

Guida allo scripting in FreeCAD

Autore: Carlo Dormeletti

Versione di riferimento di **FreeCAD**: **0.19**

Versione della guida: **0.50**

Data di stampa: **8 maggio 2021**

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 italiano di **FreeCAD**.

Un grazie generale a quanti hanno segnalato errori, omissioni e brutture varie, ai “primi lettori”, e quanti mi hanno aiutato a rendere meno contorte le mie spiegazioni.

Non posso esimermi dal citare in particolare:

- Luixx
- renatorivo

Che hanno incoraggiato ed aiutato i primi passi di questo lavoro.

Un riconoscimento speciale a Giorgio (Moebius); Senza il suo aiuto la grammatica e la punteggiatura sarebbero state approssimative anzi decisamente obbrobriose.

Un ultimo ringraziamento a \TeX , $\text{\LaTeX 2}_{\epsilon}$ e \TikZ senza i quali questa guida non sarebbe così ricca di riferimenti, note, tabelle, graficie ed 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.

v0.25 – 16 marzo 2020 Non pubblicata - correzioni di errori e qualche precisazione, piccole modifiche all'impaginazione, aggiunta di parti e di descrizioni migliorate.

v0.30 - 11 maggio 2020 Revisione profonda di alcuni capitoli, modifica di molti listati eliminando le impostazioni comuni all'inizio del file, ora sc-base.py va aggiunto a tutti i listati, in questo modo si uniformano le operazioni di impostazione iniziale, che hanno dato problemi su alcuni sistemi e versioni. Alcune migliorie sono possibili.

v0.31 - Piccole modifiche e correzioni.

v0.40 - 29 marzo 2021 Revisione del testo e aggiornamento dei numeri delle versioni stabili e di sviluppo

Indice

Prefazione	IX
Convenzioni usate nel testo	X
1. Primi passi	3
1.1. Impostazioni	5
passo 1	5
passo 2	5
passo 3	5
passo 4	5
1.2. L'editor delle proprietà	6
1.3. L'editor delle Macro	6
2. Scripting, chi era costui	8
2.1. Breve introduzione a Python	9
2.1.1. La sintassi	10
Indentazione	11
Spazi, a capo e righe bianche	12
Commenti	13
Le <code>docstring</code>	14
2.1.2. Variabili	14
2.1.3. Le liste e le tuple	15
2.1.4. I metodi e le funzioni	17
2.2. Antipasto	18
2.2.1. Le direttive <code>import</code>	20
2.2.2. I parametri	21
2.2.3. I cicli	22
2.2.4. Le condizioni	23
2.3. Nota metodologica	24
2.4. Listato - schema base	26
3. Modellazione di Solidi 3D	29
3.1. Spazio tridimensionale	29
3.1.1. I Vettori	30
3.2. Visualizzazione	30
3.2.1. Dati e Vista	31
3.3. Geometria, Topologia, Solidi e altro	32
3.3.1. Geometria	32
3.3.2. Topologia	33

3.3.3. Maglie	34
3.3.4. Tassellazione	36
3.4. Modellazione 3D	37
3.4.1. Geometria e topologia in FreeCAD	37
3.4.2. Geometria solida costruttiva CSG	40
3.4.3. Boundary representation BREP	40
4. Modellazione CSG	42
4.1. I primi oggetti	42
4.1.1. Listato - figure-base.py	44
5. Posizionamento	46
5.1. Il riferimento di costruzione	47
5.2. Posizionamento	49
5.2.1. La proprietà Placement	50
5.2.2. Listato - Riferimento costruzione	53
6. Le operazioni booleane	55
6.1. Unione	55
6.2. Sottrazione	57
6.3. Intersezione	58
6.4. Operazioni booleane - esempio completo	59
7. Modellazione BREP	62
7.1. Estrusione	65
7.1.1. Listato - estrusione	67
7.2. Rivoluzione	70
7.3. Loft	72
7.4. Un esempio avanzato	74
7.4.1. Listato - esempio BREP	77
8. La componente Vista	80
9. Programmazione modulare	83
9.1. Modularizzazione	84
9.1.1. Listato - Modulo GfcMod	86
9.1.2. Listato - sardine.py	90
9.2. Modellazione parametrica	93
9.2.1. Creazione “automatica” di geometrie	95
9.2.2. Listato - cubo_stondato modificato	97
9.3. Librerie di oggetti	98
9.3.1. Esempio di Libreria	100
10. Proprietà degli “oggetti 3D”	102
10.1. Conoscere il nome delle proprietà	102
10.1.1. Listato - aereo	106
10.2. Il nome della Rosa	108

11. Stampa 3D	110
11.1. La catena di Stampa	110
11.2. Progettare per la stampa	111
11.3. Produzione di file per la stampa	112
11.3.1. Trasformazione in Mesh	112
11.3.2. Formato AMF	114
11.3.3. Formato STL	114
A. Elementi Interfaccia Utente	115
B. Glossario	116
C. Voci di menù	117
D. Oggetti di FreeCAD	118

Elenco delle figure

1.1. L'interfaccia Utente al primo avvio	3
1.2. L'interfaccia Utente	4
1.3. Scheda Vista e Scheda Dati	6
2.1. schema a blocchi condizioni if	25
3.1. Lo spazio 3D	29
3.2. BdS Viste	30
3.3. Faccetta di una maglia.	35
3.4. Facce <i>manifold</i> e <i>non manifold</i>	36
3.5. Modelli <i>manifold</i> e <i>non manifold</i>	36
3.6. Operazioni Booleane.	41
5.1. Posizionamento	49
5.2. L'editor delle proprietà	50
6.1. Le operazioni Booleane.	58
7.1. Estrusione	66
7.3. Lo strumento Loft	74
7.4. L'oggetto creato usando la tecnica BREP	77
8.1. La Scheda Vista	81
9.1. Scatola sardine	84
10.1. Derivazione delle proprietà.	103

Elenco delle tabelle

3.1. Geometrie	38
5.1. punti di riferimento	47
5.2. proprietà Placement	51
5.3. Angoli di Eulero	51
5.4. Rotazione usando gli Angoli di Eulero	52

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 ¹.

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

¹Essendo 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** 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 oppure 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.

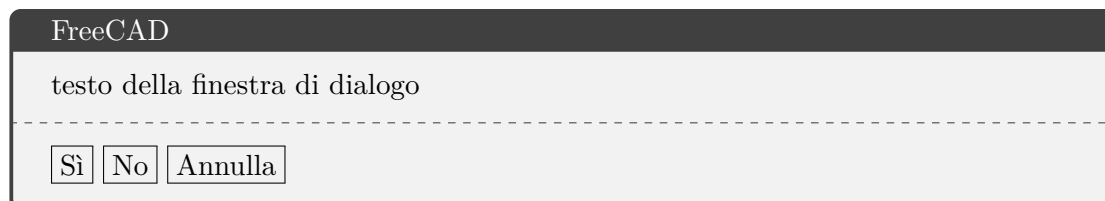
```
obj_b.Placement = FreeCAD.Placement(Vector(0, 0, 0), FreeCAD
    ↪ .Rotation(0,0,0))
```

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 0.19 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 come questa:



Messaggi testuali

Quello che viene visualizzato da **FreeCAD**, in modo testuale, viene contrassegnato nel seguente modo:

```
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 un'applicazione per la modellazione parametrica di tipo CAD/CAE (Computer Aided Design e Computer Aided Engineering). È stato realizzato per la progettazione meccanica, ma si comporta egregiamente in tutti casi in cui è necessario modellare degli oggetti 3D con precisione ed poter controllare la sequenza della modellazione.

Le parole chiave sono:

- **Modellatore**, cioè un creatore di modelli 3D.
- **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 molte parti scritte in Python, che si interfacciano con questa API.

FreeCAD è dotato di diversi “**ambienti di lavoro**” definiti in inglese “**WorkBench**” letteralmente ***banchi da lavoro*** creati per facilitare lo svolgimento di diversi compiti: modellazione 3D, redazione di progetti, architettura, ma anche analisi dei modelli tramite tecniche di CAE, nonché la creazione di tassellazioni (Mesh) finalizzate alla stampa 3D o al rendering grafico.

Questa guida è incentrata sullo “Scripting”, ovvero la 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 introduzione generale all'uso di **FreeCAD**, per questo ci sono già altre guide; vuole essere prevalentemente una “guida alla programmazione”, poiché la documentazione su questo argomento presente sul sito di **FreeCAD** in genere confonde il principiante.

I motivi di questa confusione sono molteplici:

- La documentazione è stata realizzata da molte persone diverse, è organizzata in un wiki e quindi modificata in tempi diversi e da molte “mani”.
- **FreeCAD** non è ancora arrivato alla sua versione stabile, per cui alcune cose cambiano velocemente, e la documentazione a volte non “tiene il passo” con le modifiche del codice.
- Le tecniche con cui automatizzare il lavoro attraverso lo scripting sono molteplici, e dipendono anche dallo “stile” di chi scrive lo script..

- Lo scripting è considerato da molti un aspetto secondario rispetto all'interfaccia grafica, alcuni non conoscono nemmeno il fatto che sia possibile creare modelli scrivendo un "programma".

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 **0.19**, mentre quella di sviluppo è la **0.20**.

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 raramente sono aggiornate alla versione stabile.
- **Mac**: ad oggi è consigliabile usare la versione **0.19**.
- **Windows**: troverete sia l'installer, sia le "portable builds", per la versione stabile e anche per quella di sviluppo.

Secondo il mio parere il sistema migliore per installare **FreeCAD** sui sistemi Linux è quello di usare le **AppImage**.

Una AppImage è un file che contiene tutto il necessario per far girare **FreeCAD** in modo quasi indipendente dalle librerie installate sul computer, è possibile anche caricare una AppImage su una chiavetta e usarla dove si vuole, sempre che sul computer sia installata un versione recente di Linux.

Le Appimage hanno anche il vantaggio non secondario di poter essere "aggiornate" utilizzando le istruzioni e gli strumenti presenti sul sito di **FreeCAD**, che permettono di scaricare solo le "differenze" tra l'AppImage posseduta e quella aggiornata. Dato che la dimensione di una AppImage si aggira attorno ai 500MB, il risparmio di dati è notevole.

Per Windows consiglio di usare le portable builds, seguendo le istruzioni sul sito di **FreeCAD**.

Il maggior vantaggio di usare le Appimage o le portable builds è quello di avere sempre la versione aggiornata, un secondo vantaggio è quello di poter far convivere sullo stesso sistema sia la versione stabile che quella di sviluppo senza avere troppi problemi.

Capitolo 1

Primi passi

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

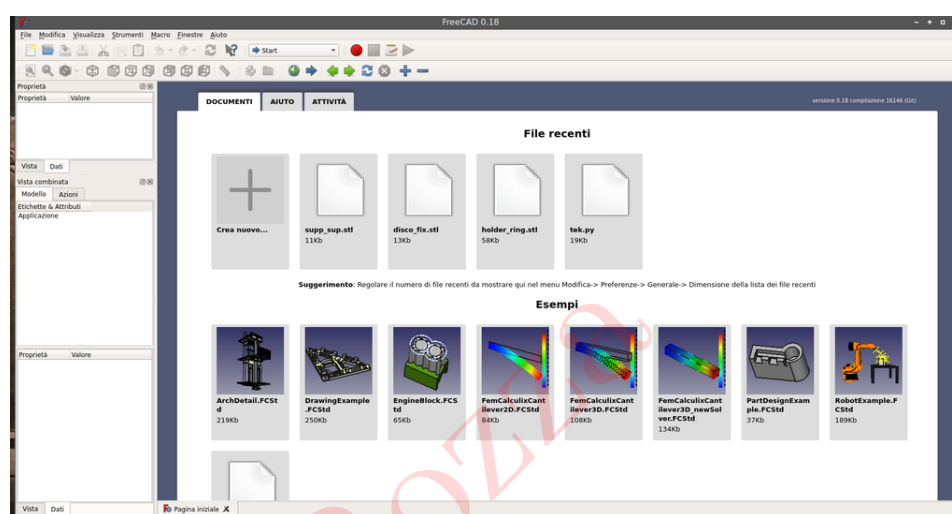


Figura 1.1.: L'interfaccia Utente al primo avvio

Prima di poter operare in modo proficuo, 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, soprattutto a proposito della navigazione con il mouse, nella **vista 3D**. Una spiegazione approfondita del funzionamento di **FreeCAD** va oltre gli scopi di questa guida e risulterebbe un inutile doppione della buona documentazione fornita sul sito del programma.

Vediamo 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.

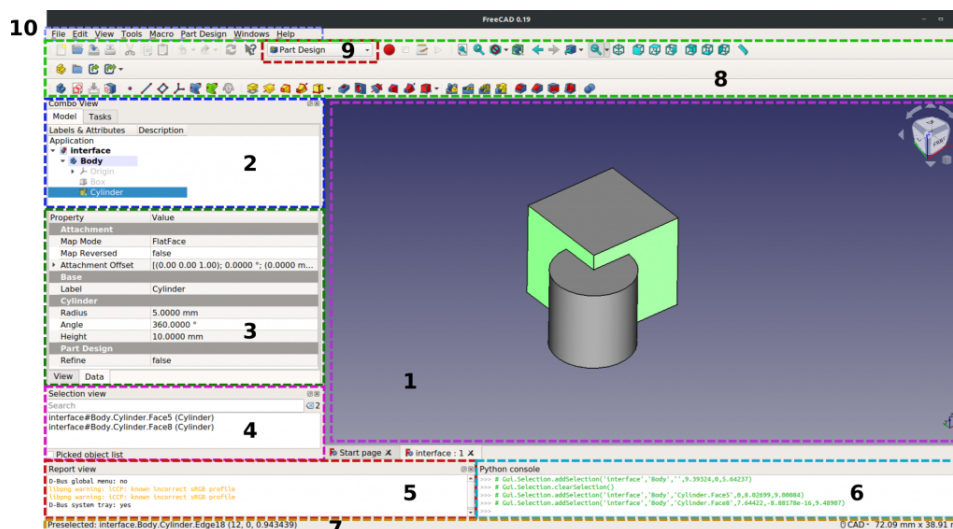


Figura 1.2.: L'interfaccia Utente

1. La **vista 3D**, che visualizza gli oggetti geometrici contenuti nel documento, in questa posizione ci potrebbe essere anche la finestra 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'**editor delle proprietà** (abbreviato a volte in **EdP**) (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 “Schede” che indicano i documenti aperti, possiamo alternarci tra loro cliccandoci sopra con il mouse.

Se il programma fosse in lingua Inglese, per cambiarla selezionate le voci di menù **Edit** ⇒ **Preferences**, scegliere l'icona **General** e la scheda **Language** dove potete scegliere la lingua Italiana.

1.1. Impostazioni

Riportiamo di seguito le impostazioni necessarie per poter operare in modo proficuo, torneranno utili nel caso dobbiate installare e configurare **FreeCAD** in futuro.

passo 1

Nel menu **Visualizza** ⇒ **Pannelli**, selezionare:

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

passo 2

Nel menu **Visualizza** ⇒ **Barre degli Strumenti**, selezionare:

- **Macro**

passo 3

Nel menu **Modifica** ⇒ **Preferenze** alla sezione **Generale** nella scheda **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

passo 4

Nel menu **Modifica** ⇒ **Preferenze** alla sezione **Generale** nella scheda **Editor**

- Nel riquadro **Indentazione**:
 - Mettere il valore 4 a **Dimensione dell'indentazione**.
 - Mettere il valore 4 a **Dimensione della tabulazione**
 - Scegliere l'opzione **Inserisci gli spazi** (automaticamente viene disabilitato **Mantieni le tabulazioni**)
- Nel riquadro **Opzioni** selezionare:
 - **Abilita la numerazione delle righe**.

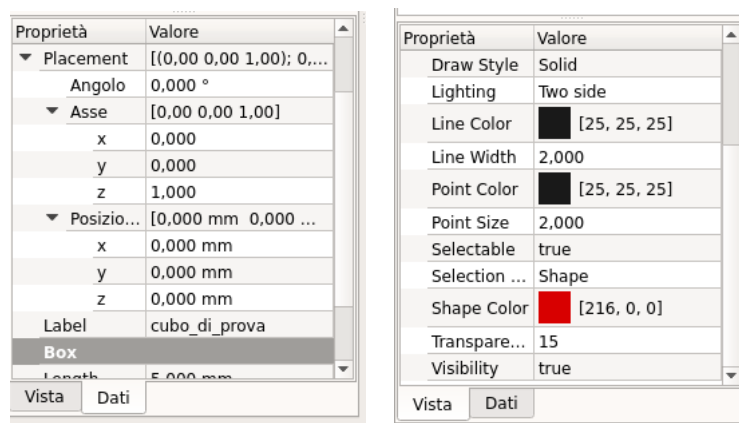
A questo punto dovrete avere la finestra del programma esattamente come nella figura fig. 1.2 nella pagina precedente.

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.

1.2. L'editor delle proprietà

L'**editor delle proprietà** presenta nella parte bassa due schede:

- La **Scheda Dati** contiene le informazioni “geometriche” dell’oggetto creato.
- La **Scheda Vista**, contiene le informazioni “grafiche” dell’oggetto, come il colore e lo stile grafico delle linee.



(a) EdP Scheda Dati.

(b) EdP Scheda Vista.

Figura 1.3.: Scheda Vista e Scheda Dati

Usiamo la denominazione abbreviata **Scheda Dati** per indicare **editor delle proprietà** parte **Dati** perché ne parleremo molte volte ed è meglio usare una denominazione convenzionale più corta che ripetere ogni volta la descrizione completa. Facciamo lo stesso per **Scheda Vista**.

1.3. L'editor delle Macro

All'interno di **FreeCAD** trovate già un decente editor per scrivere i programmi, 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 inoltre una barra degli strumenti (che abbiamo attivato con i comandi che descriviamo più sotto) con un bottone (triangolo verde), che permette di lanciare il programma e di vedere il risultato nella **vista 3D**.

Questa barra degli strumenti è mostrata nell'**area della barra degli strumenti**, quando ci si trova nell'**editor delle Macro**, e si presenta come nella figura qui a fianco, il bottone con la freccia (che nella immagine è grigio), diventa verde quando nell'editor è caricata una Macro od un programma Python.



Questo bottone serve per eseguire il file caricato 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 caricare la nuova versione.

L'**editor delle Macro** almeno nella versione **0.19** 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 semplicemente 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, nella directory che preferite, un file con estensione **.py** 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”.

Il termine Macro, in genere identifica programmi scritti in un linguaggio dedicato che gira all'interno di un interprete interno al programma, generalmente usato per eseguire comandi che “mimano” le sequenze dei comandi interni.

Gli script che scriveremo saranno dei veri e propri programmi scritti in Python, che potrebbero essere facilmente usati anche esternamente all'ambiente Python interno di **FreeCAD**, usando **FreeCAD** come se fosse una “semplice” libreria Python.

La potenza di **FreeCAD** sta proprio nell'usare non un linguaggio proprietario interno oppure un insieme ridotto di un linguaggio, ma un vero e proprio interprete Python che gira all'interno di **FreeCAD** e ne è una componente integrale, infatti 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.

Fornire una definizione di Script, è relativamente complicato, le traduzioni sono molte, quella più adeguata ed evocativa è “Copione teatrale”.

Facciamo una piccola digressione, 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.

Lo script è un testo scritto che va passato ad un “interprete dei comandi”; Python infatti è un linguaggio interpretato.

il testo di un programma nel gergo dei programmatori viene chiamato “sorgente”, per cui potremo usare indifferentemente le parole “programma”, “script”, “sorgente” o “codice” per riferirci semplicemente a quanto scritto in questo “file di testo”.

Accenniamo per amor della comprensione alla distinzione tra “linguaggio interpretato” e “linguaggio compilato”.

In un linguaggio compilato il sorgente passa attraverso un programma chiamato appunto compilatore che lo trasforma in un file eseguibile.

Questo file eseguibile, è in pratica il programma che gira sulla macchina, direttamente senza dover usare il compilatore.

Se ad esempio dobbiamo cambiare qualcosa nel comportamento del programma, dobbiamo passare per la trafila:

- Modificare i sorgenti.
- Ricompilare il programma.

- Eseguirlo e verificare che tutto vada come previsto.

Altro problema è che questo programma girerà solo su una macchina simile a quella su cui è stato creato, per cui un sorgente compilato usando Windows, non girerà su un Mac o su un computer che usa Linux.

Un linguaggio interpretato invece farà “girare” il programma all’interno del suo “interprete”.

I vantaggi sono molteplici:

- Il programma potrà girare indipendentemente dall’architettura della macchina usata, basta che esista un interprete per quel sistema e il programma girerà nello stesso modo su macchine diverse.
- Durante l’esecuzione gli errori eventualmente presenti possono essere mostrati e corretti “al volo”, senza dover passare da tutta la catena di compilazione.

Questi vantaggi possono sembrare minimi, ma 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. Controllando immediatamente la correttezza del codice ed eventualmente apportare immediatamente le modifiche e verificarne la correttezza.

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?

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 come abbiamo già accennato all’interno di **FreeCAD** si trova realmente un completo interprete Python, con tutte le sue librerie standard e qualche altra in più, per cui uno script è a tutti gli effetti un programma.

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**.

Introduciamo ora alcuni concetti, in modo molto grossolano, per una trattazione più formale e per ogni altro dubbio le pagine ufficiali 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 0.19** 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 cosmetiche, ma comunque possono generare “errori di sintassi”.

2.1.1. La sintassi

Per sintassi intendiamo il modo di scrivere un programma, cioè le regole che vanno seguite per scrivere correttamente il codice.

Le regole da tenere a mente sono poche.

Elencheremo le più “importanti” per non appesantire 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.

Introduciamo lo **Zen di Python**, cioè la filosofia dietro alla composizione del codice di Python, scritto in 20 aforismi da Tim Peters, che trovate nel PEP20.

1. Bello è meglio che brutto.
2. Esplicito è meglio che implicito.
3. Semplice è meglio che complesso.
4. Complesso è meglio che complicato.
5. Piatto è meglio che nidificato.
6. Sparso è meglio che denso.
7. La leggibilità è importante.
8. I casi particolari non sono abbastanza particolari per rompere le regole.
9. Considerato tutto, la praticità batte la purezza.
10. Gli errori non devono passare in silenzio.
11. A meno di non silenziarli in modo esplicito.
12. Davanti all'ambiguità, rifiutati di indovinare.
13. Deve esistere almeno un metodo, e preferibilmente solo uno, ovvio per fare una cosa.
14. Nonostante questo, non può essere ovvio al primo sguardo a meno di non essere Olandese.
15. Ora è meglio che Mai.
16. Comunque Mai è spesso meglio che “proprio” ora.
17. Se l'implementazione è difficile da spiegare, probabilmente è una cattiva idea.

18. Se l'implementazione è facile da spiegare, potrebbe essere una buona idea.

19. Lo “spazio dei nomi” è un'idea grandiosa – facciamone di più.

Sono dei buoni detti, anche se a volte difficilmente traducibili tanto sono permeati di slang da programmatori, molti sono ottimi, almeno i primi 7, il 17 e il 18 andrebbero seguiti, gli altri sono più degli scherzi o delle contorsioni mentali da programmatore.

Qualcuno noterà che sono solo 19, le spiegazioni sono diverse, riporto le due più diffuse:

- Il 20esimo posto è lasciato libero per il prossimo buon consiglio.
- Un lista con un indice massimo di 19 possiede 20 elementi (da 0 a 19 contati fanno 20 elementi), peccato che si parta in genere da 1.
Questo che potrebbe apparire una cosa contorta invece è una delle fondamenta della programmazione;
Quando si usano gli indici, `p[0]` significa il primo elemento di qualcosa che può essere indicizzato per cui `p[19]` è il 20esimo elemento.

Elenchiamo ora un paio di concetti generali. Dobbiamo necessariamente usare qualche termine “tecnico” anche se non ancora introdotti, ad esempio il termine “variabile” e “commento” non spaventatevi diverranno chiari fra pochissimo.

- L'importante è la “leggibilità” del programma
- La seconda cosa importante è l'omogeneità. Se si sceglie un modo per fare qualcosa, utilizzare quel modo in tutto il codice.
- Usare per quanto possibile nomi di variabili significativi, ad esempio non usare **pippo** per definire un diametro, magari usare `c_dia` oppure `dia_cono`.
- Utilizzare in modo ampio i commenti. (Riporteremo in seguito alcune regole da tenere a mente)

Indentazione

L'indentazione è il rientro delle righe, che evidenziano in modo visivo una parte di codice. Essa assume molta importanza in Python, in quanto definisce un **blocco** di codice.

È meglio usare 4 spazi per ogni indentazione, per fare in modo che l'**editor delle Macro** utilizzi gli spazi al posto delle tabulazioni, dovete andare nel menu **Modifica** ⇒ **Preferenze** alla sezione **Generale** nella scheda **Editor** e modificare 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**

Dato che ci troviamo in quella scheda, spuntate anche nel riquadro **Opzioni** la casella **Abilita la numerazione delle righe**, tornerà utile nel caso di errori per trovare più facilmente il punto che ha generato l'errore.

Un **blocco** (di codice) contiene una serie di linee di codice e viene distinto dal blocco successivo 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, che aiuta anche la leggibilità, è la spaziatura. Quando scriviamo un codice, cerchiamo di evitare di mettere gli spazi a caso.

Ecco 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 (vedi 2.1.3) 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 usate come eccezione alla convenzione che il carattere di fine 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:

```
FreeCADGui.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:

```
FreeCADGui.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 in 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.

Durante lo sviluppo di un programma risulta a volte utile “commentare” porzioni di codice che al momento non vogliamo usare, o che causano errore, per poi ritornarci sopra in seguito.

I commenti si possono inserire in Python usando il carattere “cancelletto” `#`.

Usato all’inizio di una riga questo carattere è utile per “commentare” porzioni di codice, cioè escludere quelle parti dall’esecuzione del programma, per commentare molte righe possiamo usare altri metodi.

Usate dopo la riga di codice è utile per ricordare qualcosa ad esempio “non toccare” oppure “da correggere”.

Usando un IDE, ¹ 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.

Risultano utili anche quando si usa un normale editor di testo, usando la funzione **Cerca**, per ritrovare questi punti “critici” in seguito.

Alcune regole per i commenti:

- Commentate molto, specie le parti che eseguono calcoli ed usano formule matematiche, per voi possono essere chiare, per altri può essere utile sapere cosa fanno.
- Scrivere per ricordare, quello che oggi è chiarissimo, magari fra sei mesi non lo sarà più, anche per l’autore del codice; ricordate la regola numero 2 dello Zen di

¹**Integrated Development Environment**, (ambiente di sviluppo integrato); Una specie di editor di testo specializzato usato dai programmatori per comporre codice

Python.

- Quando modificate il codice, analizzate anche i commenti; Un commento che contraddice il codice genera solo confusione.

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**. Si tratta di un commento particolare che è “BENE” esista per ogni **metodo** e serve a descrivere quello che fa il metodo e anche gli eventuali **parametri** del metodo.

Le **docstring** sono identificate e riconosciute dall’interprete perché cominciano e finiscono con tre virgolette `"""` le descriveremo meglio poco più avanti quando illustreremo i metodi.

Questo tipo di commento serve anche per escludere parti importanti di codice, ad esempio una decina righe di codice posso essere commentate semplicemente inserendo all’inizio e alla fine del codice le tre virgolette come delimitatore.

Attenzione però alle indentazioni e a non spezzare un blocco di codice.

2.1.2. Variabili

Le variabili sono dei contenitori. Possono essere descritte come delle scatole che possiedono una etichetta, questa etichetta è il nome della variabile, e dentro la scatola ci può essere di tutto, in Python tutto è un oggetto, anche **FreeCAD** stesso diventa un oggetto che si chiama appunto **FreeCAD**, lo vedremo più avanti.

I nomi delle variabili in Python seguono alcune convenzioni:

- Un nome scritto “TUTTO IN MAIUSCOLO” in genere si usa per le “costanti”, almeno dal punto di vista logico, anche se in Python dove “tutto è un oggetto” non esistono delle vere e proprie “costanti”, cioè oggetti che una volta assegnati non possono più essere modificati pena un “errore di sintassi”.
- Un nome che comincia con un carattere di “_” (*underscore* o “trattino basso”) in genere designa una variabile “locale” che è “BENE” non sia usata all’esterno dell’ambito in cui è definita, ad esempio un metodo od un modulo (concetti che spiegheremo più avanti).
- Una variabile scritta in questo modo `--nome--`, (notate che nonostante la grafica non esistono spazi tra i `_` e “nome”) ha un significato particolare ed è BENE non sia usata a meno di non sapere esattamente a cosa serve.
- Se si è proprio costretti ad usare un nome di variabile che genera 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 dichiararne 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”
- **string** ossia caratteri “stampabili” destinati a contenere del testo, quando perciò parliamo di stringa non ci riferiamo a quella delle scarpe.

Se scriviamo **alt = 0** definendola implicitamente come un “intero”. Quando poi operiamo su questa variabile usando 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.

Se vogliamo ottenere risultati consistenti, è meglio non abusare della flessibilità di Python, e operare in modo corretto, se definiamo un numero che useremo come numero in virgola mobile, cioè di tipo “float” 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.

2.1.3. Le liste e le tuple

Un modo di esprimere i dati con cui avremo a che fare spesso, sono le liste, in estrema sintesi sono degli elenchi di dati.

Potete pensarli come a dei vagoni di un treno, questa similitudine non è affatto azzardata ed infantile, perché a differenza di altri linguaggi di programmazione, in Python, una lista assomiglia proprio ad un treno merci, ogni vagone può avere un carico diverso.

Una lista viene semplicemente creata (dichiarata, assegnata) con:

```
miaLista = []
```

Con questa scrittura, creiamo una lista vuota. Potete notare la compattezza della scrittura ed il fatto che non contenga nessuna indicazione sul tipo di dati che sono contenuti nella lista.

Riprendendo l'esempio del treno merci, infatti, in caso di necessità possiamo comporre la lista usando tipi di dati diversi, questa scrittura nel linguaggio Python è perfettamente "legale":

```
miaLista = ["paperino", "via delle rose" , 46, "Topolinia",  
           ↪ "555-45634", "Indirizzo e telefono di  
           ↪ casa"]
```

La prima scrittura creava una lista vuota; in genere si creano liste vuote per riempirle, ma anche per svuotare una lista esistente e riutilizzarne il contenitore, cioè la variabile.

Vediamo ora come fare per riempirla.

L'istruzione che ci permette di aggiungere un elemento in coda alla lista è l'istruzione **append**, vediamo come usarla:

```
miaLista.append("pippo")
```

Attenzione, parliamo di elemento, per cui scrivendo:

```
miaLista.append(["paperino", "pluto", "topolino"])
```

Viene aggiunto un elemento alla lista **miaLista**, ma questo elemento è composto da un'altra lista, questa scrittura la incontreremo più avanti, ricordatela.

L'altra operazione importante, è quella di ottenere il valore immagazzinato in un certo elemento della lista, lo facciamo usando la cosiddetta indicizzazione, parola difficile per dire che specifichiamo l'indice ovvero la posizione che l'elemento che ci interessa occupa nella sequenza dei dati.

In questa scrittura:

```
nome = miaLista[0]
```

La doppia parentesi quadra dopo il nome di una variabile, significa che vogliamo l'elemento della lista **miaLista** che si trova nella posizione indicata dal numero all'interno della parentesi.

Abbiamo già accennato quando abbiamo parlato dello Zen di Python che gli indici delle liste partono da 0 per cui il primo elemento ha indice 0, il secondo indice 1 e così via.

Quando vogliamo aggiungere più elementi, usando una sola istruzione possiamo usare:

```
miaLista.extend(['paperino', "pluto", "topolino"])
```

mentre se vogliamo sapere quante elementi contiene una lista, basta scrivere:

```
numElem = len(miaLista)
```

Le liste sono molto usate perché sono delle sequenze e le sequenze sono iterabili, un brutto termine gergale che significa che possiamo scansionarle in ordine usando alcune istruzioni.

Una cosa particolare che esiste in Python e ha un nome particolare è la **tupla**, questo termine indica una lista immutabile, è usata quando i dati sono statici, cioè una volta

scritti devono essere solamente letti, si avvicina al concetto che in altri linguaggi viene chiamato **array**.

Non è comunque una “costante”, in quanto può semplicemente essere riscritta per intero assegnando al nome della variabile una nuova tupla.

si dichiara scrivendo:

```
miaTupla = ("pippo", "pluto", "paperino")
```

Essendo immutabile non esistono istruzioni per aggiungere elementi o per estenderla, però se ne può ottenerne un elemento, scorrerla oppure ottenere il numero dei suoi elementi esattamente come se si trattasse di una lista.

Si usa principalmente per un motivo, in termini di memoria è molto più efficiente, cioè ne consuma di meno.

Una cosa interessante è che queste due scritture:

```
miaLista.extend(["paperino", "pluto", "topolino"])
```

```
miaLista.extend(("paperino", "pluto", "topolino"))
```

Sono perfettamente equivalenti, cioè aggiungono gli elementi alla lista **miaLista**, “estendendo” la lista, cioè incrementandone il numero di elementi, per cui anche gli elementi “estesi” sotto forma di tupla diventano modificabili.

mentre:

```
miaLista.append(["paperino", "pluto", "topolino"])
```

```
miaLista.append(("paperino", "pluto", "topolino"))
```

Non sono per nulla equivalenti, tutte e due le scritture aggiungono un elemento alla lista, questo elemento è costituito nel primo caso dalla lista **["paperino", "pluto", "topolino"]** mentre la seconda scrittura aggiunge una tupla alla lista. Questa tupla rimane una tupla cioè immutabile.

2.1.4. 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:  
        DOC.removeObject(obj.Name)
```

mostra un **metodo** altrimenti chiamato **funzione**.

Possiamo usare indifferentemente sia “metodo” che “funzione” per riferirci genericamente a qualcosa che viene definito con la parola chiave **def()**.

Una cosa importante da notare è che alla fine della definizione è necessario mettere un carattere di due punti (:).

Altra cosa importante da ricordare è che un metodo possiede il proprio “spazio dei nomi”.

All'interno di un metodo una variabile 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

Forniremo alla fine di questo capitolo, nella sezione sezione 2.4 a pagina 26 con il titolo **Listato - schema base**, uno script di massima, potete pensarlo come uno schema “vuoto”, ma con i riquadri già disegnati.

Procederemo per gradi, cominciando con un programma che fa poche cose e aumentando mano a mano la complessità.

Presentiamo di seguito 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()
```

Le due righe dopo **import FreeCAD** eseguono queste operazioni:

- Riempio la variabile **DOC** con il risultato della chiamata alla funzione **FreeCAD.activeDocument()**, che restituisce l'oggetto del documento attivo.
- Riempio la variabile **DOC_NAME** con una stringa di testo contenente il nome del documento.

Notate l'uso dei nomi tutti in maiuscolo.

Abbiamo detto che questa convenzione indica qualcosa che concettualmente è una costante.

Se per la variabile **DOC_NAME**, appare abbastanza ovvio, per la variabile **DOC** può apparire non corretto, in quanto **DOC** viene modificato quando ad esempio creeremo degli oggetti.

Viene considerato una “costante”, perché ai fini logici rappresenta un “oggetto” che 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 dalle righe successive:

```
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 (cioè non esiste un documento attivo) 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*

Nelle prime righe di codice, mostrate sopra, potete vedere alcune direttive **import**.

Queste direttive contengono sempre un'istruzione **import**, ma hanno delle scritture leggermente diverse, spiegheremo fra poco il perché di queste differenze di scrittura.

```
import FreeCAD
```

Questa scrittura, la più semplice, importa il modulo **FreeCAD**.

Nel linguaggio python il codice viene diviso in moduli.

Per poter usare un 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 proviamo a digitare **dir()** nella **console Python**, vedremo 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, i più interessanti ai nostri fini sono:

- **FreeCAD** che contiene gli elementi “geometrici” di **FreeCAD**.
- **FreeCADGui** che contiene invece la parte “interfaccia Grafica” e soprattutto le componenti di visualizzazione degli oggetti.

Internamente, da qualche parte nel codice di **FreeCAD** viene eseguito questo tipo di import:

```
import FreeCAD as App
import FreeCADGui as Gui
```

Per questo motivo in molta documentazione ci si riferisce alle componenti **App** e **Gui**, i cui veri nomi sono **FreeCAD** e **FreeCADGui**, noi utilizzeremo la dicitura corretta fin dalla prima riga di codice.

Dovendo utilizzare molte volte un oggetto, potrebbe essere utile evitare di battere il riferimento completo all'oggetto nella forma **modulo.oggetto**, usando ad esempio solo **oggetto**.

Questo modo di procedere in genere viene sconsigliato perché espone ad alcuni rischi:

- Sostituire il nome dell'oggetto ad un nome interno od ad un nome di un altro modulo importato.
- Non capire a quale modulo appartiene quell'oggetto.

Conoscendone i pericoli possiamo però usare questa scrittura:

```
from FreeCAD import Vector, Rotation
```

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

Analogamente importiamo dal modulo **math** alcune funzioni matematiche che ci torneranno utili, ad esempio **pi** che è il valore di π (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**.

Analizziamo la scrittura, cominciando da sinistra: troviamo **FreeCAD** seguito da un punto e da **newDocument**, con questa scrittura 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".

Per illustrare questo concetto analizziamo la dichiarazione del metodo di esempio **miaScatola**:

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

Questo metodo possiede 3 parametri:

1. **miaLargh** che stabilisce la larghezza della scatola
2. **miaLungh** che ne stabilisce la lunghezza
3. **miaAlt** che ne stabilisce l'altezza

Ognuno di questi parametri però viene definito facendolo seguire da un segno di uguale e da un valore, questo significa che se invochiamo il metodo con:

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

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

Come mai?

Perché la funzione definisce tutti e tre i parametri come facoltativi e assegna dei valori per per difetto o per usare la terminologia inglese di “default”.

Se non forniamo nessun parametro per i valori `miaLargh`, `miaLungh`, `miaAlt`, il metodo usa quelli predeterminati al momento della sua creazione, altrimenti utilizza quelli forniti nell’invocazione ad esempio, invocando il metodo con:

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

Otterremo 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.

Lo scopo della funzione `clear_doc` è quello di riutilizzare il documento che abbiamo creato in precedenza.

Alla prima esecuzione dello script, abbiamo creato un documento, se eseguiamo una seconda volta lo script, ed il documento è già presente, utilizzando questo sistema risparmiamo tempo quanto :

- Ad ogni esecuzione dello script non viene creato un nuovo documento, infatti se esiste un documento viene automaticamente riutilizzato.
- Stiamo creando uno script che compone un oggetto, e non vogliamo certo che gli oggetti creati vengano aggiunti agli oggetti già presenti. Per giunta verranno creati con lo stesso nome, dato che ad ogni lancio dello script ricreiamo gli oggetti da capo; questo porterà **FreeCAD** a rinominare i nuovi oggetti aggiungendo un suffisso numerico.

Questa funzione, apparentemente semplice, contiene invece un costrutto importante.

All’interno della funzione `removeObject()` troviamo il parametro `obj.Name` che non è qualcosa di statico; al contrario questo parametro rappresenta 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, come se stessimo parlando al computer dicendo:

- 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, avvertono Python che la dichiarazione è finita, (l’abbiamo già visto anche nelle dichiarazione di **if**), se mancano Python segnala un errore, provate a toglierli e vedrete cosa succede quando lancio il programma.

Nel linguaggio Python, non esiste solo il ciclo **for**, ma non potendo analizzare in dettaglio tutto il linguaggio Python, per ogni approfondimento rimandiamo il lettore 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 delle condizioni, ad esempio:

- **SE** succede questo **allora** fai quello.
- **SE** questa cosa esiste **allora** fai così **altrimenti** fai cosà.
- **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 esempio di 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**.

Molto importante 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**. Esso 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")**, questa scrittura ci evita di scrivere tre confronti diretti e ci permette di aggiungere in seguito altri valori, se necessario.

Queste scritture evidenziano la presenza di una sola condizione.

Se dobbiamo eseguire una certa azione se il risultato della condizione è **True** e una diversa azione se il risultato è **False**, dobbiamo usare una scrittura leggermente diversa, presentiamo la “forma completa” di una struttura **if .. then .. else**:

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

Nella figura fig. 2.1 nella pagina seguente 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 istruzioni del blocco associato; tutte le altre condizioni che seguono vengono semplicemente ignorate, indipendentemente dal fatto che siano **elif** o **else**.

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 opera su un'insieme di API (Application Program Interface) molto potente, come quelle di **FreeCAD**, abbiamo molti modi per ottenere un risultato.

Per la creazione degli oggetti 3D è stato scelto l'approccio di usare, per quanto possibile, il metodo **DOC.addObject()**, per alcuni motivi:

- La definizione diventa facile da ricordare
- Viene creato l'oggetto 3D voluto, direttamente nella **vista ad albero**, senza dover invocare il metodo **Part.show** come molto spesso viene fatto negli esempi sul sito e nel forum di **FreeCAD**.

Questo metodo, se usato in un progetto complesso che comporta la creazione di molti oggetti “intermedi” potrebbe presentare problemi di utilizzo efficiente della memoria.

Dal punto di vista didattico presenta una notevole uniformità di approccio e meglio aiuta a comprendere i meccanismi di funzionamento della modellazione.

Presenteremo comunque più avanti la tecnica che fa uso dei metodi **Part.make...**, che permette di usare in modo più efficiente le risorse.

Nella costruzione del programma utilizzeremo molto la tecnica dei blocchi logici:

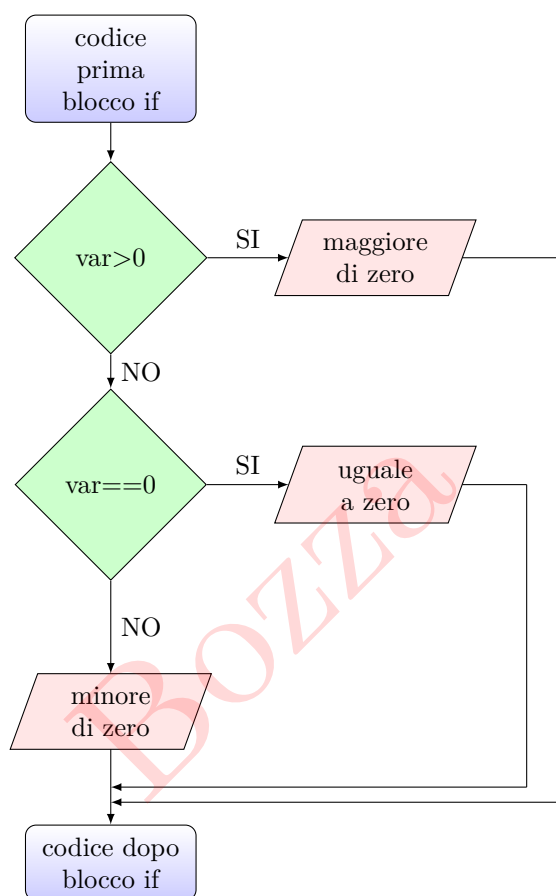


Figura 2.1.: schema a blocchi condizioni if

- Creeremo metodi, lunghi poche righe di codice, che creano forme base o che compiono operazioni semplici.
- Richiameremo questi metodi con una chiamata a quel metodo, in genere lunga una riga di codice.
- Non metteremo all'interno di un metodo solamente le chiamate necessarie alla creazione delle forme finali: ad esempio `crea_forma` che diventerà facile escludere dall'esecuzione usando un semplice `#` all'inizio della riga.

Questo modo di procedere permette di essere molto flessibili e di riutilizzare parti di codice, mediante semplici operazioni di Copia e Incolla.

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 scaricandolo dalla pagina GitHub) il codice presente nel listato presentato nella sezione sezione 2.4 con il titolo **Listato - schema base** e salvandolo.

Dopo alcune discussioni avute, e qualche errore scoperto, abbiamo preferito concepire questa guida utilizzando un codice base, a cui verranno aggiunte le porzioni di codice descritte nel testo, salvando le versioni intermedie con il nome che preferite, oppure con il nome suggerito all'interno del testo, in modo da poterle riprendere in mano e poi modificarle quando necessario.

Data la concezione modulare del codice, molte parti verranno riutilizzate per cui il Copia Incolla, diventerà un prezioso alleato per evitare di ribattere decine di linee di codice.

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 Vector, Rotation
14 import Part
15 from math import pi, sin, cos
16
```

```
17 DOC_NAME = "Pippo"
18 CMPN_NAME = None
19
20
21 def activate_doc():
22     """activate document"""
23     FreeCAD.setActiveDocument(DOC_NAME)
24     FreeCAD.ActiveDocument = FreeCAD.getDocument(DOC_NAME)
25     FreeCADGui.ActiveDocument = FreeCADGui.getDocument(
        ↪ DOC_NAME)
26     print("{0} activated".format(DOC_NAME))
27
28
29 def setview():
30     """Rearrange View"""
31     DOC.recompute()
32     VIEW.viewAxometric()
33     VIEW.setAxisCross(True)
34     VIEW.fitAll()
35
36
37 def clear_doc():
38     """Clear the active document deleting all the objects"""
39     for obj in DOC.Objects:
40         DOC.removeObject(obj.Name)
41
42 if FreeCAD.ActiveDocument is None:
43     FreeCAD.newDocument(DOC_NAME)
44     print("Document: {0} Created".format(DOC_NAME))
45
46 # test if there is an active document with a "proper" name
47 if FreeCAD.ActiveDocument.Name == DOC_NAME:
48     print("DOC_NAME exist")
49 else:
50     print("DOC_NAME is not active")
51     # test if there is a document with a "proper" name
52     try:
53         FreeCAD.getDocument(DOC_NAME)
54     except NameError:
55         print("No Document: {0}".format(DOC_NAME))
56         FreeCAD.newDocument(DOC_NAME)
57         print("Document Created".format(DOC_NAME))
58
59 DOC = FreeCAD.getDocument(DOC_NAME)
60 GUI = FreeCADGui.getDocument(DOC_NAME)
61 VIEW = GUI.ActiveView
```



```
62 #print("DOC : {0} GUI : {1}".format(DOC, GUI))
63 activate_doc()
64 #print(FreeCAD.ActiveDocument.Name)
65 clear_doc()
66
67 if CMPN_NAME is None:
68     pass
69 else:
70     DOC.addObject("App::DocumentObjectGroup", CMPN_NAME)
71
72 # EPS= tolerance to uset to cut the parts
73 EPS = 0.10
74 EPS_C = EPS * -0.5
75 VZOR = Vector(0, 0, 0)
76 ROTO = Rotation(0, 0, 0)
77
78 # IL CODICE COMINCIA DA QUI
79 # MAIN CODE START HERE
```

Capitolo 3

Modellazione di Solidi 3D

Lo scopo di un modellatore 3D è quello di creare **solidi** tridimensionali; per convenzione si abbrevia la parola tridimensionale con 3D (bidimensionale viene convenzionalmente abbreviato con 2D).

Nel resto della guida per abbreviare parleremo semplicemente di **solidi**, per due motivi: 1) un solido è per natura tridimensionale, aggiungere 3D è un per lo meno un "rinforzo" non necessario; 2) è più corto.

In matematica esistono spazi con più di tre dimensioni, noi ci limitiamo ad usarne tre.

Dalla quarta dimensione in poi gli spazi si chiamano "iperspazi" e diventano interessanti per i matematici e gli appassionati di fantascienza.

3.1. Spazio tridimensionale

Nella figura fig. 3.1 potete vedere una rappresentazione convenzionale dello spazio tridimensionale, ogni punto nello spazio 3D è definito da una tripla di coordinate (X, Y, Z).

In questa guida, se non spiegato diversamente, quando 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.

La convenzione legata agli assi è la stessa utilizzata da **FreeCAD**; la trovate rappresentata nella **vista 3D**.

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

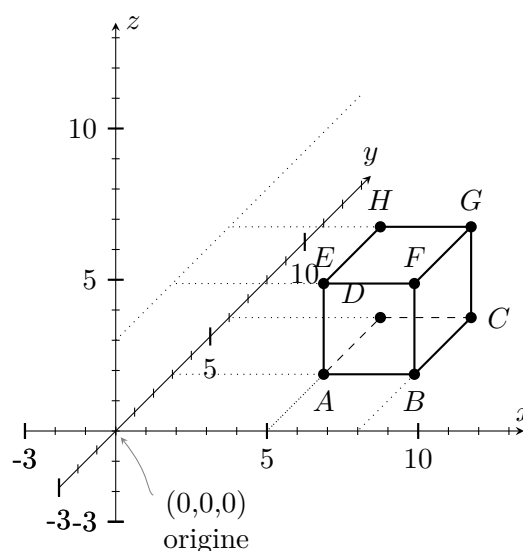


Figura 3.1.: Lo spazio 3D

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).

3.1.1. I Vettori

I **vettori** sono molto usati in **FreeCAD**.

Possiamo pensare ad un vettore come ad una lista che contiene le coordinate 3D di un punto nello spazio, espresse con valori in virgola mobile (*float*).

La scrittura più comune che incontriamo è questa **Vector(val_x, val_y, val_z)**.

È un concetto comune a tutti i modellatori 3D, un modo per passare le coordinate 3D di un punto in maniera compatta.

Un vettore può contenere anche valori diversi dalle coordinate di un punto, ad esempio gli angoli di rotazione per ogni asse e non necessariamente contiene 3 elementi.

Vengono usati i vettori al posto di altri contenitori come ad esempio una tupla, per un motivo ben preciso:

Il vettore è un elemento su cui si può operare con opportune operazioni ‘algebriche’.

Un secondo motivo risiede nella possibilità di fare riferimento ad una coordinata spaziale con un nome.

3.2. Visualizzazione

Per poter rappresentare uno spazio tridimensionale su di uno schermo (ma anche su un foglio di carta) diventa necessario usare le proiezioni.

FreeCAD permette di usare diversi tipi di proiezione, nel linguaggio della grafica 3D queste proiezioni sono chiamate *viste* o *scene*.

Il lavoro di effettuare queste proiezioni viene affidato ad un motore grafico, che partendo dal modello geometrico dell’oggetto, stabilisce un punto di ripresa come su un set cinematografico e visualizza il risultato sul monitor.

Il motore grafico compie delle trasformazioni geometriche effettuando complessi calcoli matematici.

Questo motore di visualizzazione, in **FreeCAD** è affidato alle librerie **Coin3D** che utilizzano lo standard di *OpenInventor*.

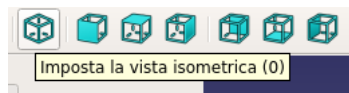


Figura 3.2.: BdS Viste

Possiamo accedere ad alcune di queste viste direttamente attraverso il menù **Visualizza**, ad esempio **Vista Ortografica**, oppure **Vista Prospettica**.

Altre viste sono accessibili come voci del menù **Visualizza** ⇒ **Viste Standard**: “dall’alto”, “dal basso”, “da destra”,

“da sinistra”, “di fronte” e “da dietro”.

Altre ancora sono accessibili attraverso il sotto menu **Visualizza** ⇒ **Viste Standard** ⇒ **Axonometric**.

Molte di queste viste sono accessibili anche tramite la **Barra degli strumenti Viste** dove visualizzate come un cubo con la faccia evidenziata che indica la posizione della vista richiesta. Tra queste icone trovate anche quella della vista **Isometrica** (figura 3.2).

3.2.1. Dati e Vista

FreeCAD era stato inizialmente creato per lavorare come applicazione a riga di comando, senza la sua attuale interfaccia utente. Successivamente si sono aggiunte le funzionalità dell'interfaccia grafica ed è stato scelto per motivi tecnici un motore grafico.

La conseguenza di questa scelta porta ad una importante dicotomia, ogni “oggetto” di **FreeCAD** possiede una componente “dati” e una componente “vista”.

Anche nel codice troveremo questa dicotomia, abbiamo infatti una componente **App** e una componente **Gui**

Ogni oggetto di **FreeCAD** possiede dunque due parti:

- Una parte **Object** contenente i dati dell'oggetto, che sono utilizzati dal motore di modellazione e che ritroviamo nella **Scheda Vista** dell'**editor delle proprietà**.
- Una parte **ViewObject** contenente i dati di visualizzazione, usati dal motore grafico, come ad esempio il colore e la trasparenza; questa parte si trova nella **Scheda Dati** dell'**editor delle proprietà**.

Nello **schema base** potete vedere un utilizzo della parte di visualizzazione nel metodo **setview()**, attraverso alcune funzioni di visualizzazione accessibili attraverso **FreeCAD.Gui**.

Nella **vista 3D** a volte potete notare che alcuni oggetti curvi sembrano essere rappresentati con poligoni, dando l'impressione di aver realizzato un oggetto poligonale al posto di uno circolare.

Questo comportamento è legato alla componente “Vista”, che per velocizzare il *rendering* grafico, utilizza dei valori che stabiliscono una tolleranza di visualizzazione.

Il motore di modellazione, basato su OpenCascade, internamente usa un cerchio definito come curva, infinitamente scalabile e quindi implicitamente preciso.

Possiamo regolare, utilizzando appositi parametri, la precisione di visualizzazione del motore grafico Coin3D:

- A livello generale, attraverso i valori situati in: menu **Modifica** ⇒ **Preferenze**, alla sezione **Part design** nella scheda **Visualizzazione della figura**, nel riquadro **Tassellazione**.

- A livello di ogni singolo oggetto, nella **Scheda Vista**.

I parametri che regolano questa precisione sono:

- La **deviazione massima dal modello** espressa come valore percentuale, è accessibile:
 - A livello generale, attraverso il valore chiamato **Deviazione massima secondo il riquadro di delimitazione del modello**.
 - A livello di singolo oggetto, attraverso la proprietà **Deviation**, il valore è espresso in percentuale.
- La **deviazione angolare** espressa in gradi (di default è impostata a 28.5 ° cioè 0.5 radianti) è accessibile attraverso:
 - A livello generale usando il valore chiamato **Massima deflessione angolare**
 - A livello di singolo oggetto, attraverso la proprietà **Angular Deflection**.

Alcuni suggeriscono di modificare **Deviation** attorno al 0.100 % e il valore **Angular Deflection** fino a 5°.

3.3. Geometria, Topologia, Solidi e altro

FreeCAD possiede una serie abbastanza completa di solidi di base, partendo da questi oggetti semplici possiamo costruirne altri molto più complessi.

Abitudini tecniche differenti portano a differenti concezioni della stessa parola, pur usando lo stesso termine un architetto, un ingegnere e un matematico danno un significato diverso ad esempio alla parola *spigolo*.

3.3.1. Geometria

La geometria, è la componente più a basso livello del motore di modellazione.

Permette di definire dei concetti di base:

- **Punti** una coordinata nello spazio 2D o 3D, non possiede dimensioni.
- **Curve** ad esempio un **cerchio**, che possiede una precisa rappresentazione matematica e di cui ad esempio fornendo due soli dati: un **centro** ed un **raggio**, possiamo definire tutti i punti che lo compongono. Analogamente se vogliamo definire una **linea** o meglio un **segmento**, ci basta definire i suoi estremi, cioè il **punto** di inizio e quello di fine.
- **Superfici** ad esempio un **piano**, ma anche **superfici** più complessa come una **Superfici** BSpline.

Le **curve** possono essere anche di altro tipo, ad esempio la parabola, l'iperbole, ma anche concetti più complessi come le BSpline, anche quelle hanno una precisa formula

matematica che possiamo usare per calcolarne ogni punto, partendo da una serie di dati di base.

Alcune di queste rappresentazioni hanno un senso nello spazio a due dimensioni (2D) altre hanno un senso solo in uno spazio a tre dimensioni (3D).

3.3.2. Topologia

(Topologia): branca della geometria che studia le proprietà delle figure.

In sintesi la topologia è il modo in cui gli elementi costitutivi dei **solidi** vengono descritti nello spazio 3D reale.

La topologia è molto importante, in quanto quasi tutto in **FreeCAD** è basato sulla topologia.

Alcuni di questi concetti, usano delle parole, usate quotidianamente nel linguaggio comune, ma che assumono una definizione precisa dal punto di vista topologico:

Possiamo affermare che la **Topologia** si trova ad un livello superiore rispetto alla **Geometria** che descrive le **Curve** e le **superfici**, possiamo individuare 3 "componenti base" di ogni oggetto che creeremo con **FreeCAD**:

- **Vertice (Vertex)**: un elemento topologico corrispondente ad un **punto**.
- **Bordo (Edge)** altrimenti detto **spigolo**: un elemento topologico corrispondente ad una curva limitata. Un **bordo** è generalmente limitato da **vertici**.
- **Faccia (Face)**: una parte di **superfici**. La sua geometria è vincolata (delimitata/-tagliata) dai suoi **bordi**; Possiede anche un significato nello spazio bidimensionale se viene considerata come una parte di piano;

Queste tre **Forme topologiche** le possiamo concepire come gli elementi costitutivi di tutte le geometrie sia in 2D che in 3D.

Gli elementi topologici visti sopra li possiamo considerare come i mattoncini che ci permettono di costruire altre cose più complesse:

- **Contorno (Wire)**: una serie di **bordi** collegati tra di loro nei **vertici**. Viene definito aperto se i **bordi** non sono interamente concatenati (se il punto di partenza e quello finale non coincidono). Viene definito chiuso, se i **bordi** sono interamente concatenati (il punto di partenza e quello finale coincidono).
- **Guscio (Shell)**: una serie di **facce** connesse nei loro **bordi**. Un guscio (shell) può essere aperto o chiuso.
- **Solido**: una parte di spazio limitato da un **guscio**.

Traduciamo il termine **wire** con **contorno**. Alcune traduzioni usano il termine "polilinea" che però ci sembra fuorviante, in quanto contiene la parola linee, mentre un contorno può essere formato anche da segmenti di *curve*.

Il concetto più vicino a quello di wire, usato in topologia generale, è quello di “**Boundary**” chiamato sia **contorno** che **confine**; a volte viene usato anche il termine **frontiera**, dovendo scegliere, si è deciso di usare **contorno**, che ci sembra più evocativo.

Parlando di **Solidi** è importante fare una distinzione:

Solido reale In cui ogni singolo **bordo** ha sempre due e solo due **facce** che si uniscono per creare il solido. Un solido reale è anche chiamato **solido manifold**. Ogni suo bordo è come in figura fig. 3.4a a pagina 36.

Solido aperti In cui la condizione sopra riportata non è soddisfatta per cui nel solido esistono bordi aperti, in altre parole alcune facce non sono tra loro connesse. I solidi aperti sono una sottocategoria dei cosiddetti **solidi non manifold**.

La condizione di **non manifold** si verifica quando non viene soddisfatta la condizione che in ogni bordo uniscono solo due facce, vedi la figura fig. 3.4b a pagina 36, che verrà spiegata in dettaglio più avanti.

L’errore “**non manifold**” è il tipico errore che si incontra nella modellazione 3D, quando si uniscono **facce** per creare un **guscio**.

Se creiamo un **guscio** aperto e lo trasformiamo in un modello per la stampa 3D, il programma che si occupa di prepararlo per la stampa potrebbe evidenziare un errore ed in genere questo errore è proprio quello di “**non manifold**”.

3.3.3. Maglie

Uno dei concetti chiave della modellazione è il concetto di **maglia** (**mesh**), usato sia nella modellazione sia nel “rendering” grafico, cioè la tecnica che permette all’elaboratore di creare immagini tridimensionali su uno schermo bidimensionale.

Usando **FreeCAD** abbiamo a che fare con il rendering grafico, ogni oggetto che creiamo e che poi guardiamo sullo schermo passa attraverso un processo di rendering.

Concetti analoghi li ritroveremo nella modellazione BREP, che introdurremo tra poco.

Possiamo approssimare un solido, definendo una serie di punti sulla sua superficie e unendoli, senza dover per forza definire tutti i punti intermedi.

Il concetto chiave di questo approccio è quello della **tassellazione** (**tessellation**).

Un **Tassello**, detto anche **Faccetta** (**Facet**), nella sua forma più semplice è un triangolo e lo possiamo sovrapporre quasi esattamente al concetto di faccia.

Convenzionalmente l’ordine dei vertici indica l’*orientamento* della faccetta:

- Il senso antiorario indica che la faccetta possiede un orientamento verso l’esterno del modello. Si definisce **normale**. Vedi figura fig. 3.3a nella pagina successiva
- Il senso orario indica che la faccetta possiede un orientamento verso l’interno del modello. Si definisce **invertita**. Vedi figura fig. 3.3a nella pagina seguente

La superficie della faccetta può assumere diverse configurazioni:

- **piatta**: semplicemente una parte di piano, contenuta nel poligono formato dai suoi vertici, tipica delle maglie.
- **curva**: la sua superficie è definita da curve matematiche. Gli oggetti BREP di **FreeCAD** hanno le faccia definite per mezzo di curve; questo le rende scalabili in modo illimitato, senza perdere di precisione.

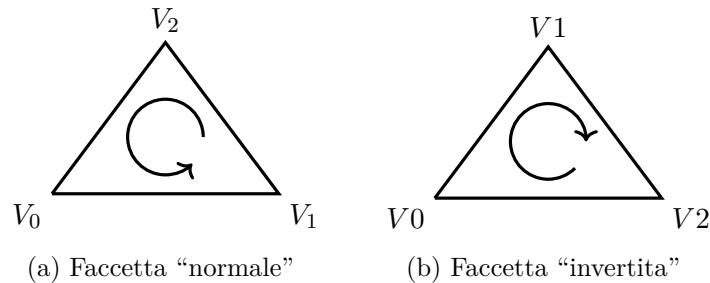
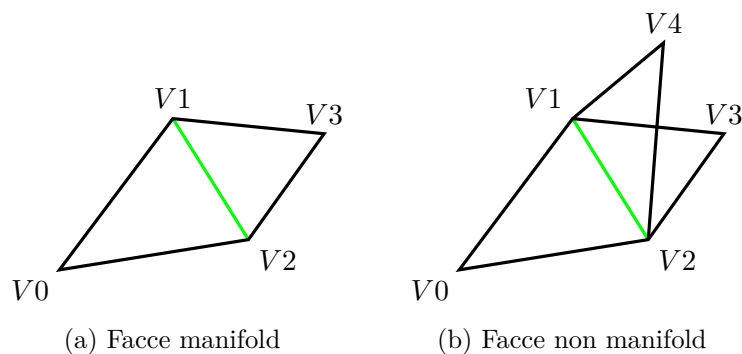
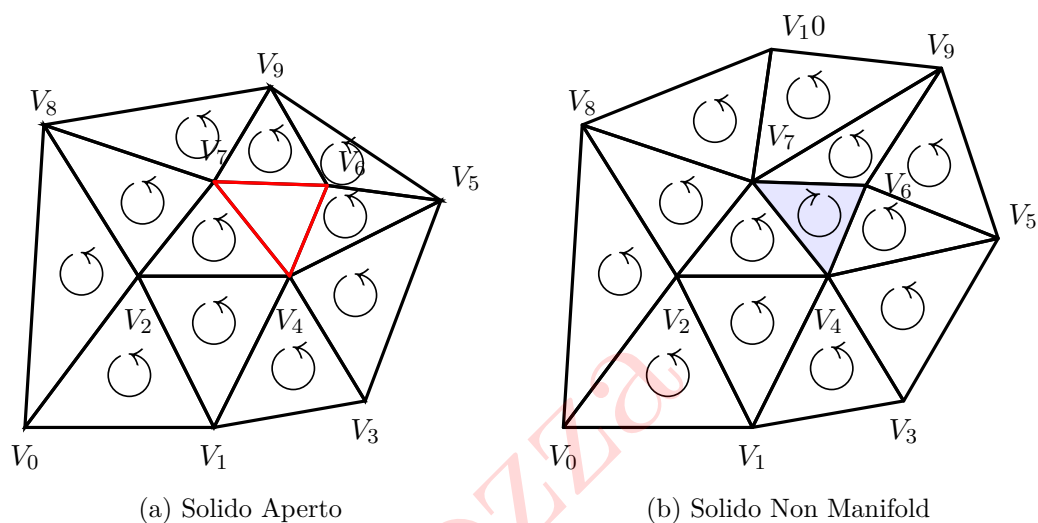


Figura 3.3.: Faccetta di una maglia.

I concetti di manifold o non manifold, sono presenti anche nelle maglie; potete facilmente trovare a volte il termine di *watertight*, cioè a “tenuta d’acqua” per indicare la condizione di **manifold**.

A volte più genericamente si usano i termini di maglia *aperta* o maglia *chiusa*, riassumiamo i vari concetti:

- *manifold* e *chiuso* sono concetti equivalenti, un esempio viene mostrato nella figura fig. 3.4a nella pagina successiva, la riga verde tra i vettori V_1 e V_2 indica un bordo in cui si incontrano due (e solo due) faccette; questo vale per tutti i bordo di tutti le facce, che compongono il guscio del solido.
- *aperto* semplicemente vuol dire che esistono dei “buchi” nella maglia, cioè delle facce non collegate ad altre facce, potete vedere una rappresentazione nella figura fig. 3.5a nella pagina seguente; le linee rosse indicano che questi bordi, non sono connessi a nessuna faccia; infatti per chiudere il solido mancherebbe la faccia definita dai vettori (V_4 , V_6 , V_7).
- *non manifold* può significare due cose:
 - Un significato viene illustrato nella figura fig. 3.4b nella pagina successiva; la linea verde tra i vertici V_1 e V_2 indica un bordo in cui si incontrano tre facce, violando la convenzione di *manifold*.
 - Un secondo significato viene invece raffigurato in figura fig. 3.5b nella pagina seguente; la faccia blue chiaro indica una faccia “invertita” notare che il simbolo dell’ordine dei vertici indica il senso orario, contrario a quello di tutte le altre facce.

Figura 3.4.: Facce *manifold* e *non manifold*Figura 3.5.: Modelli *manifold* e *non manifold*

3.3.4. Tassellazione

L'operazione, che trasforma o per meglio dire approssima un solido in una maglia, si chiama tassellazione.

Questa operazione prende la superficie del solido e la approssima calcolando le coordinate dei vertici di ogni tassello per ottenere maglie che è costituita da tasselli triangolari o poligonal, a seconda del formato desiderato.

Questa operazione in genere considera le facce come planari, in **FreeCAD** il solido viene definito con la superficie delle facce definite da curve.

Approssimare una curva usando una linea, lo potete vedere semplicemente disegnando su carta, porta ad una perdita di precisione. Per ridurre questa perdita dobbiamo moltiplicare il numero di linee che approssimano una curva.

Per effettuare la conversione si rende necessario stabilire una misura della “perdita” di precisione causata dall'esportazione. Vedremo nel capitolo dedicato alla trasformazione

dei solidi creati con **FreeCAD** in una Mesh per la stampa 3D che dovremo inserire alcuni parametri per stabilire la precisione del modello creato.

La stessa cosa vale, al contrario, quando importiamo un modello in **FreeCAD** ed il formato del file usa una **tassellazione** con faccette planari, la qualità del modello importato risulta inferiore a quella interna di **FreeCAD** e dovremo usare delle funzioni per migliorare il modello.

3.4. Modellazione 3D

Il processo di modellazione, può essere basato su diversi tipi di concetti di base, che generano diverse tecniche di costruzione, i due concetti fondamentali sono:

- CSG (**C**o**s**tructive **S**olid **G**eometry) ovvero **geometria solida costruttiva**.
- BREP (**B**oundary **r**epresentation) abbreviato in B-REP o BREP.

Molti considerano l'approccio BREP come un approccio "moderno" mentre la CSG viene considerata "vecchio stile".

Entrambi gli approcci hanno vantaggi e svantaggi, fortunatamente **FreeCAD** ci permette di usarli entrambi.

Questa libertà di scelta impone però di conoscerli entrambi, e decidere, se e quando usarli per ottenere il risultato voluto.

3.4.1. Geometria e topologia in FreeCAD

Una cosa importante da tenere presente quando usiamo lo scripting, è la distinzione tra:

- **Primitive geometriche**
- **Forma topologica**
- **Oggetto documento**

Questi concetti vengono "nascosti" nell'utilizzo attraverso la GUI, perché ogni oggetto che creiamo in genere finisce nella vista 3D, diventa quindi un **Oggetto documento**.

Non entreremo troppo in dettaglio nella spiegazione, perché andremmo molto oltre gli scopi di questa guida, ma una breve spiegazione è d'obbligo.

Nella terminologia di OpenCascade, distinguiamo tra "primitive geometriche" ("geometric primitives") e "forma topologica".

Elenchiamo le primitive geometriche più comuni disponibili in **FreeCAD** nella tabella 3.1 nella pagina successiva.

Una forma topologica può essere un **vertice**, un **bordo**, un **contorno**, una **faccia**, un **solido** oppure un "composto" di altre **forme topologiche**.

Tipo Geometria	Sottotipo	
punto		
curva		
	linea	
	cerchio	ellisse
	parabola	iperbole
	curva Bezier	curva B-Spline
superficie		
	superficie B-Spline	superficie Bezier
	piano	

Tabella 3.1.: Geometrie

Le primitive geometriche non sono create per essere visualizzate direttamente, ma piuttosto per essere usate per costruire le **forme topologiche**. Ad esempio un **bordo** può essere costruito a partire da una **linea** oppure da una porzione di cerchio.

Questo concetto a volte porta a molti problemi di comprensione ed a considerare **FreeCAD** come "inutilmente" complesso.

Dal punto di vista della modellazione, quando definiamo un cerchio fornendo un **centro** e un **raggio**, diciamo a **FreeCAD** di calcolare le coordinate dei punti che costituiscono il cerchio, volendo usare una terminologia tecnica, definiamo il "luogo dei punti della circonferenza".

Analogamente per una linea non serve memorizzare tutte le posizioni dei punti che la compongono, basta sapere che la linea dal "punto A" al "punto B".

Se vogliamo ad esempio conoscere dove la "linea A" interseca il "cerchio B" ci sarà una funzione che confronta tutti i "luoghi dei punti" delle due **curve** e restituisce i punti di intersezione.

Per illustrare meglio la cosa usiamo Python e la funzione print:

```
circle = Part.Circle()

print(circle.TypeId)

circle.Center = Vector(0, 0, 0)
circle.Radius = 10

cir1 = circle.toShape()

print(cir1.TypeId)

Part.show(cir1)
```

Vedremo nella **finestra dei rapporti** stampate queste cose:

```
Part::GeomCircle
Part::TopoShape
```

Siamo stati abbastanza prolissi nella scrittura del codice, proprio per evidenziare la sequenza dei comandi:

- creato la "primitiva geometrica" **circle** vuota (è perfettamente legittimo farlo)
- stampato il suo **TypeId** che vale **Part::GeomCircle**.
- assegnato i valori a **Center** e **Radius**.
- "trasformato" la "primitiva geometrica" in una **forma topologica** usando il metodo **.toShape()**.
- stampato il suo **TypeId** che in questo caso è diventato **Part::TopoShape**.
- "visualizzato" la **forma topologica** usando il metodo **.show()**, questo metodo crea un **Oggetto documento**.

Un **Oggetto documento** è qualcosa che viene visualizzato nella vista 3D e finisce anche nel file **.FCStd** quando salviamo il documento.

In questa guida faremo spesso uso di **Oggetti documento** derivati da **Part::Feature** che avranno una propria **forma topologica**, che normalmente sarà accessibile attraverso la proprietà **Shape**.

Quando si costruiscono cose complesse, a volte è meglio generare un **Oggetto documento** derivato da **Part::Feature** solo alla fine del processo di creazione, per non utilizzare troppe risorse, infatti se non vengono trasformate in **oggetti documento** le **forme topologiche** vengono "buttate via" e la memoria viene liberata.

Ogni **oggetto documento** invece occupa una certa quantità di memoria, e spazio nel file **.FCStd**, per cui molti oggetti occupano molta memoria e se sono degli "oggetti intermedi" o "componenti per la costruzione" non ha senso tenerli a disposizione, avere molti **oggetto documento** rallenta anche l'apertura del file **.FCStd**.

Per motivi "didattici" la prima parte di questa guida utilizza il metodo "meno efficiente" di creare **oggetto documento** e di renderli visibili ed ispezionabili nella vista 3D.

Riteniamo che per capire bene il funzionamento del programma e i concetti alla base di questa guida e della modellazione in **FreeCAD** sia utile avere tutti gli oggetti accessibili e visualizzabili nella vista 3D, nel corso della modellazione, ogni passaggio risulterà ispezionabile e visualizzabile, per cui in caso di sperimentazioni personali sarà decisamente più facile trovare il punto critico dove il risultato non è quello voluto.

La tecnica BREP è meno adatta a questo metodo di costruzione, per cui si farà ampio uso delle **forme topologiche** e solo alla fine si creerà un **oggetto documento**, descriveremo diffusamente questo modo più efficiente di creazione nel capitolo capitolo 7 a pagina 62 dedicato alla modellazione BREP.

3.4.2. Geometria solida costruttiva CSG

La CSG permette di costruire oggetti geometrici complessi partendo da un insieme ristretto di geometrie di base le cosiddette **primitive** e operando su di esse utilizzando delle **operazioni booleane**.

Le operazioni booleane sono:

- **Unione** chiamata anche **Fusione**: restituisce un **solido** ottenuto unendo (fondendo) i **solidi**. Vedi figure 3.6a e fig. 3.6b nella pagina successiva.
- **Sottrazione** chiamata anche **Taglio**: restituisce un **solido** ottenuto sottraendo (tagliando) da un **solido** un secondo **solido**. Vedi figure 3.6c e fig. 3.6d nella pagina seguente.
- **Intersezione**: restituisce la parte comune dei due **solidi**. Vedi figure 3.6e e 3.6f.

Chi ha studiato gli insiemi, troverà delle analogie, almeno nei nomi delle operazioni e non è un caso.

3.4.3. Boundary representation BREP

Nella **Boundary representation** BREP i modelli sono concettualmente composti da due parti:

Topologia: Cioè il modo in cui sono connessi: **facce**; **bordi**; **vertici**.

Geometria: Cioè il posizionamento delle superfici nello spazio (3D); si occupa di: **superfici**; **curve**; **punti**.

Possiamo definire la BREP come un insieme di regole che legano la Geometria alla Topologia, esistono delle relazioni tra i vari elementi, elenchiamone alcune:

- Una **faccia** è una parte limitata di una **superficie**.
- Un **bordo** è una parte limitata di una **curva**.
- Un **vertice** giace su un **punto**.

Un piccolo esempio utile è quello di definire un cubo utilizzando questo approccio; Un cubo è costituito da sei **facce**, unite per i loro **bordi**.

Ogni **faccia** può essere concepita anche come una **superficie** limitata dai **bordi**, ma un **bordo** è una **curva** che unisce due **vertici**.

In modo simile possiamo definire i **bordi** di un **solido** anche complesso e poi creare delle **superfici** che uniscono questi **bordi**.

Quando lo abbiamo fatto per tutti i **bordi** del **solido** otteniamo un **solido** reale.

I confini esterni di questo **solido** costituiscono il suo **guscio**

Usando questo modo di costruzione e operando attraverso l'uso di opportuni strumenti, la tecnica BREP permette di creare **solidi**.

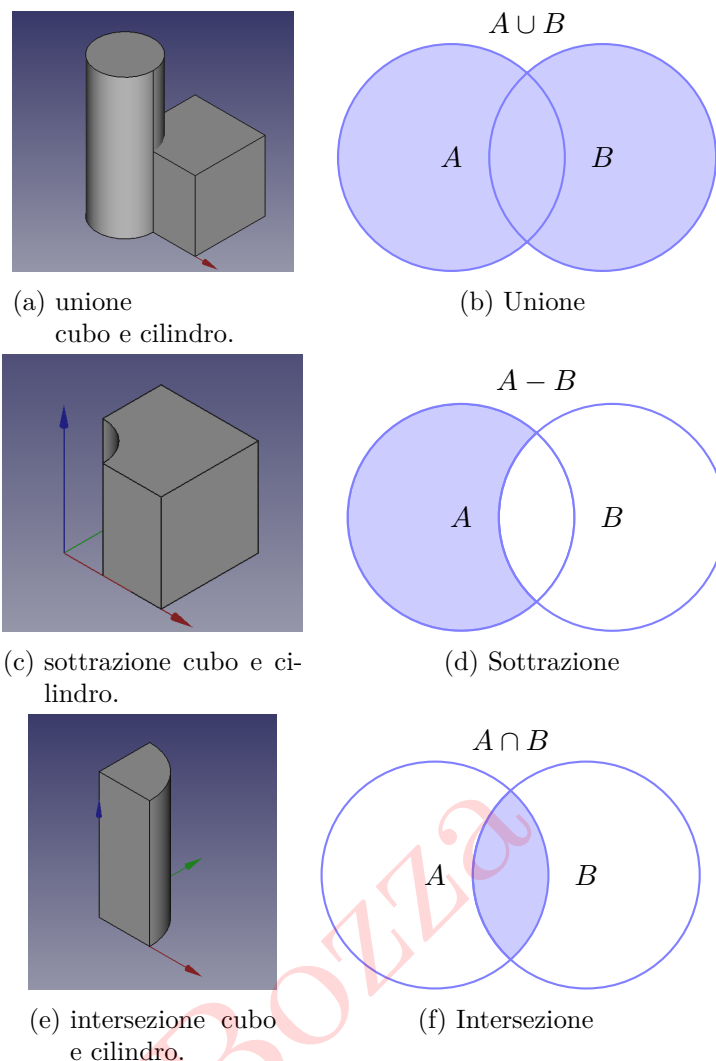


Figura 3.6.: Operazioni Booleane.

Il rischio che si corre usando questo metodo è quello di non definire correttamente tutte le superfici e di ottenere un solido aperto.

Per costruire oggetti secondo la tecnica BREP si parte da una serie di contorni, si uniscono questi contorni per creare delle superfici che definiscono il guscio.

Questi strumenti possono essere semplici come quello che permette di unire due bordi attraverso una superficie, oppure strumenti molto più complessi che “trasformano” una serie di contorni in un guscio.

Capitolo 4

Modellazione CSG

Il concetto base per costruire un **solido** complesso, assomiglia al vecchio indovinello “come fa una formica a mangiare un elefante?”.

La risposta è ovvia: “a piccoli pezzi”.

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”.

4.1. I primi oggetti

Introduciamo ora il primo oggetto, o meglio la prima **primitiva**, un parallelepipedo, per **FreeCAD** si tratta di un oggetto di tipo **Part::Box**;

Per costruire il nostro **solido**, creiamo 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 parallelepipedo, (il cubo è un caso particolare di 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

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

setview()
```

Eseguendo il programma usando la freccia verde, otterremo un cubo visualizzato nella **vista 3D**.

Ora complichiamo la faccenda. Il cubo è costruito, 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, basta guardare le frecce che indicano la direzione degli assi, che sono state attivate dall'istruzione presente nel metodo **setview()**:

```
VIEW.setAxisCross(True)
```

Questo comando visualizza l'origine e le direzioni positive dei tre assi X, Y e Z, indicate con frecce colorate.

La stessa cosa viene fatta dalla voce di menu **Visualizza ⇒ Origine degli assi**,

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

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

Cercate di capire a cosa corrispondono i valori **lung**, **larg** e **alt**, e che relazione hanno con gli assi **X**, **Y** e **Z**.

Per poter proseguire nella trattazione, abbiamo necessità di avere almeno due **primitive**; creiamo dunque la seconda **primitive**, questa volta useremo un cilindro; Volendo esprimerci nel gergo di **FreeCAD** un oggetto di tipo **Part::Cylinder**.

Inseriamo quindi subito dopo al metodo **cubo** le righe riportate nel listato elenco 4.1 nella pagina successiva denominato **metodo cilindro**.

Il metodo creato **base_cyl** restituisce il nostro **solido**;

Invochiamolo inserendo gli appropriati parametri, ad esempio:

```
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.

Programma 4.1: metodo cilindro

```
def base_cyl(nome, ang, rad, alt ):
    obj = DOC.addObject("Part::Cylinder", nome)
    obj.Angle = ang
    obj.Radius = rad
    obj.Height = alt

    DOC.recompute()

    return obj
```

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 sezione 4.1.1, che una volta lanciato dovrebbe mostrare nella **vista 3D**, qualcosa che assomiglia alla figura fig. 6.1a a pagina 58.

Ovviamente manca tutta la “parte comune” presente nel file **sc-base.py**, questo vale anche per molti listati di questa guida se non diversamente indicato.

4.1.1. 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 Attenzione: Questo listato va usato aggiungendo le linee
12 da 18 in poi al codice presente in sc-base.py
13
14 Warning: This code has to be adding the lines starting
15 from 18 to the code in sc-base.py
16 """
17
18 def cubo(nome, lung, larg, alt):
19     obj_b = DOC.addObject("Part::Box", nome)
```

```
20     obj_b.Length = lung
21     obj_b.Width = larg
22     obj_b.Height = alt
23
24     DOC.recompute()
25
26     return obj_b
27
28 def base_cyl(nome, ang, rad, alt ):
29     obj = DOC.addObject("Part::Cylinder", nome)
30     obj.Angle = ang
31     obj.Radius = rad
32     obj.Height = alt
33
34     DOC.recompute()
35
36     return obj
37
38
39 obj = cubo("cubo_di_prova", 5, 5, 5)
40 obj_1 = base_cyl("primo cilindro", 360, 2, 25)
41
42 setview()
```

Capitolo 5

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 il **vertice** sinistro, frontale, basso 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 **primitive** sono costruite prendendo come punto di partenza un punto preciso che purtroppo da quanto visto finora non è il medesimo per ogni **primitiva**, la cosa diventa complicata quando vogliamo costruire **solidi** complessi.

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

Per complicare meno il codice proposto abbiamo usato un trucchetto:

Nella definizione degli import dello **schema base** trovate questa riga:

```
from FreeCAD import Vector, Rotation
```

Come abbiamo accennato quando abbiamo parlato delle direttive **import** questo ci permette di scrivere:

- **Vector(...)** al posto di **FreeCAD.Vector(...)**
- **Rotation(...)** al posto di **FreeCAD.Rotation(...)**

Dato che poi anche scrivere **Vector(0,0,0)** e **Rotation(0, 0, 0)** diventa lungo e noioso, abbiamo aggiunto due “costanti”:

```
VZOR = Vector(0, 0, 0)
ROTO = Rotation(0, 0, 0)
```

Questo ci permette di abbreviare le scritture rispettivamente del vettore di coordinate dell'origine e della rotazione nulla, che quando dobbiamo specificare un posizionamento dobbiamo per forza avere, ma non sempre vanno riempite con dei valori.

Rotazione e Posizionamento sono forse gli argomenti più “complicate” di **FreeCAD**, infatti molti post dei vari forum relativi a **FreeCAD** sono fatti proprio perché gli utenti non “capiscono” il modo di specificare questi valori.

Non ci viene in aiuto il fatto che esistano una varietà di modi per specificare una rotazione, ad esempio il kernel OCCT supporta 24 modi per specificare una rotazione secondo gli angoli di Tait-Bryant, che è uno solo dei modi possibili per specificare una rotazione tridimensionale.

Cercheremo per quanto possibile di semplificare la questione.

5.1. Il riferimento di costruzione

Una delle particolarità di **FreeCAD** che all'inizio può destare qualche perplessità è il riferimento in base al quale vengono costruite la **primitive**.

Nella tabella tabella 5.1, elenchiamo i punti di riferimento delle varie **primitive**.

Geometria	Punto di riferimento
Part::Box	vertice sinistro (minimo x), frontale (minimo y), in basso (minimo z)
Part::Sphere	centro della sfera (centro del suo contenitore cubico)
Part::Cylinder	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	vertice di Xmin Zmin

Tabella 5.1.: punti di riferimento

Concettualmente parlando, il miglior punto di riferimento è avere l'origine **(0,0,0)** nel “baricentro” del **solido** che andiamo a costruire.

Per fortuna alcune **primitive** hanno già il loro riferimento di costruzione in quel punto (ad esempio la sfera ed il toro); Altre **primitive** hanno l'origine in un punto “ragionevole”, come il centro della **faccia** di base (ad esempio il cilindro o il cono), e diventa facile spostare il centro nel “baricentro” semplicemente modificando la posizione Z.

Per altre **primitive** come il cubo (parallelepipedo o cuboide che dir si voglia) che hanno il punto di riferimento in una posizione “particolare” possono esistere (come molte cose in **FreeCAD**) diverse soluzioni:

- Tenerne semplicemente conto quando si costruisce il modello e soprattutto quando si modifica il posizionamento (Descriveremo formalmente il concetto di posizionamento tra poco).
- Costruire la **primitiva** e modificare il suo posizionamento già in fase di creazione.

Facciamo un esempio.

Utilizziamo il listato sezione 4.1.1 a pagina 44 denominato **Listato - figure-base.py** e modifichiamo il metodo **cubo** nel modo seguente:

```
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)

    rot_c = VZOR # Rotation center
    rot = ROTO # Rotation angles
    obj_b.Placement = FreeCAD.Placement( posiz, rot, rot_c)

    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**, e un secondo parametro opzionale chiamato **off_z**, che descriveremo poco oltre.

L’inserimento di parametri opzionali ci permette di riutilizzare tutto il codice precedente, solo quando servono “prestazioni particolari” possiamo aggiungere i parametri rilevanti alla chiamata del metodo.

Ad esempio se quando serve centrare il cubo sull’origine basta aggiungere alla chiamata un **True** dopo la definizione delle dimensioni.

Invocando il metodo nel modo classico con:

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

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

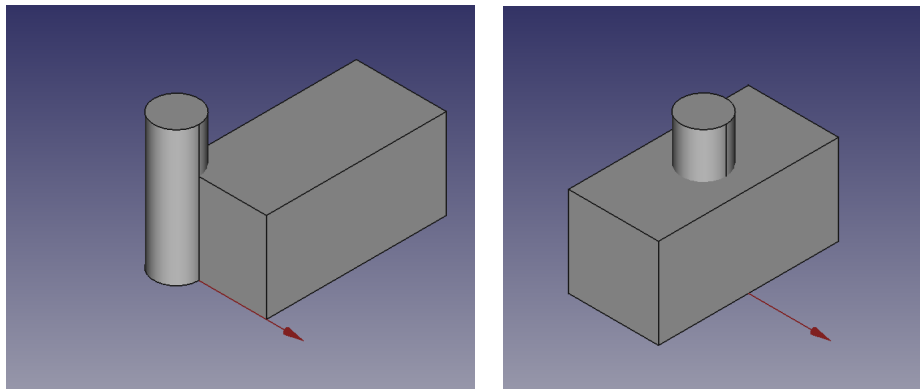
otteniamo il cubo e il cilindro come possiamo vedere in figura 5.1a.

Mentre invocando il metodo in questo modo:

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

obj2 = base_cyl("clindro", 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 5.1b.



(a) Cubo e cilindro con rdc originali.

(b) Cubo e cilindro con posizionamento modificato.

Figura 5.1.: Posizionamento

5.2. Posizionamento

Il parametro **off_z** che abbiamo aggiunto, serve per modificare la posizione Z della **primitiva**.

in questo modo possiamo alzare o abbassare il nostro cubo di quanto vogliamo, ad esempio modifichiamo 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 assegnato al parametro **off_z** il valore di **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.

5.2.1. 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:

```
rot_c = VZOR # Rotation center
rot = ROTO # Rotation angles
obj_b.Placement = FreeCAD.Placement( posiz, rot, rot_c)

DOC.recompute()
```

Questo è uno dei tanti metodi per specificarla, abbiamo usato il trucco descritto all’inizio del capitolo per accorciare la definizione, anche se qui poi abbiamo assegnato quelle costanti ad alcune variabili, più che altro per evidenziare il significato dei valori inseriti.

Una delle “scritture” più comuni è la seguente:

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

La trovate abbastanza spesso in giro, nella documentazione di **FreeCAD**, in quanto riflette le proprietà che trovate nella **vista combinata** nella parte **editor delle proprietà** come potete vedere nell’immagine fig. 5.2 che visualizza la **Scheda Vista**.

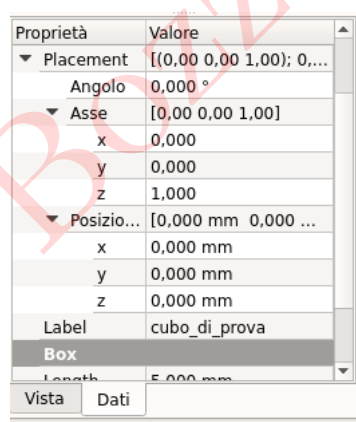


Figura 5.2.: L’editor delle proprietà

L’ordine delle voci nella **Scheda 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 fig. 5.2 e mostra 3 sotto proprietà, **Angolo**, **Asse**, **Posizione**, che elenchiamo nella tabella tabella 5.2 nella pagina seguente.

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

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

Tabella 5.2.: proprietà Placement

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

Possiamo facilmente riconoscere un componente chiamato **Pos**, che in questo contesto è il punto di **origine locale** seguito da una **tupla** di tre valori chiamata **Yaw-Pitch-Roll**.

Questo tipo di posizionamento usa gli **angoli 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	Psi ψ
Pitch (beccheggio)	rotazione rispetto a Y (alzare o abbassare il muso)	Phi φ
Roll (rollio)	rotazione rispetto a X (dondolare le ali)	Theta θ

Tabella 5.3.: Angoli di Eulero

Il modo più comodo per manipolare la rotazione di un solido è quello di usare (ed immaginare) la rotazione attorno al centro geometrico del solido, 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),
    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 tabella 5.4 nella pagina successiva

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

Tabella 5.4.: Rotazione usando gli Angoli di Eulero

La posizione che viene restituita dall'istruzione **print()**, non è nello stesso formato con cui stiamo specificando il posizionamento.

La differenza è piccola, ma importante, in questa spiegazione abbiamo scelto di specificare il posizionamento mediante le tre componenti elencate sopra.

Quanto restituito dall'istruzione **print()** è espresso in termini di **origine locale dell'oggetto** e **Rotazione**, specificata usando gli **angoli di Tait-Bryan**.

Nell'interfaccia grafica, abbiamo un modo per poter accedere ai diversi modi di descrivere un posizionamento rotazione è usare quando si è selezionato un oggetto nella **vista ad albero** il comando di menù **Modifica** ⇒ **Posizionamento**.

Il **posizionamento** e la rotazione sono argomenti relativamente complicati, probabilmente tra i più ostici da comprendere nella modellazione 3D.

Esistono molti modi di specificare un **Posizionamento**, ognuno legato ai diversi modi di specificare un sistema di coordinate, ad esempio globali o locali, il modo più completo, ma decisamente complicato da capire, se non avete delle buone basi matematiche è la **Matrice di Trasformazione**, che non tratteremo, ora ma che verrà trattato in (TODO: aggiungere riferimento).

Non deve perciò sorprendere che un modo scrittura della proprietà, ad esempio il modo che abbiamo usato, non corrisponda a quanto si ottiene quando chiediamo ad esempio di stampare le coordinate (che elenca solo due componenti).

Questa è una delle particolarità di **FreeCAD**, non deve essere considerato un errore, ma chiaramente il frutto della natura collaborativa dello sviluppo di **FreeCAD**, che è stato scritto "a più mani", ed evidentemente sono stati scelti modi diversi per parti diverse del programma.

Questo non è un indicatore di cattiva bontà del software, ma anzi della notevole versatilità di **FreeCAD**, che per molti aspetti è considerato superiore a molti software commerciali.

Per inciso durante la modellazione risulta molto utile, inserire delle istruzioni di **print()** all'interno del codice, per visualizzare ad esempio il valore di alcune variabili o proprietà degli oggetti:

```
print("Mio_valore = ", mio_valore)
```

Quando usate l'istruzione **print()** tra le parentesi potete metterci molte cose, nell'esempio precedente abbiamo messo una stringa esplicativa e un nome di variabile, la virgola separa i due elementi, la possiamo pensare come ad una lista, se concatenate più valore,

ricordatevi di separare le stringhe, nel caso dobbiate andare a capo, potete fare in questo modo:

```
print("Mio_valore = ", mio_valore, "\n altro valore = ",
      ↪ altro_valore)
```

Il carattere `\n` forza l'inserimento di un "a capo", indipendentemente dal sistema operativo usato.

Quando eseguiamo alcune operazioni (ad esempio le operazioni booleane che vedremo fra poco), il solido risultante possiede una propria proprietà **Placement**.

Questa cosa non deve sembrare strana, indipendentemente dal riferimento di costruzione delle primitive, il solido finale possiede un proprio riferimento di costruzione, che è (0,0,0) indipendente da quello delle primitive generatrici, anche se quel punto è esterno al solido, per questo è sempre meglio curare il posizionamento in fase di costruzione del solido, in modo da avere il riferimento di costruzione nel punto meno problematico.

5.2.2. Listato - Riferimento costruzione

```
1 #
2 """rif-cost.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     Attenzione: Questo listato va usato aggiungendo le linee
12     da 18 in poi al codice presente in sc-base.py
13
14     Warning: This code has to be adding the lines starting
15     from 18 to the code in sc-base.py
16 """
17
18 def cubo(nome, lung, larg, alt, cent = False, off_z = 0):
19     obj_b = DOC.addObject("Part::Box", nome)
20     obj_b.Length = lung
21     obj_b.Width = larg
22     obj_b.Height = alt
23
24     if cent == True:
25         posiz = Vector(lung * -0.5, larg * -0.5, off_z)
26     else:
```

```
27         posiz = Vector(0, 0, off_z)
28
29     rot_c = VZOR # Rotation center
30     rot = ROTO # Rotation angles
31     obj_b.Placement = FreeCAD.Placement( posiz, rot, rot_c)
32
33     DOC.recompute()
34
35     return obj_b
36
37
38 def base_cyl(nome, ang, rad, alt ):
39     obj = DOC.addObject("Part::Cylinder", nome)
40     obj.Angle = ang
41     obj.Radius = rad
42     obj.Height = alt
43
44     DOC.recompute()
45
46     return obj
47
48 # definizione oggetti
49
50 obj1 = cubo("cubo_cyl", 10, 20, 10, True, 10)
51 obj2 = base_cyl("cilindro", 360, 2.5, 15 )
52
53 print("Cubo Base = ", obj1.Placement)
54 print("Cilindro = ", obj2.Placement)
55
56 setview()
```

Capitolo 6

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.

Vediamo le **operazioni booleane**:

Nome	Descrizione
Unione	Part::Fuse oppure Part::MultiFuse
Sottrazione	Part::Cut
Intersezione	Part::MultiCommon

Questa spiegazione presuppone un metodo di lavoro che parte dal listato nella sezione 2.4 a pagina 26 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 sezione 6.4 a pagina 59 e i numeri di riga mostrati fanno riferimento a quel listato.

6.1. 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 elenco 6.1 nella pagina seguente.

Questa porzione di codice la inseriamo subito dopo il metodo **base_cyl**, nel listato **rif-cost.py**.

Descriviamo brevemente le proprietà:

- **Base** è la geometria a cui dobbiamo aggiungere la seconda geometria.
- **Tool** è la geometria da aggiungere.

Programma 6.1: Unione

```

38 def fuse_obj(nome, obj_0, obj_1):
39     obj = DOC.addObject("Part::Fuse", nome)
40     obj.Base = obj_0
41     obj.Tool = obj_1
42     obj.Refine = True
43     DOC.recompute()
44
45     return obj

```

- 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 subito dopo aver creato le geometrie, quindi subito dopo la riga che crea la geometria **primo cilindro** invocando il metodo **base_cyl**.

A questo proposito facciamo notare il fatto che ci riferiamo all'oggetto con il nome della sua variabile, a questo serve l'istruzione **return** presente alla fine dei metodi di creazione.

Anche il metodo che definisce l'unione, possiede un'istruzione **return** e quindi “restituisce” una istanza dell'oggetto creato pronta per essere inserita in una variabile e usata per modificare le proprietà dell'oggetto restituito.

Lanciano il programma apparentemente non otteniamo nulla, infatti il risultato è la figura fig. 6.1b a pagina 58, notiamo però 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.

L'operazione **unione** possiede anche una seconda forma, adatta per quando è necessario unire più di due oggetti, la presentiamo nel Programma elenco 6.2 nella pagina seguente.

Dal listato possiamo notare alla riga 46 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

Programma 6.2: Unione multipla

```
47 def mfuse_obj(nome, obj_0, obj_1):
48     obj = DOC.addObject("Part::MultiFuse", nome)
49     obj.Shapes = (obj_0, obj_1)
50     obj.Refine = True
51     DOC.recompute()
52
53     return obj
```

Programma 6.3: Sottrazione

```
55 def cut_obj(nome, obj_0, obj_1):
56     obj = DOC.addObject("Part::Cut", nome)
57     obj.Base = obj_0
58     obj.Tool = obj_1
59     obj.Refine = True
60     DOC.recompute()
61
62     return obj
```

lista di oggetti creati in modo automatico. (lo vedremo più avanti quanto affronteremo alcune tecniche di creazione “avanzata”).

6.2. 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 elenco 6.3.

Inseriamo le righe del Programma elenco 6.3 subito dopo al metodo **base_cyl**.

Lo usiamo come 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 fig. 6.1c nella pagina successiva.

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.

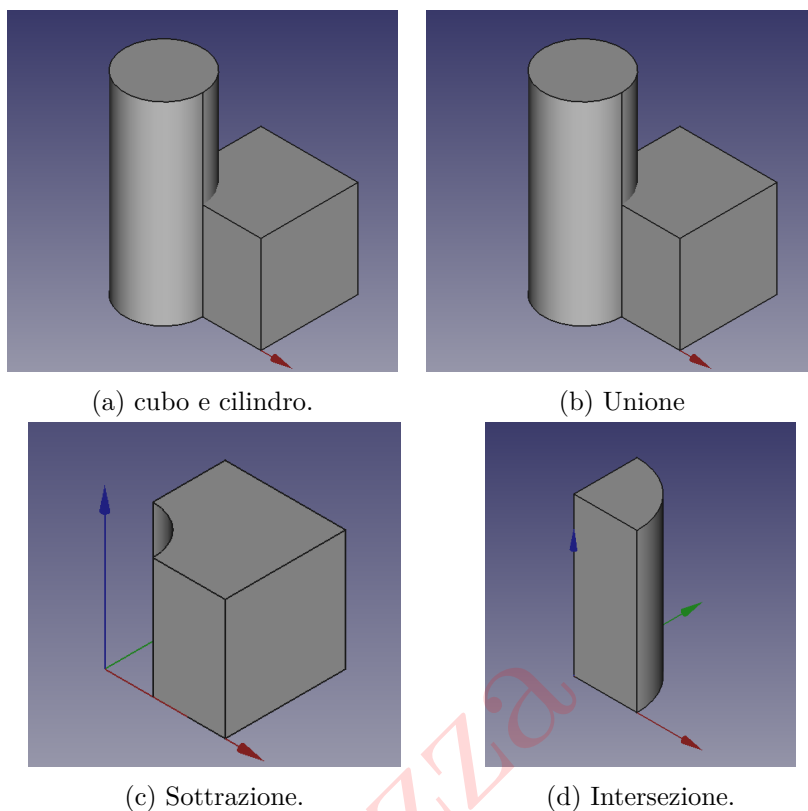


Figura 6.1.: Le operazioni Booleane.

6.3. Intersezione

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

Creiamo questo metodo come nel Programma elenco 6.4 nella pagina seguente.

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

Lo usiamo come 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 fig. 6.1d.

Programma 6.4: Intersezione

```
64 def int_obj(nome, obj_0, obj_1):
65     obj = DOC.addObject("Part::MultiCommon", nome)
66     obj.Shapes = (obj_0, obj_1)
67     obj.Refine = True
68     DOC.recompute()
69
70     return obj
```

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

6.4. 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     Attenzione: Questo listato va usato aggiungendo le linee
12     da 18 in poi al codice presente in sc-base.py
13
14     Warning: This code has to be adding the lines starting
15     from 18 to the code in sc-base.py
16 """
17
18 def cubo(nome, lung, larg, alt):
19     obj_b = DOC.addObject("Part::Box", nome)
20     obj_b.Length = lung
21     obj_b.Width = larg
22     obj_b.Height = alt
23
24     DOC.recompute()
25
26     return obj_b
27
28 def base_cyl(nome, ang, rad, alt):
29     obj = DOC.addObject("Part::Cylinder", nome)
```



```
30     obj.Angle = ang
31     obj.Radius = rad
32     obj.Height = alt
33
34     DOC.recompute()
35
36     return obj
37
38 def fuse_obj(nome, obj_0, obj_1):
39     obj = DOC.addObject("Part::Fuse", nome)
40     obj.Base = obj_0
41     obj.Tool = obj_1
42     obj.Refine = True
43     DOC.recompute()
44
45     return obj
46
47 def mfuse_obj(nome, obj_0, obj_1):
48     obj = DOC.addObject("Part::MultiFuse", nome)
49     obj.Shapes = (obj_0, obj_1)
50     obj.Refine = True
51     DOC.recompute()
52
53     return obj
54
55 def cut_obj(nome, obj_0, obj_1):
56     obj = DOC.addObject("Part::Cut", nome)
57     obj.Base = obj_0
58     obj.Tool = obj_1
59     obj.Refine = True
60     DOC.recompute()
61
62     return obj
63
64 def int_obj(nome, obj_0, obj_1):
65     obj = DOC.addObject("Part::MultiCommon", nome)
66     obj.Shapes = (obj_0, obj_1)
67     obj.Refine = True
68     DOC.recompute()
69
70     return obj
71
72 # definizione oggetti
73
74 obj = cubo("cubo_di_prova", 5, 5, 5)
75
```

```
76 obj1 = base_cyl('primo cilindro', 360,2,10)
77
78 # mfuse_obj("cubo-cyl-fu", obj, obj1)
79
80 #cut_obj("cubo-cyl-cu", obj, obj1)
81
82 int_obj("cubo-cyl-is", obj, obj1)
83
84 setview()
```

Capitolo 7

Modellazione BREP

Finora abbiamo creato direttamente gli oggetti usando le forma base, applicando ad esse delle operazioni booleane.

Anche se l'approccio alla modellazione è totalmente differente, la potenza di **FreeCAD** si rivela nel fatto che per utilizzare questo approccio più "moderno" non ci dobbiamo muovere affatto dal modulo Part.

Questo modo di comporre, utilizza un approccio leggermente diverso da quello visto finora, e lo possiamo considerare più "moderno" e più "efficiente", in termini di memoria, abbiamo spiegato alcuni concetti base e la ragione di questo approccio nel capitolo sezione 3.4.1 a pagina 37.

Passa per la creazione di componenti intermedie che solo alla fine vengono trasformate in "solidi".

Per questa parte di spiegazione partiremo dal listato mostrato nella sezione sezione 7.1.1 a pagina 67 con il titolo **Listato - estrusione**.

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

```
18 def reg_poly(center=Vector(0, 0, 0), sides=6, dia=6,  
19             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**

Una piccola nota sul nome delle variabili, meglio che sia il più possibile esplicativo, se proprio non è possibile per motivi di spazio è meglio aggiungere una piccola spiegazione nella docstring per spiegarlo.

Ad esempio nella descrizione del parametro **outer** viene usata la parola apotema, L'apotema è il **raggio della circonferenza interna** del poligono, nel programma è usato in modo improprio, in quanto è inteso come **diametro della circonferenza interna**, ne diamo ragione in nota.¹

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).

¹Non possiamo scrivere un trattato in una **docstring** per cui dobbiamo semplificare, la scelta della 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.

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 23 (**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
- 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 **Contorno (Wire)**, mentre usando il metodo **Face**:

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

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

7.1. Estrusione

Introduciamo ora lo strumento “estrazione”, 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:

```
61 def dado(nome, dia, spess):
62     polyg = reg_poly(Vector(0, 0, 0), 6, dia, 0, 0)
63
64     nut = DOC.addObject("Part::Feature", nome + "_dado")
65     nut.Shape = polyg.extrude(Vector(0, 0, spess))
66
67     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 documento appartiene al tipo `Part::Feature`, che possiamo considerare come la classe “generica” degli oggetti 3D, a cui dobbiamo fornire un appropriato **forma topologica** attraverso la proprietà `Shape`, in questo modo:

```
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 creato invocando il metodo:

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

Dovreste ottenere il risultato mostrato nella figure fig. 7.1a nella pagina successiva.

Una volta averlo creato noterete che nella **Scheda 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 trafilata, 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.

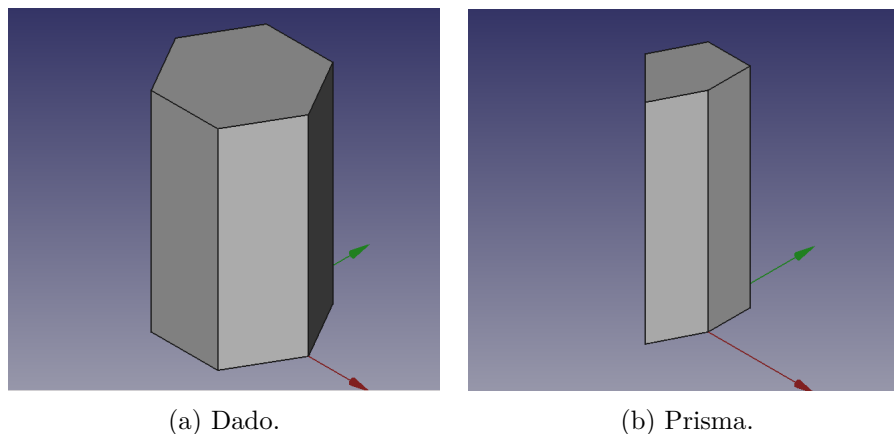


Figura 7.1.: Estrusione

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 è dato dal metodo `estr_comp` che permette di ottenere il prisma mostrato nella figura fig. 7.1b.

```

70 def estr_comp(nome, spess):
71     vertex = ((-2,0,0), (-1, 2, 0), (1, 2, 0), (1, 0, 0),
72              (0, -2, 0), (-2,0,0))
73
74     obj = Part.makePolygon(vertex)
75     wire = Part.Wire(obj)
76     poly_f = Part.Face(wire)
77
78     cexsh = DOC.addObject("Part::Feature", nome)
79     cexsh.Shape = poly_f.extrude(Vector(0, 0, spess))
80
81     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, meglio se in senso antiorario, seguendo la convenzione “trigonometrica”.²

²Secondo la convenzione trigonometrica una rotazione “positiva” è in senso antiorario, in genere in

Vediamo la creazione della tupla in queste righe:

```
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**.

7.1.1. 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     Attenzione: Questo listato va usato aggiungeno le linee
12     da 18 in poi al codice presente in sc-base.py
13
14     Warning: This code has to be adding the lines starting
15     from 18 to the code in sc-base.py
16 """
17
18 def reg_poly(center=Vector(0, 0, 0), sides=6, dia=6,
19             align=0, outer=1):
20     """
21     This return a polygonal shape
22
23     Keywords Arguments:
```

moltissimi programmi di grafica si usa questa convenzione, notate che è in senso inverso rispetto alla usuale scala di un goniometro.


```

24         center    - Vector holding the center of the polygon
25         sides     - the number of sides
26         dia       - the diameter of the base circle
27                     (apothem or externa diameter)
28         align     - 0 or 1 it try to align the base with one
                     ↪ axis
29         outer     - 0: apothem 1: outer diameter (default 1)
30     """
31
32
33     ang_dist = pi / sides
34
35     if align == 0:
36         theta = 0.0
37     else:
38         theta = ang_dist
39
40     if outer == 1:
41         rad = dia * 0.5
42     else:
43         # dia is the apothem, so calculate the radius
44         # outer radius given the inner diameter
45         rad = (dia / cos(ang_dist)) * 0.5
46
47     vertex = []
48
49     for n_s in range(0, sides+1):
50         vpx = rad * cos((2 * ang_dist * n_s) + theta) +
                     ↪ center[0]
51         vpy = rad * sin((2 * ang_dist * n_s) + theta) +
                     ↪ center[1]
52         vertex.append(Vector(vpx, vpy, 0))
53
54     obj = Part.makePolygon(vertex)
55     wire = Part.Wire(obj)
56     poly_f = Part.Face(wire)
57
58     return poly_f
59
60
61 def dado(nome, dia, spess):
62     polyg = reg_poly(Vector(0, 0, 0), 6, dia, 0, 0)
63
64     nut = DOC.addObject("Part::Feature", nome + "_dado")
65     nut.Shape = polyg.extrude(Vector(0, 0, spess))
66

```

```
67     return nut
68
69
70 def estr_comp(nome, spess):
71     vertex = ((-2,0,0), (-1, 2, 0), (1, 2, 0), (1, 0, 0),
72             (0, -2, 0), (-2,0,0))
73
74     obj = Part.makePolygon(vertex)
75     wire = Part.Wire(obj)
76     poly_f = Part.Face(wire)
77
78     cexsh = DOC.addObject("Part::Feature", nome)
79     cexsh.Shape = poly_f.extrude(Vector(0, 0, spess))
80
81     return cexsh
82
83
84 # definizione oggetti
85
86 dado("Dado", 5.5, 10)
87
88 #estr_comp("complesso", 10)
89
90 setview()
```

Programma 7.1: Rivoluzione

```

85 def manico(nome):
86     """Revolve a face"""
87     face = reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0)
88     # base point of the rotation axis
89     pos = Vector(0,10,0)
90     # direction of the rotation axis
91     vec = Vector(1,0,0)
92     angle = 360 # Rotation angle
93
94     point("punto_rot", pos)
95
96     obj = DOC.addObject("Part::Feature", nome)
97     obj.Shape = face.revolve(pos, vec, angle)
98
99     DOC.recompute()
100
101     return obj

```

7.2. 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 85 in poi con quelle visualizzate nel elenco 7.1 denominato **Rivoluzione**.

Infatti nella funzione **manico**, troviamo alla riga 87, 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(1,0,0)
```

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

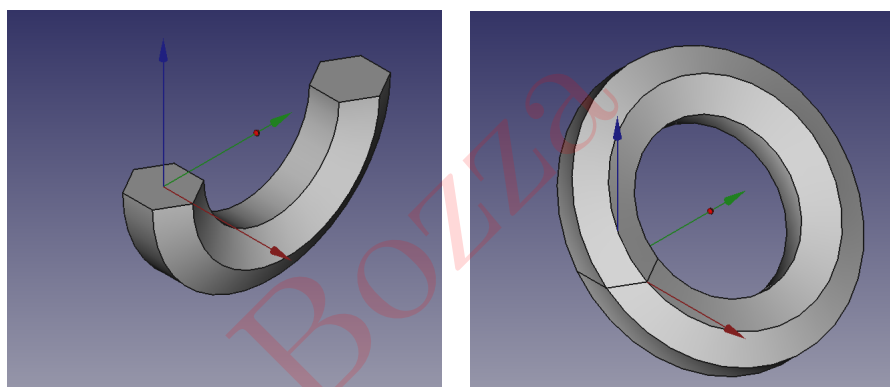
```
angle = 360 # Rotation angle
```

Questo è ovviamente l'angolo di rotazione 180°

Lanciamo l'esecuzione attraverso la linea 105:

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

Il risultato sarà una cosa come quella in figura fig. 7.2a.



(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 fig. 7.2b.

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

7.3. 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:

```
104 def loft(nome):
105     """Loft a face"""
106     faces = []
107     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
108     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
109     faces.append(reg_poly(Vector(0, 0, 0), 6, 8, 0, 0))
110     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
111     faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
112
113     sections = []
114
115     for idx, face in enumerate(faces):
116         sect = DOC.addObject(
117             "Part::Feature",
118             nome + "_sezione_" + str(idx))
119         sect.Shape = face
120         sect.Placement = FreeCAD.Placement(
121             Vector(0,0, 25 * idx),
122             ROT0, VZOR)
123         sections.append(sect)
124
125     obj = DOC.addObject("Part::Loft", nome)
126     obj.Sections=sections
127     obj.Solid = False
128     obj.Ruled = True
129     obj.Closed = False
130     obj.MaxDegree = 10
131
132     DOC.recompute()
133
134     print(obj.Placement)
135     return obj
136
```

```

137
138 # definizione oggetti
139
140 loft("Loft")
141
142 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).

```

106 faces = []
107 faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
108 faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
109 faces.append(reg_poly(Vector(0, 0, 0), 6, 8, 0, 0))
110 faces.append(reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0))
111 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 degli **oggetti documento** appartenenti al tipo **Part::Feature**.

Usiamo la nostra pre-lista, per creare la lista di forme in questo modo:

```

113 sections = []
114
115 for idx, face in enumerate(faces):
116     sect = DOC.addObject(
117         "Part::Feature",
118         nome + "_sezione_" + str(idx))
119     sect.Shape = face
120     sect.Placement = FreeCAD.Placement(
121         Vector(0,0, 25 * idx),
122         ROT0, VZOR)
123     sections.append(sect)

```

Notate nella riga 115 il “solito” metodo per scorrere la lista ottenendo anche un numero progressivo di posizione che utilizziamo in due modi:

- alla riga 118 per modificare il nome dell'oggetto.
- alla riga 121 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 119 si occupa di passare la forma selezionata alla proprietà **Shape** dell'oggetto **Part::Feature**.

Mentre la linea 123 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 fig. 7.1b a pagina 66, replicando in un certo senso le operazioni che compie uno slicer per la stampa 3D.

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

```
138 # definizione oggetti
139
140 loft("Loft")
141
142 setview()
```

Passando alla proprietà **Sections** la lista **sections** creata in precedenza, e impostando le altre proprietà; Il funzionamento di queste altre proprietà va studiato nella documentazione ufficiale di **FreeCAD**, che contiene indicazioni sulle limitazione e le complicazioni d'uso che si possono incontrare, i valori proposti permettono di ottenere la figura fig. 7.3

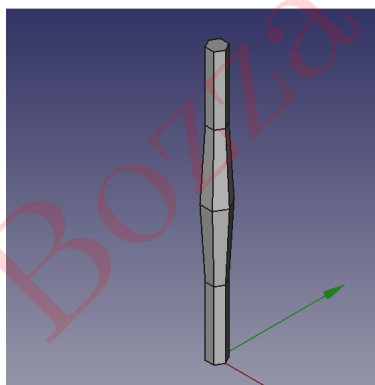


Figura 7.3.: Lo strumento Loft

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

7.4. Un esempio avanzato

Ora mettiamo assieme alcune tecniche viste fin ora per creare un modello avanzato usando la tecnica BREP.

Troverete nella pagina GitHub il listato completo con il nome di **brep-adv.py**.

Cominciamo con un esempio classico, un parallelepipedo, abbiamo già **Part::Box**, si ma quello è un oggetto costruito con la tecnica giurassica del CSG, ora vediamo la tecnica “moderna” del BREP.

Nel listato sezione 7.4.1 a pagina 77 denominato **Listato - esempio BREP** possiamo vedere una definizione di un parallelepipedo

La parte rilevante sono le righe:

```
19     h_lung = lung * 0.5
20     h_larg = larg * 0.5
21
22     vertex = (
23         (h_lung * -1, h_larg * -1, 0),
24         (h_lung * 1, h_larg * -1, 0),
25         (h_lung * 1, h_larg * 1, 0),
26         (h_lung * -1, h_larg * 1, 0),
27         (h_lung * -1, h_larg * -1, 0))
```

Abbiamo semplicemente definito alle righe 19 e 20 la metà delle dimensioni della lunghezza e della larghezza, in modo da centrare la base del cubo rispetto all’origine.

Abbiamo poi alla riga 22 e seguenti definito i cinque punti della base, ricordate che dobbiamo avere una superficie chiusa, per cui il primo punto deve essere uguale all’ultimo, i punti della base sono ordinati in senso antiorario.

Il resto del metodo è praticamente lo stesso del metodo **estr_comp**, quindi nulla di sconosciuto.

Aggiungiamo alla fine

```
# definizione oggetti

cubo_brep("cubo-brep", 20, 10, 15)

setview()
```

Lanciamo il programma e otteniamo il nostro **solido** reale che assomiglia a quanto possiamo ottenere con **Part::Box**.

Ovviamente il guadagno ottenuto compiendo questa operazione per un **solido** così semplice è praticamente nullo.

Se le cose diventano complesse si cominciano ad apprezzare i vantaggi della tecnica BREP.

Creiamo una secondo metodo facendo copia e incolla del metodo **cubo_brep**, rinomando il metodo copiato **forma_brep**.

Modificate il listato in modo da leggere dopo la definizione della lista dei vertici **vertex**

```
= (...:
```

```
39     obj = Part.makePolygon(vertex)
40     wire = Part.Wire(obj)
```



```

41
42     hole = Part.makeCircle(1.5)
43     h_wire = Part.Wire(hole)
44     h_wire.reverse()
45
46     h_wire1 = h_wire.copy()
47     h_wire1.translate(Vector(3.5,0,0))
48
49     h_wire2 = h_wire.copy()
50     h_wire2.translate(Vector(-3.5,0,0))
51
52     poly_f = Part.Face([wire, h_wire, h_wire1, h_wire2])
53
54     cexsh = DOC.addObject("Part::Feature", nome)
55     cexsh.Shape = poly_f.extrude(Vector(0, 0, alt))
56
57     return cexsh

```

Spieghiamo in dettaglio cosa abbiamo fatto.

Abbiamo definito 3 buchi nella faccia di base; Questa “faccia con i buchi” la useremo poi per estrarre il **solido** finale.

Analizziamo in dettaglio il procedimento:

Linea 42 Creiamo un cerchio usando **Part.makeCircle()**, otteniamo una **forma topologica**.

Linea 43 Trasformiamo la **forma topologica** in un **contorno**.

Linea 44 Invertiamo la **normale** del nostro **contorno**.

La normale è per così dire l’orientamento della **faccia**, anche se topologicamente possiamo pensare che per un **contorno** non abbia senso avere un orientamento, l’oggetto **contorno** possiede una proprietà **Orientation**

```

print("h_w Orient prima = ", h_wire.Orientation)
h_wire.reverse()
print("h_w Orient dopo = ", h_wire.Orientation)

```

e vedrete che appare stampato nella **finestra dei rapporti**:

```

h_w Orient prima = Forward
h_w Orient dopo = Reversed

```

in pratica stiamo dicendo a **FreeCAD** che questo **contorno** sarà un “buco”.

Ottenuto un **contorno** con la corretta normale, semplicemente duplichiamo il **contorno** alla riga 46 e poi alla linea 47 lo portiamo nella posizione voluta.

Facciamo la medesima cosa con le linee 49 e 50 per una terzo “buco”

Alla linea 52 creiamo la nostra **faccia** passando una lista di **contorni**, (una tupla darebbe errore).

FreeCAD userà i **contorni** con le normali invertite per creare i “buchi” nella **faccia**.

Se commettete un errore, ad esempio con le normali, il **solido** finale non sarà corretto ed in genere lo vedrete visualizzato in modo strano perché probabilmente sarà un **solido non manifold**.

Nella figura fig. 7.4 potete vedere il risultato di tutta la fatica fatta.

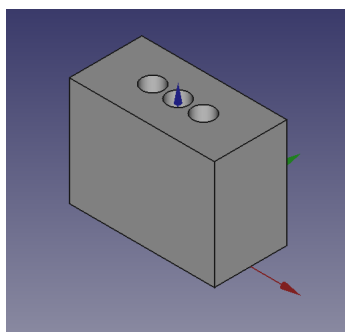


Figura 7.4.: L'oggetto creato usando la tecnica BREP

Usando l'approccio CSG, avremmo dovuto creare tre cilindri e sottrarli dal parallelepipedo.

7.4.1. Listato - esempio BREP

```

1  #
2  """brep-adv.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     Attenzione: Questo listato va usato aggiungendo le linee
12     da 18 in poi al codice presente in sc-base.py
13
14     Warning: This code has to be adding the lines starting
15     from 18 to the code in sc-base.py
16 """
17
18 def cubo_brep(nome, lung, larg, alt):
19     h_lung = lung * 0.5

```

```
20     h_larg = larg * 0.5
21
22     vertex = (
23         (h_lung * -1, h_larg * -1, 0),
24         (h_lung * 1, h_larg * -1, 0),
25         (h_lung * 1, h_larg * 1, 0),
26         (h_lung * -1, h_larg * 1, 0),
27         (h_lung * -1, h_larg * -1, 0))
28
29     obj = Part.makePolygon(vertex)
30     wire = Part.Wire(obj)
31     poly_f = Part.Face(wire)
32
33     cexsh = DOC.addObject("Part::Feature", nome)
34     cexsh.Shape = poly_f.extrude(Vector(0, 0, alt))
35
36     return cexsh
37
38
39 def forma_brep(nome, lung, larg, alt):
40     h_lung = lung * 0.5
41     h_larg = larg * 0.5
42     vertex = (
43         (h_lung * -1, h_larg * -1, 0),
44         (h_lung * 1, h_larg * -1, 0),
45         (h_lung * 1, h_larg * 1, 0),
46         (h_lung * -1, h_larg * 1, 0),
47         (h_lung * -1, h_larg * -1, 0))
48
49     obj = Part.makePolygon(vertex)
50     wire = Part.Wire(obj)
51
52     hole = Part.makeCircle(1.5)
53     h_wire = Part.Wire(hole)
54     h_wire.reverse()
55
56     h_wire1 = h_wire.copy()
57     h_wire1.translate(Vector(3.5,0,0))
58
59     h_wire2 = h_wire.copy()
60     h_wire2.translate(Vector(-3.5,0,0))
61
62     poly_f = Part.Face([wire, h_wire, h_wire1, h_wire2])
63
64     cexsh = DOC.addObject("Part::Feature", nome)
65     cexsh.Shape = poly_f.extrude(Vector(0, 0, alt))
```

```
66
67     return cexsh
68
69 # definizione oggetti
70
71 forma_brep("cubo-brep", 20, 10, 15)
72
73 setview()
```

Capitolo 8

La componente Vista

Ai più acuti occhi di falco non sarà sfuggito, nella figura fig. 7.2a a pagina 71, 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 elenco 7.1 a pagina 70 denominato **Rivoluzione** che non abbiamo spiegato:

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     DOC.recompute()
100
101     return obj
102
103 # definizione oggetti
104
105 manico("Manico")
106
107 setview()
```

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 ??.

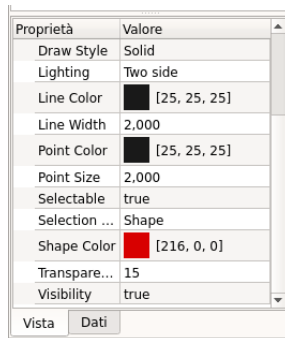


Figura 8.1.: La Scheda Vista

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 inserire dei riferimenti ben visibili, come assi e punti colorati, aiuta nei momenti critici che si possono verificare durante la modellazione.

Descriviamo sommariamente la parte **ViewObject** che riflette le proprietà che trovate nella **Scheda Vista** di ogni oggetto, come vedete nell'immagine fig. 8.1.

Abbiamo già accennato che ogni oggetto comprende una parte relativa alla **Geometria** e una parte **Vista**, ricordiamo che la parte **Geometria** possiamo modificarla nella **Scheda Dati** come abbiamo già visto nell'immagine fig. 5.2 a pagina 50.

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 **Scheda 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 9

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.

9.1. 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.

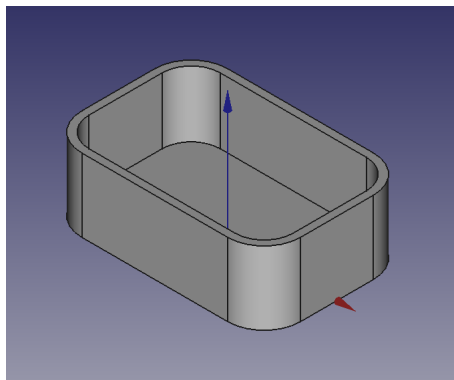


Figura 9.1.: Scatola sardine

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; Molta parte del codice iniziale ed alcuni metodi servono per iniziare a costruire il modello, oppure per definire delle forme base, questo tipo di metodi 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 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.

Altro svantaggio che potete incontrare è legato al modo con cui **FreeCAD** importa i moduli esterni, dovuto alla gestione dei moduli di Python, se effettuate modifiche nel codice del modulo esterno, dovete uscire da **FreeCAD** e rientrare e riaprire il file su cui state lavorando.

Utilizzando una tecnica che illustreremo, è possibile superare questa limitazione.

In caso particolari, però l'unica risorsa è quella descritta prima.

Trovate il listato del modulo che sposta all'esterno del nostro programma alcune routine complesse sotto il nome **Listato - Modulo GfcMod** nell'Appendice sezione 9.1.1 nella pagina seguente.

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 nome del modulo non deve andare in conflitto con un modulo di Python, in genere per i nomi dei moduli, si usa una convenzione, quella di usare il camelCase cioè scrivere il nome del modulo con delle lettere maiuscole all'interno.
I nomi dei moduli di python devono avere le lettere minuscole per cui diventa meno facile sovrapporre i nomi, nel caso usare nomi significativi come **miaLib** oppure **miaLibreriaDiOggetti** può aiutare.

Trovate il listato sezione 9.1.2 a pagina 90 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:

```
22 scripts_path = os.path.join(os.path.dirname(__file__),)
23 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 ti segnalerebbe un errore, quella scrittura invece permette inserire una tupla composta da un solo elemento.

Usando un solo elemento come argomento della funzione **os.path.join()** il suo lavoro di "unione" (join vuol dire unire in Inglese) viene eseguito sulle due parti, una delle quali è **None** in virtù del passaggio della virgola, una cosa apparentemente inutile; Ottengo però un importante effetto collaterale, il percorso che mi viene restituito è corretto in relazione al sistema operativo presente.

Secondo alcuni post trovati nel forum di **FreeCAD**, questo sistema sembra essere quello con meno punti problematici; Utilizzando il modulo **os** dovrebbe (il condizionale è d'obbligo) recuperare il percorso secondo le convenzioni del sistema operativo.¹

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 14, questo dovrebbe permettere all'interprete Python interno di recuperare il percorso del nome del file che stiamo eseguendo.

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
25 import GfcMod as 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**.

9.1.1. 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
```

¹Purtroppo 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.

```
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                   (aphotem or externa diameter)
71         align     - 0 or 1 it try to align the base with one
72                   ↪ axis
73         outer     - 0: aphotem 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)

```

```
98     poly_f = Part.Face(wire)
99
100     return poly_f
101
102
103 def cubo_stondato(doc, nome, lung, larg, alt, raggio, off_z)
104     ↪ :
105
106     c_c1 = Vector((lung * -0.5) + raggio, (larg * -0.5) +
107     ↪ raggio, 0)
108     c_c2 = Vector((lung * -0.5) + raggio, (larg * 0.5) -
109     ↪ raggio, 0)
110     c_c3 = Vector((lung * 0.5) - raggio, (larg * 0.5) -
111     ↪ raggio, 0)
112     c_c4 = Vector((lung * 0.5) - raggio, (larg * -0.5) +
113     ↪ raggio, 0)
114
115     obj_dim = c_c3 - c_c1
116     fi_lung = obj_dim[0] + raggio * 2
117     fi_larg = obj_dim[1] + raggio * 2
118
119     obj_c1 = base_cyl(doc, nome + '_cil1', 360, raggio, alt)
120     obj_c1.Placement = FreeCAD.Placement(
121         c_c1,
122         FreeCAD.Rotation(0, 0, 0),
123         FreeCAD.Vector(0,0,0))
124
125     obj_c2 = base_cyl(doc, nome + '_cil2', 360, raggio, alt)
126     obj_c2.Placement = FreeCAD.Placement(
127         c_c2,
128         FreeCAD.Rotation(0, 0, 0),
129         FreeCAD.Vector(0,0,0))
130
131     obj_c3 = base_cyl(doc, nome + '_cil3', 360, raggio, alt)
132     obj_c3.Placement = FreeCAD.Placement(
133         c_c3,
134         FreeCAD.Rotation(0, 0, 0),
135         FreeCAD.Vector(0,0,0))
136
137     obj_c4 = base_cyl(doc, nome + '_cil4', 360, raggio, alt)
138     obj_c4.Placement = FreeCAD.Placement(
139         c_c4,
140         FreeCAD.Rotation(0, 0, 0),
141         FreeCAD.Vector(0,0,0))
```

```

139     obj1 = cubo(doc, nome + "_int_lu", fi_lung , obj_dim[1],
        ↪ alt, False)
140     obj1.Placement = FreeCAD.Placement(Vector(fi_lung * -0.5
        ↪ , obj_dim[1] * -0.5, 0), FreeCAD.
        ↪ Rotation(0, 0, 0), FreeCAD.Vector(0,0,0
        ↪ ))
141
142     obj2 = cubo(doc, nome + "_int_la", obj_dim[0] , fi_larg,
        ↪ 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

```

9.1.2. 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 Vector, Rotation
18 import Part
19
20 from math import pi, sin, cos
21
22 scripts_path = os.path.join(os.path.dirname(__file__),)
23 sys.path.append(scripts_path)
24
25 import GfcMod as GFC
26
27 importlib.reload(GFC)
28
29 DOC_NAME = "Pippo"
30 CMPN_NAME = None
31
32
33 def activate_doc():
34     """activate document"""
35     FreeCAD.setActiveDocument(DOC_NAME)
36     FreeCAD.ActiveDocument = FreeCAD.getDocument(DOC_NAME)
37     FreeCADGui.ActiveDocument = FreeCADGui.getDocument(
38         ↪ DOC_NAME)
39     print("{0} activated".format(DOC_NAME))
40
41 def setview():
42     """Rearrange View"""
43     DOC.recompute()
44     VIEW.viewAxometric()
45     VIEW.setAxisCross(True)
46     VIEW.fitAll()
47
48
49 def clear_doc():
50     """Clear the active document deleting all the objects"""
51     for obj in DOC.Objects:
52         DOC.removeObject(obj.Name)
53
54 if FreeCAD.ActiveDocument is None:
55     FreeCAD.newDocument(DOC_NAME)
56     print("Document: {0} Created".format(DOC_NAME))
57
58 # test if there is an active document with a "proper" name
59 if FreeCAD.ActiveDocument.Name == DOC_NAME:
60     print("DOC_NAME exist")
```



```

61 else:
62     print("DOC_NAME is not active")
63     # test if there is a document with a "proper" name
64     try:
65         FreeCAD.getDocument(DOC_NAME)
66     except NameError:
67         print("No Document: {0}".format(DOC_NAME))
68         FreeCAD.newDocument(DOC_NAME)
69         print("Document Created".format(DOC_NAME))
70
71 DOC = FreeCAD.getDocument(DOC_NAME)
72 GUI = FreeCADGui.getDocument(DOC_NAME)
73 VIEW = GUI.ActiveView
74 #print("DOC : {0} GUI : {1}".format(DOC, GUI))
75 activate_doc()
76 #print(FreeCAD.ActiveDocument.Name)
77 clear_doc()
78
79 if CMPN_NAME is None:
80     pass
81 else:
82     DOC.addObject("App::DocumentObjectGroup", CMPN_NAME)
83
84 # EPS= tolerance to uset to cut the parts
85 EPS = 0.10
86 EPS_C = EPS * -0.5
87 VZOR = Vector(0, 0, 0)
88 ROTO = Rotation(0, 0, 0)
89
90
91 def sardine(nome, lung, prof, alt, raggio, spess):
92     """sardine
93         crea una geometria a forma di scatola di sardine
94
95         Keywords Arguments:
96         nome      = nome della geometria finale
97         lung      = lunghezza della scatola, lungo l'asse X
98         larg      = larghezza della scatole, lungo l'asse Y
99         alt       = altezza della scatola, lungo l'asse Z
100        raggio     = raggio dello "stondamendo" della scatola
101        spess      = spessore della scatola
102     """
103
104     obj = GFC.cubo_stonato(
105         DOC, nome + "_est", lung, prof, alt, raggio, 0.0)
106

```

```

107     lung_int = lung - (spess * 2)
108     prof_int = prof - (spess * 2)
109     alt_int = alt + EPS - spess
110     raggio_int = raggio - spess
111
112     obj_int = GFC.cubo_stondato(
113         DOC, nome + "_int",
114         lung_int, prof_int, alt_int, raggio_int, spess)
115
116     obj_f = DOC.addObject("Part::Cut", nome)
117     obj_f.Base = obj
118     obj_f.Tool = obj_int
119     obj_f.Refine = True
120     DOC.recompute()
121
122     return obj_f
123
124 sardine("scatola", 30, 20, 10, 5, 1)
125
126 setview()

```

9.2. 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 sezione 9.1.2 a pagina 90 denominato **Listato - sardine.py** la parte che crea tutto è costituita da solo una riga di codice:

```

124 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(...**:

```

92     """sardine
93         crea una geometria a forma di scatola di sardine
94
95         Keywords Arguments:

```

```

96         nome      = nome della geometria finale
97         lung       = lunghezza della scatola, lungo l'asse X
98         larg       = larghezza della scatole, lungo l'asse Y
99         alt        = altezza della scatola, lungo l'asse Z
100        raggio     = raggio dello "stondamendo" della scatola
101        spess      = spessore della scatola
102        " " "

```

vediamo che i parametri passati sono abbastanza esplicativi anche nel nome, ma la docstring 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:

```

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

```

mentre la seconda dalle linee:

```

112        obj_int = GFC.cubo_stonato(
113            DOC, nome + "_int",
114            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.

```

107        lung_int = lung - (spess * 2)
108        prof_int = prof - (spess * 2)
109        alt_int = alt + EPS - spess
110        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.

9.2.1. 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 sezione 9.2.2 a pagina 97 denominato **Listato - cubo_stondato modificato**, la modifica al metodo **cubo_stondato** che va a sostituire il metodo con lo stesso nome nel listato sezione 9.1.1 a pagina 86 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
108                 ↪ , 0),
109         Vector((lung * 0.5) - raggio, (larg * 0.5) - raggio,
110                 ↪ 0),
111         Vector((lung * 0.5) - raggio, (larg * -0.5) + raggio
112                 ↪ , 0)
113     ]

```

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**

9.2.2. 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) +
      ↪ 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     ]
111
112     obj_dim = c_pos[2] - c_pos[0]
113     fi_lung = obj_dim[0] + raggio * 2
114     fi_larg = obj_dim[1] + raggio * 2
115
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)
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     comps.append(obj1)
135
136     obj2 = cubo(doc, nome + "_int_la", obj_dim[0] , fi_larg,
137                 ↪ alt, False)
138     obj2.Placement = FreeCAD.Placement(
139         Vector(obj_dim[0] * -0.5, fi_larg * -0.5, 0),
140         FreeCAD.Rotation(0, 0, 0), FreeCAD.Vector(0,0,0))
141     comps.append(obj2)
142
143     obj_int = doc.addObject("Part::MultiFuse", nome)
144     obj_int.Shapes = comps
145     obj_int.Refine = True
146     doc.recompute()
147
148     obj_int.Placement = FreeCAD.Placement(
149         Vector(0, 0, off_z),
150         FreeCAD.Rotation(0, 0, 0),
151         FreeCAD.Vector(0,0,0)
152     )
153
154     return obj_int

```

9.3. 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 sezione 9.1.2 a pagina 90 denominato **Listato - sardine.py**.

```

22 scripts_path = os.path.join(os.path.dirname(__file__),)
23 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 sezione 9.3.1 nella pagina successiva

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 mettetetele lì; Ricordate che le variabili comuni saranno accessibili in lettura e scrittura. Ad esempio, se la libreria si chiama **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 lasciare **due righe bianche** tra i vari metodi; Questo permette di identificare visivamente i metodi, in genere non si lasciano mai due righe bianche 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 ad esempio **pos-x**, **pos-y**, **pos-z** (magari usate l'underscore o "trattino basso"), già dal nome potete intuire cosa fa la variabile, Cercate di evitare però nomi lunghissimi, sia per motivi di spazio, che per motivi di leggibilità.
- Non fate righe più lunghe di 78 caratteri, quando dovete comporre il codice con un carattere decentemente leggibile, il numero di caratteri a disposizione su una riga, finisce presto, anche nella finestra dell'editor interno di **FreeCAD**; Nel caso poi lo dobbiate stampare, i problemi aumentano.
- Per conoscere i modi corretti per "andare a capo" quando scrivete righe di codice vi consiglio di cercare nella documentazione di Python il PEP8 e leggerlo.

Nulla ci vieta di creare tante librerie di oggetti diversificate da usare al bisogno, per esempio un modulo conterrà i metodi per la realizzazione di viti e dadi, un secondo modulo conterrà quelli per realizzare 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 modo di procedere possiede il vantaggio aggiuntivo di poter correggere un eventuale errore solo nella libreria e automaticamente sarà corretto in tutti gli script che utilizzano quella libreria.

Nella creazione delle librerie si devono usare alcune accortezze:

- Quando si fanno modifiche, vanno documentate in modo corretto; Un esempio su tutti, non lasciare un vecchio commento che contraddice il nuovo codice.
- Quando si modifica un metodo aggiungendo nuovi parametri, dare ai nuovi parametri dei valori di default; In questo modo se il metodo dovesse per caso essere richiamato da un programma realizzato precedentemente alla modifica la chiamata non dovrà essere modificata. Curare anche che l'oggetto ottenuto sia lo stesso di quello del metodo pre-modifica.
- In ragione del punto precedente, sarebbe meglio riportare nella docstring della libreria un numero di versione che indichi l'evoluzione, ed sarebbe meglio che anche nel programma chiamante, sia riportato nella docstring, il numero di versione della libreria usata.

9.3.1. 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 10

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 **Scheda Dati** e **Scheda Vista**, che abbiamo già visto nel capitolo 8 a pagina 80 denominato **La componente Vista**.
- Quando qualche componente non è accessibile probabilmente fa parte di un workbench 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 sezione 10.1.1 a pagina 106 denominato **Listato - aereo**, presente anche in GitHub con il nome **aereo.py**

10.1. 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:

```
50 def sfera(nome, rad):
51     obj = DOC.addObject("Part::Sphere", nome)
52     obj.Radius = rad
53
54     DOC.recompute()
55
56     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 scheda **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:

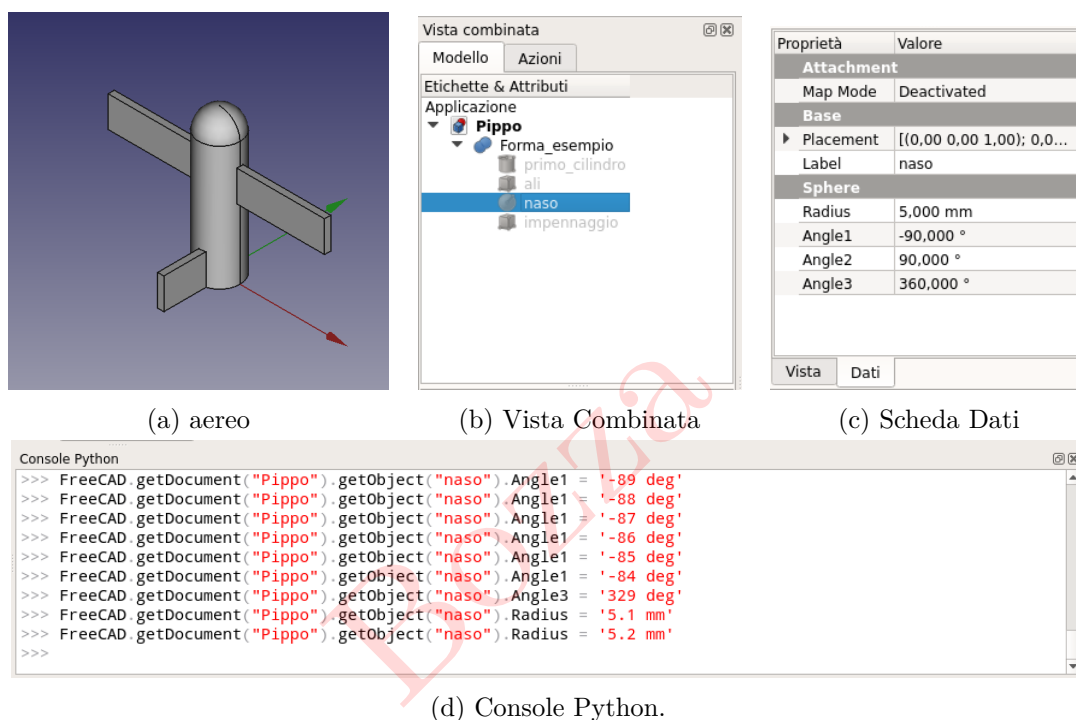


Figura 10.1.: Derivazione delle proprietà.

Nella figura 10.1a, potete vedere il modello finale generato dal codice proposto, nella figura 10.1b 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 10.1c potete vedere la parte dell'**editor 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 **Scheda 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 tre cose:

- Esistono anche se non le abbiamo utilizzate.
- Prendono in automatico alcuni valori di default.
- Controllano la generazione della sfera.

Non avendole esplicitamente definite nel codice, ci si potrebbe chiedere come fare per utilizzarle, in altre parole che nome dobbiamo usare per assegnare loro un valore.

Possiamo “intuire” che si chiamino come mostrato nell'**editor delle proprietà** anche quando usate in uno script, ad esempio:

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

L'intuizione in questo caso ha indovinato, ma se per caso i nomi fossero stati tradotti come nel caso delle componenti di **Placement**, rimarrebbe il dubbio di come chiamarle quando invochiamo il metodo di creazione della sfera.

Ricordiamo per inciso che l'**editor delle proprietà** possiede le due schede: **Scheda Vista** e **Scheda Dati** e che l'immagine mostra la **Scheda 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 10.1d.

```
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'
```

Queste righe sono semplicemente l'eco dei comandi impartiti andando a modificare le singole proprietà mediante l'interfaccia grafica usando le freccette di incremento e decremento presenti accanto ad ogni valore.

Sono però scritte usando il nome dei moduli Interni di **FreeCAD**, per usarle negli script usando i nomi che normalmente utilizziamo, 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:

```
83      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`.

- Finalmente troviamo il nome della proprietà; In questo nostro caso **Radius**, **Angle1** e **Angle3**.

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**.

Quasi esattamente perché il nome dell'oggetto è il nome della "primitiva", cioè nella

figura 10.1b le forme con l'icona , cioè **primo.cilindro**, **ali**, **naso**, **impennaggio**; Quando visualizziamo la **vista ad albero** prima di quel nome c'è **Forma esempio**, che 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**.

Il nome dell'oggetto non riflette sempre l'ordine della **vista ad albero**, ma l'ordine degli oggetti nel documento, ai fini della sequenza dei nomi, tutti gli oggetti sono figli del documento che li contiene, **Forma esempio**, **ali**, **naso** e **impennaggio** sono "fratelli".

Ai fini logici però **ali**, **naso**, ecc sono i **componenti** dell'operazione booleana unione **Forma esempio** e nell'albero infatti sono messi come se fossero suoi "figli".

Provate come esercizio ad inserire all'inizio dello script dopo aver importato **FreeCAD**:

```
import FreeCADGui
```

E poi tra `aeroplano()` e `setview()` alla riga 100:

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

(A patto che il vostro documento si chiami "Pippo" e il vostro oggetto si chiami "Forma_esempio", se avete cambiato i nomi non ditemi poi che il codice non funziona!)

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

Approfittiamo per notare una particolarità che è meglio far notare:

notate alle righe 79 e 91 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** apparentemente trasforma i caratteri spazio presenti nel nome passato come argomento, in un carattere di trattino basso (-).

10.1.1. 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     Attenzione: Questo listato va usato aggiungeno le linee
12     da 18 in poi al codice presente in sc-base.py
13
14     Warning: This code has to be adding the lines starting
15     from 18 to the code in sc-base.py
16 """
17
18 def cubo(nome, lung, larg, alt, cent = False, off_z = 0):
19     obj_b = DOC.addObject("Part::Box", nome)
20     obj_b.Length = lung
21     obj_b.Width = larg
22     obj_b.Height = alt
23
24     if cent == True:
25         posiz = Vector(lung * -0.5, larg * -0.5, off_z)
26     else:
27         posiz = Vector(0, 0, off_z)
```

```
28
29     obj_b.Placement = FreeCAD.Placement(
30         posiz,
31         FreeCAD.Rotation(0, 0, 0),
32         FreeCAD.Vector(0,0,0)
33     )
34
35     DOC.recompute()
36
37     return obj_b
38
39
40 def base_cyl(nome, ang, rad, alt ):
41     obj = DOC.addObject("Part::Cylinder", nome)
42     obj.Angle = ang
43     obj.Radius = rad
44     obj.Height = alt
45
46     DOC.recompute()
47
48     return obj
49
50 def sfera(nome, rad):
51     obj = DOC.addObject("Part::Sphere", nome)
52     obj.Radius = rad
53
54     DOC.recompute()
55
56     return obj
57
58
59 def mfuse_obj(nome, objs):
60     obj = DOC.addObject("Part::MultiFuse", nome)
61     obj.Shapes = objs
62     obj.Refine = True
63     DOC.recompute()
64
65     return obj
66
67
68 def aeroplano():
69
70     lung_fus = 30
71     diam_fus = 5
72     ap_alare = lung_fus * 1.75
73     larg_ali = 7.5
```



```

74     spess_ali = 1.5
75     alt_imp = diam_fus * 3.0
76     pos_ali = (lung_fus*0.70)
77     off_ali = (pos_ali- (larg_ali * 0.5))
78
79     obj1 = base_cyl('primo cilindro', 360, diam_fus,
80                     ↪ lung_fus)
81
82     obj2 = cubo('ali', ap_alare, spess_ali, larg_ali, True,
83               ↪ off_ali)
84
85     obj3 = sfera("naso", diam_fus)
86     obj3.Placement = FreeCAD.Placement(Vector(0,0,lung_fus),
87               ↪ FreeCAD.Rotation(0,0,0), Vector(0,0,0)
88               ↪ )
89
90     obj4 = cubo('impennaggio', spess_ali, alt_imp, larg_ali,
91               ↪ False, 0)
92     obj4.Placement = FreeCAD.Placement(Vector(0,alt_imp * -1
93               ↪ ,0), FreeCAD.Rotation(0,0,0), Vector(0,
94               ↪ 0,0))
95
96     objs = (obj1, obj2, obj3, obj4)
97
98     obj = mfuse_obj("Forma esempio", objs)
99     obj.Placement = FreeCAD.Placement(Vector(0,0,0), FreeCAD
100               ↪ .Rotation(0,0,-90), Vector(0,0,pos_ali)
101               ↪ )
102
103     DOC.recompute()
104
105     return obj
106
107 aeroplano()
108
109 setview()

```

10.2. 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 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 **Scheda 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 assegnate durante la creazione di 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()`

Capitolo 11

Stampa 3D

In questa trattazione si suppone di dover utilizzare una stampante hobbistica di tipo FDM, alcuni concetti possono variare se usate altri tipi di stampanti o una CNC.

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.

11.1. 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.

11.2. 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
- Se 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

11.3. Produzione di file per la stampa

L’operazione che in genere deve essere svolta prima di usare un file per la stampa è quella di esportare il modello in un file con formato STL o AMF.

Dovremo usare alcune cautele, siamo comunque facilitati dal fatto che usando lo scripting possiamo controllare meglio ed in un solo luogo i parametri di esportazione.

Usando l’interfaccia grafica questi parametri sono sparsi in giro per l’interfaccia principale e quelli dei workbench interessati.

11.3.1. 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> potete trovare maggiori informazioni sull’argomento.

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

11.3.2. 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 98.
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**.

11.3.3. Formato STL

Usando il formato **STL**, 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. 31

Barra di stato - p. 4

Console Python - p. 4, 20, 103–105, 109

Editor delle Macro - p. 6, 7

Editor delle proprietà - p. 4, 6, 31, 50, 103, 104

Editor Python - p. 7

Finestra dei rapporti - p. 4, 39, 49, 76

Menu standard - p. 4

Scheda Dati Scheda dell'EdP - p. 6, 31, 81, 102–104, 109

Scheda Vista Scheda dell'EdP - p. 6, 31, 32, 50, 81, 82, 102, 104

Selettore degli Ambienti - p. 4

Vista 3D La finestra 3D - p. 3, 4, 6, 31, 43, 113

Vista ad albero - p. 4, 24, 42, 52, 103, 105, 106, 109

Vista combinata - p. 4, 42, 50

Vista selezione - p. 4

Appendice **B**

Glossario

Angoli di Tait-Bryan - p. 51, 52

Operazioni booleane - p. 40, 55

Intersezione - p. 40, 55

Sottrazione - p. 40, 55

Unione - p. 40, 55

Bozza

Appendice C

Voci di menù

File Voce di menù - p. 7

Apri - p. 7

Modifica Voce di menù (*Orig: Edit*) - p. 4, 5, 11, 31, 52, 103

Posizionamento (*Orig: Placement*) - p. 52

Preferenze (*Orig: Preferences*) - p. 4, 5, 11, 31, 103, 104

Generale Menù Preferenze (*Orig: General*) - p. 4, 5, 11, 103

Editor Scheda - p. 5, 11

Finestra di Output Scheda - p. 5

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

Macro Scheda (*Orig: Macro*) - p. 103

Part design Menù Preferenze (*Orig: Part design*) - p. 31

Visualizzazione della figura Scheda (*Orig: Shape view*) - p. 31

Visualizza Voce di menù - p. 5, 30, 31, 43

Barre degli Strumenti - p. 5

Axonometric - p. 31

Isometrica - p. 31

Macro - p. 5, 7

Vista Ortografica - p. 30

Vista Prospettica - p. 30

Viste Standard - p. 30, 31

Origine degli assi - p. 43

Pannelli - p. 5

Console Python - p. 5

Report - p. 5

Appendice D

Oggetti di FreeCAD

Angle Proprietà - p. 44

Angle1 Proprietà - p. 103, 105

Angle2 Proprietà - p. 103

Angle3 Proprietà - p. 103, 105

Base Proprietà - p. 55, 57, 65

Bordo Componente Geometria (*Orig: Edge*) - p. 33, 34, 37, 38, 40

Center Proprietà - p. 39

Cerchio Geometria di tipo "Part::GeomCircle" (*Orig: Circle*) - p. 38

Contorno Componente Geometria (*Orig: Wire*) - p. 33, 34, 37, 64, 76, 77

Curva Componente Geometria (*Orig: Curve*) - p. 32, 38, 40

Curva Bezier Geometria di tipo "Part::GeomBezierCurve" (*Orig: B-Spline curve*) - p. 38

Curva B-Spline Geometria di tipo "Part::GeomBSplineCurve" (*Orig: B-Spline curve*) - p. 38

Ellisse Geometria di tipo "Part::GeomEllipse" (*Orig: Ellipse*) - p. 38

Extrude funzione - p. 65

Faccia Componente Geometria (*Orig: Face*) - p. 33, 34, 37, 40, 46–48, 64, 67, 76, 77

Forma topologica (*Orig: TopoShape*) - p. 33, 37–39, 65, 76

Guscio Componente Geometria (*Orig: Shell*) - p. 33, 34, 40

Height Proprietà - p. 44

Iperbole Geometria di tipo "Part::GeomHyperbola" (*Orig: Hyperbola*) - p. 38

Label Proprietà di un oggetto - p. 109, 113

Linea Geometria di tipo "Part::GeomLine" o "Part::GeomLineSegment" (*Orig: Line*) - p. 38

Maglia Concetto Geometrico (*Orig: Mesh*) - p. 34

Mesh Proprietà - p. 113

Oggetto documento Elemento della vista 3D (*Orig: Document object*) - p. 37, 39

Orientation Proprietà di un oggetto - p. 76

Parabola Geometria di tipo "Part::Parabola" (*Orig: Parabola*) - p. 38

Part::Box Geometria - p. 42, 47, 75

Part::Cut Op. booleana - p. 55, 57, 58

Part::Cylinder Geometria - p. 43, 47

Part::Feature Geometria - p. 39, 65, 70, 73, 74

Part::Fuse Op. booleana - p. 55, 57, 58

Part::Loft Strumento - p. 72–74

Part::MultiCommon Op. booleana - p. 55, 58

Part::MultiFuse Op. booleana - p. 55, 59, 97, 105

Part::Sphere Geometria - p. 47, 80, 103

Part::TopoShape Geometria - p. 39

Piano Geometria di tipo "Part::GeomPlane" (*Orig: Plane*) - p. 38

Placement Proprietà di un oggetto - p. 49–51, 53, 65, 73, 97, 104

Angolo (*Orig: Angle*) - p. 50, 51

Asse - p. 50, 51

Posizione - p. 50, 51

Primitiva Solido di base (*Orig: Primitive*) - p. 40, 42, 43, 46, 47, 49

Punto Componente Geometria (*Orig: Point*) - p. 32, 38, 40

Radius Proprietà - p. 39, 44, 80, 103, 105

Refine Proprietà - p. 56, 57

Revolve funzione - p. 70, 71

Sections Proprietà - p. 74

Shape Proprietà - p. 39, 65, 70, 74, 113

ShapeColor Proprietà - p. 81

Shapes Proprietà - p. 56, 59, 97

Solido altro nome di Geometria (*Orig: Solid*) - p. 29, 33, 34, 37, 40, 42, 43, 46, 47, 75–77

Superficie Componente Geometria (*Orig: Surface*) - p. 32, 33, 38, 40

Superficie Bezier Geometria di tipo "Part::GeomBezierSurface" (*Orig: Bezier surface(s)*) - p. 38

Superficie B-Spline Geometria di tipo "Part::GeomBSplineSurface" (*Orig: B-Spline surface(s)*) - p. 38

Tassellazione Concetto Geometrico (*Orig: Tessellation*) - p. 34, 37

Tool Proprietà - p. 55, 57

Transparency Proprietà - p. 81

Vertice Componente Geometria (*Orig: Vertex*) - p. 33, 37, 40, 46, 47

Vettore Componente Geometria - p. 30, 51, 52

ViewObject Proprietà - p. 31, 80, 81

Visibility Proprietà - p. 81