

RoboFab 1.1

Guida pratica

Premessa

La presente guida va intesa come un riferimento non ufficiale per l'utilizzo di RoboFab. Per ogni tipo di informazione si raccomanda sempre di fare riferimento alla documentazione ufficiale di RoboFab (consultabile all'indirizzo Internet *robofab.org*) e di FontLab (*www.fontlab.com*).

1. Che cos'è RoboFab

Dal sito ufficiale robofab.org:

“RoboFab è una libreria di Python costruita per dialogare con informazioni generalmente associate ai caratteri e al loro disegno.”

Che cosa significa tutto ciò? Andiamo con ordine...

Questa guida parlerà principalmente di scripting. Il termine scripting viene spesso inteso come “la scrittura di piccoli programmi”, evitando così il termine più spaventoso e complicato di programmazione. Chiaramente scripting è programmazione e noi useremo un linguaggio di programmazione, Python, per scrivere degli script (dei codici) che verranno eseguiti da un interprete di Python. Si tratta di un programma che cercherà di eseguire le vostre istruzioni, avvisandovi qualora si presentassero problemi o interruzioni inaspettate. I problemi non vi devono spaventare, Python vi segnalerà esattamente quale parte del codice ha causato l'inconveniente, così da sapere precisamente dove intervenire nella correzione. Vedremo più avanti come interpretare e capire gli avvisi che Python ci mostrerà.

Sì, ma i caratteri?

Scrivere script non fa parte del disegno dei caratteri. I codici non vi aiuteranno a trovare nuove idee o a disegnare meglio una curva. Potranno invece fornirvi degli utili strumenti per semplificare alcuni aspetti.

Python è un linguaggio di programmazione opensource, interpretato ed orientato agli oggetti. Cerchiamo di spiegare questi concetti anche a chi non ha mai avuto nozioni di programmazione.

Ogni programma viene sviluppato scrivendo il codice in un opportuno linguaggio con una sua sintassi definita. Alcuni linguaggi hanno bisogno di una compilazione, cioè un'operazione che trasforma il codice utilizzato in un programma eseguibile direttamente dal pc.

Un linguaggio interpretato invece, consente di salvare direttamente il codice così com'è, per essere poi letto da un interprete, un programma che gira sul sistema operativo in uso. Questo significa che non viene generato un file eseguibile dal file con il nostro codice, ma, di volta in volta, il nostro file viene interpretato. L'HTML può essere considerato un linguaggio interpretato. Infatti il nostro browser (Internet Explorer, FireFox...) legge il codice html e lo interpreta al momento, restituendoci la pagina web.

Probabilmente l'invenzione più utile nella programmazione sin dai tempi delle schede perforate è stata l'introduzione della programmazione ad **oggetti**. Il termine è usato per descrivere un modo di programmare in cui i dati su cui state lavorando e il codice relativo ad essi restano uniti. Si tratta di un modo molto utile per organizzare tutti i codici e i dati evitando di ritrovarsi con un'eccessiva quantità di dati difficile da gestire. Com'è fatto un oggetto e cosa ci potete fare è scritto nella sua **classe**, che

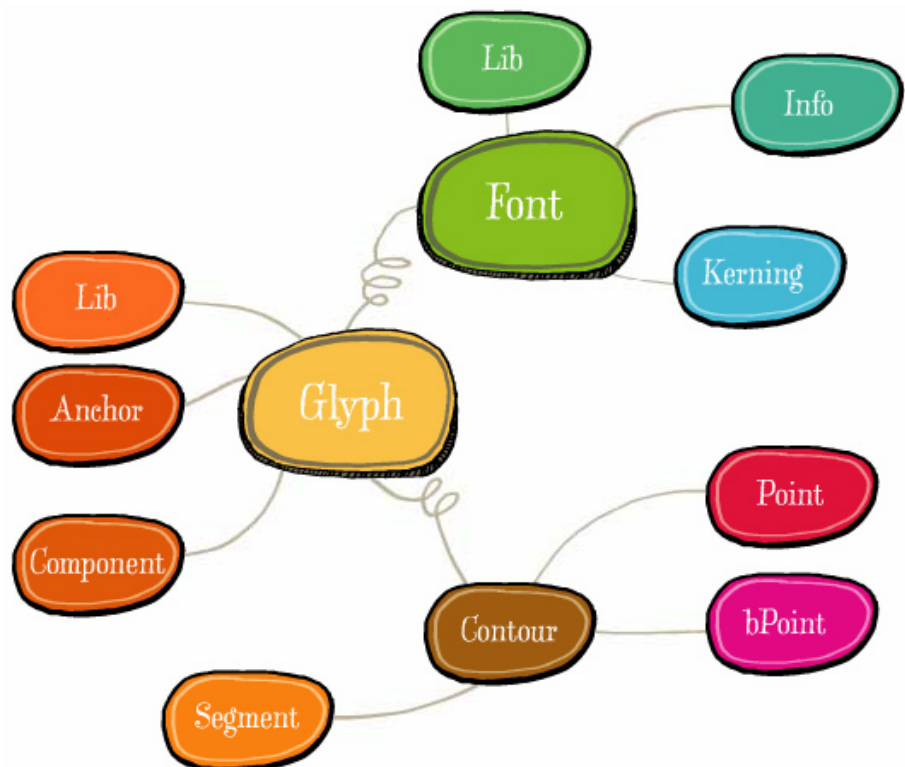
possiamo intendere come una vera e propria descrizione.

Nella terminologia esatta, un oggetto possiede:

- **attributi:** le cose che conosciamo di un oggetto, i suoi dati o valori. Ogni informazione relativa ad un oggetto è memorizzata nei suoi attributi. Gli attributi ci permettono di accedere ad una serie di informazioni che possono essere di sola lettura, (senza possibilità di modifica) o di lettura/scrittura (modificabili).
- **metodi:** le cose che un oggetto può fare. Il codice per manipolare un oggetto, le sue funzioni.

Azzardando un esempio: un oggetto della classe automobile può avere un attributo colore e un metodo guida().

Oggetti di grandi dimensioni (che contengono cioè grandi quantità di informazioni) sono generalmente suddivisi in oggetti più piccoli e specifici. L'oggetto Font per esempio permette di accedere all'oggetto Glyph. Il modo in cui i vari oggetti si relazionano tra essi, quale oggetto contiene cosa, il modo in cui vengono astratti, viene chiamato modello. Qui sotto trovate la mappa del modello degli oggetti utilizzati in RoboFab.



Tutte le informazioni necessarie per un corretto funzionamento degli oggetti, dei loro attributi e metodi sono contenute all'interno di un sistema di file che prende il nome di **libreria**. Nei file di una libreria troviamo tutte le informazioni che descrivono le classi, le funzioni e le relazioni che le legano. Insomma, tutto ciò che serve per

utilizzare gli oggetti.

Importando semplicemente la libreria all'interno di un codice, potremmo utilizzarne gli oggetti senza per questo dover riscrivere la loro parte di codice.

Per meglio capire il funzionamento di una libreria, possiamo paragonarla ad una piccola biblioteca, dove le informazioni sono già scritte all'interno dei libri. Noi possiamo usufruirne senza dover per questo riscriverle ogni volta.

Dopo questa premessa possiamo capire meglio cos'è RoboFab e a che cosa ci servirà. RoboFab è una libreria di Python che ci permette di riconoscere, modificare e gestire ogni aspetto dei caratteri tipografici digitali. Il suo grosso vantaggio sta nel poter fare tutto ciò attraverso il codice, trattando le font, i glifi etc.. come oggetti informatici.

Tutto questo non deve mai prescindere dalla forma dei caratteri, ma al contrario, deve essere inteso come un supporto al loro disegno.

RoboFab può lavorare sia con un suo specifico formato di file, l'Unified Font Object (.ufo), che interfacciarsi con altri software per il disegno dei caratteri. Nel nostro caso prenderemo in considerazione il suo utilizzo all'interno di FontLab 5.0.

Ciò significa che una volta installato RoboFab, esso è in grado di dialogare con le font su cui state lavorando all'interno di FontLab, manipolandole e gestendole attraverso il suo codice.

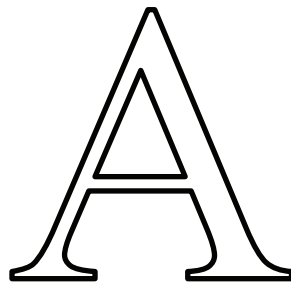
Il modello degli oggetti

Nella figura della pagina precedente abbiamo visto la struttura del modello di RoboFab. Conviene approfondire subito questa gerarchia, in quanto la sua comprensione sta alla base di un buon utilizzo della libreria. Possiamo riassumere così:

- una Font è un insieme di Glifi
- un Glifo è formato da Contorni
- un Contorno è formato da Punti e Segmenti

Font, Glifo, Contorno, Segmento e Punto sono tutti oggetti differenti. Quello che è importante capire qui è che l'accesso a tali oggetti, cioè il loro utilizzo, è possibile solo se conosciamo il percorso che ci porta ad essi. In breve, non potrò accedere ad un singolo segmento, per esempio, senza passare per la font, il glifo e il contorno che lo contengono. Questo perché una font possiede più glifi, un glifo più contorni... e così via. Inoltre ogni oggetto contiene degli attributi specifici che prescindono dai sottoggetti contenuti in essi. (Ad esempio il nome di un glifo non varierà in base ai contorni che contiene).

In ogni caso vedremo meglio con gli esempi successivi il funzionamento di questo modello.



questo è un glifo
(della font)



◀ questo è un contorno
(del glifo)



◀ questo è un segmento
(del contorno)



◀ questo è un punto
(del contorno)

2. Script in Python

Premettiamo che gli script (codici) che andremo a scrivere possono essere scritti su qualsiasi editor di testo esterno, (per es. Bloc Note su Windows o TextEdit su Mac), salvati con estensione `.py` e richiamati dentro FontLab. Tuttavia, il vantaggio di scrivere il codice direttamente all'interno di FontLab è che così facendo si ha un primo controllo sulla sintassi del codice, in quanto FontLab riconosce i comandi scritti in Python. Attraverso l'highlight syntax, i nostri comandi verranno infatti colorati in base alla categoria a cui appartengono.

Perciò, da ora in poi consideriamo il pannello Macro di FontLab il luogo predefinito dove scrivere i codici.

Il Pannello Macro



Qui sopra potete vedere il pannello Macro di FontLab 5.0. Per mostrare o nascondere il pannello, andate su `Window > Panels > Edit Macro`. Il suo utilizzo è molto semplice. Possiede un menù principale con le seguenti funzioni:

New Program

Apri un nuovo documento vuoto su cui lavorare. Poiché il pannello Macro può aprire uno script alla volta, dovremmo salvare e chiudere il precedente se non vorremmo perderlo.

Open Program

Apri uno script presente su computer, con estensione `.py`.

Save Program

Salva lo script corrente.

Save Program As...

Per salvare lo script corrente con un nome differente.

Show Line Numbers

Mostra o nasconde i numeri di linea del nostro script. I numeri di linea possono tornare utili per individuare in fretta un errore quando Python ci comunica la linea (riga) che l'ha provocato.

Close Panel

Nasconde il pannello Macro. Questa operazione non elimina il codice presente nel pannello.

Accanto a questo menù troviamo le icone:



Run macro

Per eseguire lo script corrente



Stop macro

Per interrompere lo script in esecuzione.

È attivabile durante l'esecuzione dello script.



Refresh macro system

Riavvia il sistema per le macro. (Di poco interesse per noi)

Tutte le stampe a video che faremo e gli avvisi di Python compariranno all'interno del pannello Output, che comparirà automaticamente se necessario.

La sintassi

Gli oggetti, gli attributi e i metodi seguono la sintassi del punto. Essa rappresenta un pratico sistema di riferimento per accedere al metodo o all'attributo che volete utilizzare. Altri linguaggi utilizzano la sintassi del punto, come JavaScript o PHP.

```
# attributi
unOggetto.unAttributo
unOggetto.unAltroAttributo
font.path
glyph.width

# metodi
unOggetto.unMetodo(unParametro)
unOggetto.unAltroMetodo()
font.generate()
glyph.clear()
```

Notate come il punto connetta i nomi. Questo sistema funziona anche più in profondità. A volte infatti gli oggetti contengono altri oggetti, che a loro volta ne contengono altri... Non abbiate paura di perdervi tra gli oggetti, la documentazione è qui apposta.

```
# attributi
# unOggetto.unAltroOggetto.unAttributoAltroOggetto
font.info.fullName
font.info.familyName

# metodi
# unOggetto.unAltroOggetto.unMetodoAltroOggetto(unParametro)
font.kerning.update()
font.info.autoNaming()
```

Avrete notato che in alcune righe di codice sono presenti delle parentesi, mentre in altre no. Perché? Scrivere le parentesi () significa che avete intenzione di usare un metodo e di eseguirlo. In altre parole eseguite una chiamata del metodo.

```
# un metodo NON richiamato: state solamente guardando
# l'oggetto Python che contiene quel metodo
font.update

# richiamare un metodo è molto più utile,
# significa: prendi quel codice ed eseguillo!
font.update()
```

Nomi delle variabili e dei metodi

In RoboFab esistono alcune convenzioni per quanto riguarda i nomi delle classi, degli attributi, dei metodi e delle funzioni. Questo rende più facile prevedere cosa viene richiamato e mantenere un certo ordine all'interno del codice:

- notazioneCamel: significa che quando un nome è formato da alcune parole, ogniParolaSuccessivaCominciaConLaMaiuscola. ad esempio: glyphName, kernTable, groupList, fontTools.
- i nomi degli attributi e dei metodi cominciano sempre con una minuscola, poi si segue la notazioneCamel. ad esempio: glyph.drawPoints()

Nota: queste convenzioni sono interne a RoboFab, tuttavia non significa che siano indispensabili per Python. I nomi delle vostre variabili, funzioni od altro, potranno seguire qualsiasi regola, in linea con quelle generali di Python:

- i nomi possono avere lunghezza arbitraria
- i nomi possono contenere numeri, lettere e underscore “_”
- il primo carattere deve essere una lettera
- i nomi possono contenere minuscole e maiuscole
- maiuscole e minuscole vengono riconosciute come diverse (case-sensitive: casa è diverso da Casa)
- lo spazio non è ammesso

Commenti

Possiamo inserire dei commenti all'interno di uno script, ovvero righe che non saranno eseguite dall'interprete ma che serviranno a noi come promemoria o per disabilitare, senza cancellarle, alcune parti di codice.

I commenti in Python devono essere preceduti dal simbolo ‘#’. Ad esempio:

```
# prova commento
# commento ignorato da Python
```

Un'ultima considerazione sulla sintassi. Python è molto preciso. Minuscole / maiuscole, spazi, parentesi... basta una di queste cose scritte nel modo sbagliato per bloccare l'intero script. Fate molta attenzione e controllate sempre per prima cosa la sintassi.

Detto questo possiamo provare a scrivere i nostri primi script.

3. Inizio

Ci siamo, anche se tutto non vi è chiaro al 100%, con questi primi esercizi vedremo di mettere in pratica quello di cui abbiamo parlato finora.

Un'ultima precisazione prima di cominciare. Gli script che scriviamo equivalgono a dei comandi. Ogni volta che eseguiamo uno script questo svolgerà una serie di funzioni con effetti immediati. Una sua successiva esecuzione prenderà in considerazione lo stato aggiornato delle cose. Alcuni script, esattamente come alcuni comandi, se eseguiti più volte restituiscono sempre uno stesso risultato. Per esempio stampare a video qualcosa. Per altri invece non è così. Pensiamo per esempio alla rotazione di un contorno. Se eseguiamo più volte lo script, il contorno verrà ruotato ogni volta, le rotazioni cioè si sommeranno!

Ok, adesso possiamo davvero cominciare!

- Se non l'avete ancora fatto... installate RoboFab! (Vedete le appendici A e B in fondo alla guida per installare RoboFab sia su Pc che Mac).
- Se non è ancora aperto, apriamo FontLab 5.0
- Non conviene mai lavorare sulle font originali, ma è meglio sempre creare una copia di sicurezza dei file e lavorare su quella, per non rischiare di perdere le font (alcuni comandi di RoboFab infatti sono irreversibili!).


Sotto Windows le font di sistema si trovano nella cartella

C:\windows\Fonts\.

Sotto Mac OS X le font si trovano in hd/Libreria/Fonts.

Copiate una font a vostro piacere e incollatela in una nuova cartella.

- Apriamo la font di prova, dal menù File > Open, cercandola dentro la nuova cartella appena creata.
- FontLab aprirà una finestra con tutti i glifi della font. Attiviamo il pannello per le macro, su cui andremo a scrivere i nostri codici. Per fare ciò andiamo su windows > Panels > Edit Macro.

Oppure selezioniamo questa icona 

Ok. Per prima cosa controlliamo che non ci siano stati problemi nell'installazione. Nella finestra Macro scriviamo, tutto in minuscolo:

```
import robofab
```

L'istruzione `import` dovrebbe colorarsi di blu. Questo, come dicevamo prima, per aiutarvi nella comprensione del codice (`import` è infatti una parola chiave).

Per eseguire lo script clicchiamo sull'icona Run Macro 

Se Python è installato in modo corretto, FontLab non segnalerà alcun errore. Se all'interno della finestra Output dovesse apparire un messaggio simile al seguente:

```
Traceback (most recent call last):
  File "<string>", line 1, in ?
ImportError: No module named robofab
```

significherebbe che l'installazione di RoboFab non è stata effettuata correttamente o che la cartella di RoboFab è stata spostata. Python infatti non riesce a trovare alcuna libreria con quel nome.

Provate a ripetere l'esecuzione del file `install.py` (che trovate nella cartella RoboFab) all'interno di PythonWin se siete sotto Windows, o da Terminale sotto Mac Os X.

Che cosa abbiamo fatto?

Con il comando precedente abbiamo segnalato a Python che andremo ad utilizzare la libreria RoboFab. Senza questa semplice riga, Python non saprebbe dove andare a trovare i comandi che scriviamo e che si riferiscono a RoboFab. Se vi ricordate, una libreria è un insieme di classi e funzioni che svolgono dei compiti precisi, ogni volta che vogliamo utilizzare una funzione o una classe dobbiamo importarla. Tranquilli, la maggior parte delle nostre funzioni sta nella libreria `world`, inoltre per velocizzare questa operazione possiamo importare l'intera libreria, con un unico comando, invece dei singoli componenti, uno alla volta. Vediamo meglio di approfondire questi concetti nel prossimo esempio.

Scriviamo:

```
import robofab.world
from robofab.world import CurrentFont
print CurrentFont()
```

Esguiamo lo script.

All'interno del `Panel Output` apparirà un testo simile al seguente:

```
<RFont font for NomeFontApertaInFontLab>
```

Con queste tre righe di codice abbiamo importato una libreria di RoboFab, chiamata `world`, che contiene pressochè tutte le più importanti istruzioni che utilizzeremo nei nostri prossimi script. Nella seconda riga abbiamo però specificato a Python che useremo una specifica funzione, tra tutte quelle presenti dentro la libreria `world`, chiamata `CurrentFont`.

In questo modo avvisiamo prima Python delle effettive funzioni che si servono, semplificando ed accelerando l'importazione. Questo è possibile attraverso la coppia di istruzioni `from... import`.

CurrentFont()

La funzione `CurrentFont` è fondamentale poichè ci consente di interfacciarci con la font aperta in questo momento su FontLab e di restituirci un oggetto `Font` per la font che abbiamo di fronte.

Notate che mentre nell'importazione non sono presenti le parentesi tonde, aperte e chiuse, dopo il nome della funzione, durante la sua chiamata, nella terza riga, le parentesi compaiono.

La terza riga di codice non fa altro che stampare a video il risultato della funzione `CurrentFont()` che, come abbiamo detto, è il nome della font aperta. Per fare ciò

abbiamo utilizzato l'istruzione `print`, che stampa sempre a video, all'interno della finestra Output.

Notate che non viene stampato direttamente il nome della font.

Ogni volta che stampiamo direttamente un oggetto e non un suo attributo o metodo, Python ci restituisce il nome della classe dell'oggetto, in questo caso `RFont` seguita dal nome dell'oggetto.

Nello script precedente potevamo anche importare tutta la libreria `world`, pur utilizzando esclusivamente la funzione `CurrentFont`.

Il seguente codice restituisce infatti lo stesso risultato:

```
import robofab.world
from robofab.world import *
print CurrentFont()
```

L'asterisco sta per "tutta la libreria", in questo modo in un'unica riga di comando abbiamo importato tutte le funzioni presenti, compresa la nostra `CurrentFont`.

CurrentGlyph()

Per accedere invece al glifo aperto in FontLab utilizziamo la funzione `CurrentGlyph()` che restituisce un oggetto glifo per il glifo che abbiamo di fronte. Questa funzione risulta utile quando vogliamo scrivere uno script che elabori un singolo glifo e vogliamo ottenere un oggetto dedicato ad esso.

```
# aprite un glifo in FontLab prima!!
from robofab.world import CurrentGlyph
print CurrentGlyph()
```

Nota! Il glifo non deve essere necessariamente aperto, basta anche averlo selezionato nella tabella dei glifi.

Restituire un oggetto?!

Ok ora siamo riusciti a stampare sia il nome della font che del glifo aperti in FontLab, ma come facciamo ad usarli?

Per le funzioni precedenti, abbiamo detto che sono in grado di "restituire" un oggetto... Cosa significa? Proviamo ad eseguire questo script:

```
from robofab.world import CurrentFont
nostraFont = CurrentFont()
print nostraFont.path
```

Avrete sicuramente notato la presenza di `nostraFont`. Di cosa si tratta? Abbiamo semplicemente creato un nuovo oggetto Font chiamato `nostraFont` il quale punta alla font attualmente in uso, richiamata dalla funzione `CurrentFont`. In pratica `nostraFont` è diventato la nostra font attuale e tutto ciò che andremo a fare su di esso verrà eseguito sulla font aperta in FontLab. Questa è una bella comodità, infatti ora basterà richiamare `nostraFont` per accedere direttamente alla font attuale, senza passare più per la funzione `CurrentFont`.

Nella 3 riga troviamo un primo esempio di questo utilizzo. L'attributo `path`, che si

riferisce al percorso del file aperto, è accessibile direttamente da `nostraFont`. Chiaramente il nome `nostraFont` è del tutto arbitrario, in quanto si tratta di una variabile, o meglio, di un oggetto creato da noi. Esso vale solamente per questo script.

La stessa cosa può essere fatta per il glifo:

```
from robofab.world import CurrentGlyph
nostroGlifo = CurrentGlyph()
print nostroGlifo.name
```

Per comodità, nei codici successivi utilizzeremo spesso `font` come nome dell'oggetto `Font`. (Cercate di non confondervi!)

Alcuni attributi per l'oggetto Font

La maggior parte degli attributi dell'oggetto `Font`, soprattutto riguardanti nomi e dimensioni, sono contenuti dentro un sottooggetto di `Font`, chiamato `Info`. Vediamo i principali:

- `.info.familyName`: il nome della famiglia della font
- `.info.styleName`: il peso della font (Roman, Bold...)
- `.info.fullName`: il nome completo della font
- `.info.unitsPerEm`: altezza (intesa come la differenza tra la linea delle ascendenti e la linea delle discendenti)
- `.info.ascender`: altezza delle ascendenti
- `.info.descender`: altezza delle discendenti

```
from robofab.world import CurrentFont

font = CurrentFont()
# nomi
print font.info.familyName
print font.info.styleName
print font.info.fullName
# dimensioni
print font.info.unitsPerEm
print font.info.ascender
print font.info.descender
```

Notate che gli attributi non appartengono direttamente all'oggetto `font` ma al sotto-oggetto `info`.

Alcuni attributi della font sono modificabili e possiamo modificarli direttamente dagli script. Fate attenzione però, perchè andrete a modificarli nel file che avete aperto! In questo script, per esempio, assegniamo dei nuovi valori e li stampiamo immediatamente.

```
from robofab.world import CurrentFont

font = CurrentFont()
```

```
# nomi
font.info.familyName = "Mia Famiglia"
print font.info.familyName
font.info.styleName = "Roman"
print font.info.styleName
font.info.fullName = font.info.familyName + '-' + font.info.
styleName
print font.info.fullName

# dimensioni
font.info.ascender = 600
print font.info.ascender
font.info.descender = -400
print font.info.descender
font.update()
```

Il metodo `update()` è necessario per aggiornare la font ogni volta che andiamo a modificare qualcosa.

4. Glifi e contorni

Abbiamo visto le funzioni `CurrentFont` e `CurrentGlyph` che permettono di interfacciarsi con la font o il glifo attualmente aperti in FontLab. Ma una font possiede centinaia di glifi e altre informazioni, dovremmo aprirle una alla volta, per accedervi? Decisamente no.

Proviamo questo script:

```
from robofab.world import CurrentFont
font = CurrentFont()
print font['A']
print font['two']
```

il risultato sarà qualcosa di simile:

```
<RGlyph for nomeFont.A>
<RGlyph for nomeFont.two>
```

Attraverso le parentesi quadre possiamo accedere a qualsiasi glifo della font, inserendo il suo nome (il nome postscript) tra gli apici.

Se volete conoscere il nome dei glifi, selezionate 'Name' nella quarta cella sul fondo della finestra della font (quella con tutti i glifi). Di default dovrebbe essere 'Unicode'.

Notate che nello script precedente non abbiamo utilizzato la funzione `CurrentGlyph`, in quanto volevamo accedere ad un glifo particolare e non più a quello aperto in FontLab.

Per accedere ad un attributo di un singolo glifo non faremo altro che utilizzare la solita sintassi del punto:

```
from robofab.world import CurrentFont
font = CurrentFont()
print font['A'].name
```

Il ciclo for... in

Vediamo ora un sistema per accedere in modo pratico e veloce a tutti i glifi di una font. Scrivete il seguente script rispettando gli spazi:

```
from robofab.world import CurrentFont

font = CurrentFont()
for glyph in font:
    print glyph.name
```

Nello script precedente abbiamo realizzato la nostra prima iterazione. Il termine indica la presenza di un ciclo di istruzioni eseguito ripetutamente in base a determinate condizioni. Nel nostro caso abbiamo stampato il nome di tutti i glifi presenti nella font attuale. Analizziamo meglio il codice.

Per prima cosa creiamo l'oggetto font e gli assegniamo la font attuale.

```
font = CurrentFont()
```

Successivamente, con il blocco di istruzioni

```
for glyph in font:  
    print glyph.name
```

diciamo di stampare il nome di tutti i glifi presenti nell'oggetto `font`. Il nome `glyph` è arbitrario, esso rappresenta un oggetto glifo. Ma non uno in particolare, infatti esso si "sposta" da glifo a glifo grazie alla riga

```
for glyph in font:
```

che possiamo interpretare come "per tutti i glifi che trovi all'interno dell'oggetto `font` fai qualcosa...". Quel qualcosa sono le istruzioni successive contraddistinte da uno **spazio ad inizio riga**, che verranno eseguite tante volte quanti sono i glifi. Nel nostro caso la stampa a video del nome del glifo.

Generalizzando, l'istruzione `for . . in` prevede la seguente sintassi:

```
for (sotto-oggetto) in (oggetto più grande):  
    istruzione1  
    istruzione2  
    ...  
    istruzione3
```

Il *sotto-oggetto* prenderà il posto di tutti i sotto-oggetti presenti in *oggetto*. Ogni volta che termina il ciclo di istruzioni (`istruzione1`, `istruzione2`,...), si passerà al sotto-oggetto successivo e così via, eseguendo nuovamente il ciclo.

L'istruzione3 non verrà eseguita tutte le volte, ma solo quando sarà terminato l'intero ciclo `for`. Questo perchè non fa parte del ciclo, non avendo alcuno spazio davanti.

Se non vi è del tutto chiaro il funzionamento del ciclo `for`, non disperate, col tempo e con l'esperienza prenderete confidenza con questi concetti e questa sintassi. Non abbiate paura di fare errori, se siete indecisi lavorate su delle copie e provate!

Alcuni attributi dell'oggetto Glifo

Anche gli attributi dell'oggetto Glifo sono modificabili

- `.width`: la larghezza del glifo
- `.name`: il nome del glifo
- `.leftMargin`: la posizione del margine sinistro
- `.rightMargin`: la posizione del margine destro

```
from robofab.world import CurrentFont  
  
font = CurrentFont()
```

```

glyph = font['A']
glyph.width = 200
print glyph.width
glyph.leftMargin = 50
print glyph.leftMargin
glyph.rightMargin = 50
print glyph.rightMargin

glyph.unicode = 666
glyph.update()

```

Alcuni metodi dell'oggetto Glifo

`.move((x,y))`: sposta il glifo di *x* sull'asse orizzontale e di *y* sull'asse verticale (Valori relativi!!)

`.scale((scalax, scalay))`: scala il glifo orizzontalmente di un valore pari a *scalax* e verticalmente di un valore pari a *scalay*

`.rotate(angolo)`: ruota il glifo di un valore pari ad *angolo* (in gradi) (Valori relativi!!!)

`.update()`: aggiorna il glifo

```

from robofab.world import CurrentFont

font = CurrentFont()

# richiamiamo un glifo usando il suo nome
glyph = font['A']

# ora abbiamo un oggetto glifo
# richiamiamo alcuni metodi
glyph.move((100, 75))
glyph.scale((.5, 1.5))

glyph.update()

```

Contorni

Un glifo è costituito a sua volta da uno o più contorni. Possiamo definire un contorno come una linea chiusa, un tracciato. La *c* minuscola, per intenderci, è formata da un unico contorno, mentre la *a* è formata da due contorni, uno più esterno pieno e uno interno vuoto. Come si accede ai contorni?

Il principio è lo stesso dei glifi, tutti i contorni (che ricordiamo, sono sotto-oggetti dei glifi) sono accessibili con la sintassi delle parentesi quadre, ma a differenza dei glifi non vi si accede attraverso un nome ma con un numero intero progressivo:

```

from robofab.world import CurrentFont

font = CurrentFont()
glyph = font['A']
contour = glyph[0]
print contour.points

```

Nello script precedente l'oggetto `contour` è un oggetto `Contorno` e punta al contorno nr. `o` del glifo `glyph`, che a sua volta è un oggetto `Glifo` che punta al glifo `'A'` della font.

Successivamente stampiamo a video tutta la lista dei punti che formano il contorno `o`. I contorni infatti, in quanto linee sono formati da una serie di punti, ogni punto ha precise coordinate all'interno del glifo. Ah, chiaramente ogni punto è un sotto-oggetto del contorno.

Come per i glifi anche con i contorni possiamo utilizzare il ciclo `for` per accedere in fretta a tutti gli oggetti. Vediamo come:

```
from robofab.world import CurrentFont

font = CurrentFont()
glyph = font['A']
for p in glyph:
    print p
```

In questo modo otteniamo una stampa di tutti i contorni presenti nel glifo `A`. Con lo script successivo invece accediamo a tutti i punti del contorno `o`, del glifo `A` e ne stampiamo le coordinate `x` e `y`.

```
from robofab.world import CurrentFont

font = CurrentFont()
glyph = font['A']
contorno = glyph[0]
for p in contorno.points:
    print p.x, p.y
```

5. Disegniamo

Finora abbiamo visto come utilizzare RoboFab per osservare e manipolare oggetti già esistenti. Ma se volessimo disegnarne uno noi? Possiamo inserire e modificare dei nuovi tracciati? E dei nuovi glifi? Assolutamente sì.

Per fare tutto ciò abbiamo bisogno dell'oggetto Pen (pennino).

RoboFab mette a disposizione una serie di pennini, ognuno con una sua particolare funzione. Si tratta di oggetti in grado di riprodurre i tracciati dei glifi o di muoversi e disegnare da zero nuovi tracciati.

I pennini funzionano per spostamento e tracciamento. In pratica, prima decidiamo dove vogliamo che il nostro tracciato abbia origine, poi disegniamo le curve, indicando il prossimo punto di 'arrivo' e le caratteristiche della curva.

Ma andiamo con ordine.

Se vogliamo creare da zero la nostra nuova font, conviene aprirne una nuova e completamente vuota, andando, in FontLab, su `File > New`. Tuttavia possiamo aggiungere dei nuovi glifi ad una font già esistente, utilizzando il metodo `newGlyph()` dell'oggetto `Font` che ci permette di inserire, all'interno della font un nuovo glifo.

Per prima cosa dobbiamo richiamare un pennino. Possiamo utilizzare direttamente il metodo `getPen()` dell'oggetto `Glifo`. I pennini così richiamati sono strutturati apposta per disegnare dei tracciati dentro ai glifi. In questo esempio lavoriamo sul glifo selezionato in FontLab:

```
from robofab.world import CurrentGlyph
g = CurrentGlyph()
mioPennino = g.getPen()
```

A questo punto possiamo disegnare direttamente all'interno del nostro glifo utilizzando i comandi per spostare e tracciare.

Fate attenzione ai valori, sono tutti assoluti!

`.moveTo((x,y))`: sposta il cursore al punto (x,y) . Questo comando non traccia alcun segno, ma semplicemente posiziona il pennino nel punto in cui partirà il tracciato.

`.lineTo((x,y))`: traccia una linea retta sino al punto (x,y) , partendo dalla posizione attuale

`.curveTo((x1,y1), (x2,y2), (x3,y3))`: traccia una curva di Bezier cubica, dalla posizione attuale sino al punto $(x3,y3)$, utilizzando i punti $(x1,y1)$ e $(x2,y2)$ come punti di controllo.

`pCurveTo((x1,y1), (x2,y2), ..., (xz,yz))`: traccia una curva di Bezier quadratica sino al punto (xz,yz) con un numero arbitrario di punti di controllo.

`.closePath()`: chiude il tracciato che abbiamo disegnato, congiungendo il primo con l'ultimo punto. È obbligatorio, altrimenti non vedrete nulla!

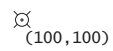
Cominciamo con qualcosa di semplice.

```
import robofab.world
from robofab.world import *
glifo = CurrentGlyph()
pen = glifo.getPen()
pen.moveTo((100,100))
pen.lineTo((600,100))
pen.lineTo((600,600))
pen.lineTo((100,600))
pen.lineTo((100,100))
pen.closePath()
glifo.update()
```

Con la funzione precedente abbiamo creato un quadrato all'interno del nostro glifo.
Vediamo i vari passaggi:

```
glifo = CurrentGlyph()
creiamo l'oggetto glifo che conterrà il glifo attuale

pen = glifo.getPen()
richiamiamo il pennino del glifo
```



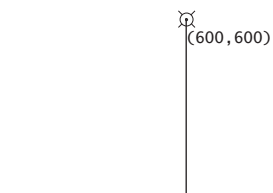
A small cursor icon (a circle with a cross) is positioned above the coordinates (100, 100).

```
pen.moveTo((100,100))
spostiamo il cursore al punto (100,100)
```



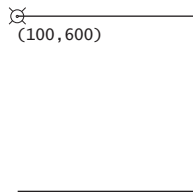
A horizontal line starts from the left and ends at a cursor icon positioned above the coordinates (600, 100).

```
pen.lineTo((600,100))
tracciamo una linea sino al punto (600,100)
```



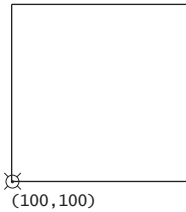
An L-shaped line is shown. It starts from the left, goes horizontally to the right to a cursor icon at (600, 100), and then continues vertically upwards to another cursor icon at (600, 600).

```
pen.lineTo((600,600))
tracciamo una linea verticale sino a (600,600)
```



```
pen.lineTo((100, 600))
```

tracciamo una linea verticale sino a (100,600)



```
pen.lineTo((100, 100))
```

tracciamo una linea per chiudere il quadrato a (100,100)

```
pen.closePath()
```

chiudiamo il tracciato. Avremmo potuto chiuderlo già ad un passaggio prima, evitando di disegnare noi l'ultima linea. Questo comando infatti unisce l'ultimo e il primo punto, chiudendo il contorno.

Abbiamo appena creato il primo contorno del nostro `gli fo`. Per accedere ad esso possiamo ricorrere alle parentesi quadre, utilizzando come indice il valore o in quanto si tratta del primo contorno.

```
gli fo[0].name='tracciato1'
```

In questo modo abbiamo appena nominato il nostro contorno.
Attenzione, non è il nome del glifo! Infatti possiamo nominare diversamente il glifo:

```
gli fo.name='quadrato'
print gli fo[0].name
print gli fo.name
```

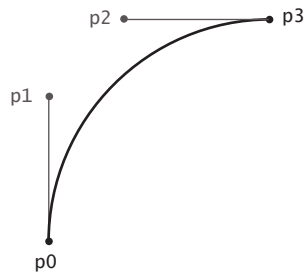
Curve di Bézier?!?

Nell'introdurre il metodo `curveTo` abbiamo accennato alle curve di Bézier. `curveTo` infatti permette di tracciare una curva di Bézier cubica. Di cosa si tratta?

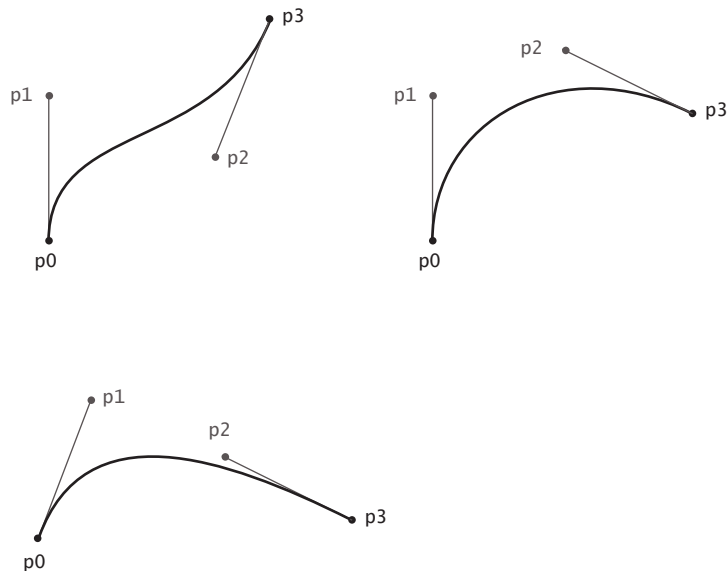
Una curva di Bézier cubica è una curva che prevede due punti di controllo.

Vediamo di capire.

Prendiamo la seguente curva:



Essa ha un punto di origine p_0 , un punto finale p_3 e i rispettivi punti di controllo p_1 e p_2 . Una curva con queste caratteristiche prende il nome di curva cubica di Bézier. Agendo sulla posizione dei punti di controllo e dei punti di origine della curva possiamo modificare il suo aspetto:

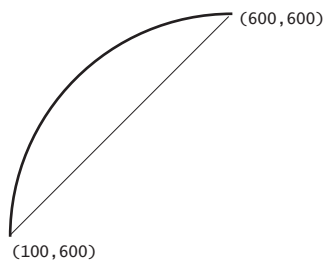


Come possiamo disegnarla con RoboFab?

Il metodo `curveTo()` ha tre parametri di ingresso, che rappresentano rispettivamente i punti p_1 , p_2 e p_3 . Il punto p_0 corrisponde all'attuale posizione del cursore. Per disegnare la curva non faremo altro che richiamare il metodo con dentro questi valori, espressi come punti.

```
import robofab.world
from robofab.world import *
glifo = CurrentGlyph()
pen = glifo.getPen()
pen.moveTo((100,100))
pen.curveTo((100,300), (300,600), (600,600))
pen.closePath()
glifo.update()
```

Lo script precedente traccia una curva cubica di Bézier con $p_0(100,100)$, $p_1(100,300)$, $p_2(300,600)$, $p_3(600,600)$



Appendice A

Installazione per PC (Windows 98 / Xp / Vista)

Per un corretto funzionamento RoboFab ha bisogno dei seguenti componenti:

- Compilatore di Python per Windows
- Pacchetto RoboFab dei LettError
- FontLab

Per prima cosa vediamo nel dettaglio i passi da seguire per installare questi componenti:

- Poiché RoboFab è scritto in linguaggio Python, la prima cosa da fare è installare un interprete Python.

È free ed opensource. Consigliamo di installare la versione 2.2.1, scaricabile dal sito di Python a questo indirizzo:

<http://www.python.org/ftp/python/2.2.1/Python-2.2.1.exe>

Questa versione tuttavia è command-line quindi, dopo averla installata conviene scaricare ed installare anche l'estensione per Windows a questo indirizzo:

http://downloads.sourceforge.net/pywin32/pywin32-210.win32-py2.2.exe?modtime=1159009204&big_mirror=0

- Il pacchetto di RoboFab si può scaricare dal sito dei LettError, al seguente link: http://robofab.org/download/current/RoboFab_1.1.1.PlusFontTools.zip dopo aver letto le note di licenza (<http://robofab.org/download/license.html>). Una volta scaricato il file .zip estraiamolo in una cartella con nome RoboFab (ad esempio C:\Programmi\RoboFab).

- Come versione di FontLab, per problemi di compatibilità con Python è preferibile la versione 5.0.

E' possibile scaricare una demo per Windows dal sito di FontLab, all'indirizzo:

<http://www.font.to/downloads/demos/FLS5WinDemo.exe>

Per ogni eventuale approfondimento o problema rimandiamo al link <http://robofab.org/install.html>

Una volta installati i componenti precedenti, vediamo come procedere. Verifichiamo che FontLab abbia riconosciuto l'installazione di Python, aprendo FontLab e andando su Edit > Properties.

Nella finestra che appare dovrebbe comparire la scritta Python is installed. Qualora Python non risulti installato, procedere a disinstallare e reinstallare Python.

Per installare RoboFab bisogna eseguire lo script `install.py`, che creerà il file `robofab.pth`. Questo file contiene il percorso dell'attuale posizione della libreria RoboFab, necessario per consentire a Python di localizzare correttamente i comandi. Per fare ciò:

- Apriamo l'estensione Pythonwin, installata nel punto 2. Questa applicazione si trova sotto `Start > Tutti i programmi > Python 2.2 > Pythonwin`.
Se avete difficoltà a trovare il programma provate a cercare sotto la cartella di Python, generalmente `C:\Programmi\Python22\Lib\site-packages\pythonwin\Pythonwin.exe`.
- All'apertura di Pythonwin troviamo una finestra con il prompt `>>>`. Andiamo su `File > Open` e cerchiamo il file `install.py`, sotto la cartella dove abbiamo installato RoboFab (vedi punto precedente). Apriamolo.
- Apparirà una nuova finestra chiamata `install.py`. Andiamo su `File > Run` e clicchiamo su OK all'eventuale apertura di una finestra, senza modificare nulla. Chiudiamo Pythonwin e riapriamolo.
- Posizioniamoci sulla finestra che compare (quella caratterizzata da `>>>`) e digitiamo, tutto in minuscolo:

```
import robofab
```

Premiamo invio. Se non compare alcun messaggio RoboFab è stato installato correttamente.

Per ogni problema rinviamo ai due siti

<http://robofab.org/index.html>

<http://just.letterror.com/ltrwiki/RoboFab>

e alla mail [type.polimi@gmail.com](mailto:polimi@gmail.com)

Appendice B

Installazione per Mac Os X

Per un corretto funzionamento RoboFab ha bisogno dei seguenti componenti:

- Compilatore di Python per Mac
- Pacchetto RoboFab dei LettError
- FontLab

Per prima cosa vediamo nel dettaglio i passi da seguire per installare questi componenti:

- Poiché RoboFab è scritto in linguaggio Python, la prima cosa da fare è avere un interprete Python. Per fortuna è assolutamente free e dovrebbe essere già installato con Mac Os X.

A questo punto possiamo passare a leggere il punto successivo, ma se vogliamo esserne sicuri possiamo aprire il Terminale (Applicazioni / Utility / Terminale) digitare `python` e poi invio.

Se si ha installato Python si otterrà una risposta positiva e il Terminale indicherà la versione installata. Come per esempio:

```
Python 2.3.5 (#1, Mar 20 2005, 20:38:20)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1809)] on
darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Se sulla macchina si ha installato Mac Os X Panther (10.3.x) la versione di Python sarà precedente (2.3.0). Se non si ha installato Python si può scaricarlo dal sito: <http://www.pythonmac.org/packages/> anche in questo caso aprire il disco immagine e procedere con l'installazione aprendo il file `MacPython.mpkg`

- Il pacchetto di RoboFab si può scaricare dal sito dei LettError, al seguente link http://robofab.com/download/current/RoboFab_1.1.1.PlusFontTools.dmg dopo aver letto le note di licenza (<http://robofab.org/download/license.html>). Una volta scaricato il disco immagine (un file .dmg) apriamolo e trasciniamo tutto il contenuto in una nuova cartella fuori dal disco immagine con nome RoboFab (per esempio ~/Documents/RoboFab).

- Come versione di FontLab, per problemi di compatibilità con Python è preferibile la versione 5.0. È possibile scaricare una demo per Mac dal sito di FontLab, all'indirizzo:
<http://www.font.to/downloads/demos/FLS5MacDemo.hqx>

Per ogni eventuale approfondimento o problema rimandiamo al link <http://robofab.org/install.html>

Una volta installati i componenti precedenti, vediamo come procedere. Verifichiamo che FontLab abbia riconosciuto l'installazione di Python, aprendo FontLab e andando su `Edit > Properties`. Nella finestra che appare dovrebbe comparire la scritta `Python is installed`. Qualora Python non risulti installato, procedere a disinstallare e reinstallare Python.

Per installare RoboFab bisogna eseguire lo script `install.py`, che creerà il file `robofab.pth`. Questo file contiene il percorso dell'attuale posizione della libreria RoboFab, necessario per consentire a Python di localizzare correttamente i comandi. Per fare ciò:

- Apriamo il Terminale. Scriviamo `cd` più uno spazio. Trascinare l'icona del file `install.py` nella finestra del Terminale.

Si otterrà una stringa di testo (il percorso del file, cioè dove si trova nelle cartelle) che potrebbe essere simile a questa:

```
cd /Users/nomeutente/Documents/RoboFab/install.py
```

non schiacciamo invio ma cancelliamo `install.py` in modo da ottenere `cd /Users/nomeutente/Documents/RoboFab/`

Diamo l'invio.

- A questo punto digitiamo: `python install.py`

Se tutto è andato bene avremmo un risultato analogo a questo:

```
Installing RoboFab: about to write a path to '/Users/nomeutente/Documents/RoboFab/Lib' in
'/System/Library/Frameworks/Python.framework/Versions/2.3/
lib/python2.3/sitepackages/robofab.pth'...
Robofab is now installed.
(Note that you have to run the install script again if you
move your RoboFab folder)
```

Ora abbiamo installato RoboFab. Chiudiamo il terminale.

- Apriamo Fontlab. Richiamiamo la finestra Macro dal menu: `window > Panels > Edit Macro`. Posizioniamoci sulla finestra che compare e digitiamo, tutto in minuscolo:

```
import robofab
#commento
```

Premiamo invio. Se non compare alcun messaggio RoboFab è stato installato correttamente!

Per qualsiasi problema rinviamo ai due siti

<http://robofab.org/index.html>

<http://just.lettererror.com/ltrwiki/RoboFab>

e alla mail type.polimi@gmail.com

Appendice C

Elenco (incompleto) dei metodi e degli attributi dei principali oggetti.

Font

L'oggetto Font contiene tutti i glifi, le informazioni sulla nominazione e sul kerning. Per accedere ad esso possiamo utilizzare la funzione `CurrentFont()` che restituisce la font correntemente aperta in FontLab:

```
from robofab.world import CurrentFont
font = CurrentFont()
```

Attributi

`.path`

Il percorso del file aperto (sola lettura).

`.kerning`

L'oggetto Kerning, contenente tutte le informazioni per i kerning della font.

`.info`

L'oggetto Info con tutti i nomi della font e le dimensioni delle chiavi.

`.fileName`

Il nome e il percorso della font.

Metodi

`nomeFont[nomeGlifo]`

Richiede alla font un glifo. L'accesso avviene con il nome del glifo.

`.has_key(nomeGlifo)`

Restituisce True se tra la font esiste il glifo *nomeGlifo*

`.keys()`

Restituisce una lista di tutti i nomi dei glifi della font

`.newGlyph(nomeGlifo, clear=True)`

crea un nuovo glifo vuoto chiamato *nomeGlifo*. Se viene impostato `clear=True` il glifo verrà sovrascritto qualora già esistesse. Di default è *True*.

`.removeGlyph(nomeGlifo)`

Rimuove un glifo dalla font.

`.insertGlyph(unGlifo, as=None)`

Inserisce *unGlifo* nella font e restituisce l'oggetto Glifo relativo. Se la font possiede già un glifo con lo stesso nome, quest'ultimo verrà sostituito. Il parametro *as*

viene usato per dare un nome alternativo al glifo.

`.compileGlyph(nomeGlifo, nomeBase, nomiAccenti, adjustwidth=False, preflight=False, printErrors=True)`

Compila i componenti dentro un nuovo glifo utilizzando componenti e punti di ancoraggio. *nomeGlifo* è il nome del glifo in cui dovrà convogliare il tutto. *nomeBase* è il nome del glifo base. *nomiAccenti* è una lista di nomi di Accenti [['acute','top']].

`.generateGlyph(nomeGlifo, replace=True, preflight=False, printErrors=True)`

Genera un glifo e lo restituisce. Con `replace = True` la font sostituirà il glifo qualora esistesse già. Con `preflight = True` la font tenterà di generare il glifo senza però aggiungerlo alla font. (Utile per testare la possibilità di creare il glifo).

`.save(destDir=None, doProgress=False, saveNow=False)`

Salva la font.

`.autoUnicode()`

Assegna i codici Unicode automaticamente.

`.round()`

Arrotonda tutte le coordinate del glifo a dei numeri interi.

Per esempio il punto (12.3, -10.99) diventerà (12,11).

`.update()`

Aggiorna la font in FontLab. Potete richiamarlo dopo una serie di manipolazioni.

`.copy()`

Restituisce una copia della font. Tutti i glifi e tutte le informazioni associate verranno duplicate.

`.getCharacterMapping()`

Restituisce un dizionario con i valori Unicode e i nomi dei glifi.

`.naked()`

Restituisce l'oggetto Font di FontLab. Può tornare utile per settare alcuni valori specifici in FontLab che non sono gestiti negli oggetti RoboFab.

`.close()`

Chiude la finestra 'Font' di FontLab.

`.appendHGuide()`

Inserisce una guida orizzontale.

`.appendVGuide()`

Inserisce una guida verticale.

Glifo

L'oggetto Glifo permette di accedere ad un glifo della font. L'accesso è possibile attraverso il nome del glifo, da un oggetto Font, secondo la sintassi:

```
nomefont['nomeGlifo']
```

o in modo diretto, attraverso la funzione `CurrentGlyph()` per accedere al glifo attualmente aperto in FontLab:

```
from robofab.world import CurrentFont  
glifo = CurrentGlyph()
```

Attributi

`.components`

La lista dei componenti del glifo.

`.anchors`

La lista dei punti di ancoraggio.

`.len(aGlyph)`

Il numero dei contorni presenti nel glifo.

`glifo[indice]`

Restituisce l'oggetto Contorno *indice* presente nel *glifo*.

`.width`

La larghezza del glifo.

`.leftMargin`

Il margine sinistro del glifo.

`.rightMargin`

Il margine destro del glifo.

`.name`

Il nome del glifo.

`.box`

La 'scatola' che racchiude il glifo. I valori sono (xMin, yMin, xMax, yMax). Rappresentano i limiti attuali della forma del glifo.

Metodi

`.getParent()`

Restituisce il 'genitore' del glifo, cioè l'oggetto Font a cui appartiene.

`.appendContour(unContorno)`

Aggiunge *unContorno* al glifo.

`.appendGlyph(unGlifo)`

Aggiunge un intero glifo. Vengono aggiunti tutti i contorni, gli ancoraggi e i componenti al glifo.

`.appendAnchor(nome, posizione)`

Crea un nuovo punto di ancoraggio al glifo con *nome* e *posizione*.

`.removeAnchor(ancoraggio)`

Rimuove il punto *ancoraggio* dal glifo.

`.autoUnicode()`

Tenta di trovare i valori Unicode per questo glifo. Cerca di trovare una corrispondenza tra il nome del glifo e un valore conosciuto.

`.copy()`

Restituisce una copia del glifo. La copia comprende tutte le parti del glifo.

`.correctDirection()`

Corregge la direzione di tutti i contorni del glifo.

`.autoContourOrder()`

Ordina automaticamente i contorni in base: al punto di contorno, al segmento di contorno, al valore della x del centro, al valore y del centro del contorno e l'area del rettangolo del glifo.

`.pointInside((x,y))`

Restituisce *True* se il punto è all'interno dell'area 'nera' del glifo o *False* se il punto si trova all'interno dell'area 'bianca'.

`.draw(unPennino)`

Il glifo disegna se stesso utilizzando *unPennino*.

`.drawPoints(unPenninoPerPunti)`

Il glifo disegna se stesso con *unPenninoPerPunti*.

`.getPen()`

Restituisce un oggetto Pennino già settato per disegnare sul glifo.

`.getPointPen()`

Restituisce un oggetto `PenninoPerPunti` già settato per disegnare sul glifo.

`.interpolate(fattore, minGlifo, maxGlifo, suppressError=True, analyzeOnly=False)`

Trasforma il glifo nell'interpolazione tra `minGlifo` e `maxGlifo` attraverso un `fattore`. Con `suppressError = True` questo metodo non considera se l'interpolazione sia fattibile. Con `analyzeOnly = True` il metodo analizzerà solamente se l'interpolazione è possibile e provvederà a mostrarci gli eventuali errori.

`.isCompatible(altraGlifo, report=True)`

Restituisce `True` se il glifo ha una struttura dei punti compatibile con un `altraGlifo`. Se `report = True`, restituisce un report con gli eventuali problemi.

`.isEmpty()`

Restituisce `True` se il glifo non contiene alcun contorno, componente o ancoraggio.

`.move((x,y), contours=True, components=True, anchors=True)`

Sposta un glifo o solamente parte di esso. Di default vengono compresi nello spostamento i contorni, i componenti e i punti di ancoraggio.

`.scale((x,y), center=(o,o))`

Scala il glifo di `x` e `y`. Opzionale è la scelta del centro della scala.

`.rotate(angolo, offset=None)`

Ruota il glifo di un `angolo` (in gradi). Opzionale la scelta di un valore `offset`.

`.skew(angolo, offset=None)`

Inclina il glifo di un `angolo`. Opzionale la scelta di un valore `offset`.

`.rasterize(cellSize=50, xMin=None, yMin=None, xMax=None, yMax=None)`

Taglia il glifo lungo una griglia basata sul valore di `cellSize`. Restituisce una lista contenente valori booleani che indicano se ogni cella è nera (`True`) o bianca (`False`). La lista sono organizzate dall'alto verso il basso del glifo e procedono da destra a sinistra. È un'operazione molto elaborata!

`.removeOverlap()`

Rimuove delle sovrapposizioni nel glifo.

`.naked()`

Restituisce l'oggetto Glifo di `FontLab`.

`.update()`

Aggiorna la font in `FontLab`.

Potete richiamarlo dopo una serie di manipolazioni. Potrebbe rallentare lo script.

`.getVGuides()`

Restituisce la lista delle guide verticali per l'oggetto Glifo.

`.getHGuides()`

Restituisce la lista delle guide orizzontali per l'oggetto Glifo.

`.appendVGuide()`

Aggiunge una guida verticale al glifo.

`.appendHGuide()`

Aggiunge una guida orizzontale al glifo.

`.clearVGuides()`

Rimuove le guide verticali dal glifo.

`.clearHGuides()`

Rimuove le guide orizzontali dal glifo.

Contorno

Il contorno è identificabile come un tracciato chiuso, all'interno del glifo. Vi si accede attraverso la sintassi:

`nomeGlifo['indiceContorno']`

Dal contorno è possibile accedere ai suoi punti e ai suoi segmenti.

Attributi

`.index`

L'indice del contorno all'interno del Glifo. È indispensabile per accedere al contorno.

`.selected`

Restituisce '1' se il contorno è selezionato o '0' se non lo è.

`.box`

Il rettangolo che contiene il contorno (sola lettura)

`.clockwise`

La direzione del contorno: '1' se in senso orario o '0' se antiorario

`.points`

La lista dei punti del contorno.

Metodi

`.reverseContour()`

Ribalta la direzione del contorno

`.copy()`

Duplica il contorno.

`.draw(unPennino)`

Disegna il contorno con *unPennino* di RoboFab per i segmenti

`.drawPoints(unPenninoPunti)`

Disegna il contorno con *unPenninoPunti*.

`.move((x,y))`

Muove il contorno di *x* sull'asse orizzontale e di *y* su quello verticale

`.pointInside((x,y), evenOdd=o)`

Determina se il punto *(x,y)* si trova all'interno o all'esterno del contorno.

`.round()`

Arrotonda tutti i punti del contorno ad interi.

`.scale((x,y), center=(o,o))`

Scala il contorno di *x* e *y*. Opzionale è la scelta del centro della scala.

`.rotate(angolo, offset=None)`

Ruota il contorno di un *angolo* (in gradi). Opzionale la scelta di un valore *offset*.

`.skew(angolo, offset=None)`

Inclina il contorno di un *angolo*. Opzionale la scelta di un valore *offset*.

`.transform(matrice)`

Trasforma il contorno. Usa una *matrice* di trasformazione (oggetto).

`.appendSegment(tipoSegmento, points, smooth=False)`

Aggiunge un segmento al contorno.

`.insertSegment(indice, tipoSegmento, points, smooth=False)`

Inserisce un segmento all'interno del contorno.

`.removeSegment(indice)`

Rimuove il segmento *indice* dal contorno.

`.setStartSegment(indiceSegmento)`

Definisce *indiceSegmento* come il primo nodo del contorno.

