

# Guida allo scripting in FreeCAD

Autore: Carlo Dormeletti

Versione di riferimento di **FreeCAD**: **0.18**

Versione della guida: **0.20**

Data di stampa: **11 marzo 2020**

Distribuito sotto Licenza **CC BY-NC-ND 4.0 IT**

## Licenza

Distribuito sotto licenza **CC BY-NC-ND 4.0 IT**- vedi

<https://creativecommons.org/licenses/by-nc-nd/4.0/>

## Disclaimer

Questa guida viene fornita “così com’è” in buona fede e senza nessuna pretesa di completezza, nessuna responsabilità per danni diretti od indiretti può essere attribuita all’autore. Nel dubbio non si utilizzino le informazioni qui contenute.

## Ringraziamenti

Prima di tutto un ringraziamento agli autori di **FreeCAD**.

Mi ritengo debitore di tutti coloro che hanno postato materiale nel forum dedicato a **FreeCAD**.

Un ultimo ringraziamento a  $\text{\TeX}$  e a  $\text{\LaTeX 2}_{\epsilon}$  senza i quali questa guida non sarebbe così ricca di riferimenti, note, tabelle indici e quanto altro ancora distinguono un buon prodotto grafico da un file prodotto con un word processor.

## Contatti, segnalazione di errori e comunicazioni

Per errori, omissioni o problemi nella presente guida, posso essere contattato tramite le “issues” di GitHub al seguente indirizzo web:

<https://github.com/onekk/freecad-doc>

Buon divertimento con **FreeCAD**.

Carlo Dormeletti (onekk)

# Storico della modifiche

## Note per questa versione

Posso ritenere questa versione della guida una versione Alpha, usando il linguaggio dei programmatori, una versione grezza o una bozza nel linguaggio degli scrittori e degli editori.

Mi interessano molto i vostri suggerimenti relativi a:

- Stile del testo per quanto riguarda la grafica e l'impaginazione.
- Punti poco chiari nelle spiegazioni.
- La chiarezza e l'opportunità dei rimandi nel testo (Ad esempio: Vedi a pagina XXX)

Ringrazio anticipatamente per la segnalazione degli inevitabili errori. Usate le indicazioni contenute in **Contatti, segnalazione di errori e comunicazioni**.

Piccola nota sul numero di versione, in genere il numero di versione viene composto nel seguente modo:

- Il numero prima del punto indica la revisione definitiva del testo per cui 1.0 sarà la prima versione dell'opera completa. Se questo numero vale 0 l'opera è da considerarsi una bozza.
- Il primo numero dopo il punto (il decimo) indica una importante revisione del testo.
- il secondo numero dopo il punto (il centesimo) indica una revisione "minore".

Non necessariamente viene rispettato l'ordine dei centesimi per cui ad esempio la 0.17 succede una 0.18, si può passare direttamente al decimo superiore nel caso di esempio da 0.17 a 0.20 quando c'è stata una revisione importante.

Non necessariamente viene rispettato l'ordine del decimo, la convenzione vuole che più il numero di avvicina all'intero più la versione si avvicini alla pubblicazione, per cui ad esempio da 0.40 si può passare alla 0.80 volendo indicare che l'opera è vicina al completamento.

Un decimo di 0.9 indica che è quasi pronta, cioè rimane solo la fase di "correzione delle bozze", dove si correggono gli errori di ortografia e si sistemano solamente i piccoli errori di impaginazione, i rimandi ecc., per cui non vi stupitevi se ad esempio da una versione 0.60 si passerà ad esempio ad una 0.90

## Storico

**v0.09 – 19 febbraio 2020** Prima stesura.

**v0.15 – 28 febbraio 2020** Pubblicazione di preliminare su forum **FreeCAD** e su forum RepRap Italia.

**v0.16 – 01 marzo 2020** Pubblicazione su GitHub versione Bozza.

**v0.17 – marzo 2020** Non pubblicata - Piccole correzioni di errori, modificati tutti i riquadri di codice per migliorare la leggibilità e per compattare l'impaginazione.

**v0.20 – 11 marzo 2020** Seconda Bozza preliminare su GitHub, modifica totale dell'ordine dei capitoli, aggiunte e spostamenti di codice, aggiunta di capitoli, all'opera "completa", manca la parte conversione del modello per la stampa 3D.

Bozza

# Indice

<b>Prefazione</b>	<b>VIII</b>
<b>Convenzioni usate nel testo</b>	<b>IX</b>
<b>1. Primi passi</b>	<b>3</b>
1.1. L'editore delle proprietà . . . . .	5
1.2. L'editor delle Macro . . . . .	5
<b>2. Scripting, chi era costui</b>	<b>7</b>
2.1. Breve introduzione a Python . . . . .	8
2.1.1. La sintassi . . . . .	9
Indentazione . . . . .	9
Spazi, a capo e righe bianche . . . . .	10
Commenti . . . . .	11
Le <code>docstring</code> . . . . .	11
2.1.2. Variabili . . . . .	11
2.1.3. I metodi e le funzioni . . . . .	12
2.2. Antipasto . . . . .	13
2.2.1. Le direttive <code>import</code> . . . . .	15
2.2.2. I parametri . . . . .	15
2.2.3. I cicli . . . . .	16
2.2.4. Le condizioni . . . . .	17
2.3. Nota metodologica . . . . .	18
2.4. Listato - schema base . . . . .	20
<b>3. Creazione di Oggetti 3D</b>	<b>22</b>
3.1. Gli Oggetti 3D . . . . .	22
3.1.1. Spazio tridimensionale . . . . .	22
3.1.2. Proiezioni e viste . . . . .	23
3.1.3. I <b>Vettori</b> . . . . .	23
3.1.4. Topologia . . . . .	24
3.1.5. Geometria e Vista . . . . .	24
3.1.6. I primi oggetti . . . . .	25
3.1.7. Listato - figure-base.py . . . . .	26
<b>4. Posizionamento</b>	<b>29</b>
4.1. Il riferimento di costruzione . . . . .	29
4.2. Posizionamento . . . . .	31
4.2.1. La proprietà <b>Placement</b> . . . . .	32

4.2.2. Listato - Riferimento costruzione . . . . .	35
<b>5. Le operazioni booleane</b>	<b>38</b>
5.1. Unione . . . . .	39
5.2. Sottrazione . . . . .	40
5.3. Intersezione . . . . .	41
5.4. Operazioni booleane - esempio completo . . . . .	42
<b>6. Modellazione avanzata</b>	<b>46</b>
6.1. Estrusione . . . . .	48
6.1.1. Listato - estrusione . . . . .	51
6.2. Rivoluzione . . . . .	54
6.3. Loft . . . . .	56
<b>7. La componente Vista</b>	<b>60</b>
<b>8. Programmazione modulare</b>	<b>63</b>
8.1. Modularizzazione . . . . .	64
8.1.1. Listato - Modulo GfcMod . . . . .	66
8.1.2. Listato - sardine.py . . . . .	70
8.2. Modellazione parametrica . . . . .	72
8.2.1. Creazione “automatica” di geometrie . . . . .	74
8.2.2. Listato - cubo_stondato modificato . . . . .	76
8.3. Librerie di oggetti . . . . .	77
8.3.1. Esempio di Libreria . . . . .	79
<b>9. Proprietà degli “oggetti 3D”</b>	<b>81</b>
9.1. Conoscere il nome delle proprietà . . . . .	81
9.1.1. Listato - aereo . . . . .	84
9.2. Il nome della Rosa . . . . .	88
<b>10. Stampa 3D</b>	<b>90</b>
10.1. La catena di Stampa . . . . .	90
10.2. Progettare per la stampa . . . . .	91
10.3. Produzione di file per la stampa . . . . .	92
10.3.1. Trasformazione in Mesh . . . . .	93
10.3.2. Formato AMF . . . . .	94
10.3.3. Formato STL . . . . .	94
<b>A. Elementi Interfaccia Utente</b>	<b>96</b>
<b>B. Glossario</b>	<b>97</b>
<b>C. Voci di menù</b>	<b>98</b>
<b>D. Oggetti di FreeCAD</b>	<b>99</b>

## Elenco delle figure

1.1. L'interfaccia Utente al primo avvio . . . . .	3
1.2. L'interfaccia Utente . . . . .	4
1.3. . . . .	5
2.1. schema a blocchi condizioni if . . . . .	19
3.1. Lo spazio 3D . . . . .	23
4.1. . . . .	31
4.2. L'editor delle proprietà . . . . .	33
5.1. Le operazioni Booleane. . . . .	41
6.1. . . . .	50
6.3. Lo strumento Loft . . . . .	59
7.1. La linguetta Vista . . . . .	61
8.1. Scatola sardine . . . . .	64
9.1. Derivazione delle proprietà. . . . .	82

## Elenco delle tabelle

4.1. punti di riferimento . . . . .	30
4.2. proprietà Placement . . . . .	33
4.3. Angoli di Eulero . . . . .	34
4.4. Rotazione usando gli Angoli di Eulero . . . . .	34

BOZZA



## Prefazione

Questa guida vuole essere un concreto aiuto nell'utilizzo dello scripting in **FreeCAD**, finalizzato alla modellazione 3D per ottenere modelli usabili nella stampa 3D.

Si è scelto di usare alcune convenzioni grafiche, che elencherò più estesamente più avanti, le copia delle schermate sono prese dal mio desktop Linux.

Tradurre significa anche interpretare, non sono un laureato in ingegneria, e nemmeno un laureato per inciso, tanto meno un traduttore, per cui il lavoro conterrà necessariamente delle imprecisioni e dei tecnicismi, se non delle vere e proprie parti in “dialetto tecnologico”.

La traduzione del linguaggio tecnico è un lavoraccio, i puristi si scandalizzano per l'uso di alcuni termini e i tecnici si scandalizzano perché si sono tradotti gli stessi termini che “fanno parte del linguaggio del settore”, in più alcuni termini sono usati nella lingua originale in quanto il corrispondente italiano non è sufficiente <sup>1</sup>.

Ho scelto di accompagnare alcuni termini con il corrispondente termine usati nell'originale Inglese messo tra parentesi, alcuni termini sono stati lasciati in inglese, non traduco i termini inglesi comunemente utilizzati nella lingua italiana come mouse, computer, script, thread, output o altri ancora.

Mi scuso per gli errori e gli svarioni nella presente guida, se avete suggerimenti o segnalazioni di errore da fare contattatemi attraverso i metodi citati in **Contatti, segnalazione di errori e comunicazioni**.

Carlo Dormeletti

onekk

---

<sup>1</sup>Essendo figlio di un meccanico di auto ho sentito parlare fin da piccolo di “gigleur” del carburatore, per riferirsi ai getti di minimo, di massimo o di compensazione, o di “pivot” per riferirsi a quello che in italiano viene definito “perno fuso”.

## Convenzioni usate nel testo

I colori di alcune parti del testo hanno un significato preciso:

- **Finestra dei rapporti** riferimento ad un elemento dell'interfaccia di **FreeCAD**.
- **Visualizza** voci di menù o elementi di una rappresentazione ad albero.  
**Visualizza** ⇒ **Barre degli Strumenti**, indica una sequenza di voci di menù o di rami di un albero di scelta.
- **Part::Box** i nomi dei metodi (funzioni) e le geometrie (oggetti 3D) di **FreeCAD**.
- **Placement** le proprietà di un metodo di **FreeCAD**.
- **variabile** i nomi delle variabili o i riferimenti di parti di codice all'interno della spiegazione testuale.

Le sequenze di tasti o combinazioni del mouse sono indicate in questo modo:

- **CTRL+SHIFT+F** indica che vanno premuti assieme Ctrl, Shift e il tasto F
- **Clic destro/sinistro/altro** vuole dire di cliccare il tasto del mouse.
- **Doppio Clic destro/sinistro/altro** vuole dire di cliccare due volte in modo veloce il tasto del mouse.
- **Destro/sinistro/altro premuto** vuole dire di tenere premuto il tasto del mouse mentre si fa qualcosa.
- Se viene indicato nel testo di **trascinare** qualcosa oppure si parla di **trascinamento** vuol dire selezione un oggetto e mentre si tiene **sinistro premuto** si muove il mouse nel punto desiderato (Questa funzione in inglese è chiamata *Drag and Drop*).

I riquadri colorati vengono usati per alcuni scopi:

Questo riquadro rosso viene utilizzato per gli esercizi proposti.

Il riquadro grigio viene per evidenziare una nota generica.

Questo riquadro giallo viene utilizzato quando è necessario evidenziare un paragrafo per illustrare meglio alcune particolarità del programma.

Questo riquadro ciano viene utilizzato per le altre note sul funzionamento del programma.

Questo riquadro verde scuro viene utilizzato per evidenziare una nota relativa a comportamenti particolari del programma e per eventuali variazioni riscontrate tra i vari Sistemi Operativi.

### Porzioni di codice

Questo è l'aspetto di una porzione di codice senza i numeri di linea:

```
for obj in DOC.Objects :
    DOC.removeObject(obj.Name)
```

Questo è invece l'aspetto di una porzione di codice che riporta i numeri di linea:

```
1 for obj in DOC.Objects :
```

In questa porzione di codice potete vedere un esempio di codice con la riga che viene mandata a capo dal programma di impaginazione.

```
obj_b.Placement = FreeCAD.Placement(Vector(0, 0, 0), FreeCAD
    ↪ .Rotation(0,0,0))
```

La freccia rossa indica dove il programma ha interrotto per motivi di impaginazione la linea, quando scrivete il codice dovete scrivere tutto di seguito, senza andare a capo.

### Immagini e copie delle schermate

Per la copia delle schermate viene utilizzata la versione di **FreeCAD 0.18 per Linux**, alcune schermate potrebbero non riflettere in maniera accurata le versioni per altri sistemi operativi o versioni diverse del programma, in genere preferisco utilizzare le AppImage distribuite direttamente da **FreeCAD** che la versione pre compilata per il sistema operativo, perché più aggiornate ed perché se diventa necessario chiedere informazioni una delle cose più comuni che ti viene detta è usa una versione AppImage per verificare che l'errore non sia già stato corretto.

Dove non risulta strettamente necessario usare una copia della schermata viene mostrata una finestra di dialogo stilizzata:



### Messaggi testuali

L'esempio qui sotto indica che questo è quanto viene visualizzato da **FreeCAD**, in modo testuale:

```
Placement [Pos=(0,0,0), Yaw-Pitch-Roll=(0,0,0)]
```

### Listati dei programmi e parti di codice

I listati presentati sono di creazione originale dell'autore e appositamente realizzati per questa guida, i colori delle varie parti del codice, non rispecchiamo perfettamente quelli che sono utilizzati all'interno dell'editor di **FreeCAD**, si è cercato comunque di approssimarli il più possibile a quanto mostrato da **FreeCAD**.

Molte parti di codice non sono da considerarsi un programma a sé stante, per cui per la comprensione e l'utilizzazione va letto assolutamente il testo che lo accompagna.

Per esigenze di impaginazione, nonché per evitare che l'inserimento del codice, rompesse il filo del discorso, i listati completi dei programmi in genere sono riportati alla fine del capitolo o della sezione, sono comunque disponibili alla pagina web:

<https://github.com/onekk/freecad-doc>

Il nome del file è in genere riportato nella “testatina” del programma e dove non possibile, indicato nel testo.

# Introduzione

**FreeCAD** è una applicazione per la modellazione parametrica di tipo CAD/CAE (Computer Aided Design e Computer Aided Engineering). È fatto principalmente per la progettazione meccanica, ma serve anche in tutti casi in cui è necessario modellare degli oggetti 3D con precisione e avere il controllo dello storico della modellazione.

Le parole chiave sono **Modellatore**, cioè un creatore di modelli 3D e **Parametrico**, perché permette di modificare facilmente le creazioni modificando i loro parametri..

**FreeCAD** è costruito attorno ad un motore CAD 3D, **Open Cascade Technology (OCCT)**.

**FreeCAD** è composto da una serie di API scritte in C++ e da molto codice si interfaccia con questa API scritto in Python.

**FreeCAD** è dotato di diversi “**ambienti di lavoro**” definiti in inglese “**WorkBench**” letteralmente tavoli da lavoro creati per svolgere diversi compiti, dalla modellazione 3D alla redazione di progetti, all’architettura e anche all’analisi dei modelli tramite tecniche di CAE.

Questa guida è incentrata sullo “Scripting”, cioè sulla scrittura di “programmi” che comandano il modellatore 3D di **FreeCAD** e permettono la creazione di modelli 3D.

Questa attività è possibile perché **FreeCAD** contiene al suo interno un potente interprete **Python**, la cui presenza ci consente di operare sul modellatore 3D.

Questo documento non vuole essere una completa introduzione all’uso del programma, per questo ci sono già altre guide, ma quasi esclusivamente una “guida alla programmazione” perché la documentazione presente sul sito di **FreeCAD** su questo argomento rende un principiante confuso, e lo allontana da questo potente mezzo di creazione.

I motivi di questa confusione sono molteplici:

- La documentazione è stata realizzata a molte mani.
- **FreeCAD** non è ancora arrivato alla sua versione stabile, per cui alcune cose cambiano.
- Le tecniche con cui automatizzare il lavoro attraverso lo scripting sono molteplici.
- Lo scripting è considerato a volte un aspetto secondario rispetto all’interfaccia grafica.

Per contro la sezione **Documentazione** del sito ufficiale, contiene molte introduzioni teoriche fatte molto bene e da gente competente, per cui potete sicuramente fare riferimento al sito ufficiale:

<https://www.freecadweb.org/>

## Installazione di FreeCAD

La prima cosa da fare è procurarsi **FreeCAD**, la versione stabile attuale è la **0.18**, mentre quella di sviluppo è la **0.19**.

Come installarlo dipende in parte dal sistema operativo che usate:

- **Linux** troverete alcune “AppImage”, in genere trovate anche delle versioni nei repository della distribuzione Linux usata, ma altrettanto in genere non sono aggiornate alla versione stabile.
- **Mac** ad oggi (febbraio 2020) è consigliato usare la versione di sviluppo **0.19**.
- **Windows**, troverete sia l’installer nelle versioni per win32 e win64, sia le “portable builds” sia per la versione stabile che per quella di sviluppo.

Secondo il mio parere il sistema migliore per installare **FreeCAD** sui sistemi Linux è quello di usare le **AppImage** cioè un file che contiene tutto quello che serve per far girare **FreeCAD** in modo dipendente il meno possibile dalle librerie installate sul computer, lo si può tranquillamente caricare su una chiavetta e usarlo dove si vuole, sempre che sul computer che volete usare sia installata un versione recente di Linux.

In più le Appimage hanno il vantaggio di poter essere “aggiornate” scaricando solo la parte modificata, usando le istruzioni e gli strumenti presenti sul sito di **FreeCAD**, dato che una AppImage attualmente si aggira attorno ai 500MB, il risparmio di dati è notevole.

Per Windows consiglio di usare le portable builds va estratto il file scaricato secondo le istruzioni sul sito di **FreeCAD**, usando 7zip (se non lo avete dovete scaricarlo al sito ufficiale di 7zip), fatto questo si avvia il programma direttamente dalla cartella dove vengono estratti i file.

Il maggior vantaggio di usare le Appimage o le portable builds è quello di avere sempre la versione aggiornata e di poter tenere sia la versione di sviluppo che quella stabile senza avere rogne.

# Capitolo 1

## Primi passi

Al primo avvio vi troverete una schermata simile a quella della figura 1.1.

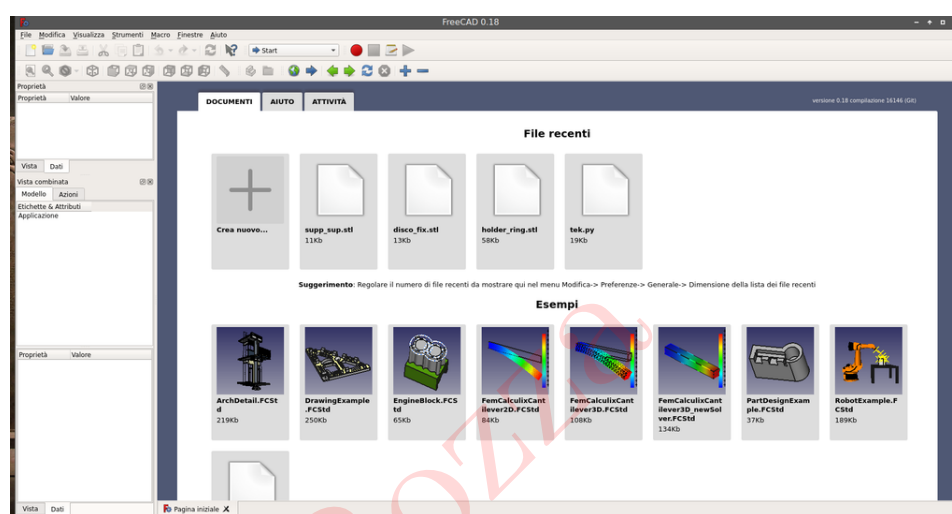


Figura 1.1.: L'interfaccia Utente al primo avvio

Prima di poter operare è meglio attivare alcuni elementi, cercherò di guidarvi passo passo, almeno nelle prime fasi, consiglio caldamente di leggere la documentazione di **FreeCAD** per poter operare in modo corretto, non mi dilungherò sulla navigazione con il mouse, nella **vista 3D**, una spiegazione approfondita va oltre gli scopi di questa guida e risulterebbe un inutile doppione della buona documentazione fornita sul sito del programma.

Vediamo comunque in modo molto veloce e schematico l'interfaccia Utente di **FreeCAD**, per lo meno per stabilire il nome degli elementi che ci serviranno nella spiegazione, userò la stessa descrizione presente sul sito ufficiale.

La figura 1.2 mostra uno schema dell'interfaccia utente.

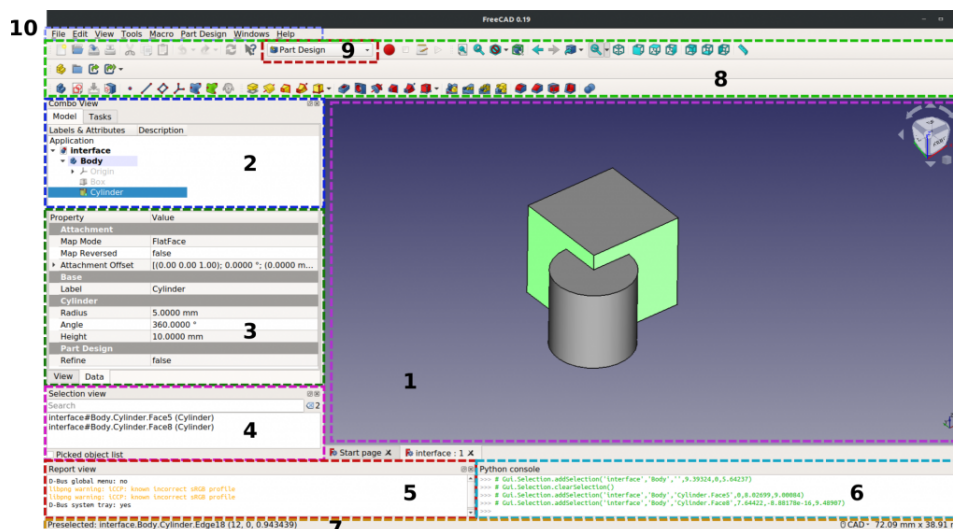


Figura 1.2.: L'interfaccia Utente

1. La **vista 3D**, che visualizza gli oggetti geometrici contenuti nel documento, in questa posizione ci potrebbe essere anche la inestra dell'editor.
2. **vista ad albero** (parte della **vista combinata**), che mostra la gerarchia e lo storico di costruzione degli oggetti nel documento; può anche visualizzare il pannello delle azioni per i comandi attivi.
3. L'**editore delle proprietà** (parte della **vista combinata**), che consente di visualizzare e modificare le proprietà degli oggetti selezionati.
4. Il **vista selezione**, che indica gli oggetti o i sotto-elementi degli oggetti (facce, vertici) che sono selezionati.
5. La **finestra dei rapporti**, dove **FreeCAD** stampa i messaggi di avvisi o di errori.
6. La **console Python** dove sono visibili tutti i comandi eseguiti da **FreeCAD**, e in cui è possibile inserire il codice Python.
7. La **barra di stato**, dove compaiono alcuni messaggi e suggerimenti.
8. L'**area della barra degli strumenti**, dove sono ancorate le barre degli strumenti.
9. Il **selettore degli Ambienti** che mostra quello attivo.
10. Il **menu standard**, che ospita le operazioni di base del programma.

Nella parte inferiore della **vista 3D**, troviamo le “linguette” che indicano i documenti aperti, possiamo alternarci tra loro cliccandoci sopra con il mouse.

Il programma dovrebbe essere in lingua Inglese, per cambiarla selezionate le voci di menù **Edit** ⇒ **Preferences**, scegliere l'icona **General** e la linguetta **Language** dove potete scegliere la lingua che più vi piace (e che riuscite a capire ovviamente), noi per semplicità useremo l'Italiano.

Per poter operare in modo efficace con lo scripting, è necessario attivare due **Panelli**, per fare questo dovete selezionare **Visualizza** ⇒ **Panelli** e poi mettere il segno di



spunta se non già presente su:

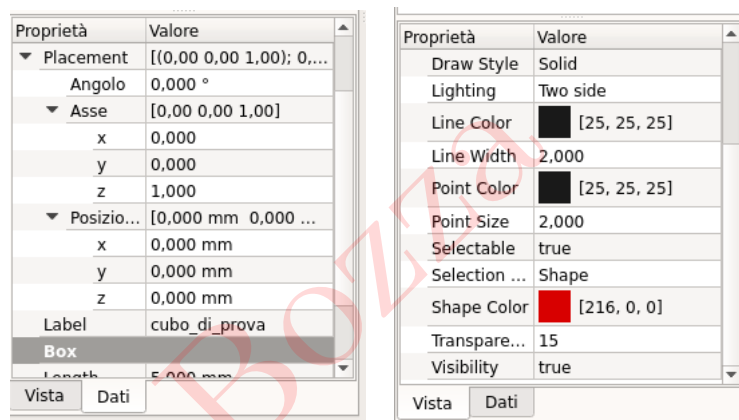
- **Report**
- **Console Python**

A questo punto dovreste avere la finestra del programma esattamente come nella figura 1.2 nella pagina precedente.

Nella finestra **Report** troverete gli errori e una “traccia” che vi aiuterà a risolverli. La finestra **Console Python** mostrerà anche i comandi che eseguirete attraverso i menù e vi potrà mostrare i metodi e le proprietà degli oggetti che avete creato.

## 1.1. L'editore delle proprietà

L'**editore delle proprietà** presenta nella parte bassa due linguetta, quella denominata **Dati** che chiameremo **Ling. Dati** contiene gli elementi che appartengono alle proprietà dell'oggetto creato, per la parte Dati, mentre la linguetta **Vista, Ling. Vista**, contiene le caratteristiche “grafiche” dell'oggetto, come il colore e lo stile grafico delle linee.



(a) Edp Linguetta Dati.

(b) EdP Linguetta Vista.

Figura 1.3.

Usiamo questa denominazione perché ne parleremo molte volte ed è meglio usare una denominazione convenzionale più corta che ripetere ogni volta la descrizione completa **editore delle proprietà** parte **Dati** ecc.

## 1.2. L'editor delle Macro

All'interno di **FreeCAD** trovate già tutto questo, l'**editor delle Macro** che appare quando caricate un file con estensione **.py** oppure una macro di **FreeCAD**, contiene appunto un editor che appare al posto della **vista 3D**.

L'**editor delle Macro** possiede poi una barra degli strumenti (da attivare con i comandi che vi descrivo sotto) con un bottone (triangolo verde) che permette di lanciare il programma e di vedere il risultato nella **vista 3D**.

Se questa barra non è attiva, basta fare **Visualizza** ⇒ **Barre degli Strumenti** e mettere il segno di spunta accanto a **Macro**, nell'**area della barra degli strumenti** appariranno dei bottoni come in nella figura qui a fianco, il bottone con la freccia che diventa verde quando nell'editor è caricata una Macro od un programma Python serve per eseguire quello che vedete nell'editor.



Niente vieta di usare un editor Python esterno, **FreeCAD** è abbastanza intelligente da vedere che il programma caricato è stato modificato da un programma esterno e da chiedervi se volete ricaricarlo:

Prima di iniziare gli esercizi di script Python, dovete andare nel menu **Modifica** ⇒ **Preferenze** alla sezione **Generale** nella linguetta **Finestra di Output** e attivare queste due caselle nel riquadro **Interprete Python**:

- Reindirizzare l'output interno di Python nella vista report
- Reindirizzare gli errori interni di Python alla finestra di report

L'**editor delle Macro** almeno nella versione **0.18** non possiede una voce di menù diretta per accedere all'editor, anche l'icona con la figura della carta e della penna nella barra degli strumenti **Macro** apre un dialogo che chiede di selezionare una Macro.

Il modo più veloce e “pulito” per accedere all'**editor delle Macro** quello di aprire un file con estensione **.py**

Dovete semplicemente creare un file anche vuoto con estensione **.py** dove preferite e caricarlo in **FreeCAD** con il comando **File** ⇒ **Apri**.

A volte ci riferiremo all'**editor delle Macro** come **editor Python**, per motivi di brevità e anche per motivi “semantici” infatti le **Macro** di **FreeCAD** sono scritte in Python, ma semanticamente non stiamo scrivendo Macro, cioè programmi che girano solo all'interno dell'interprete interno, ma veri e propri script Python, che potrebbero essere facilmente usati come programmi Python che usano **FreeCAD** come se fosse una “semplice” libreria Python.

Il termine Macro nel nostro caso è riduttivo, in origine il termine Macro era riferito a “registrazioni” di sequenze di comandi, oppure a comandi scritti in un linguaggio interno al programma, in genere diverso da programma a programma, la potenza di **FreeCAD** sta proprio nell'usare non un linguaggio proprietario interno oppure un dialetto proprietario di un linguaggio, ma un vero e proprio interprete Python che gira all'interno di **FreeCAD** e ne è una componente integrale, come abbiamo già accennato, una buona parte di **FreeCAD** è scritta proprio in Python.

# Capitolo 2

## Scripting, chi era costui

Citiamo il buon Alessandro Manzoni, per introdurre una parola inglese.

Oibò direbbero alcuni, talatri farebbero smorfie di disgusto, purtroppo, oggi abbiamo a che fare molto spesso con la lingua della “perfida Albione” tanto per citare un termine in uso quasi un centinaio di anni per chiamare l’isola che ha dato filo da torcere anche a Giulio Cesare.

Dobbiamo convivere con queste stranezze e non ripeterò le considerazioni fatte in precedenza sulla traduzione, qui dobbiamo intenderci, e per far questo non possiamo distaccarci molto dalla terminologia originale e fare giri di parole per denominare un concetto.

Che cos’è uno Script, le traduzioni sono molte, cominciamo con una suggestione, quando il gioco si fa duro, l’inglese usa il latino o il latineggiante, per cui script suona di più vicino a scrivere, che a write, piccola rivincita della cultura latina.

Quindi è un testo scritto che va passato secondo il linguaggio dei programmatori ad un “interprete dei comandi”, infatti Python è un linguaggio interpretato.

La maggiore distinzione tra un linguaggio interpretato e uno compilato è che un linguaggio compilato passa attraverso un compilatore e viene trasformato in un eseguibile, cioè un programma che gira sulla macchina, in modo “statico”, una volta fatte le decisioni nel “sorgente” del programma e compilato, queste decisioni non sono modificabili ma bisogna passare per la trafila:

- modificare i sorgenti
- ricompilare il programma
- eseguirlo e verificare che tutto vada come previsto

Un linguaggio interpretato invece farà “girare” il programma all’interno del suo “interprete” e mostrerà durante l’esecuzione gli errori eventualmente presenti.

Il vantaggio può sembrare minimo, ma questo ad esempio vi consente di tenere aperto nel computer un editor per modificare il programma e in contemporanea una console (linea di comando nella terminologia Windows) in cui lanciare il programma, e controllare subito la correttezza di quanto scritto ed eventualmente apportare al volo delle modifiche e verificarne immediatamente la bontà.

Lo svantaggio in generale è dato dalla minore velocità di esecuzione.

Tutto questo discorso solo per introdurre il termine “Scripting”, senza poi in definitiva nemmeno spiegarlo?

Non possiamo dar tutto per scontato se parliamo di una guida introduttiva, dobbiamo pur introdurre.

Finita l'introduzione diamo finalmente una definizione, uno “**Script**” è:

Un “piccolo programma” che fatto passare attraverso un interprete esegue un'azione.

piccolo perché in genere non possiede migliaia di righe di codice (che poi è l'unità di misura che usano i programmatori), programma perché in definitiva all'interno di **FreeCAD** abbiamo realmente un vero e proprio interprete Python, con tutte le sue librerie standard e qualche altra in più.

Basta infatti digitare **dir()** nella **console Python**, per vedere apparire questo:

```
>>>dir()
['App', 'Err', 'FreeCAD', 'FreeCADGui', 'Gui', 'Log', 'Msg',
  ↳ 'PathCommandGroup', 'Start', 'StartPage', 'WebGui
  ↳ ', 'WebPage', 'WebView', 'Workbench', 'Wrn', '
  ↳ __annotations__', '__builtins__', '__doc__', '
  ↳ __loader__', '__name__', '__package__', '__spec__',
  ↳ 'cmake', 'removeFromPath', 'sys', 'traceback', '
  ↳ webView']
```

Questi sono i nomi dei moduli interni di **FreeCAD**, che sono resi disponibili dall'interprete Python interno.

Dovendo “scrivere” un programma dobbiamo conoscere il linguaggio in cui va scritto, per cui introduciamo brevemente le basi del linguaggio Python.

## 2.1. Breve introduzione a Python

Python è un linguaggio che integra diversi elementi della **programmazione ad oggetti**, ora dobbiamo per forza accennare ad alcuni concetti, in modo molto sommario e impreciso, altrimenti rischiamo di non capirci.

Se siete interessati e per ogni dubbio le pagina ufficiale del linguaggio contengono una completa documentazione e una corretta spiegazione di tutte le cose e anche dei tutorial introduttivi fatti molto bene.

Una piccola ma doverosa nota: in **FreeCAD** 0.18 l'interprete Python è il 3.x quindi la documentazione di riferimento è quella relativa alla versione 3.x di Python, in genere l'AppImage indica nel nome anche la versione di Python che è contenuta, essendo terminato il supporto per Python 2.x naturalmente la versione di riferimento è la 3.x, attenzione però che in giro ci sono molte guide che si riferiscono ancora a Python 2.x

che ha delle piccole ma importanti differenze con Python 3.x, alcune differenze sono importanti altre solo delle cose cosmetiche, che comunque possono generare “errori di sintassi”.

### 2.1.1. La sintassi

La sintassi, cioè il modo di scrivere un programma, in genere ci sono poche regole da tenere a mente, niente di trascendentale, a volte qualche cosa si dimentica, però la leggibilità migliora notevolmente, se si seguono poche regole.

Elencheremo le più “importanti” per non appensantire l’introduzione, sappiate che esiste un documento ufficiale il PEP8 per Python che elenca le regole di sintassi più importanti. Fate riferimento a quello per maggiori informazioni.

La cosa basilare è la “leggibilità” del programma, ed usare una certa omogeneità, in più non dare mai per scontato che una cosa sia chiara, elenco un paio di concetti generali, ho dovuto usare qualche termine “tecnico” anche se non li ho introdotti, ad esempio il termine “variabile” e “commento” non spaventatevi diverranno chiari fra pochissimo.

- Se possibile cercate di usare nomi di variabili con un significato, ad esempio non usare **pippo** per definire una diametro, magari usare **c.dia** oppure **dia.cono**.
- Usate in modo ampio i commenti, quello che per voi oggi è chiarissimo, magari fra sei mesi lo sarà di meno.

### Indentazione

L’indentazione, cioè il rientro delle righe che evidenziano in modo visivo una parte di codice, assume molta importanza in Python, infatti essa definisce un **blocco** di codice.

Per convenzione è meglio usare 4 spazi per ogni indentazione per dire all’ di usare gli spazi al posto delle tabulazioni dovete andare nel menu **Modifica** ⇒ **Preferenze** alla sezione **Generale** nella linguetta **Editor** e mettere alcuni valori nel riquadro **Indentazione**

- **Dimensione dell’indentazione** mettete 4
- **Dimensione della tabulazione** mettete 4
- Scegliete l’opzione **Inserisci gli spazi** questa azione automaticamente toglie la spunta a **Mantieni le tabulazioni**

Già che ci siete spuntate nel riquadro **Opzioni** la casella **Abilita la numerazione delle righe**, torna utile nel caso di errori per trovare più facilmente il punto che ha generato l’errore.

Un blocco contiene una serie di linee di codice e viene distinto dal blocco successiva dalla diversa indentazione, un esempio lo potete vedere proprio nella finestra delle preferenze dell’Editor, nel riquadro **Anteprima**, poco più sopra potete scegliere un carattere decente per migliorare la leggibilità, fate alcune prove.

Le cose importanti sono:

- scegliete un carattere “**monospace**” altrimenti detto a larghezza fissa, sono i più indicati per la scrittura del codice
- fate attenzione che le lettere l (elle minuscola) e I (i maiuscola) si possano distinguere tra di loro
- fate attenzione che il simbolo 0 (zero) si possa distinguere dalla O (o maiuscola), in genere lo zero ha un puntino oppure è barrato proprio per distinguerlo immediatamente dalla o maiuscola

Non faremo un trattato su Python, piuttosto cercheremo di spiegare i concetti basilari per poter scrivere uno script per **FreeCAD**, facendo esempi e spiegando dove serve un minimo di “teoria”, cercando di tenere ben presente la “chiarezza” ed “agilità”.

### Spazi, a capo e righe bianche

Altro concetto importante, almeno nella forma sono le spaziature, quando scriviamo un codice, cerchiamo di evitare di mettere gli spazi a caso, poche regole da tenere a mente:

1. Attorno ad un operatore, ad esempio il segno di uguale (=) nell’assegnazione del valore di una variabile, serve un solo spazio a destra e un solo spazio a sinistra.
2. Dopo la virgola ci vuole uno spazio, tranne nelle tuple con un singolo elemento, ad esempio (**"baco"**,) è legittimo, ma ad esempio (**"mela"**, **"pera"**,) non lo è.
3. Non vanno messi spazi dopo la parentesi aperta e prima della parentesi chiusa, si può però andare a capo perché la parentesi aperta e chiusa vengono usati per interrompere la convenzione che la fine della riga segna la fine del comando.
4. Tra un **#** messo di seguito al codice e il codice stesso vanno messi due spazi dopo quello che preferite, ma meglio almeno uno spazio.

Un piccolo esempio per chiarire il punto 3:

```
FreeCAD.ActiveDocument().activeView().setAxisCross(True)
```

Se ad esempio avessimo necessità di spezzare la linea perché troppo lunga, e **True** dovesse andare a capo, le convenzioni di Python prevedono un rientro (indentazione) tipo quello mostrato qui sotto:

```
FreeCAD.ActiveDocument().activeView().setAxisCross(  
    True)
```

Potete andare a capo dopo una parentesi aperta, mettendo almeno una indentazione per separare in modo visivo la riga dalla riga seguente che contiene in genere una istruzione appartenente allo stesso blocco di codice.

Per le righe bianche, cercate di tenere una certa uniformità, lo so che è solo una questione estetica, ma migliora la leggibilità.

Potete usare le righe bianche liberamente, per separare blocchi logici di codice, però con alcune cautele:

- Evitate di usare più di tre righe bianche consecutive.
- Mettete due righe bianche prima e dopo una funzione o metodo che dir si voglia (cosa sono lo vedremo tra poco).

## Commenti

In Python come tutti i linguaggi di programmazione sono importanti i “commenti”, cioè le righe che l’interprete non interpreta, ed in cui si mettono informazioni che spiegano il codice oppure porzioni di codice che al momento non vogliamo usare, oppure pezzi di codice come promemoria, o che causano errore, magari durante lo sviluppo di un programma.

I commenti si possono inserire in Python usando `#`, se usate ad inizio della riga sono utili per commentare porzioni di codice, mentre se usate dopo la riga di codice sono utili per ricordare qualcosa ad esempio “non toccare” oppure “da correggere”, se usate un IDE ad esempio alcuni commenti sono evidenziati in modo particolare, sono delle vere e proprie “parole chiave”, in un sorgente non è infrequente trovare `#FIXME` per ricordare che questo può generare un errore oppure `#TODO` per ricordare che c’è qualcosa da aggiungere, in futuro.

Ricordate che quando documentate bene le cose anche a distanza di tempo potete riusare il codice perché sapete cosa significa ogni parametro e che operazione esegue ogni porzione di codice.

## Le docstring

Un particolare tipo di commenti sono le **docstring**, è un commento particolare che è “BENE” esista per ogni **metodo**, serve a descrivere quello che fa il metodo e anche eventualmente i **parametri** del metodo.

Le **docstring** sono identificate e riconosciute dall’interprete perché cominciano e finiscono con tre virgolette `"""` le potete vedere poco più sotto quando vengono spiegati i metodi.

### 2.1.2. Variabili

Le variabili sono dei contenitori, questi contenitori possono contenere di tutto, da numeri ad oggetti anche complessi, Python.

I nomi delle variabili in Python seguono alcune convenzioni:

- se il nome è scritto “TUTTO IN MAIUSCOLO” in genere si usa per le “costanti”, almeno dal punto di vista logico, anche se in Python “tutto è un oggetto” e vere e proprie “costanti”, cioè cose che una volta dichiarate non possono più essere cambiate pena un “errore di sintassi” non esistono

- se il nome comincia con un carattere di “\_” in genere è considerata una variabile “locale” che è “BENE” non sia usata all’esterno dell’ambito in cui è definita ed usata.
- le variabili scritte in questo modo `_nome_`, (notate che nonostante la grafica non esistono spazi tra i \_ e “nome”) hanno un significato particolare ed è BENE non siano usate se non si sa a cosa servono.
- nel caso ci siano ambiguità con parole chiave del linguaggio potete usare un solo “\_” dopo il nome della variabile per eliminare l’ambiguità
- usate nomi di variabili più lunghi di tre caratteri e usate liberamente i “\_” all’interno del nome, ad esempio `nome_della_variabile` è perfettamente legale e significativo.

Le variabili di Python, non sono “fortemente tipizzate”, cioè non è necessario dichiarare il tipo durante la creazione.

Le variabili comunque possiedono dei tipi, elenchiamo i principali:

- `int` ossia **integer** un numero intero cioè senza virgola
- `float` ossia **floating point** un numero con la virgola detto anche “in virgola mobile”
- `stringa` ossia caratteri “stampabili” destinati a contenere del testo.

se scriviamo `alt = 0` e poi operiamo su di essa con operazioni che presuppongono che il numero sia in virgola mobile, non succede nulla, semplicemente l’interprete traduce il numero intero in un numero in virgola mobile.

Questa però se vogliamo ottenere risultati consistenti, è meglio non abusare della flessibilità di Python, e operare in modo corretto, se definiamo un numero che sappiamo essere in virgola mobile, meglio definirlo come `alt = 0.0`.

Nei listati dei programmi potete trovare molte “divisioni per due” scritte come “moltiplicazioni per 0.5” la ragione è un “trucco salvavita” da programmatore.

Alcuni linguaggi interpretano la divisione in modo strano (fortunatamente non è il caso di Python) per cui se io divido un valore `float` per 2 che è un `int` l’interprete o il compilatore convertono l’operazione in una divisione di `int`, restituendo un `intero` invece che un `float`, facendo diventare poi matti a trovare l’errore nel codice, usando la moltiplicazione si moltiplica un `float` per un altro valore `float` evitando ogni possibile errore di conversione implicita.

Una seconda ragione è che in genere la moltiplicazione è meno costosa in termini di risorse elaborative di una divisione.

### 2.1.3. I metodi e le funzioni

Questa porzione di codice:

```
def clear_doc():  
    """
```



```
Clear the active document deleting all the objects
"""
for obj in DOC.Objects:
```

Mostra un **metodo** altrimenti chiamato **funzione**.

La distinzione che alcuni fanno è che un **metodo** possiede dei parametri, mentre una **funzione** non ne possiede; Non avendo parametri tra le parentesi tonde questo metodo è una funzione.

La distinzione però non è affatto rigida per cui possiamo sentire parlare indifferentemente di metodo e funzione per riferirci genericamente a qualcosa definito con la parola chiave **def()**.

Notate che alla fine della definizione è necessario mettere un (:) due punti.

Una cosa importante, un metodo possiede il suo “spazio dei nomi”, cioè una variabile definita in un metodo, possiede il suo proprio valore anche se ha lo stesso nome di una variabile presente in un altro metodo, lo spazio dei nomi comprende il nome dei parametri che passiamo nella definizione del metodo:

Ad esempio se abbiamo un metodo definito come:

```
def pippo(pluto, paperino, topolino = 0):
    paperoga = pluto * paperino - topolino
    ...

def nonna_papera(qui, quo, qua):
    paperoga = qui * quo + qua
    ...
```

il valore delle due variabili **paperoga** è diverso.

## 2.2. Antipasto

Vediamo ora un piccolo pezzo di codice da inserire nell’editor.

Qui sotto vedete un esempio minimale di codice.

```
import FreeCAD

DOC = FreeCAD.activeDocument()
DOC_NAME = "Pippo"

if DOC is None:
    FreeCAD.newDocument(DOC_NAME)
    FreeCAD.setActiveDocument(DOC_NAME)
    DOC = FreeCAD.activeDocument()
```

Il programma fa poco o nulla, assegna alla variabile **DOC**, il contenuto della funzione **FreeCAD.activeDocument()**, che contiene l’oggetto del documento attivo.

Assegna poi alla variabile **DOC\_NAME** il nome del documento.

Notate l'uso dei nomi tutti in maiuscolo, per la variabile **DOC\_NAME**, appare naturale, ma per il documento attivo può sembrare non corretto, in quanto è soggetto a modifiche durante la creazione di oggetti.

Viene considerato una “costante”, in quanto considerato ai fini logici come “oggetto” non viene modificato, (**DOC**) rappresenta sempre il “documento attivo”.

Se premete la famosa freccia verde della **Barra degli strumenti Macro**, che ho descritto più sopra vedrete che viene creato un nuovo documento e viene aperta una **vista 3D** vuota.

La “magia” viene fatta da:

```
if DOC is None:
    FreeCAD.newDocument(DOC_NAME)
    FreeCAD.setActiveDocument(DOC_NAME)
    DOC = FreeCAD.activeDocument()
```

La prima riga controlla se il documento esiste confrontando il valore di **DOC** con l'espressione **is None** attraverso la parola chiave **if**, se l'espressione è vera allora:

- Crea un nuovo documento chiamato con il nome contenuto nella variabile **DOC\_NAME**.
- Lo rende il documento attivo.
- assegna alla variabile **DOC** il contenuto del documento attivo.

Semplice semplice, però avete fatto fare qualcosa a **FreeCAD** e avete creato il vostro primo script in Python.

Complichiamo di poco le cose:

```
import FreeCAD

DOC = FreeCAD.activeDocument()
DOC_NAME = "Pippo"

def clear_doc():
    """
    Clear the active document deleting all the objects
    """
    for obj in DOC.Objects:
        DOC.removeObject(obj.Name)

if DOC is None:
    FreeCAD.newDocument(DOC_NAME)
    FreeCAD.setActiveDocument(DOC_NAME)
    DOC = FreeCAD.activeDocument()
```

```
else:

    clear_doc()
```

Per capire questa parte di codice dobbiamo spiegare ancora poche cose.

### 2.2.1. Le direttive *import*

Le righe di codice mostrate sopra fanno parte dello **schema base** a cui abbiamo accennato.

Se osservate quel codice nelle prime righe vedete alcune direttive **import**, con scritture leggermente diverse, spieghiamo a grandi linee il perché di queste differenze di scrittura.

```
import FreeCAD
```

Questo semplicemente importa il modulo **FreeCAD**, per poter usare oggetto contenuto all'interno del modulo, sia esso un metodo (ne parleremo fra poco) o una variabile, dobbiamo “riferirci” ad esso con **FreeCAD.qualcosa**.

Se dobbiamo però usare molte volte un oggetto potrebbe essere utile evitare di battere il riferimento al modulo, questo ci espone al rischio di non capire a chi appartiene quell'oggetto.

Conoscendone i pericoli possiamo però usare questa scrittura:

```
from FreeCAD import Base, Vector
```

che importa i metodi **Base** e **Vector** all'interno dello “spazio dei nomi” permettendoci di riferirci ad essi semplicemente come **Base** e **Vector** e non come **BaseFreeCAD.Base** e **FreeCAD.Vector**, risparmiandoci molte battute sulla tastiera.

Analogamente importiamo dal modulo **math** alcune cose che torneranno utili come **pi** che è il valore di  $\pi$  (pi greco) e le funzioni seno e coseno.

```
from math import pi, sin, cos
```

### 2.2.2. I parametri

La riga:

```
FreeCAD.setActiveDocument(DOC_NAME)
```

mostra l'uso di un **parametro**.

Cominciamo da sinistra troviamo **FreeCAD** seguito da un punto e da **newDocument** stiamo invocando il metodo **newDocument** contenuto nel modulo **FreeCAD** usando il contenuto della variabile **DOC\_NAME**

Un metodo può contenere molti parametri, alcuni possono essere “obbligatori” altri “facoltativi”,

In questa porzione di codice vediamo la dichiarazione del metodo **miaScatola**:

```
def miaScatola(mioNome, miaLargh=50.0, miaLungh=30.0,
               miaAlt=70.0):
```

Questo metodo possiede 3 paramteri:

1. **miaLargh** che è la larghezza
2. **miaLungh** che è la lunghezza
3. **miaAlt** che è l'altezza

ognuno di questi parametri però ha un segno di uguale ed un valore, ciò significa che se io invoco il metodo con:

```
miaScatola("Scatola")
```

otterrò un parallelepipedo con larghezza = 50, lunghezza = 30 e altezza = 70 (mm nel nostro caso).

Perché? Semplicemente perché la funzione definisce tutti e tre i parametri come facoltativi e assegna dei valori per difetto o per usare la terminologia inglese di "default".

Se non fornisco nessun parametro per i valori **miaLargh**, **miaLungh**, **miaAlt** il metodo usa quelli predeterminati al momento della sua creazione, altrimenti glieli devo fornire io e lui usa quelli che gli fornisco al momento dell'invocazione, ad esempio:

```
miaScatola("Scatola", 20, 20, 20)
```

otterrò un parallelepipedo, in questo caso un cubo con larghezza = 20, lunghezza = 20 e altezza = 20 (mm nel nostro caso).

### 2.2.3. I cicli

Analizziamo la funzione:

```
def clear_doc():
    """
    Clear the active document deleting all the objects
    """
    for obj in DOC.Objects:
        DOC.removeObject(obj.Name)
```

Questa funzione ha il compito di svuotare il documento attivo, e viene invocata da una versione leggermente modificata della condizione che abbiamo visto poco sopra.

Lo scopo di questa funzione è quello di riutilizzare il documento che creiamo la prima volta se il documento attivo è vuoto, ci risparmia tempo in quanto :

- Ad ogni lancio dello script non viene creato un nuovo documento, infatti se esiste un documento viene automaticamente riutilizzato.
- Stiamo creando un script che compone un oggetto, e non vogliamo certo che gli oggetti creati vengano aggiunti agli oggetti già presenti con lo stesso nome alla

nostra vista, dato che in pratica ad ogni lancio dello script ricreiamo gli oggetti da capo.

Questa funzione è apparentemente semplice, ma contiene un costrutto importante.

All'interno della funzione `removeObject()` abbiamo il parametro `obj.Name` che non è qualcosa di statico.

Infatti questo parametro è un membro di una lista di oggetti che si chiama `DOC.Objects` che noi stiamo scansionando usando il ciclo `for`:

```
for obj in DOC.Objects :
```

Spieghiamo a parole quello che stiamo facendo stiamo dicendo al programma:

- prendi un oggetto contenuto in `DOC.Objects` e rendilo disponibile con il nome `obj`, alle istruzioni del blocco.
- continua fino alla fine della lista `DOC.Objects`

Notate il “due punti” alla fine, è facile dimenticarli a volte, dicono a Python che la dichiarazione è finita, (l'abbiamo già visto anche nelle dichiarazioni di `if`), se mancano segna errore, provate a toglierli e vedete il risultato al lancio del programma.

Non esiste solo il ciclo `for` nel linguaggio Python, ma non dobbiamo fare un trattato sul linguaggio Python, per ogni approfondimento rimando alle ottime guide e tutorial presenti sul sito ufficiale del linguaggio.

Da questo piccolo esempio potete capire la potenza di uno script in Python.

#### 2.2.4. Le condizioni

Durante la scrittura di uno script può rendersi necessario compiere delle scelte.

Le scelte in genere sono basate su alcune condizioni:

- **se** succede questo **allora** fai quello
- **se** questa cosa esiste **allora** fai così **altrimenti** fai cos'altro
- **se** una certa variabile ha un valore **allora** comportati così

Potete notare che ho evidenziato tre parole: **se** ... **allora** ... **altrimenti**.

Tradotte in inglese diventano **if** ... **then** ... **else**.

Abbiamo già visto un piccolo esempio di una condizione **if** ... **then**.

In Python a differenza di altri linguaggi la parola chiave **then** non esiste, è sostituita dal blocco di codice che segue la condizione.

La parola chiave **if** introduce una condizione booleana in base alla quale viene eseguita o meno una porzione di codice.

Ricordiamo per inciso che un valore booleano può assumere solo due valori, vero **True** o falso **False**.

La chiave è la scrittura della **condizione**, diamone alcuni esempi:

```
if DOC is None:

if var >= 0:

if nome in ("paperino", "pippo", "pluto"):
```

Notiamo che la sintassi è **if condizione** :

Il (:) due punti è importante, perché dice all'interprete che la definizione della condizione è finita e che il blocco che segue va eseguito se la condizione è **True** (Vera).

il primo esempio è un semplice confronto con **None** cioè **Nulla**, è vero se il confronto verifica la non esistenza di una cosa, se la variabile non esiste (per meglio dire assume il valore di **None**) allora si eseguono le istruzioni del blocco.

Il secondo esempio controlla che il valore di **var** sia maggiore o uguale a 0.

Il terzo esempio controlla che la stringa **nome** abbia uno dei valori presenti nella tupla (**"paperino"**, **"pippo"**, **"pluto"**), ci evita di scrivere tre confronti diretti e ci permette di aggiungere in seguito altri valori.

Queste scritture evidenziano la presenza di una prima condizione, ci si può fermare lì, però se dobbiamo agire in un certo modo se il risultato della condizione è **True** e in un modo diverso se è **False**, dobbiamo introdurre una scrittura leggermente diversa:

```
if var > 0:
    print("maggiore di zero")
elif var == 0:
    print("uguale a zero")
else: # var < 0
    print("minore di zero")
```

Nella figura 2.1 nella pagina successiva viene rappresentato lo schema a blocchi della condizione **if** presentata qui sopra.

Una piccola nota se una condizione viene valutata come vera, vengono eseguite tutte le istruzione del blocco associato e tutte le altre condizioni che seguono la condizione vera semplicemente non vengono valutate, siano esse **elif** o **else**, e il programma continua dopo la fine del blocco **else:**, se esiste.

## 2.3. Nota metodologica

Avendo a disposizione un linguaggio molto potente e articolato (Python), che “siede” sopra ad un’insieme di API (Application Program Interface) molto potente come quelle di **FreeCAD**, abbiamo molti modi per ottenere una cosa.

Ho scelto con questa guida di usare principalmente un modo, quello di usare il più possibile l’approccio di usare il metodo **DOC.addObject()**, alla prova dei fatti questo metodo è risultato meno problematico di altri, e meno involuto, questo metodo crea

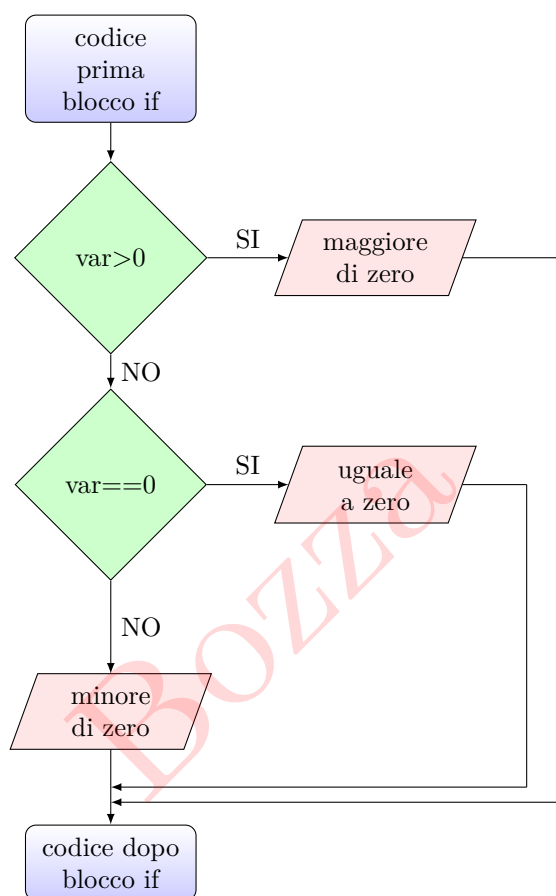


Figura 2.1.: schema a blocchi condizioni if

direttamente nel documento aperto, l'oggetto 3D voluto, senza poi dovere richiamare una secondo metodo per visualizzare esplicitamente la geometria invocando il metodo **show** come molto spesso si legge nella documentazione ufficiale.

Nella costruzione del programma utilizzeremo molto la tecnica dei blocchi logici, creando metodi che creano forme base o che compiono operazioni semplici, e che sono lunghi poche righe e poi richiamandole con un semplice riga di codice, anche questo modo si è rivelato utile e proficuo, rende possibile creare varie versioni più o meno complesse, permettendo di analizzare alternative mettendo semplicemente un **#** all'inizio della riga che non vogliamo eseguire.

Nel corso del discorso faremo spesso riferimento a "listati" dei programmi che sono riportati per ragioni di impaginazione alla fine del capitolo dove vengono presentati. Forniremo comunque all'interno del testo le porzioni di codice rilevanti e le aggiunte graduali, nonché il riferimento alle pagine dove vengono riportati.

Questa guida andrebbe seguita scrivendo (o scaricando il codice) inizialmente il listato presentato nella sezione 2.4 con il titolo **Listato - schema base**, salvandolo ed aggiungendo via via le porzioni di codice illustrate nel testo, risulterà utile salvare le versioni intermedie con il nome che preferite, in modo da poterle riprendere poi se necessario.

Molto codice è stato concepito per essere agevolmente trasferito da un metodo all'altro mediante il copia-incolla, lo potete anche capire dal riutilizzo di molti nomi di variabile all'interno dei metodi.

## 2.4. Listato - schema base

```
1 #
2 """sc-base.py
3
4     This code was written as an sample code
5     for "FreeCAD Scripting Guide"
6
7     Author: Carlo Dormeletti
8     Copyright: 2020
9     Licence: CC BY-NC-ND 4.0 IT
10 """
11
12 import FreeCAD
13 from FreeCAD import Base, Vector
14 import Part
15 from math import pi, sin, cos
16
17 DOC = FreeCAD.activeDocument()
18 DOC_NAME = "Pippo"
19
20 def clear_doc():
```



```
21     """
22     Clear the active document deleting all the objects
23     """
24     for obj in DOC.Objects:
25         DOC.removeObject(obj.Name)
26
27 def setview():
28     """Rearrange View"""
29     FreeCAD.Gui.SendMsgToActiveView("ViewFit")
30     FreeCAD.Gui.activeDocument().activeView().viewAxometric(
31         ↵ )
32
33 if DOC is None:
34     FreeCAD.newDocument(DOC_NAME)
35     FreeCAD.setActiveDocument(DOC_NAME)
36     DOC = FreeCAD.activeDocument()
37
38 else:
39
40     clear_doc()
41
42
43 # EPS= tolerance to uset to cut the parts
44 EPS = 0.10
45 EPS_C = EPS * -0.5
```

# Capitolo 3

## Creazione di Oggetti 3D

### 3.1. Gli Oggetti 3D

Gli **oggetti** di cui parliamo sono oggetti 3D (tridimensionali), cioè le cose che poi diventeranno reali una volta stampati.

In **FreeCAD** abbiamo una serie abbastanza completa di oggetti e possiamo addirittura definirne altri molto più complessi usando varie tecniche.

Quando gli oggetti creati con questo metodo sono usati come componenti di una costruzione complessa, semplicemente diventano parte dell'albero dell'oggetto.

Nelle descrizioni, a volte potrete trovare la parola **geometrie** per definire gli **oggetti 3D**, ma anche la parola **oggetti** oppure a volte **forme** che poi è la traduzione italiana di **Shapes**, purtroppo **FreeCAD** è stato sviluppato da molte persone e le definizioni a volte variano, un piccolo esempio in molti metodi, ad esempio `DOC.addObject()`, chiaramente si chiama quello che si sta creando **Object** cioè **oggetto**, poi però nella definizione di molte funzioni si usa la proprietà **Shape** per assegnare una geometria e **Shapes** per assegnare più geometrie, dobbiamo tenerne conto, sappiate che in pratica sono sinonimi.

Faremo anche largo uso delle istruzioni del linguaggio Python creando metodi e poi chiamandoli al bisogno, alla prova dei fatti si è dimostrato il modo più maneggevole per la costruzione di elementi anche molto complessi.

Il concetto base per costruire un oggetto complesso, assomiglia al vecchio indovinello “come fa una formica a mangiare un elefante?”, la risposta ovvia, una volta che la si è sentita è “a piccoli pezzi”.

Partiremo illustrando un paio di concetti, ma procederemo con esempi “concreti”, non essendo un **Manuale** dove è necessario essere sistematici, ma piuttosto una **Guida** cercheremo appunto di guidare il lettore partendo dal “semplice” per arrivare al “complesso”.

Non possiamo fare a meno ora di richiamare alcuni concetti relativi alla geometria 3D.

#### 3.1.1. Spazio tridimensionale

Nella figura 3.1 qui accanto potete vedere una rappresentazione convenzionale dello spazio tridimensionale, ogni punto nello spazio 3D è definito da una tripla di coordinate  $(X, Y, Z)$ , nella guida, in genere se non spiegato diversamente se vedete qualcosa scritto in questo modo  $(0, 0, 0)$ , significa che stiamo parlando di una posizione nello spazio 3D, per comodità in questa parte di spiegazione, le posizioni possiedono solo numeri interi, ma ovviamente i numeri decimali sono ammessi.

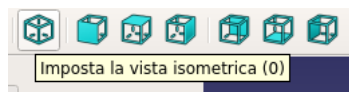
Le convenzioni sono le stesse usate da **FreeCAD**, cioè come le vedete nella **vista 3D**.

La figura mostra una schematizzazione dello spazio 3D, con raffigurato un cubo, e gli otto punti che lo definiscono.

Ogni punto è definito da una tripla di numeri che riportano le coordinate, alcuni punti li potete ricavare seguendo le linee punteggiate sul grafico, ad esempio il punto **A** ha coordinate  $(5,3,0)$ , mentre il punto **B** ha coordinate  $(8,3,0)$ , non sorprendentemente il punto corrispondente ad **A** sulla faccia superiore del cubo **E** avrà coordinate  $(5,3,3)$  mentre **F** che corrisponde a **B**  $(8,3,3)$ .

### 3.1.2. Proiezioni e viste

Per poter rappresentare uno spazio tridimensionale usando due dimensioni è necessario usare le cosiddette proiezioni, **FreeCAD** ne possiede diversi tipi, alcune accessibili direttamente attraverso il menù **Visualizza**, ad esempio **Vista Ortografica**, oppure **Vista Prospettica**, altre accessibili come voci del menù **Visualizza**  $\Rightarrow$  **Viste Standard**.



Le viste accessibili tramite il menù **Visualizza**  $\Rightarrow$  **Viste Standard** sono le viste “dall’alto”, “dal basso”, “da destra”, “da sinistra”, “di fronte” e “da dietro”, si trovano anche nella **Barra degli strumenti Viste** visualizzate come un cubo con la faccia evidenziata relativamente alla posizione della vista richiesta. possiede anche altre tre viste nel **Visualizza**  $\Rightarrow$  **Viste Standard**  $\Rightarrow$  **Axonometric**.

Tra queste trovate anche una vista importante, la vista **Isometrica** che possiede anche una icona nella **Barra degli strumenti Viste** come potete vedere nella figura qui a fianco.

### 3.1.3. I Vettori

Un **vettore** in **FreeCAD** lo possiamo concepire come un contenitore che contiene 3 elementi, in genere le coordinate 3D di un punto nello spazio definite come valori in

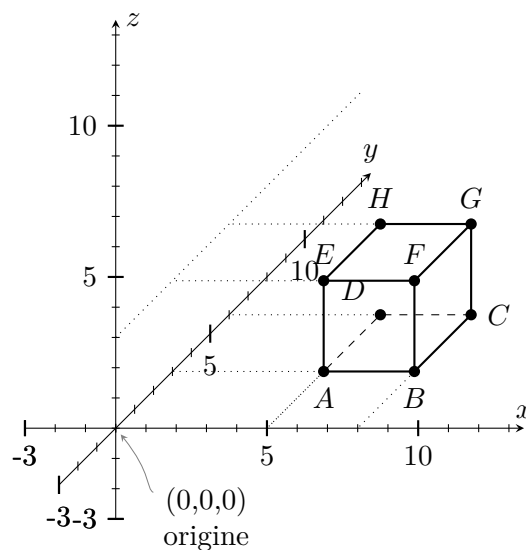


Figura 3.1.: Lo spazio 3D

virgola mobile (float) di X, valore di Y e valore di Z in questo modo **Vector(val\_x, val\_y, val\_z)**, è un concetto abbastanza comune a tutti i modellatori 3D.

A volte contiene anche cose diverse da un punto, ad esempio gli angoli di rotazione per ogni asse.

Perché un **vettore** e non qualche altra cosa, perché sui **vettori** si può operare in caso di necessità con opportune operazioni e perché ci permette di fare riferimento a quella terna di valori con un unico nome.

### 3.1.4. Topologia

La (Topologia), cioè della branca della geometria che studia le proprietà delle figure.

Usando **FreeCAD** non possiamo esimerci da introdurre alcuni concetti:

- **Vertice (Vertex)** Un elemento topologico corrispondente ad un punto. Esso non ha dimensioni.
- **Bordo (Edge)** Un elemento topologico corrispondente ad una curva limitata. Un **bordo** è generalmente delimitato dai **vertici**. Ha una dimensione.
- **Polilinea (Wire)** Una serie di **bordi** (una polilinea) collegati tra di loro nei **vertici**. Può essere aperto o chiuso, secondo se i **bordi** sono interamente concatenati oppure no.
- **Faccia (Face)** In 2D è una parte di un piano; in 3D è una parte di una superficie. La sua geometria è vincolata (delimitata/tagliata) dai suoi **bordi**.
- **Guscio (Shell)** Una serie di **facce** connesse nei loro **bordi**. Una **shell (guscio)** può essere aperta o chiusa. **solid** Una parte di spazio limitato da shell. E tridimensionale.

### 3.1.5. Geometria e Vista

**FreeCAD** è stato inizialmente creato per lavorare come applicazione a riga di comando, senza la sua attuale interfaccia utente. Di conseguenza, quasi tutto al suo interno è separato in una componente "geometria" e una componente "vista".

In **FreeCAD**, quasi tutti gli oggetti possiedono due parti distinte, una parte **Object** che contiene i dati della geometria e una parte **ViewObject** che contiene i dati di visualizzazione, ad esempio il colore e la trasparenza.

Questa distinzione è visibile anche nell'**editore delle proprietà** che presenta appunto una **Ling. Vista** e una **Ling. Dati**.

Nello **schema base** potete vedere un utilizzo della parte **ViewObject** nel metodo **setview()**.

### 3.1.6. I primi oggetti

Semplicemente partiamo definendo un oggetto, ad esempio un cubo, cioè un oggetto di tipo **Part::Box**, creiamo quindi un metodo in questo modo:

```
def cubo(nome, lung, larg, alt):
    obj_b = DOC.addObject("Part::Box", nome)
    obj_b.Length = lung
    obj_b.Width = larg
    obj_b.Height = alt

    DOC.recompute()

    return obj_b
```

Il metodo restituisce un oggetto cubo (anche se in realtà è un parallelepipedo), il perché deve restituire qualcosa diventerà chiaro più avanti.

Il metodo prevede 4 parametri:

1. **nome** il nome assegnato all'oggetto, questo nome apparirà nella **vista combinata** nella parte **vista ad albero**.
2. **lung** la misura della lunghezza
3. **larg** la misura della larghezza
4. **alt** la misura dell'altezza

estendiamo il nostro schema base aggiungendo la parte sopra, e un paio di altre righe, ottenendo il programma a pagina sezione 3.1.7 nella pagina successiva con il titolo **Li-stato - figure-base.py** che sorprendentemente e magicamente caricato e lanciato in **FreeCAD**, visualizzerà un cubo (questo sarà proprio un cubo) nella **vista 3D**.

Ora complichiamo la faccenda, il cubo è fatto, per comprendere bene le cose dobbiamo **orientarci** nello spazio, cioè dobbiamo rispondere a due domande:

- a cosa corrispondono in termini di direzione degli assi le variabili che passiamo al momento della costruzione.
- che **posizione** occupa l'oggetto nello spazio 3D.

Per rispondere alla seconda domanda, possiamo attivare la visualizzazione dell'origine nella finestra grafica, usando il menu **Visualizza** ⇒ **Origine degli assi**, oppure usando aggiungendo al metodo **setview()**

```
FreeCAD.Gui.activeDocument().activeView().setAxisCross(True)
```

in questo modo si visualizza l'origine degli assi e le direzioni positive dei tre assi X, Y e Z.

Programma 3.1: metodo cilindro

```

1 def base_cyl(nome, ang, rad, alt):
2     obj = DOC.addObject("Part::Cylinder", nome)
3     obj.Angle = ang
4     obj.Radius = rad
5     obj.Height = alt
6
7     DOC.recompute()
8
9     return obj

```

Per rispondere invece alla prima domanda e per esercitarvi, provate a modificare i valori della riga:

```
obj = cubo("cubo_di_prova", 5, 5, 5)
```

e vedere a cosa corrispondono i valori lung, larg e alt, relativamente agli assi X, Y e Z.

Per poter proseguire nella trattazione, abbiamo necessità di avere almeno due geometrie, creiamone una seconda, questa volta un cilindro, inseriamo subito al metodo **cubo** le righe riportate nel listato 3.1 denominato **metodo cilindro**.

Questo definisce il metodo **base\_cyl** che crea un secondo tipo di geometria, un cilindro cioè un oggetto **Part::Cylinder**, invocando il metodo con gli appropriati parametri:

```
obj_1 = base_cyl('primo cilindro', 360, 2, 25)
```

Posizionando la riga precedente subito dopo quella che crea il cubo.

Illustriamo brevemente i “parametri” che forniamo al metodo:

1. **nome** Il nome che desideriamo abbia la geometria
2. **ang** cioè l'angolo che viene assegnato alla proprietà **Angle**.  
Possiamo creare solo una porzione di cilindro fornendo valori inferiori a 360°
3. **rad** il raggio che viene assegnato alla proprietà **Radius**.
4. **alt** l'altezza che viene assegnata alla proprietà **Height**

Ora dovremmo avere un programma che assomiglia al listato 3.1.7, che una volta lanciato dovrebbe mostrare nella **vista combinata**, qualcosa che assomiglia alla figura 5.1a a pagina 41.

### 3.1.7. Listato - figure-base.py

```
1 #
```

```
2  """cubo-prova.py
3
4  This code was written as an sample code
5  for "FreeCAD Scripting Guide"
6
7  Author: Carlo Dormeletti
8  Copyright: 2020
9  Licence: CC BY-NC-ND 4.0 IT
10 """
11
12 import FreeCAD
13 from FreeCAD import Base, Vector
14 import Part
15 from math import pi, sin, cos
16
17 DOC = FreeCAD.activeDocument()
18 DOC_NAME = "Pippo"
19
20 def clear_doc():
21     """
22     Clear the active document deleting all the objects
23     """
24     for obj in DOC.Objects:
25         DOC.removeObject(obj.Name)
26
27 def setview():
28     """Rearrange View"""
29     FreeCAD.Gui.SendMsgToActiveView("ViewFit")
30     FreeCAD.Gui.activeDocument().activeView().viewAxometric(
31         ↵ )
32
33 if DOC is None:
34     FreeCAD.newDocument(DOC_NAME)
35     FreeCAD.setActiveDocument(DOC_NAME)
36     DOC = FreeCAD.activeDocument()
37
38 else:
39
40     clear_doc()
41
42
43 # EPS= tolerance to uset to cut the parts
44 EPS = 0.10
45 EPS_C = EPS * -0.5
46
```

```
47
48 def cubo(nome, lung, larg, alt):
49     obj_b = DOC.addObject("Part::Box", nome)
50     obj_b.Length = lung
51     obj_b.Width = larg
52     obj_b.Height = alt
53
54     DOC.recompute()
55
56     return obj_b
57
58 def base_cyl(nome, ang, rad, alt ):
59     obj = DOC.addObject("Part::Cylinder", nome)
60     obj.Angle = ang
61     obj.Radius = rad
62     obj.Height = alt
63
64     DOC.recompute()
65
66     return obj
67
68
69 obj = cubo("cubo_di_prova", 5, 5, 5)
70
71 setview()
```



# Capitolo 4

## Posizionamento

Quanto abbiamo ottenuto dall'esempio precedente, non assomiglia ad una scultura del Canova, ma già possiamo notare alcune cose:

1. Il cubo è costruito con lo spigolo in  $(0, 0, 0)$
2. il cilindro è posizionato con il centro della faccia inferiore in  $(0, 0, 0)$

Questa particolarità ci aiuta ad introdurre un concetto importante, le geometrie sono costruite prendendo come punto di partenza un punto preciso che purtroppo da quanto visto finora non è il medesimo per ogni geometria, la cosa diventa complicata quando vogliamo costruire cose complesse, perché non avendo un punto comune dobbiamo giostrarci, tra i vari riferimenti.

Per cui nella costruzione e soprattutto nel posizionamento della geometria, è necessario tenere presente questa particolarità.

Questo argomento è una delle cose “complicate” di **FreeCAD**, per cui chiariamo fin d'ora alcuni concetti.

### 4.1. Il riferimento di costruzione

Una delle particolarità di **FreeCAD** che all'inizio può destare qualche perplessità è il riferimento in base al quale viene costruito l'oggetto.

Nella tabella 4.1 nella pagina successiva, elenchiamo i punti di riferimento dei vari oggetti.

Concettualmente parlando, il miglior punto di riferimento è l'origine  $(0,0,0)$  per fortuna alcune geometrie hanno già il loro riferimento di costruzione in quel punto, una soluzione è quella di modificarne il posizionamento, questo come quasi tutto in **FreeCAD** dipende molto dai gusti di ciascuno e dalle tecniche di programmazione, però una soluzione che in genere pone meno problemi è quello di definire un punto convenzionale di costruzione per le geometrie, uniformando almeno le dimensioni X e Y del centro della geometria, infatti alcuni oggetti come si vede nella tabella usano l'origine come centro, ma se abbiamo un centro e un diametro è relativamente facile modificare la posizione Z.

Geometria	Punto di riferimento
<b>Part::Box</b>	vertice sinistro (minimo x), frontale (minimo y), in basso (minimo z)
<b>Part::Sphere</b>	centro della sfera (centro del suo contenitore cubico)
<b>Part::Cylinder</b>	centro della faccia di base
Part::Cone	centro della faccia di base (o superiore se il raggio della faccia di base vale 0)
Part::Torus	centro del toro
Part::Wedge	spigolo di Xmin Zmin

Tabella 4.1.: punti di riferimento

In questo caso la geometria più problematica è il cubo, vediamo come ridurre in parte la sua problematicità.

Partiamo dal listato 3.1.7 a pagina 26 denominato **Listato - figure-base.py** e modifichiamo il metodo **cubo** nel modo seguente:

```
def cubo(nome, lung, larg, alt, cent = False):
    obj_b = DOC.addObject("Part::Box", nome)
    obj_b.Length = lung
    obj_b.Width = larg
    obj_b.Height = alt

    if cent == True:
        obj_b.Placement = FreeCAD.Placement(
            Vector(lung * -0.5, larg * -0.5, 0),
            FreeCAD.Rotation(0, 0, 0), FreeCAD.Vector(0,
                ↪ 0,0))

    DOC.recompute()

    return obj_b
```

Abbiamo leggermente modificato il metodo introducendo un parametro opzionale **cent** a cui abbiamo assegnato un valore di default di **False**, in questo modo possiamo riutilizzare tutto il codice precedente ma quando serve centrare il cubo sull'origine basta aggiungere alla chiamata un **True** alla fine.

Invocando il metodo nel modo classico con:

```
obj1 = cubo("cubo_cent", 10, 20, 10)

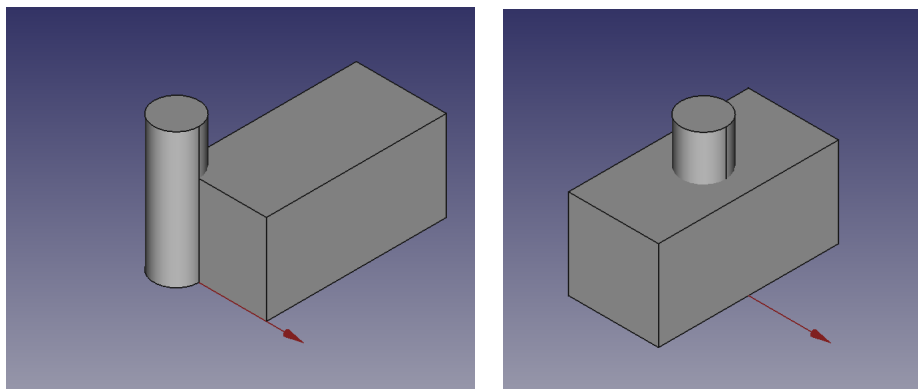
obj2 = base_cyl("cilindro", 360, 2.5, 15)
```

otteniamo il cubo e il cilindro come possiamo vedere in figura 4.1a.

Mentre invocando il metodo in questo modo:

```
obj1 = cubo("cubo_cent", 10, 20, 10, True)
obj2 = base_cyl("cilindro", 360, 2.5, 15)
```

per ritrovarci il cubo con il centro della faccia di base in (0,0,0), uniformandoci allo stesso posizionamento di base del cilindro e otterremo invece la figura 4.1b.



(a) Cubo e cilindro con rdc originali.

(b) Cubo e cilindro con posizionamento modificato.

Figura 4.1.

## 4.2. Posizionamento

Abbiamo introdotto nel metodo precedente la proprietà **Placement** della geometria, per poter posizionare il cubo con il centro della faccia base in (0, 0, 0)

Complichiamo di poco il metodo **cubo** in questo modo:

```
def cubo(nome, lung, larg, alt, cent = False, off_z = 0):
    obj_b = DOC.addObject("Part::Box", nome)
    obj_b.Length = lung
    obj_b.Width = larg
    obj_b.Height = alt

    if cent == True:
        posiz = Vector(lung * -0.5, larg * -0.5, off_z)
    else:
        posiz = Vector(0, 0, off_z)

    obj_b.Placement = FreeCAD.Placement(
        posiz,
        FreeCAD.Rotation(0, 0, 0),
        FreeCAD.Vector(0,0,0)
```

```

    )

    DOC.recompute()

    return obj_b

```

Abbiamo aggiunto un parametro **off\_z** che se presente modifica la posizione Z della geometria.

in questo modo possiamo alzare o abbassare il nostro cubo di quanto vogliamo, ad esempio scriviamo le righe finali del codice per leggere:

```

obj1 = cubo("cubo_cyl", 10, 20, 10, True, 10)
obj2 = base_cyl("cilindro", 360, 2.5, 15 )

print("Cubo Base = ", obj1.Placement)
print("Cilindro = ", obj2.Placement)

```

Lanciando il programma vedremo che viene visualizzato il cubo, posizionato più in alto rispetto all'origine, la cosa non ci stupisce perché abbiamo introdotto la proprietà **off\_z** mettendola a **10**.

Se osserviamo poi nella **finestra dei rapporti** vedremo che vengono stampati le proprietà **Placement** dei due oggetti.

```

Cubo Base = Placement [Pos=(-5,-10,10), Yaw-Pitch-Roll
↪      =(0,0,0)]
Cilindro = Placement [Pos=(0,0,0), Yaw-Pitch-Roll=(0,0,0)]

```

Potete naturalmente giocare con i vari parametri e creare cose nuove.

Vi lascio come esercizio il compito di aggiungere al metodo **base\_cyl** un parametro per il suo posizionamento in altezza, in modo da poterlo posizionare in Z a piacimento.

#### 4.2.1. La proprietà Placement

Per modificare la posizione abbiamo utilizzato la proprietà **Placement**.

Questa proprietà presenta una varietà di scritture, se notiamo nel metodo **cubo** la abbiamo specificata come:

```

obj_b.Placement = FreeCAD.Placement(
    posiz,
    FreeCAD.Rotation(0, 0, 0),
    FreeCAD.Vector(0,0,0)
)

```

Questo è uno dei tanti metodi per specificarla, lo vedremo in dettaglio più avanti.

Una delle “scritture” più comuni è però la seguente:

```
FreeCAD.Placement(
    Vector(pos_x, pos_y, pos_z),
    FreeCAD.Rotation(Vector(axis_x, axis_y, axis_z), ang)
)
```

La trovate abbastanza spesso in giro, e riflette le proprietà che trovate nella **vista combinata** nella parte **editore delle proprietà** come potete vedere nell’immagine 4.2 che visualizza la **Ling. Vista**.

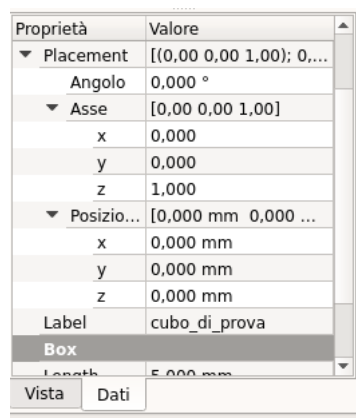


Figura 4.2.: L’editor delle proprietà

L’ordine delle voci nella **Ling. Vista** varia rispetto a quello con cui sono espressi nel codice, vediamo che **Placement** possiede una freccia sulla sinistra che se premuta espande la vista come nella figura 4.2 e mostra 3 sotto proprietà, **Angolo**, **Asse**, **Posizione**, che elenchiamo nella tabella 4.2.

Nome	variabile	Descrizione
<b>Angolo</b>	<b>ang</b>	angolo di rotazione in gradi
<b>Asse</b>	<b>axis_x</b> , <b>axis_y</b> , <b>axis_z</b>	asse di riferimento <b>Vettore</b> con valori 0 o 1 che selezionano l’asse di riferimento, sono ammessi più valori
<b>Posizione</b>	<b>pos_x</b> , <b>pos_y</b> , <b>pos_z</b>	<b>Vettore</b> di Traslazione (X, Y, Z)

Tabella 4.2.: proprietà Placement

Sorprendentemente questa “scrittura” non è quella che otteniamo se chiediamo la stampa della proprietà dell’oggetto, come abbiamo fatto poco fa:

```
Cubo Base = Placement [Pos=(-5,-10,10), Yaw-Pitch-Roll
    ↪ =(0,0,0)]
```

Possiamo facilmente riconoscere un componente chiamato **Pos**, che è il **Vettore** di Traslazione, seguito da una **tupla** di tre valori chiamata **Yaw-Pitch-Roll**.

Questo tipo di posizionamento usa gli **angoli di Eulero o di Tait-Bryan**, i nomi delle rotazioni (imbardata, rollio e beccheggio), sicuramente ricorderanno a qualcuno i termini tipici della navigazione navale o aeronautica:

Nome	Descrizione	Angolo
Yaw (imbardata)	rotazione rispetto a Z	<b>Psi</b> $\psi$
Pitch (beccheggio)	rotazione rispetto a Y (alzare o abbassare il muso)	<b>Phi</b> $\varphi$
Roll (rollio)	rotazione rispetto a X (dondolare le ali)	<b>Theta</b> $\theta$

Tabella 4.3.: Angoli di Eulero

Il modo più comodo per manipolare la rotazione di un oggetto è quello di usare (ed immaginare) la rotazione attorno al centro geometrico dell'oggetto, a meno di non avere delle ragioni per usare altri punti di riferimento.

Passando alla proprietà **Placement**, un valore scritto nel modo seguente:

```
FreeCAD.Placement(
    Vector(pos_x, pos_y, pos_z),
    FreeCAD.Rotation(Yaw, Pitch, Roll),
    Vector(c_rot_x, c_rot_y, c_rot_z)
)
```

Notate subito che sono passate tre componenti differenti, descritte nella tabella 4.4

Componente	Descrizione
<b>pos_x, pos_y, pos_z</b>	il <b>Vettore</b> di Traslazione
<b>Yaw, Pitch, Roll</b>	angoli di rotazione
<b>c_rot_x, c_rot_y, c_rot_z</b>	Centro di rotazione

Tabella 4.4.: Rotazione usando gli Angoli di Eulero

Il **posizionamento** e la rotazione sono cose relativamente complicate, forse le più ostiche da comprendere nella modellazione 3D, possono essere gestite usando qualche “astuzia” che vedremo a tempo debito.

La cosa più sorprendente è che la scrittura della proprietà comprende tre componenti, ma non corrisponde a quanto otteniamo con la stampa, che elenca solo due componenti.

Questa è una delle particolarità di **FreeCAD**, la rotazione può essere espressa in molti modi, chiaramente questo è frutto della natura collaborativa dello sviluppo di **FreeCAD**, che è stato scritto “a più mani”, e evidentemente sono stati scelti modi diversi per parti diverse del programma.

Questo comunque non inficia la bontà del software, che comunque compie il suo lavoro e lo fa in modo egregio, in certe parti in modo superiore a molti software commerciali, blasonati.

Per inciso durante la modellazione risulta molto utile, inserire delle istruzioni di **print()** all’interno del codice, per visualizzare ad esempio la correttezza dei calcoli, o come in questo caso, il valore di alcune variabili o proprietà degli oggetti.

Abbiamo scoperto uno dei segreti della modellazione, se eseguiamo una operazione booleana, il risultato possiede una propria proprietà **Placement**, e la cosa non deve sembrare strana, indipendentemente dal riferimento di costruzione delle geometrie di base, la geometria finale possiede un proprio riferimento di costruzione, che è (0,0,0) anche se quel punto non fa parte della geometria.

#### 4.2.2. Listato - Riferimento costruzione

```
1 #
2 """rif-cost-full.py
3
4     This code was written as an sample code
5     for "FreeCAD Scripting Guide"
6
7     Author: Carlo Dormeletti
8     Copyright: 2020
9     Licence: CC BY-NC-ND 4.0 IT
10 """
11
12 import FreeCAD
13 from FreeCAD import Base, Vector
14 import Part
15 from math import pi, sin, cos
16
17 DOC = FreeCAD.activeDocument()
18 DOC_NAME = "Pippo"
19
```

```
20 def clear_doc():
21     """
22     Clear the active document deleting all the objects
23     """
24     for obj in DOC.Objects:
25         DOC.removeObject(obj.Name)
26
27 def setview():
28     """Rearrange View"""
29
30     FreeCADGui.activeDocument().activeView().viewAxometric(
31         ↪ )
32     FreeCADGui.activeDocument().activeView().setAxisCross(
33         ↪ True)
34     FreeCADGui.SendMsgToActiveView("ViewFit")
35
36 if DOC is None:
37     FreeCAD.newDocument(DOC_NAME)
38     FreeCAD.setActiveDocument(DOC_NAME)
39     DOC = FreeCAD.activeDocument()
40
41 else:
42     clear_doc()
43
44
45 # EPS= tolerance to uset to cut the parts
46 EPS = 0.10
47 EPS_C = EPS * -0.5
48
49
50 def cubo(nome, lung, larg, alt, cent = False, off_z = 0):
51     obj_b = DOC.addObject("Part::Box", nome)
52     obj_b.Length = lung
53     obj_b.Width = larg
54     obj_b.Height = alt
55
56     if cent == True:
57         posiz = Vector(lung * -0.5, larg * -0.5, off_z)
58     else:
59         posiz = Vector(0, 0, off_z)
60
61     obj_b.Placement = FreeCAD.Placement(
62         posiz,
63         FreeCAD.Rotation(0, 0, 0),
```



```
64         FreeCAD.Vector(0,0,0)
65     )
66
67     DOC.recompute()
68
69     return obj_b
70
71
72 def base_cyl(nome, ang, rad, alt ):
73     obj = DOC.addObject("Part::Cylinder", nome)
74     obj.Angle = ang
75     obj.Radius = rad
76     obj.Height = alt
77
78     DOC.recompute()
79
80     return obj
81
82 # definizione oggetti
83
84 obj1 = cubo("cubo_cyl", 10, 20, 10, True, 10)
85 obj2 = base_cyl("cilindro", 360, 2.5, 15 )
86
87 print("Cubo Base = ", obj1.Placement)
88 print("Cilindro = ", obj2.Placement)
89
90 setview()
```

# Capitolo 5

## Le operazioni booleane

Una geometria complessa può essere creata in molti modi, **FreeCAD** offre molte funzioni per creare geometrie anche molto complesse, partendo da geometrie più semplici, il suo arsenale di strumenti è potente e versatile.

In questo capitolo analizzeremo alcune operazioni sulle geometrie, che vanno padroneggiate in quanto sono le tecniche fondamentali per la creazione di geometrie complesse.

Abbiamo già visto una delle geometrie di base, il parallelepipedo, altre le tratteremo nel proseguimento del discorso, per non complicare la discussione con inutili elencazioni di nozioni.

Ma se le geometrie che abbiamo a disposizione non riescono a produrre, la forma voluta, abbiamo a disposizione un set di strumenti per crearne di nuove.

Partendo da semplici geometrie e operando sulle stesse utilizzando le **operazioni booleane** possiamo creare geometrie molto complesse.

Le **operazioni booleane**, che nonostante il nome strano sono relativamente semplici, sono una delle basi per la modellazione 3D, chi ha studiato gli insiemi, troverà delle analogie, almeno nei nomi delle operazioni e non è un caso.

Vediamo le **operazioni booleane**:

Nome	Descrizione
<b>Unione</b>	<b>Part::Fuse</b> oppure <b>Part::MultiFuse</b>
<b>Sottrazione</b>	<b>Part::Cut</b>
<b>Intersezione</b>	<b>Part::MultiCommon</b>

Questa spiegazione presuppone un metodo di lavoro che parte dal listato nella sezione 2.4 a pagina 20 con il titolo **Listato - schema base**, espandendolo mediante l'aggiunta delle linee di codice proposte.

Le porzioni di codice sono tratte dal listato definitivo **Operazioni booleane - esempio completo**, listato 5.4 a pagina 42 e i numeri di riga mostrati fanno riferimento a quel listato.

Programma 5.1: Unione

```
def fuse_obj(nome, obj_0, obj_1):
    obj = DOC.addObject("Part::Fuse", nome)
    obj.Base = obj_0
    obj.Tool = obj_1
    obj.Refine = True
    DOC.recompute()
```

## 5.1. Unione

Adesso si comincia a fare sul serio.

Uniamo le due figure usando **Part::Fuse**, lo facciamo però in modo Pythonico cioè usando una bel metodo, come potete leggere nel Programma 5.1.

Questa porzione di codice la inseriamo alla riga 58 del nostro schema base, subito dopo il metodo **base\_cyl**.

Descriviamo brevemente le proprietà:

- **Base** è la geometria a cui dobbiamo aggiungere la seconda geometria.
- **Tool** è la geometria da aggiungere.
- l'uso della proprietà **Refine** settata a **True** che cerca di “affinare” la geometria risultante eliminando le facce inutili e le cuciture.

Per usarla per realizzare una geometria, invochiamo il metodo con gli appropriati parametri:

```
fuse_obj("cubo-cyl-fu", obj, obj1)
```

L'invocazione, va posizionata “ovviamente” dopo aver creato le geometrie, quindi subito dopo la riga 71 che crea la geometria **primo cilindro** invocando il metodo **base\_cyl**.

Lanciando il programma apparentemente non otteniamo nulla, infatti il risultato è la figura 5.1b a pagina 41, notiamo però che che nella **vista combinata** sono “spariti” i due oggetti e ne è apparso uno solo chiamato **cubo-cyl-fu**.

Gli oggetti però non sono “spariti”, sono diventati parte dell'oggetto **cubo-cyl-fu**, infatti cliccando sulla freccia ► essa diventa ▼ e appaiono “magicamente” i nomi dei due oggetti, in grigio chiaro, per indicare che sono i “componenti” dell'oggetto **cubo-cyl-fu**.

Soffermiamoci un attimo sul funzionamento della **vista combinata**, solo due parole, se selezionate un oggetto e premete la barra spaziatrice, essa si comporta come un interruttore per la **Visibilità** dell'oggetto.

Se ad esempio rendiamo invisibile l'oggetto **cubo-cyl-fu** e poi rendiamo visibile l'oggetto **primo cilindro**, possiamo controllare il suo posizionamento, ovviamente con solo due oggetti è quasi inutile, ma con molti diventa essenziale.

Programma 5.2: Unione multipla

```

78 def mfuse_obj(nome, obj_0, obj_1):
79     obj = DOC.addObject("Part::MultiFuse", nome)
80     obj.Shapes = (obj_0, obj_1)
81     obj.Refine = True
82     DOC.recompute()
83
84     return obj

```

Programma 5.3: Sottrazione

```

86 def cut_obj(nome, obj_0, obj_1):
87     obj = DOC.addObject("Part::Cut", nome)
88     obj.Base = obj_0
89     obj.Tool = obj_1
90     obj.Refine = True
91     DOC.recompute()
92
93     return obj

```

L'operazione **unione** possiede anche una seconda forma, adatta per quando è necessario unire più di due oggetti, la presentiamo nel Programma 5.2.

Dal listato possiamo notare alla riga 80 la proprietà **Shapes** a cui viene passata una **tupla** di geometrie, ovviamente la tupla contiene i due oggetti che abbiamo creato, ma può contenere quanti oggetti desideriamo, utile ad esempio se dobbiamo passare una lista di oggetti creati in modo automatico. (lo vedremo più avanti quanto affronteremo alcune tecniche di creazione “avanzata”).

## 5.2. Sottrazione

La sottrazione, detta anche “Taglio” traducendo letteralmente la parola **Cut**, sottrae una geometria da un'altra, si invoca creando un oggetto **Part::Cut**, come possiamo leggere nel Programma 5.3.

Inseriamo le righe del Programma 5.3 alla riga 66 del nostro schema base, subito dopo al metodo **base\_cyl**.

Lo usiamo come è dovrebbe essere oramai diventato usuale invocando il metodo con gli appropriati parametri.

```
cut_obj("cubo-cyl-cu", obj, obj1)
```

Mettiamo il segno di **#** davanti all'invocazione di **fuse\_obj**, copiamo la riga sopra e lanciamo il programma, ottenendo come risultato è la figura 5.1c nella pagina seguente.

Le proprietà dell'oggetto sono sostanzialmente le stessa del oggetto **Part::Fuse**.

- **Base** è la geometria da cui dobbiamo sottrarre
- **Tool** è la geometria da sottrarre
- **Refine** dal comportamento analogo a quella vista in **Part::Fuse**.

Una piccola nota, nell'uso di **Part::Fuse**, l'ordine degli oggetti non è importante, stiamo aggiungendo e l'ordine degli addendi non cambia il risultato dell'operazione; Ovviamente utilizzando lo strumento **Part::Cut** l'ordine è importante.

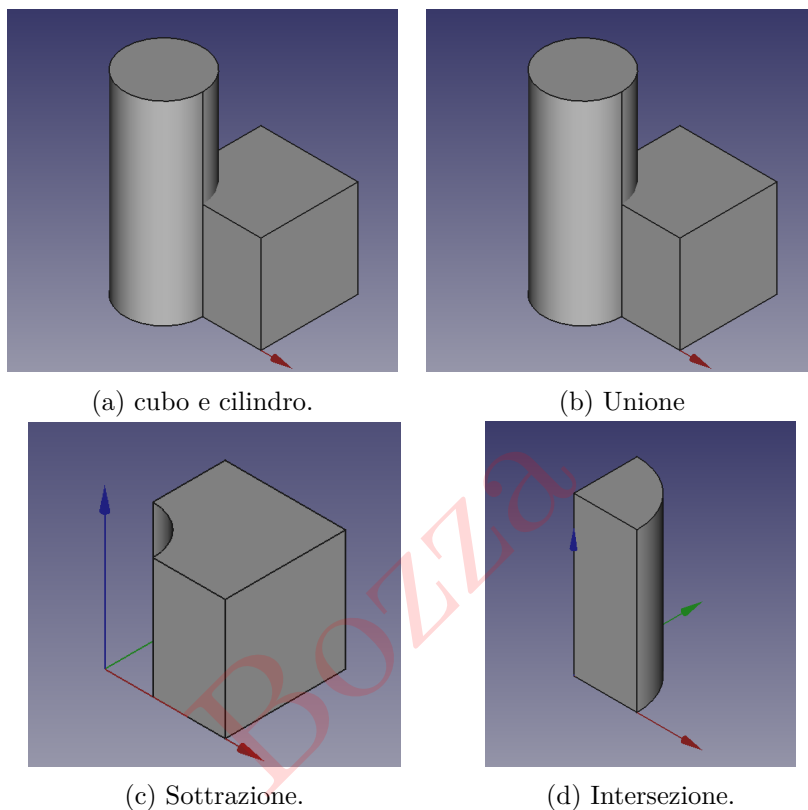


Figura 5.1.: Le operazioni Booleane.

### 5.3. Intersezione

L'Intersezione, che si ottiene creando un oggetto **Part::MultiCommon**, estrae la parte comune alle geometrie.

Creiamo questo metodo come nel Programma 5.4 nella pagina successiva.

Questa porzione di codice la inseriamo nel nostro schema base, subito dopo al metodo **cut\_obj**.

Programma 5.4: Intersezione

```

95 def int_obj(nome, obj_0, obj_1):
96     obj = DOC.addObject("Part::MultiCommon", nome)
97     obj.Shapes = (obj_0, obj_1)
98     obj.Refine = True
99     DOC.recompute()
100
101     return obj

```

Lo usiamo come è dovrebbe essere diventato usuale invocando il metodo con gli appropriati parametri.

```
int_obj("cubo-cyl-is", obj, obj1)
```

Mettiamo il segno di `#` davanti all'invocazione di `cut_obj`, copiamo la riga sopra e lanciamo il programma, ottenendo come risultato è la figura 5.1d nella pagina precedente.

La costruzione e le proprietà sono analoghi a quelli di `Part::MultiFuse`, valgono le stesse considerazioni fatte per la `tupla` passata alla proprietà `Shapes`.

## 5.4. Operazioni booleane - esempio completo

```

1 #
2 """ob-ex-full.py
3
4     This code was written as an sample code
5     for "FreeCAD Scripting Guide"
6
7     Author: Carlo Dormeletti
8     Copyright: 2020
9     Licence: CC BY-NC-ND 4.0 IT
10 """
11
12 import FreeCAD
13 from FreeCAD import Base, Vector
14 import Part
15 from math import pi, sin, cos
16
17 DOC = FreeCAD.activeDocument()
18 DOC_NAME = "Pippo"
19
20 def clear_doc():
21     """
22     Clear the active document deleting all the objects

```

```
23     """
24     for obj in DOC.Objects:
25         DOC.removeObject(obj.Name)
26
27 def setview():
28     """Rearrange View"""
29     FreeCAD.Gui.SendMsgToActiveView("ViewFit")
30     FreeCAD.Gui.activeDocument().activeView().viewAxometric(
31         ↪ )
32     FreeCAD.Gui.activeDocument().activeView().setAxisCross(
33         ↪ True)
34
35 if DOC is None:
36     FreeCAD.newDocument(DOC_NAME)
37     FreeCAD.setActiveDocument(DOC_NAME)
38     DOC = FreeCAD.activeDocument()
39 else:
40
41     clear_doc()
42
43
44 # EPS= tolerance to uset to cut the parts
45 EPS = 0.10
46 EPS_C = EPS * -0.5
47
48
49 def cubo(nome, lung, larg, alt):
50     obj_b = DOC.addObject("Part::Box", nome)
51     obj_b.Length = lung
52     obj_b.Width = larg
53     obj_b.Height = alt
54
55     DOC.recompute()
56
57     return obj_b
58
59 def base_cyl(nome, ang, rad, alt ):
60     obj = DOC.addObject("Part::Cylinder", nome)
61     obj.Angle = ang
62     obj.Radius = rad
63     obj.Height = alt
64
65     DOC.recompute()
66
```

```
67     return obj
68
69 def fuse_obj(nome, obj_0, obj_1):
70     obj = DOC.addObject("Part::Fuse", nome)
71     obj.Base = obj_0
72     obj.Tool = obj_1
73     obj.Refine = True
74     DOC.recompute()
75
76     return obj
77
78 def mfuse_obj(nome, obj_0, obj_1):
79     obj = DOC.addObject("Part::MultiFuse", nome)
80     obj.Shapes = (obj_0, obj_1)
81     obj.Refine = True
82     DOC.recompute()
83
84     return obj
85
86 def cut_obj(nome, obj_0, obj_1):
87     obj = DOC.addObject("Part::Cut", nome)
88     obj.Base = obj_0
89     obj.Tool = obj_1
90     obj.Refine = True
91     DOC.recompute()
92
93     return obj
94
95 def int_obj(nome, obj_0, obj_1):
96     obj = DOC.addObject("Part::MultiCommon", nome)
97     obj.Shapes = (obj_0, obj_1)
98     obj.Refine = True
99     DOC.recompute()
100
101     return obj
102
103 # definizione oggetti
104
105 obj = cubo("cubo_di_prova", 5, 5, 5)
106
107 obj1 = base_cyl('primo cilindro', 360, 2, 10)
108
109 # mfuse_obj("cubo-cyl-fu", obj, obj1)
110
111 # cut_obj("cubo-cyl-cu", obj, obj1)
112
```



```
113 int_obj("cubo-cyl-is", obj, obj1)
114
115 setview()
```

BOZZA

# Capitolo 6

## Modellazione avanzata

Finora abbiamo creato direttamente gli oggetti usando le forma base, per forme complesse esistono alcune tecniche molto potenti che permettono di disegnare una porzione di piano chiusa e poi di elaborarla in modo da creare un oggetto 3D.

Per utilizzare questa tecnica dobbiamo fare uso dei concetti esposti nella sezione 3.1.4 a pagina 24 **Topologia**, se non li avete ben chiari rileggeteli ora.

Non ci muoveremo dal modulo Part.

Per questa parte di spiegazione partiremo dal listato mostrato nella sezione 6.1.1 a pagina 51 con il titolo **Listato - estrusione**.

Il listato contiene abbastanza carne al fuoco, analizziamolo in dettaglio:

```
47 def reg_poly(center=Vector(0, 0, 0), sides=6, dia=6,  
48             align=0, outer=1):
```

Con queste righe definiamo il metodo che creerà il nostro oggetto, la definizione e di conseguenza la chiamata possiede di molti parametri, per cui viene dotata di una nutrita docstring, costruita secondo i “canoni” di Python, contiene la descrizione della funzione, e **Keywords Arguments:**, seguito dalla linea 53 alla linea 58 dalla lista completa dei parametri con una spiegazione sommaria di quello che fanno.

Per chi non legge bene l’inglese, cerchiamo di elencarli:

- **center** = un Vettore che contiene il centro del poligono
- **sides** = il numero dei lati del poligono da un minimo di 3 a quello che volete voi
- **dia** = il diametro del **cerchio base** con una specificazione che può trattarsi o dell’apotema (diametro interno) o del diametro esterno
- **align** = allineamento (0 oppure 1), allineare il poligono all’asse (X)
- **outer** = 0 = apotema, 1 diametro esterno, cioè specifica che di che tipo è la misura fornita con il parametro **dia**

Il parametro **outer** merita una spiegazione in merito alla parola apotema, L’apotema è il **raggio della circonferenza interna** del poligono, nel programma è usato in modo improprio. <sup>1</sup>

---

<sup>1</sup>Non possiamo scrivere un trattato in una **docstring** per cui dobbiamo semplificare, la scelta della

Segue poi la “magia nera” del calcolo dei punti di vertice del poligono e della loro assegnazione alla **lista** **vertex**.

La linea:

```
vertex = []
```

si occupa di creare la **lista** **vertex** creandola vuota, la parentesi quadra aperta e chiusa creano la lista vuota.

```
vertex.append(Vector(vpx, vpy, 0))
```

si occupa di aggiungere **append()** è il metodo per aggiungere alla lista **vertex** qualcosa.

Il qualcosa è il Vettore che contiene il punto di vertice appena calcolato dal ciclo **for**, ma potete notare che all’interno delle equazioni per calcolare i punti di vertice ci siamo riferiti alla variabile **center** usando delle parentesi quadre, questo costrutto, si chiama **indicizzazione**, in pratica dice prendi l’elemento numero x dalla lista **center**, ma se ricordate bene **center** è un Vettore non una lista.

In Python molte cose possono essere rappresentate come **sequenze** e quindi indicizzate: **center[0]** è l’elemento 0 della sequenza Vettore, quindi il valore di X, ovviamente 1 è il secondo elemento, quindi il valore di Y, **center[2]** sarebbe Z, ma qui non ci è servito.

Veniamo ora alla parte sostanziosa:

```
obj = Part.makePolygon(vertex)
```

Assegniamo alla variabile **obj** il risultato della chiamata al metodo **Part.makePolygon()**, a cui abbiamo passato l’elenco dei vertici, attraverso una **lista**.

Questo elenco di punti però deve rispettare alcune regole:

- Il punto iniziale e il punto finale devono coincidere, perché il poligono deve essere **chiuso**.
- punti devono essere ordinati tra di loro (per convenzione si usa l’ordinamento antiorario).

La funzione **Part.makePolygon()** è molto versatile, ad esempio per contenere i punti si può usare anche una **tupla** che in pratica è una lista immutabile, nel senso che una volta creata non può essere modificata.

L’utilità della **tupla** deriva dal fatto che risparmia memoria, notate che come abbiamo già accennato nella sezione 2.2.4 a pagina 17 (**Le condizioni**), la **tupla** è identificata dall’uso delle “parentesi tonde” nella sua definizione, la **lista** usa le “parentesi quadre”.

Se però avete dei vettori vanno bene lo stesso, la funzione **Part.makePolygon()**, accetta:

- Una lista di Vettori

---

parola “apotema” risiede nella sua compattezza per indicare che il dato è relativo al **diametro del cerchio interno** rispetto al **diametro del cerchio esterno**, usando la definizione corretta, si cade facilmente in confusione, perché il nostro cervello confonde molto facilmente due concetti che differiscono in pratica per solo due lettere (in/es)terno.

- Una lista di tuple
- Una tupla di tuple

Una sua particolarità, che potrebbe generare errori è il fatto che non accetta una **lista di liste**, per cui i punti vanno racchiusi tra parentesi tonde, oppure generati in modo automatico come una **tupla**.

Fatte queste considerazioni proseguiamo nel discorso, con la linea:

```
wire = Part.Wire(obj)
```

trasformiamo l'oggetto poligono in un oggetto di tipo **Polilinea (Wire)**, mentre usando il metodo **Face**:

```
poly_f = Part.Face(wire)
```

Trasformiamo l'oggetto **Polilinea (Wire)** in una **Faccia (Face)**, ricordiamo dalla definizione che la **Faccia** è una parte di piano delimitato dal una **Polilinea** chiusa.

Questa funzione è stata creata per una funzione eminentemente pratica, creare esagoni, lo si può intuire dai valori di default, e dalla scelta di quali parametri passare al metodo.

Gli esagoni sono molto usati in quanto sono la base per la costruzione di dadi, o di alloggiamenti per gli stessi, cosa molto comune nella stampa 3D.

Un dado si distingue in base al diametro e al passo della filettatura interna, ad esempio “dado 3MA”, ma la misura che più mi interessa è il diametro della chiave per manovrarlo che per inciso nei dadi 3MA è da 5,5 mm, da qui la scelta del parametro come diametro del cerchio interno.

Internamente, il metodo utilizza correttamente il raggio del poligono, i vertici infatti sono situati sul cerchio esterno del poligono, noi al metodo forniamo invece diametro interno e centro.

Nei calcoli interni sono state usate le formule per trasformare l'apotema in raggio del cerchio esterno, e i diametri vengono trasformati in raggi moltiplicando per 0.5.

La parte di piano ottenuta ci servirà per lavorarci sopra utilizzando alcuni strumenti.

## 6.1. Estrusione

Introduciamo ora lo strumento “estrusione”, utilizzando la funzione **extrude**, applicata ad una porzione chiusa di piano.

Ad esempio usando la forma esagonale creata poco fa ed “estrudendola” possiamo creare un dado. All'incirca come otteniamo quando schiacciamo il tubetto di dentifricio, il salsicciotto che esce prende la forma del buco del tappo.

Per compiere questa operazione scriviamo questo metodo:

```
90 def dado(nome, dia, spess):  
91     polyg = reg_poly(Vector(0, 0, 0), 6, dia, 0, 0)
```

```

92
93     nut = DOC.addObject("Part::Feature", nome + "_dado")
94     nut.Shape = polyg.extrude(Vector(0, 0, spess))
95
96     return nut

```

Analizziamolo in dettaglio:

```
polyg = reg_poly(Vector(0, 0, 0), 6, dia, 0, 0)
```

Questa linea di codice si occupa di creare la parte di piano, invocando la funzione **reg\_poly**

Ora creiamo il nostro oggetto, però non avendo una forma definita, dobbiamo usare la classe base, cioè un contenitore generico:

```
nut = DOC.addObject("Part::Feature", nome + "_dado")
```

L'oggetto **Part::Feature**, è la classe generica degli oggetti 3D, a cui dobbiamo fornire è una proprietà **Shape** attraverso:

```
nut.Shape = polyg.extrude(Vector(0, 0, spess))
```

Notate la “magia nera” della funzione **extrude** usata in **polyg.extrude()**, ad essa dobbiamo solo fornire un vettore che indica il punto finale dell’estrusione, nel nostro caso estrudiamo l’oggetto fino al punto con **Vector(0, 0, spess)**, ricordando che l’esagono era costruito con il centro in **(0, 0, 0)**, in pratica estrudiamo in direzione Z di **spess**.

Potete liberamente modificare il valore di questo vettore per vedere che risultati ottenete.

L’oggetto vero e proprio va come siamo abituati a fare creato invocando il metodo, in questo modo:

```
dado("Dado", 5.5, 10)
```

Dovreste ottenere il risultato mostrato nella figure 6.1a nella pagina seguente.

Una volta averlo creato noterete che nella **Ling. Vista**, il nostro oggetto avrà solo la proprietà **Base** e la proprietà **Placement**, più che sufficienti per poter operare sull’oggetto.

Usando la sola interfaccia grafica non potremmo modificare l’oggetto, pensate di dover definire tutti i punti a mano, trovando la posizione e inserendo ogni punto con click del mouse, poi estruderlo e infine assegnare la forma ad un oggetto.

Se commettete un errore dovete rifare tutta la trafilatura, a meno di non aver salvato le forme intermedie.

Usando lo scripting, basta editare un paio di parametri e rilanciare il programma, decisamente più comodo.

Nel caso di esempio l’estrusione è stata fatta con un oggetto semplice, se componete profili complessi, ovviamente il risultato sarà più complesso, un esempio semplice semplice

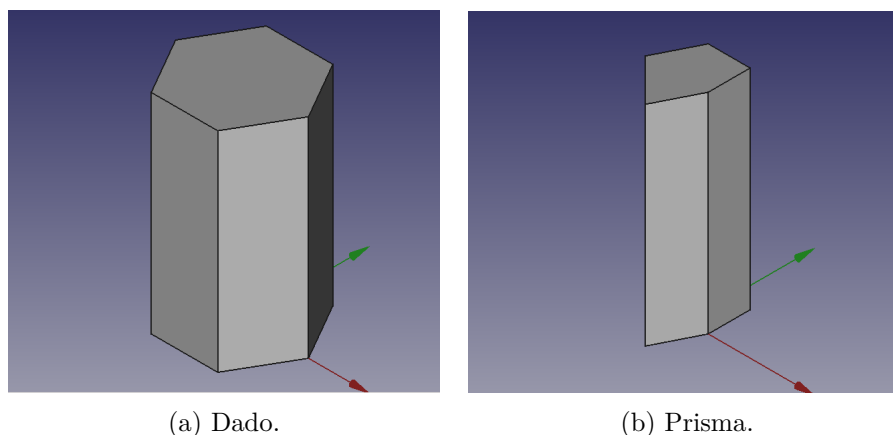


Figura 6.1.

è dato dal metodo `estr_comp` che permette di ottenere il prisma mostrato nella figura 6.1b.

```

99 def estr_comp(nome, spess):
100     vertex = ((-2,0,0), (-1, 2, 0), (1, 2, 0), (1, 0, 0),
101              (0, -2, 0), (-2,0,0))
102
103     obj = Part.makePolygon(vertex)
104     wire = Part.Wire(obj)
105     poly_f = Part.Face(wire)
106
107     cexsh = DOC.addObject("Part::Feature", nome)
108     cexsh.Shape = poly_f.extrude(Vector(0, 0, spess))
109
110     return cexsh

```

La parte rilevante è la definizione del profilo base da cui viene estruso il prisma, la cosa più semplice è creare una **tupla** contenente tante **tuple** quante sono i punti del poligono, ricordate però che vanno ordinati in, meglio se in senso antiorario, lo facciamo in questo modo:

```

vertex = ((-2,0,0), (-1, 2, 0), (1, 2, 0), (1, 0, 0),
          (0, -2, 0), (-2,0,0))

```

Questo modo di definire la lista di punti ci permette di ottenere forme molto complesse in maniera molto semplice, ed efficace, sovrapponendo o disegnando una forma su di un foglio di carta millimetrata e ricavandone il contorno, come una serie di coordinate X e Y, potete velocemente riprodurre una forma in una decina di linee di codice.

La trasformazione della lista in una faccia avviene nelle linee:

```

wire = Part.Wire(obj)
poly_f = Part.Face(wire)

```

con cui otteniamo una **Faccia** che poi estrudiamo nella stessa maniera usata nel metodo **dado**.

Il programma completo oltre che di seguito, è disponibile anche nella pagina di GitHub dedicata a questa guida sotto il nome di **ext-full.py**.

### 6.1.1. Listato - estrusione

```
1 #
2 """ext-full.py
3
4     This code was written as an sample code
5     for "FreeCAD Scripting Guide"
6
7     Author: Carlo Dormeletti
8     Copyright: 2020
9     Licence: CC BY-NC-ND 4.0 IT
10 """
11
12 import FreeCAD
13 from FreeCAD import Base, Vector
14 import Part
15 from math import pi, sin, cos
16
17 DOC = FreeCAD.activeDocument()
18 DOC_NAME = "Pippo"
19
20 def clear_doc():
21     """
22     Clear the active document deleting all the objects
23     """
24     for obj in DOC.Objects:
25         DOC.removeObject(obj.Name)
26
27 def setview():
28     """Rearrange View"""
29     FreeCAD.Gui.SendMsgToActiveView("ViewFit")
30     FreeCAD.Gui.activeDocument().activeView().viewAxometric(
31         ↪ )
32     FreeCAD.Gui.activeDocument().activeView().setAxisCross(
33         ↪ True)
34
35 if DOC is None:
36     FreeCAD.newDocument(DOC_NAME)
37     FreeCAD.setActiveDocument(DOC_NAME)
```

```

37     DOC = FreeCAD.activeDocument()
38
39     else:
40
41         clear_doc()
42
43     # EPS= tolerance to uset to cut the parts
44     EPS = 0.10
45     EPS_C = EPS * -0.5
46
47     def reg_poly(center=Vector(0, 0, 0), sides=6, dia=6,
48                 align=0, outer=1):
49         """
50         This return a polygonal shape
51
52         Keywords Arguments:
53             center    - Vector holding the center of the polygon
54             sides     - the number of sides
55             dia       - the diameter of the base circle
56                       (aphotem or externa diameter)
57             align     - 0 or 1 it try to align the base with one
58                       ↪ axis
59             outer     - 0: aphotem 1: outer diameter (default 1)
60         """
61
62         ang_dist = pi / sides
63
64         if align == 0:
65             theta = 0.0
66         else:
67             theta = ang_dist
68
69         if outer == 1:
70             rad = dia * 0.5
71         else:
72             # dia is the apothem, so calculate the radius
73             # outer radius given the inner diameter
74             rad = (dia / cos(ang_dist)) * 0.5
75
76         vertex = []
77
78         for n_s in range(0, sides+1):
79             vpx = rad * cos((2 * ang_dist * n_s) + theta) +
80                 ↪ center[0]
81             vpy = rad * sin((2 * ang_dist * n_s) + theta) +

```



```

                                ↪ center[1]
81         vertex.append(Vector(vpx, vpy, 0))
82
83     obj = Part.makePolygon(vertex)
84     wire = Part.Wire(obj)
85     poly_f = Part.Face(wire)
86
87     return poly_f
```

## 6.2. Rivoluzione

In questa sezione non parliamo del fenomeno sociale, in quanto stiamo trattando di tecniche di creazione di oggetti 3D.

Uno strumento molto potente è reso disponibile dalla funzione `revolve` che fornisce la possibilità di usare una parte di piano per definire un oggetto mediante la rotazione, da un triangolo possiamo ottenere un cono, da un parallelepipedo un cilindro, da un cerchio un toro, ecc.

Ovviamente avendo già a disposizione queste figure nell'arsenale di **FreeCAD**, useremo figure diverse, ad esempio l'esagono creato prima.

Salviamo il risultato ottenuto con le linee dell'esempio precedente, e sostituiamo le righe da 99 in poi con quelle visualizzate nel 6.1 nella pagina seguente denominato **Rivoluzione**.

Infatti nella funzione `manico`, troviamo alla riga 116, il nostro esagono.

```
face = reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0)
```

Niente di particolarmente diverso da quello che abbiamo visto prima, nemmeno nella parte finale del codice:

```
obj = DOC.addObject("Part::Feature", nome)
obj.Shape = face.revolve(pos, vec, angle)
```

Creiamo semplicemente un oggetto `Part::Feature` e assegniamo alla proprietà `Shape` il risultato della funzione `revolve`, cioè la geometria ottenuta facendo ruotare la porzione di piano scelta.

Da spiegare sono i parametri della funzione `revolve`, analizziamoli in dettaglio:

```
# base point of the rotation axis
pos = Vector(0,10,0)
```

Indica il "punto base" della rotazione, nel nostro caso la faccia ha centro in `(0, 0, 0)` e scegliamo un punto base di rotazione di `(0, 10, 0)` cioè a 10mm dal centro del poligono.

```
# direction of the rotation axis
vec = Vector(1,0,1)
```

Questo vettore fornisce l'asse di riferimento in questo caso l'asse X.

```
angle = 360 # Rotation angle
```

Questo è ovviamente l'angolo di rotazione  $180^\circ$

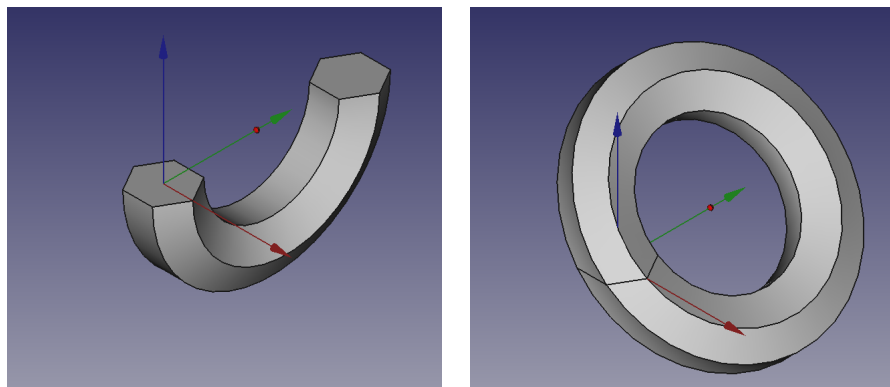
Lanciamo l'esecuzione attraverso la linea 134:

```
manico("Manico")
```

Il risultato sarà una cosa come quella in figura 6.2a a pagina 56.

## Programma 6.1: Rivoluzione

```
99 def point(nome, pos, pt_r = 0.25, color = (0.85, 0.0, 0.00),
100           tr = 0):
101     """draw a point for reference"""
102     rot_p = DOC.addObject("Part::Sphere", nome)
103     rot_p.Radius = pt_r
104     rot_p.Placement = FreeCAD.Placement(
105         pos, FreeCAD.Rotation(0,0,0), Vector(0,0,0))
106     rot_p.ViewObject.ShapeColor = color
107     rot_p.ViewObject.Transparency = tr
108
109     DOC.recompute()
110
111     return rot_p
112
113
114 def manico(nome):
115     """Revolve a face"""
116     face = reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0)
117     # base point of the rotation axis
118     pos = Vector(0,10,0)
119     # direction of the rotation axis
120     vec = Vector(1,0,1)
121     angle = 360 # Rotation angle
122
123     point("punto_rot", pos)
124
125     obj = DOC.addObject("Part::Feature", nome)
126     obj.Shape = face.revolve(pos, vec, angle)
127
128     DOC.recompute()
129
130     return obj
131
132 # definizione oggetti
133
134 manico("Manico")
135
136 setview()
```



(a) Rivoluzione.

(b) Effetto speciale.

Questo è solo un esempio, l'unico limite è la vostra fantasia.

I valori dell'asse di rotazione possono anche essere diversi da 0 o 1, e non necessariamente messi solo per un asse, sperimentando con valori diversi, si possono ottenere interessanti effetti, capire però a priori cosa si ottiene potrebbe essere complicato, provate ad inserire come:

- `vec = Vector(1,0,1)`
- `angle = 360`

e otterrete un effetto interessante, come potete vedere in figura 6.2b.

Il programma completo è disponibile nella pagina di GitHub dedicata a questa guida sotto il nome di `rev-full.py`.

### 6.3. Loft

Lo strumento **Part::Loft** di **FreeCAD**, viene utilizzato per creare cose abbastanza complesse e forme fluide, anche se possiede alcune limitazioni, che trovate ben elencate nella documentazione di **FreeCAD**, lo presentiamo comunque perché è un metodo molto interessante, è necessario studiare bene la documentazione per ottenere gli effetti desiderati.

Utilizzeremo sempre il nostro poligono ottenuto con il metodo `reg_poly`, visto che lo abbiamo a disposizione.

Riutilizziamo buon parte del codice ottenuto fin'ora, aggiungiamo a prima della riga **# definizione oggetti**, queste righe di codice:

```
133 def loft(nome):
134     """Loft a face"""
135     faces = []
136     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
137     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
138     faces.append(reg_poly(Vector(0, 0, 0), 6, 8, 0, 0))
```

```

139     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
140     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
141
142     sections = []
143
144     for idx, face in enumerate(faces):
145         sect = DOC.addObject(
146             "Part::Feature",
147             nome + "_sezione_" + str(idx))
148         sect.Shape = face
149         sect.Placement = FreeCAD.Placement(
150             Vector(0,0, 25 * idx),
151             FreeCAD.Rotation(0,0,0), Vector(0,0,0))
152         sections.append(sect)
153
154     obj = DOC.addObject("Part::Loft", nome)
155     obj.Sections=sections
156     obj.Solid = False
157     obj.Ruled = True
158     obj.Closed = False
159     obj.MaxDegree = 10
160
161     DOC.recompute()
162
163     print(obj.Placement)
164     return obj
165
166
167 # definizione oggetti
168
169 loft("Loft")
170
171 setview()

```

Analizziamo l'anatomia dello strumento, per prima cosa dobbiamo avere alcune forme, possiamo pensarle come delle sezioni della geometria finale, nel nostro caso usiamo un poligono regolare, ma potete usare anche altre forme (notate che non ho fatto riferimento a geometrie), queste forme per poter essere passate allo strumento devono essere ordinate in una lista.

Per come è costruito il metodo che utilizzeremo, creiamo una pre-lista di forme, secondo l'ordine di costruzione, ma senza necessariamente specificare una posizione, la cosa più comoda è crearle centrate all'origine (0, 0, 0).

```

135     faces = []
136     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
137     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))

```

```

138     faces.append(reg_poly(Vector(0, 0, 0), 6, 8, 0, 0))
139     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
140     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))

```

Lo strumento **Part::Loft** ha necessità di avere degli oggetti di tipo `DOC.Object` ed il modo più semplice è quello di creare dei **Part::Feature**, che come abbiamo già accennato sono il modo più “generico” per aver un oggetto, usiamo la nostra pre-lista, per creare la lista di forme in questo modo:

```

142     sections = []
143
144     for idx, face in enumerate(faces):
145         sect = DOC.addObject(
146             "Part::Feature",
147             nome + "_sezione_" + str(idx))
148         sect.Shape = face
149         sect.Placement = FreeCAD.Placement(
150             Vector(0,0, 25 * idx),
151             FreeCAD.Rotation(0,0,0), Vector(0,0,0))
152         sections.append(sect)

```

Notate nella riga 144 il “solito” metodo per scorrere la lista ottenendo anche un numero progressivo di posizione che utilizziamo in due modi:

- alla riga 147 per modificare il nome dell’oggetto.
- alla riga 150 per incrementare, usando l’indice come moltiplicatore del valore in Z della parte **Posizione** della proprietà **Placement** che definiamo per l’oggetto, ricordiamo che l’indice parte da 0.

La riga 148 si occupa di passare la forma selezionata alla proprietà **Shape** dell’oggetto **Part::Feature**.

Mentre la linea 152 si occupa di assegnare l’oggetto creato, alla lista di oggetti da passare allo strumento **Part::Loft**.

Questo è uno dei modi possibili, creato con l’intento di semplificare il codice, si possono usare altri metodi, ad esempio misurando la sezione di un oggetto a distanze note e poi ricavandone una sezione come abbiamo fatto per il prisma della Figura 6.1b a pagina 50, replicando in un certo senso le operazioni che compie uno slicer per la stampa 3D.

Fatto questo invochiamo lo strumento **Part::Loft**:

```

154     obj = DOC.addObject("Part::Loft", nome)
155     obj.Sections=sections
156     obj.Solid = False
157     obj.Ruled = True
158     obj.Closed = False
159     obj.MaxDegree = 10

```

Passando alla proprietà **Sections** la lista **sections** creata in precedenza, e impostando le altre proprietà; Il funzionamento di queste altre proprietà va studiato e compreso

nella documentazione ufficiale di **FreeCAD**, che contiene indicazioni sulle limitazioni e le complicazioni d'uso che si possono incontrare, i valori proposti permettono di ottenere la figura 6.3

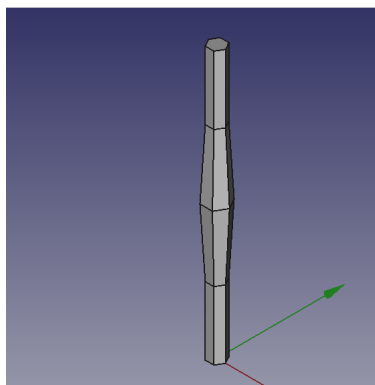


Figura 6.3.: Lo strumento Loft

Il programma completo è disponibile nella pagina di GitHub dedicata a questa guida sotto il nome di **loft-full.py**.

# Capitolo 7

## La componente Vista

Ai più acuti, occhi di falco non sarà sfuggito nella figura 6.2a a pagina 56, la presenza di un puntino rosso e a chi ha letto il listato non sarà sfuggita la presenza di una riga nel codice del metodo `manico` nel 6.1 a pagina 55 denominato **Rivoluzione** che non abbiamo citato:

```
123     point("punto_rot", pos)
```

Si occupa di creare il punto rosso della figura, è qualcosa di relativamente semplice, molto simile a quanto già visto durante la creazione di altri oggetti 3D.

La funzione è definita come:

```
99 def point(nome, pos, pt_r = 0.25, color = (0.85, 0.0, 0.00),
100           tr = 0):
101     """draw a point for reference"""
102     rot_p = DOC.addObject("Part::Sphere", nome)
103     rot_p.Radius = pt_r
104     rot_p.Placement = FreeCAD.Placement(
105         pos, FreeCAD.Rotation(0,0,0), Vector(0,0,0))
106     rot_p.ViewObject.ShapeColor = color
107     rot_p.ViewObject.Transparency = tr
108
109     DOC.recompute()
110
111     return rot_p
```

Crea un oggetto **Part::Sphere** e lo posiziona in una posizione definita, passata al metodo come vettore con il nome di `pos`, la sua incarnazione più semplice è quella vista sopra, ma possiede una serie di parametri opzionali, (a cui vengono assegnati valori di default) che lo rendono abbastanza flessibile.

Analizziamoli:

- `pt_r`: che è il raggio della sfera, passato alla proprietà **Radius**, possiede un valore di default di 0.25 (mm).
- `color`: è il colore della sfera



- **tr**: è la trasparenza della sfera

I due parametri **color** e **tr** appartengono alla componente **ViewObject**, a cui abbiamo accennato a pagina 24.

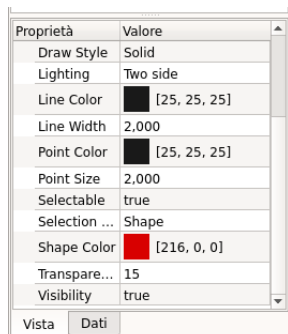


Figura 7.1.: La linguetta Vista

Il colore e la trasparenza permettono di visualizzare meglio le cose, in genere sono considerate dei meri orpelli alla modellazione, comunque hanno una discreta importanza perché aiutano durante la modellazione, poter mettere dei riferimenti, come assi e punti in una modellazione, aiuta a mettere a punto soprattutto nei momenti critici.

Descriviamo sommariamente la parte **ViewObject** che riflette le proprietà che trovate nella **Ling. Vista** di ogni oggetto, come vedete nell'immagine 7.1.

Abbiamo già accennato che ogni oggetti comprende un parte relativa alla **Geometria** e una parte

**Vista**, ricordiamo che la parte **Geometria** possiamo modificarla nella **Ling. Dati** come abbiamo già visto nell'immagine 4.2 a pagina 33.

Qui abbiamo almeno tre proprietà che potrebbero tornare utili:

- **ShapeColor** contiene il colore dell'oggetto,
- **Transparency** la trasparenza è semplicemente un valore che va da 0 a 100, occhio che è il valore della trasparenza quindi se mettete un valore alto la trasparenza è alta.
- **Visibility** semplicemente un valore booleano **True** o **False**, ovviamente **True** vuol dire **Visibile**.

All'interno di **FreeCAD** il colore può essere specificato come tupla di tre o quattro valori float che vanno da 0 a 1, (R, G, B, A).

- **R**: la componente rossa
- **G**: la componente verde
- **B**: la componente blu
- **A**: il canale Alpha cioè la trasparenza, è un valore float, 0 = completamente opaco e 1 = totalmente trasparente.

La proprietà **ShapeColor**, non sembra tenere conto della componente **A**, non viene generato nessun errore se viene passata una tupla di quattro valori ma non succede nulla, teniamo presente però la definizione del colore perché potrebbe tornare utile in altri casi.

Nel dubbio, provare a passare una tupla di quattro valori, magari con l'ultimo numero diciamo di 0.25 e vedere se viene generata una trasparenza, aggiustate poi il canale **A** come più vi aggrada, se invece viene generato un errore passate una tupla del tipo **RGB**.

Una nota di metodo, usualmente i valori per le componenti RGB sono si trovano in molte siti e manuali espressi come tripla di valori interi compresi tra 0 ... 255, per ottenere i valori da passare basta semplicemente dividere il valore RGB per 255.

Nella linguetta **Vista** i valori sono mostrati nel formato con gli interi e il dialogo che si apre se li modificate da interfaccia grafica, potete inserire sia i valori RGB sia il codice HTML (che poi sono gli stessi valori ma espressi come tre valori esadecimali uniti).

Bozza

# Capitolo 8

## Programmazione modulare

Durante la costruzione di un modello, è necessario creare molti oggetti e combinarli in vari modi, ad un certo punto diventa complicato gestire le cose, alcune tecniche ci aiutano ad evitare i maggiori inconvenienti.

- Modularizzazione.
- Modellazione parametrica.
- Librerie di oggetti.

Potrebbe sembrare una inutile complicazione introdurre questi concetti in una guida introduttiva, ma se il progetto diventa complicato, dopo aver cominciato a modellare, potrebbe diventare necessario modificare gran parte del codice per organizzarlo in modo organico e se non fatto subito all'inizio, diventa un'operazione che porta via molto tempo.

Imparare fin da subito queste tecniche permette di risparmiare molto tempo, e molti grattacapi, in futuro

Tenendo a mente che il presupposto di questa guida è la modellazione finalizzata alla stampa in 3D, dobbiamo considerare i problemi che pone la destinazione finale, ad esempio il restringimento di alcuni materiali e le differenze dimensionali di alcune forme, un esempio su tutti in genere i “fori” di un modello 3D sono sempre più piccoli del dovuto, questa è una strategia precisa di molti slicer e ne va tenuto conto, modificando alcuni parametri, se abbiamo “progettato per costruire” la cosa diventerà relativamente semplice.

## 8.1. Modularizzazione

Proviamo a realizzare una cosa particolare, una **scatola di sardine**, si tratta di un progetto relativamente complesso, che utilizza una varietà di tecniche di costruzione, ed alcune tecniche di organizzazione del codice.

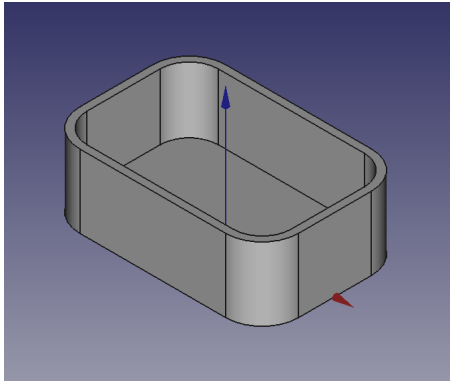


Figura 8.1.: Scatola sardine

Se vogliamo affrontare un modello complesso, dobbiamo usare qualche tecnica per organizzare il codice, in modo da poter procedere per gradi.

Non è un concetto nuovo, abbiamo già realizzato gran parte del codice mettendo le componenti in metodi e poi richiamandoli al bisogno.

Possiamo pensare ad un modello complesso come ad un insieme di scatole cinesi, noi chiamiamo l'ultima ma abbiamo anche quelle contenute al suo interno.

Una delle tecniche più utili per ridurre la complessità di un modello è quella di spezzarlo in blocchi logici che poi possiamo ricomporre in seguito per realizzare il modello finale.

Una parte di questi blocchi logici possono essere trasferiti in un **Modulo esterno** da importare con una istruzione **import** di Python, allo stesso modo con cui facciamo con i moduli di **FreeCAD**.

Questo approccio offre molti vantaggi:

- rendere la lunghezza del codice minore, molti moduli in pratica servono per iniziare a costruire il modello oppure per inizializzare alcune variabili, o per definire delle forme base, queste tipo di moduli sono i candidati migliori per essere spostati in un modulo esterno
- migliorare la leggibilità del codice, una piccola regola è quella delle 300 linee di codice, se si ha un programma con più di 300 linee di codice, qualcosa può essere spostato in un modulo esterno.
- permette di riutilizzare e di non riscrivere molte parti del codice quando si realizzano progetti con caratteristiche simili.

L'approccio presenta alcuni svantaggi, il principale è quello che per capire cosa fa una parte del codice è necessario andare a leggerlo nel modulo esterno.

Trovate il listato del modulo che sposta all'esterno del nostro programma alcune routine complesse sotto il nome **Listato - Modulo GfcMod** nell'Appendice 8.1.1 a pagina 66.

Leggendo il codice non noterete niente di eccezionale, l'unica cosa da notare è che nei metodi diventa necessario passare l'istanza di **DOC** come parametro, ad esempio:

```
22 def cubo(doc, nome, lung, larg, alt, cent = True, off_z = 0.
    ↪ 0):
```

in questo modo viene correttamente usata il contenuto della variabile DOC (definita nel programma principale), come documento attivo dove inserire le geometrie create, ma se usata da altri script il contenuto di questa variabile non viene condivisa.

Se volete creare un modulo dovete tenere presente alcune cose:

- Il percorso di ricerca dei file deve comprendere la directory in cui c'è il programma chiamante. In alcune implementazioni di **FreeCAD** questo non è automatico, per cui, in caso di problemi è meglio verificare il percorso standard, oppure inserirlo esplicitamente come vedremo fra poco.
- il modulo deve avere un nome particolare, il nome non deve contenere caratteri speciali, nemmeno il trattino e il trattino basso, in genere si usa il CamelCase, come nel modulo di esempio.

Trovate il listato 8.1.2 a pagina 70 denominato **Listato - sardine.py**.

Se andrete a leggerlo (è meglio farlo ora) potete notare la compattezza del listato, avendo “portato fuori” le routine complesse il listato diventa molto semplice.

Nel programma chiamante, le righe seguenti si occupano di impostare correttamente il percorso di ricerca:

```
21 scripts_path = os.path.join(os.path.dirname(__file__),)
22 sys.path.append(scripts_path)
```

La riga 21 si occupa di definire una variabile **scripts\_path** che viene “compilata” dall’istruzione che segue che invoca il metodo **os.path.join()** che unisce due percorsi, è un trucco perché il secondo percorso non esiste, infatti se notate l’istruzione finisce con una virgola prima della parentesi chiusa.

Perché? Dipende da una particolarità di Python, se devo passare una **tupla** e questa tupla è composta da un solo elemento, il sistema di darebbe errore, però se uso quella scrittura posso inserire una tupla composta da un solo elemento.

Questo trucco, mi serve perché usando un solo elemento e dandolo in pasto alla funzione **os.path.join()** essa compie il suo lavoro di “unire” (join vuol dire questo in Inglese) due parti, una delle quali è **None**, quindi faccio una cosa apparentemente inutile, però usando questo trucco, ottengo l’effetto collaterale di ottenere un percorso corretto in modo quasi indipendente dal sistema operativo, quasi perché ogni sistema operativo possiede comunque le sue “paturnie”, questo sistema sembra essere da alcuni post trovati nel forum di **FreeCAD** quello con meno punti problematici in quanto usando il modulo **os** dovrebbe (il condizionale è d’obbligo) recuperare il percorso usando le convenzioni in uso in quel sistema operativo.<sup>1</sup>

Il nome del file in esecuzione **\_\_file\_\_** è una convenzione di Python per definire alcuni variabili interne, come già ricordato nella sezione 2.1.2 a pagina 11, questo dovrebbe permettere all’interprete Python interno di recuperare il percorso del nome del file che stiamo eseguendo.

<sup>1</sup>Purtroppo non è sicuro al 100%, alcune modifiche alle chiamate di sistema di alcuni sistemi operativi che usando standard propri, non sempre sono coerenti tra una versione e l’altra del sistema operativo, specie se si ha a che fare con caratteri speciali e link simbolici oppure directory condivise.

La riga 22 si occupa di aggiungere il percorso della directory dove si trova il file in esecuzione alla lista dei percorsi di ricerca dei file, infatti aggiunge alla lista `sys.path` attraverso la funzione `append()` che abbiamo già visto in altre parti del codice che aggiunge un elemento ad una lista.

Queste righe di codice importano in modo “corretto” il modulo

```
14 import importlib
```

Questa riga importa un modulo che serve ad importare correttamente il modulo esterno, serve perché siamo all'interno di **FreeCAD**.

```
24 import GfcMod as GFC
25
26 importlib.reload(GFC)
```

Queste righe compiono la vera e propria importazione, la riga 24 importa il modulo **GfcMod**, assegnandoli il nome “interno” **GFC**, la riga 26 invoca il modulo che abbiamo caricato all'inizio, e sistema le chiamate in modo corretto.

Importando il modulo **GfcMod** con un nome diverso come **GFC**, quando dobbiamo riferirci ad un metodo contenuto nel file **GfcMod** ad esempio **cubo** basterà invocarlo con **GFC.cubo(...)**, questa tecnica è abbastanza usuale per ridurre la lunghezza delle righe con nomi molto lunghi, ma attenzione a non creare ambiguità con altri moduli di Python, per inciso **GFC** non è molto fantasioso perché è l'abbreviazione di **Guida di Free CAD**.

### 8.1.1. Listato - Modulo GfcMod

```
1 """GfcMod.py
2     module
3
4     This code was written as an sample code
5     for "FreeCAD Scripting Guide"
6
7     Author: Carlo Dormeletti
8     Copyright: 2020
9     Licence: CC BY-NC-ND 4.0 IT
10 """
11
12 import FreeCAD
13 from FreeCAD import Base, Vector
14 import Part
15 from math import pi, sin, cos
16
17
18 # EPS= tolerance to uset to cut the parts
19 EPS = 0.10
```

```
20 EPS_C = EPS * -0.5
21
22 def cubo(doc, nome, lung, larg, alt, cent = True, off_z = 0.
    ↪ 0):
23     obj_b = doc.addObject("Part::Box", nome)
24     obj_b.Length = lung
25     obj_b.Width = larg
26     obj_b.Height = alt
27
28     doc.recompute()
29
30     if cent == True:
31         posiz = Vector(lung * -0.5, larg * -0.5, off_z)
32     else:
33         posiz = Vector(0, 0, off_z)
34
35     obj_b.Placement = FreeCAD.Placement(
36         posiz,
37         FreeCAD.Rotation(0, 0, 0),
38         FreeCAD.Vector(0,0,0)
39     )
40
41     return obj_b
42
43 def base_cyl(doc, nome, ang, rad, alt, off_z = 0.0):
44     obj = doc.addObject("Part::Cylinder", nome)
45     obj.Angle = ang
46     obj.Radius = rad
47     obj.Height = alt
48
49     doc.recompute()
50
51     posiz = Vector(0, 0, off_z)
52
53     obj.Placement = FreeCAD.Placement(
54         posiz,
55         FreeCAD.Rotation(0, 0, 0),
56         FreeCAD.Vector(0,0,0)
57     )
58
59     return obj
60
61 def reg_poly(center=Vector(0, 0, 0), sides=6, dia=6,
62             align=0, outer=1):
63     """
64     This return a polygonal shape
```

```

65
66     Keywords Arguments:
67         center    - Vector holding the center of the polygon
68         sides     - the number of sides
69         dia       - the diameter of the base circle
70                     (apothem or externa diameter)
71         align     - 0 or 1 it try to align the base with one
72                     ↪ axis
73         outer     - 0: apothem 1: outer diameter (default 1)
74     """
75     ang_dist = pi / sides
76
77     if align == 0:
78         theta = 0.0
79     else:
80         theta = ang_dist
81
82     if outer == 1:
83         rad = dia * 0.5
84     else:
85         # dia is the apothem, so calculate the radius
86         # outer radius given the inner diameter
87         rad = (dia / cos(ang_dist)) * 0.5
88
89     vertex = []
90
91     for n_s in range(0, sides+1):
92         vpx = rad * cos((2 * ang_dist * n_s) + theta) +
93             ↪ center[0]
94         vpy = rad * sin((2 * ang_dist * n_s) + theta) +
95             ↪ center[1]
96         vertex.append(Vector(vpx, vpy, 0))
97
98     obj = Part.makePolygon(vertex)
99     wire = Part.Wire(obj)
100    poly_f = Part.Face(wire)
101
102    return poly_f
103
104 def cubo_stondato(doc, nome, lung, larg, alt, raggio, off_z)
105     ↪ :
106
107     c_c1 = Vector((lung * -0.5) + raggio, (larg * -0.5) +
108         ↪ raggio, 0)

```



```

106     c_c2 = Vector((lung * -0.5) + raggio, (larg * 0.5) -
107                  ↪ raggio, 0)
107     c_c3 = Vector((lung * 0.5) - raggio, (larg * 0.5) -
108                  ↪ raggio, 0)
108     c_c4 = Vector((lung * 0.5) - raggio, (larg * -0.5) +
109                  ↪ raggio, 0)
109
110     obj_dim = c_c3 - c_c1
111     fi_lung = obj_dim[0] + raggio * 2
112     fi_larg = obj_dim[1] + raggio * 2
113
114     obj_c1 = base_cyl(doc, nome + '_cil1', 360, raggio, alt)
115     obj_c1.Placement = FreeCAD.Placement(
116         c_c1,
117         FreeCAD.Rotation(0, 0, 0),
118         FreeCAD.Vector(0,0,0))
119
120     obj_c2 = base_cyl(doc, nome + '_cil2', 360, raggio, alt)
121     obj_c2.Placement = FreeCAD.Placement(
122         c_c2,
123         FreeCAD.Rotation(0, 0, 0),
124         FreeCAD.Vector(0,0,0))
125
126     obj_c3 = base_cyl(doc, nome + '_cil3', 360, raggio, alt)
127     obj_c3.Placement = FreeCAD.Placement(
128         c_c3,
129         FreeCAD.Rotation(0, 0, 0),
130         FreeCAD.Vector(0,0,0))
131
132     obj_c4 = base_cyl(doc, nome + '_cil4', 360, raggio, alt)
133     obj_c4.Placement = FreeCAD.Placement(
134         c_c4,
135         FreeCAD.Rotation(0, 0, 0),
136         FreeCAD.Vector(0,0,0))
137
138
139     obj1 = cubo(doc, nome + "_int_lu", fi_lung , obj_dim[1],
140                ↪ alt, False)
140     obj1.Placement = FreeCAD.Placement(Vector(fi_lung * -0.5
141                ↪ , obj_dim[1] * -0.5, 0), FreeCAD.
142                ↪ Rotation(0, 0, 0), FreeCAD.Vector(0,0,0
143                ↪ ))
141
142     obj2 = cubo(doc, nome + "_int_la", obj_dim[0] , fi_larg,
143                ↪ alt, False)
143     obj2.Placement = FreeCAD.Placement(Vector(obj_dim[0] * -

```

```

    ↪ 0.5, fi_larg * -0.5, 0), FreeCAD.
    ↪ Rotation(0, 0, 0), FreeCAD.Vector(0,0,0
    ↪ ))
144
145
146     obj_int = doc.addObject("Part::MultiFuse", nome)
147     obj_int.Shapes = [obj1, obj2, obj_c1, obj_c2, obj_c3,
    ↪ obj_c4]
148     obj_int.Refine = True
149     doc.recompute()
150
151     obj_int.Placement = FreeCAD.Placement(
152         Vector(0, 0, off_z),
153         FreeCAD.Rotation(0, 0, 0),
154         FreeCAD.Vector(0,0,0)
155     )
156
157     return obj_int

```

### 8.1.2. Listato - sardine.py

```

1  #
2  """sardine.py
3
4      This code was written as an sample code
5      for "FreeCAD Scripting Guide"
6
7      Author: Carlo Dormeletti
8      Copyright: 2020
9      Licence: CC BY-NC-ND 4.0 IT
10 """
11
12 import os
13 import sys
14 import importlib
15
16 import FreeCAD
17 from FreeCAD import Base, Vector
18 import Part
19 from math import pi, sin, cos
20
21 scripts_path = os.path.join(os.path.dirname(__file__),)
22 sys.path.append(scripts_path)
23
24 import GfcMod as GFC
25

```

```
26 importlib.reload(GFC)
27
28 DOC = FreeCAD.activeDocument()
29 DOC_NAME = "Pippo"
30
31 def clear_doc():
32     """
33     Clear the active document deleting all the objects
34     """
35     for obj in DOC.Objects:
36         DOC.removeObject(obj.Name)
37
38 def setview():
39     """Rearrange View"""
40
41     FreeCAD.Gui.activeDocument().activeView().viewAxometric(
42         ↪ )
43     FreeCAD.Gui.activeDocument().activeView().setAxisCross(
44         ↪ True)
45     FreeCAD.Gui.SendMsgToActiveView("ViewFit")
46
47 if DOC is None:
48     FreeCAD.newDocument(DOC_NAME)
49     FreeCAD.setActiveDocument(DOC_NAME)
50     DOC = FreeCAD.activeDocument()
51 else:
52     clear_doc()
53
54
55 # EPS= tolerance to uset to cut the parts
56 EPS = 0.10
57 EPS_C = EPS * -0.5
58
59
60
61 def sardine(nome, lung, prof, alt, raggio, spess):
62     """sardine
63     crea una geometria a forma di scatola di sardine
64
65     Keywords Arguments:
66     nome      = nome della geometria finale
67     lung      = lunghezza della scatola, lungo l'asse X
68     larg      = larghezza della scatole, lungo l'asse Y
69     alt       = altezza della scatola, lungo l'asse Z
```

```

70         raggio = raggio dello "stondamendo" della scatola
71         spess  = spessore della scatola
72     """
73
74     obj = GFC.cubo_stonato(
75         DOC, nome + "_est", lung, prof, alt, raggio, 0.0)
76
77     lung_int = lung - (spess * 2)
78     prof_int = prof - (spess * 2)
79     alt_int = alt + EPS - spess
80     raggio_int = raggio - spess
81
82     obj_int = GFC.cubo_stonato(
83         DOC, nome + "_int",
84         lung_int, prof_int, alt_int, raggio_int, spess)
85
86     obj_f = DOC.addObject("Part::Cut", nome)
87     obj_f.Base = obj
88     obj_f.Tool = obj_int
89     obj_f.Refine = True
90     DOC.recompute()
91
92     return obj_f
93
94 sardine("scatola", 30, 20, 10, 5, 1)
95
96 setview()

```

## 8.2. Modellazione parametrica

Nelle primissime pagine abbiamo affermato, seguendo la definizione presente sul sito, che **FreeCAD** è un modellatore parametrico, cioè permette di modificare le geometrie modificando i loro parametri, questo è vero per una serie di geometrie di base, per altre ad esempio una fusione, i parametri di origine non sono facilmente accessibili e ad esempio se costruiamo venti geometrie, e poi cambiamo idea siamo costretti ad andare recuperare ogni geometria di base nell'albero di creazione e modificarne i parametri.

Usando lo scripting, questo viene evitato, anzi il più delle volte, è possibile modificare un progetto complesso, modificando pochissime righe di codice, anche se impattano su centinaia di geometrie.

Nell'esempio 8.1.2 a pagina 70 denominato **Listato - sardine.py** la parte che crea tutto è costituita da solo una riga di codice:

```

94 sardine("scatola", 30, 20, 10, 5, 1)

```

se commentate con **#** la riga 94 non verranno create le geometrie, questa è la potenza dello scripting, e della modellazione parametrica, basta cambiare poco perché molto

venga influenzato dalla modifica effettuata.

Leggiamo la doctring del metodo `sardine(...`:

```

62     """sardine
63         crea una geometria a forma di scatola di sardine
64
65     Keywords Arguments:
66     nome      = nome della geometria finale
67     lung      = lunghezza della scatola, lungo l'asse X
68     larg      = larghezza della scatole, lungo l'asse Y
69     alt       = altezza della scatola, lungo l'asse Z
70     raggio    = raggio dello "stondamendo" della scatola
71     spess     = spessore della scatola
72     """

```

vediamo che i parametri passati sono abbastanza esplicativi anche nel nome, ma la doctstring evita ogni fraintendimento, pensate al parametro `raggio`, sembra qualcosa di poco conto, ma se guardate nel listato **Listato - Modulo GfcMod** quante geometrie vengono modificate vi rendete conto della potenza della modellazione parametrica, ovviamente se avete pianificato bene la costruzione del modello.

Il riferimento alla variabile `raggio` lo trovate in tutti i “calcoli preliminari” alla costruzione del modello, in pratica dalla riga 105 alla riga 112.

Analizziamo la logica del programma, per creare una “scatola stondata” agli angoli, in modo sintetico dobbiamo definire due geometrie e sottrarle, una geometria è quella della scatola esterna (`obj`), e una è quella del “buco” interno (`obj_int`), queste geometrie differiscono solo per le misure, che assumono.

La prima geometria è definita dalle linee:

```

74     obj = GFC.cubo_stonato(
75         DOC, nome + "_est", lung, prof, alt, raggio, 0.0)

```

mentre la seconda dalle linee:

```

82     obj_int = GFC.cubo_stonato(
83         DOC, nome + "_int",
84         lung_int, prof_int, alt_int, raggio_int, spess)

```

Come vedete entrambe sono generate dal metodo `GFC.cubo_stonato(...`, usando i parametri passati alla chiamata a `sardine(...`, per il `obj`, mentre per `obj_int`, i parametri sono ricalcolati in base a `spess` come potete facilmente vedere nelle righe da 77 a 80 del listato.

```

77     lung_int = lung - (spess * 2)
78     prof_int = prof - (spess * 2)
79     alt_int = alt + EPS - spess
80     raggio_int = raggio - spess

```

Questo è un primo assaggio della modellazione parametrica, che poi non è nulla di particolare, se non definire in modo preciso un modello “generico” e poi permettere un certo grado di personalizzazione.

Il vantaggio della modellazione parametrica è che se ben pianificata permette di riutilizzare buona parte del codice in modo da rendere la “modellazione finale” una cosa veloce.

### 8.2.1. Creazione “automatica” di geometrie

Utilizzando un approccio classico alla modellazione anche usando lo scripting, si può cadere nell’errore di ricalcare semplicemente l’abituale modo di procedere a cui siamo abituati usando l’interfaccia grafica:

- disegno la geometria A
- posiziono la geometria A
- disegno la geometria B
- posiziono la geometria B
- seleziono le due geometrie
- fondo, taglio, opero con gli strumenti voluti

Niente di particolare, avete comunque a disposizione un vantaggio, quello di poter modificare i parametri in modo veloce, senza dover spendere ore a cliccare in giro per l’interfaccia grafica.

Però ad esempio modificare i centri di 100 fori in una piastra, può diventare noioso, lungo, complicato e soggetto ad errori anche con lo scripting.

Diversamente cercando di “automatizzare” il più possibile la creazione di una geometria basterà cambiare poco per estendere le modifiche a molti oggetti.

Riprendiamo per illustrare questo concetto le righe da 105 a 136 del listato **Listato - Modulo GfcMod** che si occupano della creazione dei quattro cilindri che definiscono gli angoli della scatola.

Osservando questa parte di codice si nota subito una certa ridondanza che la rende il candidato perfetto per una semplificazione, del resto ogni cilindro differisce solo per il posizionamento e per il nome.

Per creare in modo automatico i quattro cilindri avremo necessità di automatizzare prima di tutto la creazione dei nomi, e di passare alle proprietà di ogni cilindro gli opportuni valori, sembra complicato, ma abbiamo a disposizione la potenza di Python che ci viene in soccorso.

Per seguire la presentiamo nel listato 8.2.2 a pagina 76 denominato **Listato - cubo\_stondato modificato**, la modifica al metodo **cubo\_stondato** che va a sostituire il metodo con lo stesso nome nel listato 8.1.1 a pagina 66 denominato **Listato - Modulo GfcMod**.

Abbiamo accennato nell'introduzione a Python che per Python tutto è un oggetto, per cui possiamo tranquillamente creare una lista che contiene i **Vettori** con le posizioni dei centri dei cilindri, ricordiamo che la lista è contenuta tra le parentesi quadre, Il codice sarà:

```

105     c_pos = [
106         Vector((lung * -0.5) + raggio, (larg * -0.5) +
                ↪ raggio, 0),
107         Vector((lung * -0.5) + raggio, (larg * 0.5) - raggio
                ↪ , 0),
108         Vector((lung * 0.5) - raggio, (larg * 0.5) - raggio,
                ↪ 0),
109         Vector((lung * 0.5) - raggio, (larg * -0.5) + raggio
                ↪ , 0)
110     ]

```

Abbiamo semplicemente sostituito le variabili **c\_c1**, **c\_c2**, **c\_c3**, **c\_c4** con una lista chiamata **c\_pos**.

Avendo ora a disposizione una lista, la riga 110 del modulo originale diventa:

```

112     obj_dim = c_pos[2] - c_pos[0]

```

Sostituiamo semplicemente le vecchie variabili con il riferimento al corrispondente elemento della lista, ricordiamo che gli indici delle liste in Python partono da 0, per cui **c\_c1** è diventata **c\_pos[0]** e conseguentemente **c\_c3** è diventata **c\_pos[2]**.

```

116     comps = []
117
118     for i, pos in enumerate(c_pos):
119
120         obj = base_cyl(
121             doc, nome + '_cil_' + str(i),
122             360, raggio, alt)
123         obj.Placement = FreeCAD.Placement(
124             pos,
125             FreeCAD.Rotation(0, 0, 0),
126             FreeCAD.Vector(0,0,0))
127         comps.append(obj)

```

Nella riga 116 troviamo l'inizializzazione di una lista vuota chiamata **comps**, che conterrà tutti i componenti della nostra geometria finale.

Con la riga 118 creiamo un iterabile attraverso la funzione **enumerate(c\_pos)**, questo costruito che abbiamo già visto, ritorna una coppia di dati per ogni elemento della lista, il **numero di posizione i** e l'elemento della lista originale **pos** che ricordiamo contiene il nostro vettore di posizione del centro.

Questo ci serve per fornire tutti i dati alla creazione dell'oggetto che avviene alle linee da 120 a 126.

Notate l'automatismo della composizione del nome dell'oggetto alla linea 121, composto da una stringa formata da **nome** che è la variabile nome che viene passata al metodo **cubo\_stondato**, concatenata alla stringa **\_cil\_**, che crea la parte centrale del nome e **str(i)** che trasforma il numero corrispondente alla posizione del centro del cilindro in una stringa di caratteri adatta ad essere concatenata alla parte precedente. (+ è l'operatore di concatenazione per le stringhe di Python), ogni oggetto avrà un nome significativo diverso dal precedente, e questo varrà senza dover modificare una riga di codice se cambia il numero di componenti della lista di centri.

L'altra parte significativa è la riga 124 dove viene assegnato alla proprietà **Placement** dell'oggetto cilindro che stiamo creando la posizione del centro.

Le altre piccole modifiche sono alle linee 134 e 141 dove vengono aggiunte alla lista **comps** le due geometrie dei parallelepipedi, così:

```
comps.append(obj1)
```

la linea 144 :

```
obj_int.Shapes = comps
```

assegna la lista alla proprietà **Shapes** del metodo **Part::MultiFuse**

Di seguito trovate l'intero codice del metodo modificato.

Il modulo modificato è disponibile nella pagina GitHub della guida sotto il nome di **GfcModva.py**

### 8.2.2. Listato - cubo\_stondato modificato

```
103 def cubo_stondato(doc, nome, lung, larg, alt, raggio, off_z)
104     ↪ :
105     c_pos = [
106         Vector((lung * -0.5) + raggio, (larg * -0.5) +
107             ↪ raggio, 0),
108         Vector((lung * -0.5) + raggio, (larg * 0.5) - raggio
109             ↪ , 0),
110         Vector((lung * 0.5) - raggio, (larg * 0.5) - raggio,
111             ↪ 0),
112         Vector((lung * 0.5) - raggio, (larg * -0.5) + raggio
113             ↪ , 0)
114     ]
115
116     obj_dim = c_pos[2] - c_pos[0]
117     fi_lung = obj_dim[0] + raggio * 2
118     fi_larg = obj_dim[1] + raggio * 2
119
120     comps = []
```



```

117
118     for i, pos in enumerate(c_pos):
119
120         obj = base_cyl(
121             doc, nome + '_cil_' + str(i),
122             360, raggio, alt)
123         obj.Placement = FreeCAD.Placement(
124             pos,
125             FreeCAD.Rotation(0, 0, 0),
126             FreeCAD.Vector(0,0,0))
127         comps.append(obj)
128
129     obj1 = cubo(doc, nome + "_int_lu", fi_lung , obj_dim[1],
130                ↪ alt, False)
131     obj1.Placement = FreeCAD.Placement(
132         Vector(fi_lung * -0.5, obj_dim[1] * -0.5, 0),
133         FreeCAD.Rotation(0, 0, 0), FreeCAD.Vector(0,0,0))
134
135     comps.append(obj1)
136
137     obj2 = cubo(doc, nome + "_int_la", obj_dim[0] , fi_larg,
138                ↪ alt, False)
139     obj2.Placement = FreeCAD.Placement(
140         Vector(obj_dim[0] * -0.5, fi_larg * -0.5, 0),
141         FreeCAD.Rotation(0, 0, 0), FreeCAD.Vector(0,0,0))
142
143     comps.append(obj2)
144
145     obj_int = doc.addObject("Part::MultiFuse", nome)
146     obj_int.Shapes = comps
147     obj_int.Refine = True
148     doc.recompute()
149
150     obj_int.Placement = FreeCAD.Placement(
151         Vector(0, 0, off_z),
152         FreeCAD.Rotation(0, 0, 0),
153         FreeCAD.Vector(0,0,0)
154     )
155
156     return obj_int

```

### 8.3. Librerie di oggetti

Nell'esempio della scatola di sardine abbiamo introdotto il concetto che alcune parti di codice possono essere spostate in un modulo esterno da importare nello script usando

una istruzione di import.

Questo modo di lavorare permette di creare diverse librerie di oggetti da importare al bisogno, questo comporta la definizione di alcune concetti.

Queste righe di codice nel listato 8.1.2 a pagina 70 denominato **Listato - sardine.py**.

```
21 scripts_path = os.path.join(os.path.dirname(__file__),  
22 sys.path.append(scripts_path)
```

si occupano di definire il percorso di ricerca dell'interprete Python interno di **FreeCAD**.

Dove mettere queste librerie di oggetti è una scelta che dipende da alcune considerazioni, che ognuno deve fare, i principali posti che vengono in mente sono due:

- La directory dove si trova il file principale, e le righe sopra esposte si occuperanno di aggiungere la directory dove si trova il file al “percorso” di ricerca dell'interprete Python interno, e di trovare quindi le librerie.
- Una posizione unica nel computer, in questo caso è necessario specificare il percorso duplicando le righe esposte sopra.

Come organizzare una libreria, la cosa migliore è quella di organizzarla nella maniera mostrata nel Listato 8.3.1 nella pagina seguente

Illustriamolo in dettaglio:

- dalla riga 1 alla riga 8 trovate la **docstring** generale della libreria, facciamo le cose con ordine, magari mettiamo che tipo di oggetti la libreria fornisce, mettete anche il nome dell'autore, il copyright e la licenza se dovete farla girare, non si sa mai qualcuno potrebbe rubarvi le idee e spacciarle come sue.
- dalla riga 10 alla riga 13 ci sono le usuali import di uno script FreeCAD, presumo che la libreria serva per produrre geometrie più o meno complesse con FreeCAD.
- notate alla riga 15 il commento che riporta il luogo dove è meglio che ci siano le eventuali altre istruzioni di import.

Se ci sono variabili comuni mettetele lì, ricordate una cosa che le variabili comuni saranno accessibili in lettura e scrittura attraverso l'usuale modo, se ad esempio la libreria si chiama **Pippo** e la importate come Pippo, la variabile **Pluto** sarà modificabile con **Pippo.Pluto = 5.0** e accessibile con **lungh = Pippo.Pluto**.

- dalla riga 22 in poi mettete i metodi, vi ricordo che sarebbe meglio che ogni metodo ritornasse un oggetto, e che ogni metodo è meglio sia fornito di una sua propria **docstring**.

A tale scopo ho riportato una **docstring** di esempio alle righe da 25 a 37, in sostanza dovete illustrare cosa fa e dopo **Keywords Arguments:** elencare ogni parametro e se sono ammessi dei valori come ad esempio per **cent** elencare le varie alternative.

Non è infrequente che usiate un indicatore ad esempio un intero per fornire varie possibilità di costruzione e che questo intero (0, 1, 2, 3, 99) ovviamente non abbia un valore riconoscibile, se a distanza di tempo riprenderete in mano il codice, quasi

sicuramente vi sarete scordati cosa fa e dovrete rileggere e ricostruire mentalmente tutto il filo logico del codice per capire a cosa serve il parametro.

- una nota sulla spaziatura, in genere è meglio che come nel caso di metodi separati all'interno di un programma lasciare **due spazi** tra i vari metodi, permette di identificare visivamente la separazione tra i metodi, in genere non si lasciano mai due spazi all'interno di un metodo.
- usate nomi dei parametri significativi ed in genere con più di tre caratteri, ad esempio non mettere **x, y, z** per indicare una posizione ma magari **pos-x, pos-y, pos-z** (magari usate l'underscore o "trattino basso"), già dal nome potete intuire cosa fa la variabile, evitate però nomi lunghissimi ovviamente per motivi di spazio.
- non fate righe più lunghe di 78 caratteri, questo sembra una cretinata, ma se poi dovete leggere con un carattere decentemente leggibile diventa un rognia anche nell'editor interno di FreeCAD.
- per conoscere i modi corretti per andare a capo nel codice vi consiglio di cercare nella documentazione di Python il PEP8 e leggerlo.

Nulla ci vieta di creare tante librerie di oggetti da usare al bisogno, per esempio un modulo si potrà occupare della realizzazione di viti e dadi, un secondo modulo realizzerà dei profilati, ecc.

Basterà definire le opportune istruzioni di importazione e si potranno usare i vari componenti quando necessitano, occupandosi della loro creazione una volta per tutte, questo fornirà il vantaggio di poter correggere un eventuale errore solo nella libreria e automaticamente sarà corretto in tutti gli script che utilizzano quella libreria.

Quando si creano delle librerie serve però di usare alcune cautele:

- Se si fanno modifiche, meglio documentarle in modo corretto.
- Quando ad esempio si modifica un metodo, e si aggiungono dei parametri, dare ai nuovi parametri dei valori di default, in modo che se il metodo viene richiamato da un programma realizzato precedentemente alla modifica l'oggetto ottenuto sia lo stesso del metodo pre-modifica. I nuovi programmi potranno beneficiare delle nuove funzioni, ed eventualmente i vecchi programmi potranno essere aggiornati se servirà usare le nuove funzioni.
- In ragione del punto precedente, sarà meglio riportare nella docstring della libreria un numero di versione che indichi l'evoluzione, e per lo stesso motivo questo numero di versione è bene che sia riportato magari come commento nella docstring del programma chiamante, a scanso di equivoci.

### 8.3.1. Esempio di Libreria

```
1  """libreria.py
2      module
3
```

```
4
5     Author: ----
6     Copyright: ----
7     Licence: ---
8     """
9
10    import FreeCAD
11    from FreeCAD import Base, Vector
12    import Part
13    from math import pi, sin, cos
14
15    # qui mettete altre istruzioni import
16
17    # qui mettete delle variabili comuni
18    # EPS= tolerance to use to cut the parts
19    EPS = 0.10
20    EPS_C = EPS * -0.5
21
22    # qui di seguito andranno le funzioni
23
24    def cubo(doc, nome, lung, larg, alt, cent = True, off_z = 0.
        ↪ 0):
25        """cubo
26            Il metodo ritorna un cubo
27
28            Keywords argument:
29
30            doc      = riferimento al documento attivo
31            nome     = nome della geometria finale
32            ...
33            cent     = True (centra il cubo attorno all'origine)
34                    False (lascia il cubo al riferimento di
        ↪ costruzione)
35            off_z    = offset in Z per la base del cubo
36
37        """
38        return qualcosa
39
40
41    def altro....
```

# Capitolo 9

## Proprietà degli “oggetti 3D”

Uno dei segreti più segreti di **FreeCAD**, riportato anche nella descrizione è che **FreeCAD** è nato senza interfaccia grafica, ed era utilizzabile pienamente senza di essa, anche ora è

Da questo derivano alcune considerazioni:

- La divisione dell’interfaccia tra **Ling. Dati** e **Ling. Vista**, che abbiamo già visto nel capitolo 7 a pagina 60 denominato **La componente Vista**.
- Quando qualche componente non è accessibile probabilmente fa parte di un wor-bench per cui non è stata resa disponibile l’accesso dall’esterno.
- Esistono dei metodi per derivare anche se non lo si trova nella documentazione il nome della proprietà.

Nella discussione che segue si farà uso del listato **aereo** che troverete nel listato 9.1.1 a pagina 84 denominato **Listato - aereo**, presente anche in GitHub con il nome **aereo.py**

### 9.1. Conoscere il nome delle proprietà

Presentiamo un modello complesso un aereo, simile ma non identico ad un esempio presente in **FreeCAD**.

Questo modello è composto da tre primitive geometriche, il cubo, il cilindro e una sfera.

La sfera non la abbiamo ancora vista, per cui presentiamola, “buongiorno questa è una sfera” avrebbe detto Zuzzurro, comunque eccola:

```
82 def sfera(nome, rad):
83     obj = DOC.addObject("Part::Sphere", nome)
84     obj.Radius = rad
85
86     DOC.recompute()
87
88     return obj
```

Quanto ci si può aspettare da una sfera, l’oggetto **FreeCAD** è **Part::Sphere**.

Questo ci servirà per illustrare come derivare le proprietà usando l'interfaccia grafica, e le informazioni visualizzate nella **console Python**, a patto di aver attivato una funzione di **FreeCAD**, in questo modo:

menu **Modifica** ⇒ **Preferenze** alla sezione **Generale** nella linguetta **Macro** nel riquadro **Comandi di Log** spuntare la voce “**Mostra lo script dei comandi nella console Python**”.

Questa impostazione permette di mostrare nella **console Python** l'eco dei comandi impartiti nell'interfaccia grafica.

Vediamo un esempio pratico:

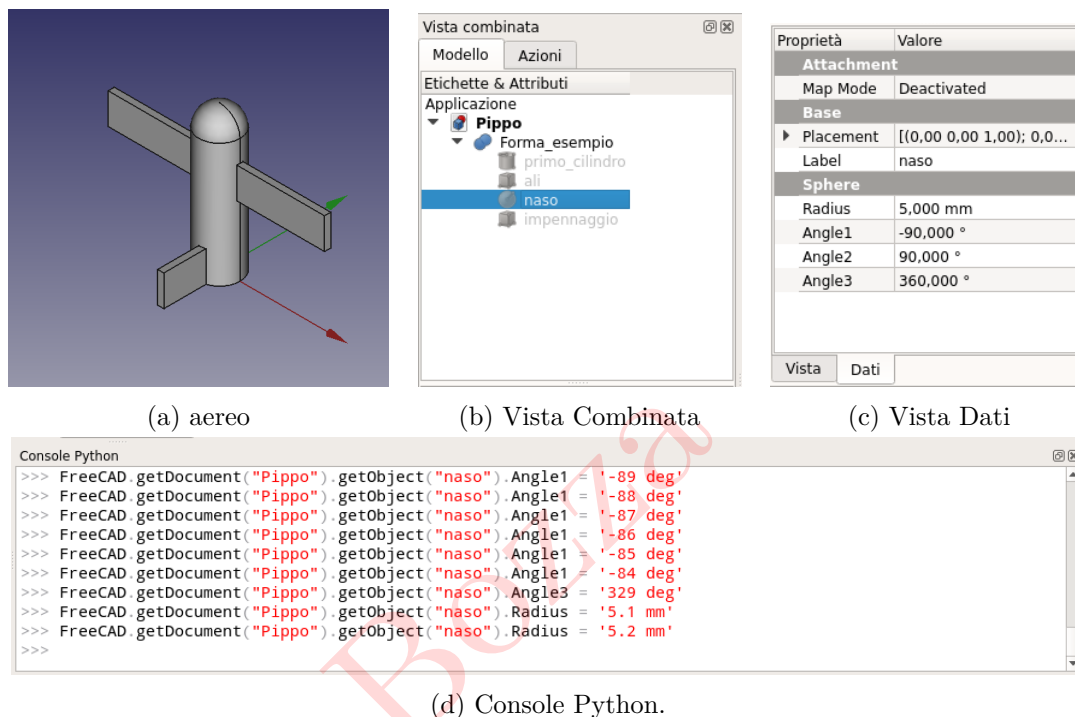


Figura 9.1.: Derivazione delle proprietà.

Nella figura 9.1a, potete vedere il modello finale generato dal codice proposto, nella figura 9.1b potete vedere quanto mostrato nella **vista ad albero**, dopo aver espanso l'albero del modello ed evidenziato la componente **naso**, che è la sfera che abbiamo presentato prima.

Nella figura 9.1c potete vedere la parte dell'**editore delle proprietà** relativa alla geometria **naso**, notate che nel codice di creazione, abbiamo specificato sola la proprietà **Radius**, La definizione della sfera come possiamo vedere dalla **Ling. Dati** comprende altre tre proprietà: **Angle1**, **Angle2** e **Angle3**, che non non abbiamo esplicitato e che “ovviamente” sono state create con i valori di default.

Queste proprietà permettono di modificare la sfera per farle assumere delle diverse configurazioni, controllando la generazione di porzioni diverse della sfera.

Al momento non interessa molto cosa fanno, ma solo il fatto che esistono, hanno preso in automatico dei valori di default, e che controllano la generazione della sfera.

Non avendole definite, potrebbe rimanere il dubbio di come fare a invocarle.

Ovviamente se seguiamo l'esempio del codice, possiamo "intuire" che si chiamino anche ai ai fini dello script:

```
obj.Angle1 = .....
obj.Angle2 = .....
obj.Angle3 = .....
```

L'intuizione in questo caso indovina, ma se ad esempio i nomi fossero stati tradotti come nel caso delle componenti di **Placement**, ci rimarrebbe il dubbio di come chiamarle, ai fini della scrittura del metodo per creare una sfera.

Ricordiamo per inciso che l'**editore delle proprietà** possiede le due linguette **Ling. Vista** e **Ling. Dati** e che l'immagine mostra la **Ling. Dati**.

Ci viene in aiuto quell'impostazione delle **Preferenze** che abbiamo consigliato di attivare.

Questo permette di mostrare l'output del comando impartito nell'interfaccia grafica nella finestra della **console Python**, come mostrato nella figura 9.1d.

```
FreeCAD.getDocument("Pippo").getObject("naso").Angle1 = '-84
    ↪ deg'
FreeCAD.getDocument("Pippo").getObject("naso").Angle3 = '329
    ↪ deg'
FreeCAD.getDocument("Pippo").getObject("naso").Radius = '5.1
    ↪ mm'
```

Per comprendere questa scrittura dobbiamo fare qualche esercizio di traduzione:

**FreeCAD** è chiaramente il nome del modulo di **FreeCAD** che abbiamo importato con la direttiva `import` all'inizio di ogni script, in molti esempi in giro per la rete e sul sito è chiamato anche **App**.

`getDocument("Pippo")` ci ritorna l'istanza del documento che si chiama **"Pippo"** che tradotto nel nostro usuale modo di chiamarlo nel codice è **DOC**.

`getObject("naso")` è l'istanza dell'oggetto che si chiama **"naso"**, che "guarda caso" è l'oggetto che viene evidenziato nella **vista ad albero**, in pratica è questo:

```
115     obj3 = sfera("naso", diam_fus)
```

Questo oggetto però nella routine di creazione si chiama **obj**, infatti se guardate alla riga 88 del codice **obj** viene "restituito" alla funzione chiamante da `return obj`.

Quello che segue è il nome della proprietà, nel nostro caso **Radius**, **Angle1** e **Angle3**.

Le righe citate non sono nient'altro che l'eco dei comandi che ho impartito andando a modificare le singole proprietà mediante l'interfaccia grafica usando le frecce di incremento e decremento presenti accanto ad ogni valore.

Facciamo qualche considerazione finale, fino a qui abbiamo:

- Trovato un modo per derivare le proprietà di un oggetto usando l'interfaccia grafica, facendoci “dire” dalla **console Python** come si chiama.
- Imparato un modo per “collegare” quanto letto nella **console Python** al nostro codice.
- Imparato il modo di recuperare l'istanza di un documento, conoscendone il “nome”.
- Imparato il modo di recuperare l'istanza di un oggetto conoscendone il “nome”.

Una volta creato un oggetto, oppure se dovessimo dover manipolare un oggetto creato dall'interfaccia grafica, ad esempio **ali**, nel nostro codice, avremmo dovuto scrivere:


```
obj = FreeCAD.getDocument("Pippo").getObject("ali")
obj....
```

Ovviamente, presupponendo che il nome del documento sia **"Pippo"**, come nella **vista ad albero** mostrata.

Da quella vista potete notare due cose, l'arborescenza corrisponde più o meno esattamente all'ordine del codice mostrato nella **console Python** in Parole: **Applicazione.Pippo.objetto**, il quasi esattamente nella frase precedente è legato al fatto che il nome dell'oggetto è il nome della “primitiva”, cioè nella figura 9.1b le forme con l'icona



, cioè **primo\_cilindro**, **ali**, **naso**, **impennaggio**.

Notiamo per che **Forma esempio**, possiede un'icona  che indica l'operazione booleana di **Unione** infatti alla riga 123 del codice quell'oggetto è stato creato invocando il metodo alla riga 91 che crea (e restituisce) un oggetto **Part::MultiFuse**.

Come riprova provate a inserire tra **aeroplano()** e **setview()** alla riga 132

```
FreeCADGui.getDocument("Pippo").getObject("Forma_esempio").
    ↪ ShapeColor = (0.333,0.333,1.000)
```

E vedrete che la l'aeroplano visualizzato cambia colore e diventa blu.

(A patto che il documento si chiami "Pippo" e l'oggetto si chiami **"Forma\_esempio"**)

Approfittiamo per notare una particolarità:

notate alle righe 111 e 123 che i nomi passati come argomento hanno uno spazio, mentre nel nome delle geometrie riportato nella **vista ad albero**, è presente il carattere trattino basso (-), **FreeCAD** trasforma i caratteri spazio presenti nel nome come argomento, in un carattere di trattino basso (-).

La cosa la potete anche derivare, e a volte può risultare utile saperlo,

### 9.1.1. Listato - aereo



```
1 #
2 """aereo.py
3
4     This code was written as an sample code
5     for "FreeCAD Scripting Guide"
6
7     Author: Carlo Dormeletti
8     Copyright: 2020
9     Licence: CC BY-NC-ND 4.0 IT
10 """
11
12 import FreeCAD
13 from FreeCAD import Base, Vector
14 import Part
15 from math import pi, sin, cos
16
17 DOC = FreeCAD.activeDocument()
18 DOC_NAME = "Pippo"
19
20 def clear_doc():
21     """
22     Clear the active document deleting all the objects
23     """
24     for obj in DOC.Objects:
25         DOC.removeObject(obj.Name)
26
27 def setview():
28     """Rearrange View"""
29
30     FreeCAD.Gui.activeDocument().activeView().viewAxometric(
31         ↪ )
32     FreeCAD.Gui.activeDocument().activeView().setAxisCross(
33         ↪ True)
34     FreeCAD.Gui.SendMsgToActiveView("ViewFit")
35
36 if DOC is None:
37     FreeCAD.newDocument(DOC_NAME)
38     FreeCAD.setActiveDocument(DOC_NAME)
39     DOC = FreeCAD.activeDocument()
40 else:
41
42     clear_doc()
43
44
```

```
45 # EPS= tolerance to uset to cut the parts
46 EPS = 0.10
47 EPS_C = EPS * -0.5
48
49
50 def cubo(nome, lung, larg, alt, cent = False, off_z = 0):
51     obj_b = DOC.addObject("Part::Box", nome)
52     obj_b.Length = lung
53     obj_b.Width = larg
54     obj_b.Height = alt
55
56     if cent == True:
57         posiz = Vector(lung * -0.5, larg * -0.5, off_z)
58     else:
59         posiz = Vector(0, 0, off_z)
60
61     obj_b.Placement = FreeCAD.Placement(
62         posiz,
63         FreeCAD.Rotation(0, 0, 0),
64         FreeCAD.Vector(0,0,0)
65     )
66
67     DOC.recompute()
68
69     return obj_b
70
71
72 def base_cyl(nome, ang, rad, alt ):
73     obj = DOC.addObject("Part::Cylinder", nome)
74     obj.Angle = ang
75     obj.Radius = rad
76     obj.Height = alt
77
78     DOC.recompute()
79
80     return obj
81
82 def sfera(nome, rad):
83     obj = DOC.addObject("Part::Sphere", nome)
84     obj.Radius = rad
85
86     DOC.recompute()
87
88     return obj
89
90
```

```
91 def mfuse_obj(nome, objs):
92     obj = DOC.addObject("Part::MultiFuse", nome)
93     obj.Shapes = objs
94     obj.Refine = True
95     DOC.recompute()
96
97     return obj
98
99
100 def aeroplano():
101
102     lung_fus = 30
103     diam_fus = 5
104     ap_alare = lung_fus * 1.75
105     larg_ali = 7.5
106     spess_ali = 1.5
107     alt_imp = diam_fus * 3.0
108     pos_ali = (lung_fus*0.70)
109     off_ali = (pos_ali- (larg_ali * 0.5))
110
111     obj1 = base_cyl('primo cilindro', 360, diam_fus,
112                    ↪ lung_fus)
113
114     obj2 = cubo('ali', ap_alare, spess_ali, larg_ali, True,
115               ↪ off_ali)
116
117     obj3 = sfera("naso", diam_fus)
118     obj3.Placement = FreeCAD.Placement(Vector(0,0,lung_fus),
119               ↪ FreeCAD.Rotation(0,0,0), Vector(0,0,0)
120               ↪ )
121
122     obj4 = cubo('impennaggio', spess_ali, alt_imp, larg_ali,
123               ↪ False, 0)
124     obj4.Placement = FreeCAD.Placement(Vector(0,alt_imp * -1
125               ↪ ,0), FreeCAD.Rotation(0,0,0), Vector(0,
126               ↪ 0,0))
127
128     objs = (obj1, obj2, obj3, obj4)
129
130     obj = mfuse_obj("Forma esempio", objs)
131     obj.Placement = FreeCAD.Placement(Vector(0,0,0), FreeCAD
132               ↪ .Rotation(0,0,-90), Vector(0,0,pos_ali)
133               ↪ )
134
135     DOC.recompute()
```

```

128     return obj
129
130
131 aeroplano()
132
133 setview()

```

## 9.2. Il nome della Rosa

*Che cosa c'è in un nome? Ciò che noi chiamiamo con il nome di rosa, anche se lo chiamassimo con un altro nome, servirebbe pur sempre lo stesso dolce profumo.*

— William Shakespeare, *Romeo e Giulietta* atto II, scena II

Analizziamo ora una particolarità di **FreeCAD**, il nome dell'oggetto.

Abbiamo citato il Bardo di Albione, o come dice afferma qualcuno l'emigrato Siciliano Guglielmo Scrollalanza, se chiamassimo rosa con un altro nome, rimarrebbe sempre il profumo.

Visualizzando le proprietà di un oggetto, nella **Ling. Dati**, nella sezione **Base**, trovate per ogni oggetto la proprietà **Label**.

Questa proprietà contiene il nome che abbiamo assegnato all'oggetto quando lo abbiamo creato.

Possiamo cambiarla liberamente usando ad esempio questo comando:

```

FreeCAD.getDocument("Pippo").getObject("Forma_esempio").
    ↳ Label = "nuovo nome"

```

Bisogna però ricordare che un oggetto possiede due proprietà:

- **Name** che contiene il riferimento interno di **FreeCAD**, che una volta creata diventa “di sola lettura”.
- **Label** che è accessibile e modificabile liberamente.

Questa cosa la potete verificare se nella **console Python**

```

FreeCAD.getDocument("Pippo").getObject("Forma_esempio").Name
'Forma_esempio'

```

mentre se provate a modificarla con:

```

FreeCAD.getDocument("Pippo").getObject("Forma_esempio").Name
    ↳ = "nome nuovo"

```

ottenete questo messaggio di errore:

```

Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: Attribute 'Name' of object 'DocumentObject'
    ↳ is read-only

```

Riepilogando:

Il nome che assegnata ad un oggetto diventa il suo “nome interno” che **non è modificabile**. Il nome visualizzato in **vista ad albero** è modificabile attraverso la proprietà **Label**.

Se avete necessità di accedere ad un “nome interno” che non riflette più quello della proprietà **Label** avete poche possibilità:

- ricordare o prendere nota del “nome interno”
- usare il trucco della modifica tramite l'interfaccia grafica di un parametro, con le opzioni citate all'inizio del capitolo e leggere il “nome interno” nel contenuto della funzione **.getObject()**

Bozza

# Capitolo 10

## Stampa 3D

Non ci occuperemo in dettaglio della stampa 3D, per la ragione che non è il principale argomento di questa guida.

Ci occuperemo di due cose principali:

- La progettazione finalizzata alla stampa 3D.
- la produzione dei modelli per la stampa 3D.

Iniziamo ad occuparci brevissimamente della catena di stampa.

### 10.1. La catena di Stampa

In genere quando si stampa un modello in 3D partendo da zero si deve passare attraverso alcune fasi:

1. Produzione del Modello
2. Creazione del file STL o AMF
3. Conversione del file STL o AMF nel “linguaggio della stampante”
4. Stampa del Modello
5. Eventuali correzioni dimensionali o per ridurre problemi in punti particolari della stampa

In genere queste fasi sono eseguite da almeno tre programmi:

- il modellatore si occupa delle fasi 1,2 e 5.
- un programma di Slicing si occupa di prendere il file prodotto al punto 2 e trasformarlo in un file pronto per la stampa
- un programma si occupa di inviarlo alla stampante, a volte è un vero e proprio programma che controlla la stampante, altre volte è semplicemente l'operazione di copia del file su una scheda SD da inserire nella stampante.

Tranne il punto 3 e 4 per cui non abbiamo molto da fare con **FreeCAD**, gli altri punti ovviamente hanno a che fare con il nostro programma.

Il punto 1 ovviamente lo abbiamo affrontato estesamente, come fare a produrre un modello dovrebbe essere chiaro. Il punto 5 sembra poco collegato con **FreeCAD**, è legato alla modellazione parametrica.

In genere è possibile introdurre durante la progettazione un parametro che entri come moltiplicatore in alcune dimensioni del modello che se messo ad 1.0 permette di creare il modello “normalmente”, se messo diciamo a 1.1 ingrandisce queste dimensioni del modello.

Queste trasformazioni lineari possono essere eseguite anche esternamente agendo direttamente in fase di conversione del modello, magari applicando un fattore di ingrandimento lineare al modello, molti programmi di Slicing presentano questa opzione.

## 10.2. Progettare per la stampa

Un problema con cui in genere ci si scontra è quello dei “fori”, cioè del fatto che un foro, o buco che dir si voglia in genere è approssimato dalla stampante con qualcosa di più piccolo, la scelta è voluta per due motivi:

- Un motivo geometrico, un foro è circolare, ma la stampante lo deve approssimare con un poligono costituito da un certo numero di lati, questo numero di lati dipende da molti fattori come lo spessore del salsicciotto che esce dall’ugello della stampante oppure dalla conversione del modello da “Solido 3D” interno a **FreeCAD** a modello Mesh che deve essere passato al programma di Slicing.
- un motivo “pratico”, molti Slicer fanno internamente la supposizione che il foro poi sia ripassato con una punta per portarlo alle dimensioni definitive, questa assunzione interna è fatta per gli stessi presupposti geometrici citati or ora ma anche per altre ragioni “storiche” legate al comportamento dei primi Slicer, oramai è diventata una specie di prassi che un foro sia leggermente più piccolo.

Per cui una delle cautele da assumere in fase di progettazione è tenere presente che i fori andrebbero specificati in modo parametrico, con parametri “pensati” in base allo loro destinazione finale, piccoli esempi:

- Se un foro è destinato ad accogliere la testa di una vite esagonale, magari potrà tornare utile impostare un diametro di riferimento maggiore, in modo da alloggiare più comodamente la testa.
- Viceversa se deve accogliere una testa esagonale o un dado e deve tenerla in posizione, il diametro nominale va bene così oppure andrà ridotto, e dovrà essere diverso da quello dell’alloggiamento del punto precedente
- il foro dovrà essere filettato, o accogliere una vite autofilettante, il diametro nominale magari va bene, o addirittura andrà previsto un diametro minore, per avere maggiore tenuta.
- Il foro deve essere passante, in questo caso un diametro maggiore eviterà di dover allargare un foro con la punta del trapano.

Altre aspetti da considerare nella progettazione che possono rivelarsi utili sono:

- Alcune parti se stampate troppo piccole risultano complicate da stampare, oppure se ci sono particolari come “i ponti” lo Slicer usato introduce troppi sostegni che rallentano la stampa, con l’esperienza si arriverà ad aggiungere un elemento di sostegno direttamente in fase di progettazione, destinato ad essere spezzato o limato in fase di rifinitura solo dove è necessario.
- Altre parti magari delle basi di piccolo diametro si aggrapperanno difficilmente alla base della stampante, potrà risultare utile creare dei cerchi o altre forme di basso spessore, 0,10 o 0,20 mm che miglioreranno l’adesione in fase di stampa

### 10.3. Produzione di file per la stampa

L’operazione principale è quella di esportazione di un file in formato STL o AMF pronto per essere dato in pasto allo Slicer per la preparazione alla stampa.

Dovremo usare alcune cautele, usando l’interfaccia grafica è possibile, ma la definizione dei parametri di esportazione è relativamente complicata perché sono sparsi in giro per l’interfaccia principale e quelli dei workbench interessati.

L’approccio attraverso lo Scriting presenta una maggiore facilità di definizione dei parametri.

Per comprendere a pieno la spiegazione dobbiamo definire alcuni termini:

**Solido** è l’oggetto prodotto con **FreeCAD** in genere è definito come un solido ottenuto con delle funzioni matematiche, per cui è costituito da linee curve (dove non esiste un piano), internamente è definito in modo abbastanza complesso, **FreeCAD** internamente usa un modellatore **BREP** che usa anche curve per definire il modello.

**Tassellazione** l’operazione che trasforma o per meglio dire approssima un **Solido** in una superficie formata da tasselli, triangolari o poligonali a seconda del modello, definendo questi tasselli usando le coordinate dei vertivi del tassello, e implicitamente considerando il tassello un poligono piano, dopo la tassellazione il modello è in pratica una serie di coordinate di ogni vertice dei tasselli.

**Mesh** letteralmente “rete” è la superficie ottenuta tassellando la superficie del Solido

Dalla descrizione possiamo intuire che l’operazione di tassellazione comporta una “perdita di precisione” dei particolari che non possono essere trasformati perfettamente in una superficie piana, e meno di non aumentare in modo esponenziale il numero dei tasselli creati.

Per compiere un operazione di tassellazione è necessario quindi fornire una qualche indicazione di quanto vogliamo che sia questo scostamento tra Solido e Mesh, questo può essere fatto in vari modi che dipendono dall’algoritmo di tassellazione.



### 10.3.1. Trasformazione in Mesh

Presentiamo un modello di modulo per l'esportazione di un file “generico”:

```

1 def create_mesh(o_name):
2     """
3     Create the mesh object
4     """
5     mesh = DOC.addObject("Mesh::Feature", "Mesh-" + o_name)
6     mesh.Mesh = MeshPart.meshFromShape(
7         Shape=DOC.getObject(o_name).Shape,
8         LinearDeflection=0.01,
9         AngularDeflection=0.025,
10        Relative=False
11    )
12    mesh.Label = "Mesh-" + o_name
13    mesh.ViewObject.CreaseAngle = 25.0

```

L'operazione presuppone che siano aggiunte queste due linee all'inizio del file che importano i necessari moduli di **FreeCAD**:

```

import Mesh
import MeshPart

```

Si presuppone di avere un modello finito di cui si conosce il “nome interno”, che è l'unico parametro passato al metodo.

Alla riga 5 potete notare la “solita” creazione di un oggetto, stavolta sarà un oggetto **Mesh::Feature**, che sarà nominato con un nome composto dal prefisso **Mesh-** e dal nome passato al metodo.

Questo oggetto viene definito con tre proprietà:

1. **Mesh** a cui viene passato oggetto Mesh di cui vediamo la creazione fra poco
2. **Label** che riporta il nome dell'oggetto pari pari al nome assegnato prima
3. **viewObject.CreaseAngle** modifica la visualizzazione dell'oggetto per migliorarne l'aspetto nella **vista 3D**

Alla proprietà **Mesh** viene assegnato l'oggetto ottenuto dalla funzione **MeshPart.meshFromShape()**, che è abbastanza complessa, notate che vengono passati alcuni parametri tra le parentesi della funzione:

1. **Shape** che è la proprietà **Shape** dell'oggetto **DOC.getObject(o\_name)**, ricordiamoci le considerazioni fatte in precedenza sul “nome interno” e il nome visualizzato.
2. **LinearDeflection** limita la distanza tra la curva del solido e la sua tassellazione.
3. **AngularDeflection** limita l'angolo tra successivi segmenti della tassellazione.
4. **Relative** se posto a **True** il massimo scostamento tra Solido e Mesh sarà il valore specificato da **LinearDeflection** moltiplicato per la lunghezza del segmento della tassellazione.

Si intuisce facilmente che i parametri **LinearDeflection**, **AngularDeflection** e **Relative**, regolano la precisione della tassellazione e quindi lo scostamento del Solido dalla Mesh.

I parametri proposti sono quelli che dopo qualche prova ho ritenuto accettabili in termini di:

- Precisione del modello.
- tempo di conversione.

In genere si consiglia in vari luoghi, che il parametro **AngularDeflection**, sia circa un quarto del parametro **LinearDeflection**, perché stiamo definendo l'angolo tra un tassello e il successivo, e di porre il parametro **Relative** a **False** in modo da limitare lo scostamento tra una faccia “lunga” e la successiva.

Con i parametri proposti, in genere si aumenta il numero dei tasselli rispetto alle impostazioni di default, potete liberamente sperimentare e trovare i valori che più soddisfano le vostre esigenze, nella documentazione di **FreeCAD** all'indirizzo [https://wiki.freecadweb.org/Mesh\\_F](https://wiki.freecadweb.org/Mesh_F) potete trovare maggiori informazioni sull'argomento.

Ora la cosa si complica in relazione al tipo di file che vogliamo ottenere:

### 10.3.2. Formato AMF

```
15     filename = u"/Model_" + o_name + u".amf"
16     Mesh.export([mesh], os.path.join(HOME_PATH, filename),
                  ↪ tolerance=0.01, exportAmfCompressed=
                  ↪ False)
```

Niente di trascendentale, analizziamo i parametri passati alla funzione **Mesh.export()**:

- Bisogna passare una **lista** come primo parametro, anche se contiene un solo oggetto.
- La creazione del nome del file potrebbe creare problemi se contiene caratteri speciali, nel caso usare le cautele proposte quando abbiamo parlato delle librerie di oggetti a pagina 77.  
Nel caso proposto il nome viene creato usando un percorso definito “sommato” al nome dell’oggetto e all’estensione **.amf**.
- il parametro **tolerance** ovviamente deve essere compatibile con la deviazione che abbiamo impostato nella trasformazione del Solido in Mesh, per non vanificare quell’operazione.
- Il valore di **exportAmfCompressed** se usate questo file con alcuni programmi di Slicing tipo Slic3R deve essere **False**.

### 10.3.3. Formato STL

Usando il formato **STL**, abbiamo invece le righe diventano:

```
15 filename = u"/Model_" + o_name + u".stl"
16 Mesh.export( [mesh], os.path.join(HOME_PATH, filename),
               ↪ tolerance=0.01)
```

Niente di particolare, valgono le stesse considerazioni fatte in precedenza per il parametro **tolerance**, non esistono altre opzioni conosciute per l'esportazione.

# Appendice **A**

## Elementi Interfaccia Utente

area della barra degli strumenti - p. 4, 6

**Barra degli strumenti Viste** - p. 23

barra di stato - p. 4

console Python - p. 4, 8, 82–84, 88

editor delle Macro - p. 5, 6

editor Python - p. 6

editore delle proprietà - p. 4, 5, 24, 33, 82, 83

finestra dei rapporti - p. 4, 32

**Ling. Dati** Linguetta dell'EdP - p. 5, 24, 61, 81–83, 88

**Ling. Vista** Linguetta dell'EdP - p. 5, 24, 33, 61, 81, 83

menu standard - p. 4

selettore degli Ambienti - p. 4

**vista 3D** La finestra 3D - p. 3–6, 23, 93

**vista ad albero** - p. 4, 25, 82–84, 89

**vista combinata** - p. 4, 25, 33

**vista selezione** - p. 4

# Appendice **B**

## Glossario

angoli di Eulero o di Tait-Bryan - p. 34

operazioni booleane - p. 38

intersezione - p. 38

sottrazione - p. 38

unione - p. 38

BOZZA

# Appendice C

## Voci di menù

**File** Voce di menù - p. 6

**Apri** - p. 6

**Modifica** Voce di menù (*Orig: Edit*) - p. 4, 6, 9, 82

**Preferenze** (*Orig: Preferences*) - p. 4, 6, 9, 82, 83

**Generale** (*Orig: General*) - p. 4, 6, 9, 82

**Editor** Linguetta - p. 9

**Finestra di Output** Linguetta - p. 6

**Lingua** Linguetta (*Orig: Language*) - p. 4

**Macro** Linguetta (*Orig: Macro*) - p. 82

**Visualizza** Voce di menù - p. 4, 6, 23, 25

**Barre degli Strumenti** - p. 6

**Axonometric** - p. 23

**Isometrica** - p. 23

**Macro** - p. 6

**Vista Ortografica** - p. 23

**Vista Prospettica** - p. 23

**Viste Standard** - p. 23

**Origine degli assi** - p. 25

**Panelli** - p. 4

**Console Python** - p. 5

**Report** - p. 5

# Appendice D

## Oggetti di FreeCAD

**Angle** Proprietà - p. 26

**Angle1** Proprietà - p. 82, 83

**Angle2** Proprietà - p. 82

**Angle3** Proprietà - p. 82, 83

**Base** Proprietà - p. 39, 41, 49

**bordo** Componente Geometria (*Orig: edge*) - p. 24

**extrude** funzione - p. 48, 49

**faccia** Componente Geometria (*Orig: face*) - p. 24, 48, 51

**guscio** Componente Geometria (*Orig: shell*) - p. 24

**Height** Proprietà - p. 26

**Label** Proprietà di un oggetto - p. 88, 89, 93

**Mesh** Proprietà - p. 93

**Part::Box** Geometria - p. 25, 30

**Part::Cut** Op. booleana - p. 38, 40, 41

**Part::Cylinder** Geometria - p. 26, 30

**Part::Feature** Geometria - p. 49, 54, 58

**Part::Fuse** Op. booleana - p. 38, 39, 41

**Part::Loft** Strumento - p. 56, 58

**Part::MultiCommon** Op. booleana - p. 38, 41

**Part::MultiFuse** Op. booleana - p. 38, 42, 76, 84

**Part::Sphere** Geometria - p. 30, 60, 81

**Placement** Proprietà di un oggetto - p. 31–35, 49, 76, 83

**Angolo** (*Orig: Angle*) - p. 33

**Asse** - p. 33

**Posizione** - p. 33

**polilinea** Componente Geometria (*Orig: wire*) - p. 24, 48

**Radius** Proprietà - p. 26, 60, 82, 83

**Refine** Proprietà - p. 39, 41

**revolve** funzione - p. 54

**Sections** Proprietà - p. 58

**Shape** Proprietà - p. 22, 49, 54, 58, 93

**ShapeColor** Proprietà - p. 61

**Shapes** Proprietà - p. 22, 40, 42, 76

**Tool** Proprietà - p. 39, 41

**Trasparency** Proprietà - p. 61

**vertice** Componente Geometria (*Orig: vertex*) - p. 24

**vettore** Componente Geometria - p. 23, 24, 33, 34

**ViewObject** Proprietà - p. 24, 61

**Visibility** Proprietà - p. 61

Bozza