractivePlane('Intersection line circle')
# disegno l'asse di simmetria e il punto
asse = ig.Line(ig.Point(-4, -11, width=5),
               ig.Point(-2, 12, width=5),
               width=3, color='DarkOrange1', name='r')
punto = ig.Point(-7, 3, width=5, name='P')
# disegno la perpendicolare all'asse passante per il punto
perp = ig.Orthogonal(asse, punto, width=1)
# trovo l'intersezione tra la perpendicolare e l'asse
piede = ig.Intersection(perp, asse)
# disegno la circonferenza di centro piede e passante per punto
circ = ig.Circle(piede, punto, width=1)
# trovo il simmetrico di punto rispetto a asse
ig.Intersection(perp, circ, -1, width=5, color='DebianRed', name="P'")
ip.mainloop()
```

Disegna un triangolo equilatero.

```
ip = ig.InteractivePlane('Intersection circle circle')
# Disegno i due primi vertici
v_0 = ig.Point(-2, -1, width=5, name='A')
v_1 = ig.Point(3, 2, width=5, name='B')
# Disegno le due circonferenze di centro v_0 e v_1 e
# passanti per v_1 e v_0
c_0 = ig.Circle(v_0, v_1, width=1)
c_1 = ig.Circle(v_1, v_0, width=1)
# terzo vertice: intersezione delle due circonferenze
v_2 = ig.Intersection(c_0, c_1, 1, width=5, name='C')
# triangolo per i 3 punti
```

```
ig.Polygon((v_0, v_1, v_2), width=4, color='DarkSeaGreen4')
ip.mainloop()
```

15.16 Text

Scopo

Crea un testo posizionato in un punto del piano.

Sintassi

```
Text(x, y, text[, iplane=None])
```

Osservazioni

- `x` e `y` sono due numeri interi o razionali relativi `x` è

l'ascissa e `y` l'ordinata del punto.

- `text` è la stringa che verrà visualizzata.
- Se sono presenti più piani interattivi, si può specificare l'argomento `iplane` per indicare in quale di questi la scritta deve essere visualizzata.

Esempio

Scrive un titolo in due finestre grafiche.

```
ip0 = ig.InteractivePlane('Text pale green', w=400, h=200)
ip1 = ig.InteractivePlane('Text blue violet', w=400, h=200)
ig.Text(-2, 2, "Prove di testo blue violet",
        color='blue violet', width=20)
ig.Text(-2, 2, "Prove di testo pale green",
        color='pale green', width=20, iplane=ip0)
ip0.mainloop()
```

15.17 Label

Scopo

Crea un testo legato p_a un oggetto.

Sintassi

```
Label(obj, x, y, text)
```

Osservazioni

La stringa contenuta in `text` viene posizionata alla distanza di `x` pixel in orizzontale e di `y` pixel in verticale dall'oggetto `obj`.

Esempio

Disegna un punto e gli appiccica un'etichetta.

```
ip = ig.InteractivePlane('Label')
p_0 = ig.Point(7, 3, color='navy', width=10, name='A')
ig.Label(p0, 0, 20, "colore di A = 'navy'")
ip.mainloop()
```

15.18 VarText

Scopo

Crea un testo variabile. Il testo contiene dei «segnaposto» che verranno sostituiti con i valori prodotti dai dati presenti nel parametro variables.

Sintassi

```
VarText(x, y, text, variables[, iplane=None])
```

Osservazioni

- x e y sono due numeri interi o razionali relativi x è

l'ascissa e y l'ordinata del punto.

- text è la stringa che contiene la parte costante e i segnaposto.
- In genere i *segnaposto* saranno nella forma: «{0}» che indica p_a Python di convertire in stringa il risultato prodotto dal dato.
- variables è un dato o una tupla di dati.
- Se sono presenti più piani interattivi, si può specificare l'argomento iplane per indicare in quale di questi la scritta deve essere visualizzata.

Esempio

Un testo che riporta la posizione dei un punto.

```
ip = ig.InteractivePlane('VarText')
p_0 = ig.Point(7, 3, color='green', width=10, name='A')
ig.VarText(-4, -3, "Posizione del punto A: ({0}; {1})",
           (p0.xcoord(), p0.ycoord()),
           color='green', width=10)
ip.mainloop()
```

15.19 VarLabel

Scopo

Testo variabile legato ad un oggetto.

Sintassi

```
VarLabel(obj, x, y, text, variables)
```

Osservazioni

vedi le osservazioni di Label e VarText.

Esempio

Disegna un punto con un'ettichetta che riporta la sua posizione.

```
ip = ig.InteractivePlane('VarLabel')
p_0 = ig.Point(7, 3, color='red', width=10, name='A')
ig.VarLabel(p0, 0, -40,
            "A{0}", p0.coords(), color='red', width='10')
ip.mainloop()
```

15.20 PointOn

Scopo

Punto disegnato su un oggetto in una posizione fissa.

Sintassi

```
PointOn(obj, parameter)
```

Osservazioni

L'oggetto deve essere una linea o una retta o una circonferenza, `parameter` è un numero che individua una precisa posizione sull'oggetto. Sia le rette sia le circonferenze hanno una loro metrica che è legata ai punti base dell'oggetto. Su una retta una semiretta o un segmento `point0` corrisponde al parametro 0 mentre `point1` corrisponde al parametro 1. Nelle circoferenze il punto di base della circonferenza stessa corrisponde al parametro 0 l'intera circonferenza vale 2. Il punto creato con `PointOn` non può essere trascinato con il mouse.

Esempio

Disegna il simmetrico di un punto rispetto ad una retta.

```
ip = ig.InteractivePlane('PointOn')
# disegno l'asse di simmetria e il punto
asse = ig.Line(ig.Point(-4, -11, width=5),
               ig.Point(-2, 12, width=5),
               width=3, color='DarkOrange1', name='r')
punto = ig.Point(-7, 3, width=5, name='P')
# disegno la perpendicolare all'asse passante per il punto
perp = ig.Orthogonal(asse, punto, width=1)
# trovo il simmetrico di punto rispetto p_a asse
ig.PointOn(perp, -1, width=5, color='DebianRed', name="P'")
ig.Text(-5, -6, """P' è il simmetrico di P.""")
ip.mainloop()
```

15.21 ConstrainedPoint

Scopo

Punto legato ad un oggetto.

Sintassi

```
ConstrainedPoint(obj, parameter)
```

Osservazioni

Per quanto riguarda `parameter`, valgono le osservazioni fatte per `PoinOn`. Questo punto però può essere trascinato con il mouse pur restando sempre sull'oggetto. Dato che può essere trascinato con il mouse ha un colore di default diverso da quello degli altri oggetti.

Esempio

Circonferenza e proiezioni sugli assi.

```
ip = ig.InteractivePlane('ConstrainedPoint', sx=200)
# Circonferenza
origine = ig.Point(0, 0, visible=False)
unix = ig.Point(1, 0, visible=False)
uniy = ig.Point(0, 1, visible=False)
```

```
circ = ig.Circle(origine, unix, color="gray10")
# Punto sulla circonferenza
cursore = ig.ConstrainedPoint(circ, 0.25, color='magenta', width=20)
# assi
assex = ig.Line(origine, unix, visible=False)
assey = ig.Line(origine, uniy, visible=False)
# proiezioni
py = ig.Parallel(assey, cursore, visible=False)
hx = ig.Intersection(assex, py, color='red', width=8)
px = ig.Parallel(assex, cursore, visible=False)
hy = ig.Intersection(assey, px, color='blue', width=8)
ip.mainloop()
```

15.21.1 parameter

Scopo

I punti legati agli oggetti hanno un metodo che permette di ottenere il parametro.

Sintassi

```
<constrained point>.parameter()
```

Osservazioni

In `PointOn` il parametro è fissato nel momento della costruzione dell'oggetto. In `ConstrainedPoint` il parametro può essere variato trascinando il punto con il mouse.

Esempio

Scrivi i dati relativi p_a un punto collegato p_a un oggetto.

```
ip = ig.InteractivePlane('parameter')
c0 = ig.Circle(ig.Point(-6, 6, width=6), ig.Point(-1, 5, width=6))
c1 = ig.Circle(ig.Point(6, 6, width=6), ig.Point(1, 5, width=6))
p_a = ig.PointOn(c0, 0.5, name='A')
p_b = ig.ConstrainedPoint(c1, 0.5, name='B')
ip.newVarText(-5, -1, 'ascissa di A: {0}', p_a.xcoord())
ip.newVarText(-5, -2, 'ordinata di A: {0}', p_a.ycoord())
ip.newVarText(-5, -3, 'posizione di A: {0}', p_a.coords())
ip.newVarText(-5, -4, 'parametro di A: {0}', p_a.parameter())
ip.newVarText(5, -1, 'ascissa di B: {0}', p_b.xcoord())
ip.newVarText(5, -2, 'ordinata di B: {0}', p_b.ycoord())
ip.newVarText(5, -3, 'posizione di B: {0}', p_b.coords())
ip.newVarText(5, -4, 'parametro di B: {0}', p_b.parameter())
ip.mainloop()
```

15.22 Angle

Scopo

Angolo dati tre punti o due punti e un altro angolo. Il secondo punto rappresenta il vertice. Il verso di costruzione dell'angolo è quello antiorario.

Sintassi

```
Angle(point0, vertex, point1[, sides])
Angle(point0, vertex, angle[, sides])
```

Osservazioni

L'argomento `sides` può valere:

- `True` (o `(0, 1)`): vengono disegnati i lati;
- `0`: viene disegnato il lato 0;
- `1`: viene disegnato il lato 1;

`Angle` fornisce i seguenti metodi dal significato piuttosto evidente:

- `extent`: ampiezza dell'angolo;
- `bisector`: bisettrice;
- `vertex`: il vertice;
- `point0`: il punto base 0;
- `point1`: il punto base 1;
- `side0`: il lato 0;
- `side1`: il lato1.

Esempio

Disegna un angolo e un angolo con i lati.

```
ip = ig.InteractivePlane('Angle(Point, Point, Point)')
ip.defwidth = 5
p_a = ig.Point(-2, 4, color="#40c040", name="A")
p_b = ig.Point(-5, -2, color="#40c040", name="B")
p_c = ig.Point(-8, 6, color="#40c040", name="C")
p_d = ig.Point(8, 6, color="#40c040", name="D")
p_e = ig.Point(5, -2, color="#40c040", name="E")
p_f = ig.Point(2, 4, color="#40c040", name="F")
# angolo senza i lati
ig.Angle(p_a, p_b, p_c, color="#40c040")
# angolo con i lati
ig.Angle(p_d, p_e, p_f, color="#c04040", sides=True)
ip.mainloop()
```

Somma di due angoli.

```
ip = ig.InteractivePlane('Angle(Point, Point, Angle)')
# i 2 angoli di partenza
p_a = ig.Angle(ig.Point(-3, 7, width=6),
               ig.Point(-7, 5, width=6),
               ig.Point(-6, 8, width=6),
               sides=(0, 1), color="#f09000", name='alfa')
p_b = ig.Angle(ig.Point(9, 2, width=6),
               ig.Point(2, 3, width=6),
               ig.Point(6, 4, width=6),
               sides=(0, 1), color="#0090f0", name='beta')
# Punti di base dell'angolo somma di p_a p_b
v = ig.Point(-11, -8, width=6)
p_0 = ig.Point(3, -10, width=6)
# la somma degli angoli
```

```
b1 = ig.Angle(p_0, v, p_b, (0, 1), color="#0090f0")
p_1 = b1.point1()
a1 = ig.Angle(p_1, v, p_a, sides=True, color="#f09000")
ig.Text(-4, -12, "Somma di due angoli")
ip.mainloop()
```

15.23 Bisector

Scopo

Retta bisettrice di un angolo.

Sintassi

```
Bisector(angle)
```

Osservazioni

Vedi `Ray`.

Esempio

Disegna l'incentro di un triangolo.

```
ip = ig.InteractivePlane('Bisector')
# I tre vertici del triangolo
p_a = ig.Point(-7, -3, color="#40c040", width=5, name="A")
p_b = ig.Point(5, -5, color="#40c040", width=5, name="B")
p_c = ig.Point(-3, 8, color="#40c040", width=5, name="C")
# Il triangolo
ig.Polygon((p_a, p_b, p_c))
# Due angoli del triangolo
cba=Angle(p_c, p_b, p_a)
bac=Angle(p_b, p_a, p_c)
# Le bisettrici dei due angoli
b1 = ig.Bisector(cba, color="#a0c040")
b2 = ig.Bisector(bac, color="#a0c040")
# L'incentro
ig.Intersection(b1, b2, color="#c040c0", width=5, name="I")
ip.mainloop()
```

15.24 Polygonal

Scopo

Poligonale data una sequenza di vertici.

Sintassi

```
Polygonal(points)
```

Osservazioni

Vedi `Polygon`.

Esempio

Disegna una linea spezzata aperta.

```
ip = ig.InteractivePlane('Polygonal')
points=(ig.Point(-8, -3), ig.Point(-6, -2), ig.Point(-5, -2),
        ig.Point(-4, 2),  ig.Point(-2, 3),  ig.Point(0, 4),
        ig.Point(2, 3),   ig.Point(4, 2),   ig.Point(5, -2),
        ig.Point(6, -2),  ig.Point(8, -3))
Polygonal(points, color='saddle brown', width=4)
ip.mainloop()
```

15.25 CurviLine

Scopo

Linea curva determinata da una sequenza di vertici.

Sintassi

```
CurviLine(points)
```

Osservazioni

Vedi `Polygon`.

Esempio

Disegna una linea spezzata aperta.

```
ip = ig.InteractivePlane('CurviLine')
points=(ig.Point(-8, -3), ig.Point(-6, -2), ig.Point(-5, -2),
        ig.Point(-4, 2),  ig.Point(-2, 3),  ig.Point(0, 4),
        ig.Point(2, 3),   ig.Point(4, 2),   ig.Point(5, -2),
        ig.Point(6, -2),  ig.Point(8, -3))
ig.CurviLine(points, color='goldenrod', width=4)
ip.mainloop()
```

15.26 Calc

Scopo

Dato che contiene il risultato di un calcolo.

Sintassi

```
Calc(function, variables)
```

Osservazioni

- `function` è una funzione python, al momento del calcolo, alla funzione vengono passati come argomenti il contenuto di `variables`.
- `variables` è un oggetto `Data` o una tupla che contiene oggetti Data. Il risultato è memorizzato all'interno dell'oggetto Calc e può essere visualizzato con `VarText` o utilizzato per definire la

posizione di un punto.

Esempio

Calcolla il quadrato di un lato e la somma dei quadrati degli altri due di un triangolo.

```
ip = ig.InteractivePlane('Calc')
ig.Circle(ig.Point(2, 4), ig.Point(-3, 4), width=1)
ip.defwidth = 5
p_a = ig.Point(-3, 4, name="A")
p_b = ig.Point(7, 4, name="B")
p_c = ig.Point(-1, 8, name="C")
ab = ig.Segment(p_a, p_b, color="#40c040")
bc = ig.Segment(p_b, p_c, color="#c04040")
ca = ig.Segment(p_c, p_a, color="#c04040")
q1 = ig.Calc(lambda p_a: p_a*p_a, ab.length())
q2 = ig.Calc(lambda p_a, p_b: p_a*p_a+p_b*p_b,
                              (bc.length(), ca.length()))
ig.VarText(-5, -5, "ab^2 = {0}", q1, color="#40c040")
ig.VarText(-5, -6, "bc^2 + ca^2 = {0}", q2, color="#c04040")
ip.mainloop()
```

15.27 Elenco Funzioni, Classi, Attributi e Metodi di `pyig`

15.27.1 Funzioni di `pyig`

`version` Restituisce la versione della libreria.

15.27.2 Classi di `pyig`

`Angle` Angolo dati tre punti o due punti e un angolo, il secondo punto rappresenta il vertice. Il verso di costruzione dell'angolo è quello antiorario.

`Bisector` Retta bisettrice di un angolo.

`Circle` Circonferenza dato il centro e un punto o il centro e un raggio (un segmento).

`ConstrainedPoint` Punto legato ad un oggetto.

`CurviLine` Linea curva determinata da una sequenza di vertici.

`Calc` Dato che contiene il risultato di un calcolo.

`InteractivePlane` Crea il piano cartesiano e inizializza gli attributi del *piano*.

`Intersection` Crea il punto di intersezione tra due rette.

`Label` Crea un testo legato p_a un oggetto.

`Line` Crea una retta per due punti.

`MidPoint` Crea il punto medio di un segmento

`MidPoints` Crea il punto medio tra due punti.

`Orthogonal` Crea la retta perpendicolare ad una retta data passante per un punto.

`Parallel` Crea la retta parallela ad una retta data passante per un punto.

`Point` Crea un *punto libero* date le coordinate della sua posizione iniziale.

`PointOn` Punto disegnato su un oggetto in una posizione fissa.

`Polygon` Crea un poligono data una sequenza di vertici.

`Polygonal` Poligonale data una sequenza di vertici.

Ray Traccia una semiretta con l'origine in un punto e passante per un altro punto.

Segment Crea un segmento dati i due estremi, i due estremi sono *punti*.

Text Crea un testo posizionato in un punto del piano.

VarText Crea un testo variabile. Il testo contiene dei «segnaposto» che verranno sostituiti con i valori prodotti dai dati presenti nel parametro variables.

VarLabel Testo variabile legato ad un oggetto.

Vector Testo variabile legato ad un oggetto.

15.27.3 Attributi

<oggetto_visibile>.color Attributo degli oggetti geometrici: imposta il colore dell'oggetto;

<oggetto_visibile>.name Attributo degli oggetti geometrici: imposta il nome dell'oggetto.

<oggetto_visibile>.visible Attributo degli oggetti geometrici: stabilisce se l'oggetto sarà visibili o invisibile;

<oggetto_visibile>.width Attributi degli oggetti geometrici: imposta la larghezza dell'oggetto.

<piano_interattivo>.defvisible Attributo del piano: stabilisce se i nuovi oggetti creati saranno, in modo predefinito, visibili o invisibili;

<piano_interattivo>.defwidth Attributo del piano: imposta la larghezza predefinita;

<piano_interattivo>.defwidthtext Attributo del piano: imposta la larghezza predefinita dei caratteri;

<piano_interattivo>.defcolor Attributo del piano: imposta il colore predefinito degli oggetti;

<piano_interattivo>.defdragcolor Attributo del piano: imposta il colore predefinito per gli oggetti che

si

possono trascinare con il mouse.

<piano_interattivo>.defintcolor Attributo del piano: imposta il colore predefinito per la superficie di circonferenze e poligoni.

15.27.4 Metodi

<circonferenza>.radius Metodo delle classi *circonferenza* che restituisce un oggetto

data

che contiene la lunghezza del raggio della circonferenza.

<figura>.perimeter Metodo delle classi *figura* che restituisce un oggetto `data` contenete la lunghezza del contorno dell'oggetto.

<figura>.surface Metodo delle classi *figura* che restituisce un oggetto `data` contenete l'area della superficie dell'oggetto.

<linea_retta>.equation Metodo delle classi *linea* che restituisce un oggetto `data` contenete l'equazione esplicita della retta p_a cui appartiene l'oggetto.

<linea_retta>.point1 Metodo delle classi *linea* che restituisce il *punto1*.

<linea_retta>.slope È il metodo presente in tutte le classi *linea* che restituisce la pendenza della retta o della retta che p_a cui appartiene l'oggetto.

<oggetto>.type Restituisce il tipo dell'oggetto.

<oggetto_visibile>.coords Restituisce un dato che contiene le coordinate.

<oggetto_visibile>.point0 Metodo delle classi *linea* che restituisce il *punto0*.

<oggetto_visibile>.xcoord Metodo degli oggetti visualizzabili: restituisce un dato che contiene l'ascissa.

<oggetto_visibile>.ycoord Metodo degli oggetti visualizzabili: restituisce un dato che contiene l'ordinata.

<punto_legato>.parameter Metodo dei punti legati agli oggetti che restituisce un oggetto `data` contenete il parametro.

<segmento>.length Metodo della classe `Segment` che restituisce un oggetto `data` contenete la lunghezza del segmento stesso.

<segmento>.midpoint Metodo della classe `Segment` che restituisce il punto medio del segmento.

CAPITOLO 16

Indici e tavole

- genindex
- modindex
- search