Guida allo scripting in FreeCAD

Autore: Carlo Dormeletti

Versione di riferimento di **FreeCAD**: **0.18**
Versione della guida: **0.16**
Data di stampa: **1 marzo 2020**

Licenza

Disclaimer

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

Ringraziamenti

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

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

Un ultimo ringraziamento a TEX e a LATEX 2_ε senza i quali questa guida non sarebbe così ricca di riferimenti, note, tabelle indici e quanto altro ancora distinguono un buon prodotto grafico da un file prodotto con un word processor.

Contatti, segnalazione di errori e comunicazioni

Per errori, omissioni o problemi nella presente guida, posso essere contattato tramite 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**.

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.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco listati

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

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

Convenzioni usate nel testo

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

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

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

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

I riquadri colorati vengono usati per alcuni scopi:

Questo riquadro rosso viene utilizzato per gli esercizi proposti.

Questo riquadro arancio viene utilizzato per i programmi e le righe di codice.

Il riquadro grigio viene utilizzato per mostrare l'output del programma.

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 sul SO Linux.

Immagini e copie delle schermate

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

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

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 quelli che potete vedrete in **FreeCAD**.

Comunque migliorano la leggibilità del codice e per questo sono stati utilizzati.

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 comodità e per esigenze di impaginazione, nonché per evitare che l'inserimento del codice, rompesse il filo del discorso, i listati dei programmi sono stati raccolti in appendice, e ne viene dato il riferimento all'interno del testo.

Introduzione

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

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

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

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

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

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

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

I motivi di questa confusione sono molteplici:

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

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

https://www.freecadweb.org/

Installazione di FreeCAD

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

Come installarlo dipende in parte dal sistema operativo che usate:

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

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

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

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

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

Capitolo 1

Primi passi

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

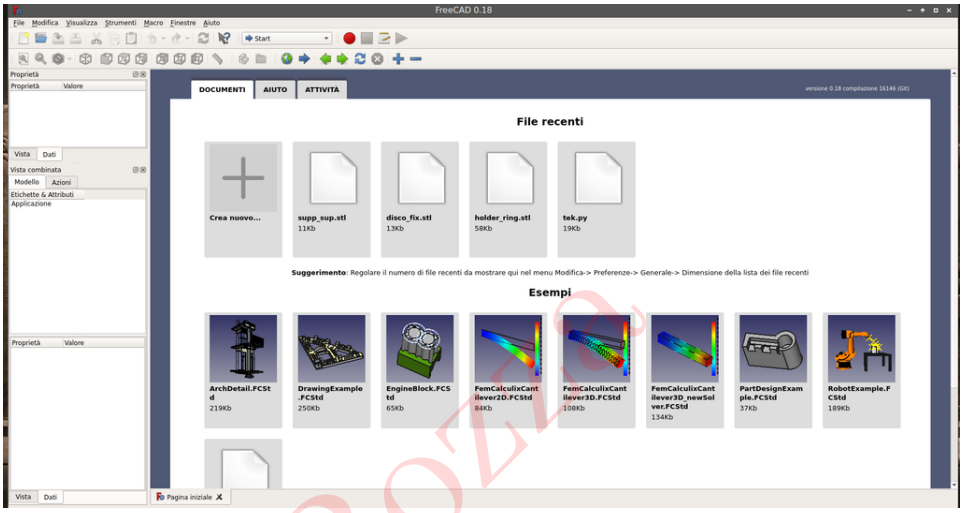


Figura 1.1.: L'interfaccia Utente al primo avvio

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

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

La figura 1.2 mostra una schema dell'interfaccia utente.

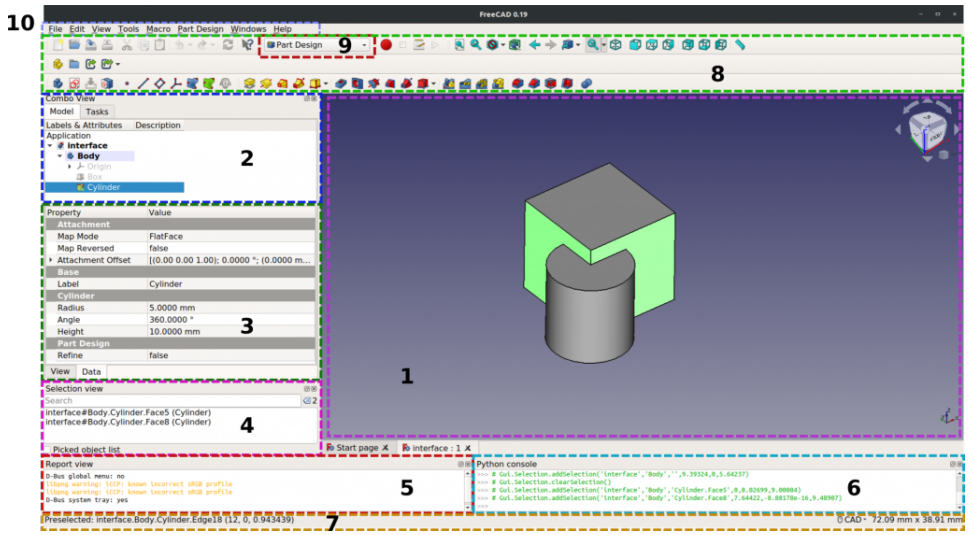


Figura 1.2.: L'interfaccia Utente

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

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

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

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

spunta se non già presente su:

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

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

Nella 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.

Parliamo ora di **Python**, che è un linguaggio di scripting, cioè viene interpretato e non compilato.

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

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

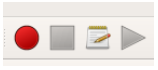
Il vantaggio maggiore di questo tipo di approccio è che una volta compilato, il porgramma ha una vita a se stante e non dipende dalla presenza del compilatore sulla macchina dove viene fatto girare.

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

Il vantaggio può sembrare minimo, ma potete avere aperto nel computer un editor che vi permette di modificare il programma e una console (linea di comando nella terminologia Windows) in cui lanciare il programma.

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

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



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

Niente vieta di usare un vero e proprio IDE per Python esterno, **FreeCAD** è abbastanza

intelligente da vedere che il programma caricato è stato modificato da un programma esterno e da chiedervi se volete ricaricarlo, in definitiva è solo questione di comodità perché un'IDE può addirittura mostrarvi l'errore immediatamente al momento della battitura almeno in ordine agli "errori di sintassi" che sono quelli più facilmente evitabili, specie se la questione è un "punto e virgola" al posto di un "due punti" o un "punto" al posto di una "virgola".

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

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

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

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

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

Capitolo 2

Scripting

2.1. Breve introduzione a Python

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

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

Una piccola ma doverosa nota: in **FreeCAD** 0.18 l'interprete Python è il 3.x quindi la documentazione di riferimento è quella relativa alla versione 3.x di Python, in genere l'AppImage indica nel nome anche la versione di Python che è contenuta, essendo terminato il supporto per Python 2.x naturalmente la versione di riferimento è la 3.x, attenzione però che in giro ci sono molte guide che si riferiscono ancora a Python 2.x che ha delle piccole ma importanti differenze con Python 3.x, alcune differenze sono importanti altre solo delle cose cosmetiche, che comunque possono generare "errori di sintassi".

2.1.1. La sintassi

2.1.2. Indentazione

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

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

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

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

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

Le cose importanti sono:

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

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

2.1.3. Spazi e righe bianche

Altro concetto importante, almeno nella forma sono le spaziature, quando scriviamo un codice, cerchiamo di evitare di mettere gli spazi a caso, ad esempio attorno al segno di uguale (=) nell'assegnazione del valore di una variabile, serve un solo spazio a destra e un solo spazio a sinistra, la stessa cosa vale per gli operatori (+,-,*,/).

Per le righe bianche, in genere è meglio usare una certa omogeneità, l'importante comunque è la "leggibilità" del programma, per fare questo sono importanti i commenti

2.1.4. Commenti

In Python come tutti i linguaggi di programmazione sono importanti i "commenti", cioè le righe che l'interprete non interpreta.

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

L'utilità dei commenti si rivela se ad esempio dopo qualche tempo (magari anni) riprendete in mano uno script, quasi sicuramente quello che vi sembrava chiarissimo al

momento della redazione dello script, risulterà decisamente oscuro, se documentate bene le cose anche a distanza di tempo potete riusare il codice perché sapete cosa significa ogni parametro e magari anche ogni porzione di codice.

2.1.5. Le `docstring`

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

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

2.1.6. Variabili

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

I nome delle variabili in Python seguono alcune convenzioni:

- se il nome è scritto "TUTTO IN MAIUSCOLO" in genere si usa per le "costanti", almeno dal punto di vista logico, anche se in Python "tutto è un oggetto" e vere e proprie "costanti", cioè cose che una volta dichiarate non possono più essere cambiate pena un "errore di sintassi" non esistono
- se il nome comincia con un carattere di "_" in genere è considerata una variabile "locale" che è "BENE" non sia usata all'esterno dell'ambito in cui è definita ed usata.
- le variabili scritte in questo modo `__nome__`, (notate che nonostante la grafica non esistono spazi tra i _e "nome") hanno un significato particolare ed è BENE non siano usate se non si sa a cosa servono.
- nel caso ci siano ambiguità con parole chiave del linguaggio potete usare un solo "_" dopo il nome della variabile per eliminare l'ambiguità
- usate nomi di variabili più lunghi di tre caratteri e usate liberamente i "_" all'interno del nome, ad esempio `nome_della_variabile` è perfettamente legale e significativo.

2.1.7. I metodi e le funzioni

Questa porzione di codice:

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

Mostra un **metodo** altrimenti chiamata **funzione**.

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

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

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

2.2. Antipasto

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

Una piccola nota sul metodo, forniremo più avanti, uno **schema base** di uno script **FreeCAD**, in pratica lo potete pensare come uno schema "vuoto", ma con i riquadri già disegnati.

Cominciamo con un programma che fa poche cose e aumentiamo mano a mano la complessità.

Qui sotto vedete un esempio minimale di codice.

```
import FreeCAD

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

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

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

Assegna poi alla variabile `DOC_NAME` il nome del documento.

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

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

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

La "magia" viene fatta da:

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

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

- Crea un nuovo documento chamato con il nome contenuto nella variabile `DOC_NAME`.
- Lo rende il documento attivo.
- assegna alla varibile `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 appena appena le cose:

```
import FreeCAD

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

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


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

else:

   clear_doc()
```

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

2.2.1. Le direttive import

Le righe di codice mostrate sopra fanno parte dello **schema base** a cui abbiamo accennato e che viene riportato per intero nell'appendice a pagina 50.

Se osservate quel codice nelle prime righe vedete delle direttive **import**, con scritture leggermente diverse.

```
import FreeCAD
```

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

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

Conoscendone i pericoli possiamo però usare questa scrittura:

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

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

Analogamente importiamo dal modulo `math` alcune cose che torneranno utili come `pi` che è il valore di 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**.

Cominciamo da sinistra troviamo `FreeCAD` seguito da un punto e da `newDocument` stiamo invocando il metodo `newDocument` contenuto nel modulo `FreeCAD` usando il contenuto della variabile `DOC_NAME`

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

In questa porzione di codice vediamo la dishiarazione del metodo `miaScatola`:

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

Questo metodo possiede 3 paramteri:

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

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

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

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

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

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

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

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

2.2.3. I cicli

Analizziamo la funzione:

```
20  def clear_doc():
21      """
22      Clear the active document deleting all the objects
23      """
24      for obj in DOC.Objects:
25          DOC.removeObject(obj.Name)
26
```

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

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

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

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

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

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

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

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

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

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

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

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

2.2.4. Le condizioni

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

Le scelte in genere sono basta su alcune condizioni:

- **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 la ho evidenziato tre parole: **se** ... **allora** ... **altrimenti**.

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

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

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

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

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

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

```
if DOC is None:

if var >= 0:

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

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

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

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

Il secondo esempio controlla che il valore di `var` sia ¿= a 0.

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

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

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

Vediamo nello schema a blocchi in figura 2.1 a pagina 16:

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

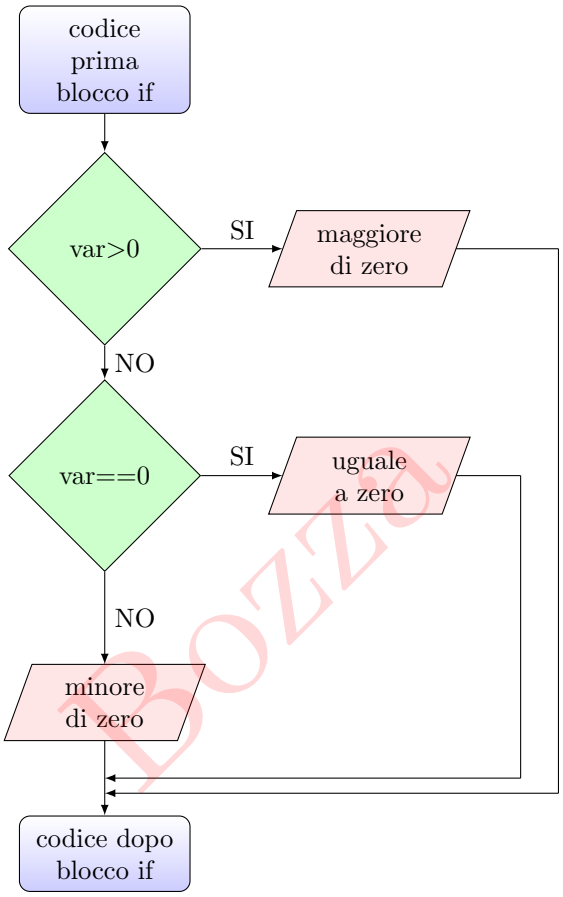


Figura 2.1.: schema a blocchi condizioni if

Capitolo 3

Creazione di Oggetti 3D

3.1. Gli Oggetti 3D

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

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

Usando lo scripting, abbiamo anche la scelta di usare molte vie, ho scelto di usare principalmente un modo, quello di usare il metodo `DOC.addObject()`, alla prova dei fatti questo metodo è risultato meno problematico di altri, e meno involuto, questo metodo crea direttamente nel documento aperto, l'oggetto voluto, senza poi doverli esplicitamente visualizzare invocando il metodo **show** si fa con altri metodi.

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

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

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

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

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

plesso".

Faremo spesso riferimento a "listati" dei programmi che sono presenti nell'Appendice A, se li avessimo riportati nel testo, avrebbero occupato a volte più di una pagina e avrebbero interrotto il filo del discorso, forniremo all'interno del testo le porzioni di codice rilevanti e le aggiunte graduali.

Questa guida andrebbe seguita scrivendo (o scaricando il codice quando verrà reso disponibile) inizialmente lo **schema base**, Listato A.1 a pagina 50, salvandolo ed aggiungendo via via le porzioni di codice illustrate nel testo, risulterà utile salvare le versioni intermedie con il nome che preferite, in modo da poterle riprendere poi se necessario.

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

3.1.1. Spazio tridimensionale

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

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

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

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

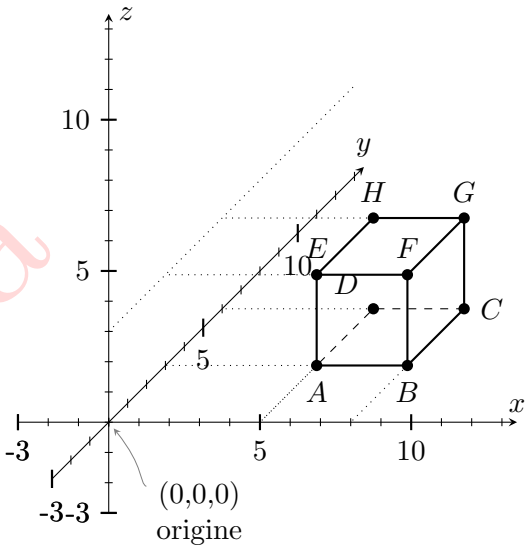
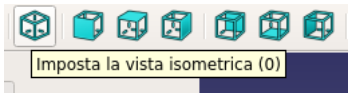


Figura 3.1.: Lo spazio 3D

3.1.2. Proiezioni e viste

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



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

Standard ⇒ **Axonometric**.

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

3.1.3. I Vettori

Un **vettore** in **FreeCAD** lo possiamo concepire come un contenitore che contiene 3 elementi, in genere le coordinate 3D di un punto nello spazio definite come valori in virgola mobile (float) di X, valore di Y e valore di Z in questo modo `Vector(val_x, val_y, val_z)`, è un concetto abbastanza comune a tutti i modellatori 3D.

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

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

3.1.4. Topologia

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

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

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

3.1.5. Geometria e Vista

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

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

Nello **schema base** a pagina 50 potete vedere un utilizzo della parte **ViewObject** nel metodo **setview()**.

3.1.6. Il riferimento di costruzione

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

Nella tabella 3.1, elenchiamo i punti di riferimento dei vari oggetti.

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	spigolo di Xmin Zmin

Tabella 3.1.: punti di riferimento

3.1.7. Il primo oggetto

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

```
49  def cubo(nome, lung, larg, alt):
50      obj_b = DOC.addObject("Part::Box", nome)
51      obj_b.Length = lung
52      obj_b.Width = larg
53      obj_b.Height = alt
54
55      DOC.recompute()
56
57      return obj_b
```

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

Il metodo prevede 4 parametri:

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

estendiamo il nostro schema base aggiungendo la parte sopra, e un paio di altre righe, ottenendo il programma **cubo di prova** a pagina 52 che sorprendentemente e magicamente caricato e lanciato in **FreeCAD**, visualizzerà un cubo (questo sarà proprio un cubo) nella **vista 3D**.

Ora complichiamo la faccenda, il cubo è fatto, ma dov'è, nel senso in che **posizione** si trova?

Questo è uno dei segreti di **FreeCAD**, e uno dei suoi aspetti più complicati, della teoria e delle particolarità parleremo più avanti a tempo debito.

Per avere una piccola idea di dove si trova l'oggetto, possiamo attivare la visualizzazione dell'origine nella finestra grafica, usando il menu **Visualizza** ⇒ **Origine degli assi**, oppure usando aggiungendo al metodo **setview()**

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

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

ora potete giocare con i valori della riga:

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

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

3.1.8. Posizionamento

Dalla tabella 3.1 a pagina 20 potete vedere che il cubo non usa il centro come riferimento di costruzione.

In una costruzione modulare, in genere, per evitare problemi, è necessario adottare una tecnica abbastanza rigorosa.

Una delle tecniche "salvavita" è quella di riportare tutti gli oggetti creati in una posizione convenzionale.

La posizione convenzionale più comoda è quella con la faccia inferiore centrata rispetto `Vector(0, 0, 0)`

Per fare questo è necessario modificare la posizione del cubo creato.

Il **vettore** è anche usato da **FreeCAD** in operazioni come la rotazione per contenere ad esempio gli angoli di rotazione per i tre assi X, Y, Z oppure per definire usando i valori (0 ed 1) l'asse a cui si riferisce una certa operazione.

Vedremo fra poco questi due utilizzi.

La scelta di lavorare attraverso esempi impone che al momento di introdurre nuovi "costrutti" dobbiamo affrontare anche qualche argomento di teoria, per cui interrompiamo momentaneamente la spiegazione per introdurre le direttive *import* di Python

La proprietà Placement

Per modificare la posizione abbiamo come al solito diversi metodi, secondo la mia opinione il metodo che porta minori problemi è quello di impostare direttamente la proprietà **Placement**.

La proprietà **Placement** può essere espressa in diversi modi, una delle "scritture" è:

```
FreeCAD.Placement(
    Vector(pos_X, pos_Y, pos_Z),
    FreeCAD.Rotation(Vector(0,0,0), 0)
    )
```

La trovate abbastanza spesso in giro, per cui spieghiamola con un certo dettaglio.

Riflette anche le proprietà che trovate nella **vista combinata** nella parte **editore delle proprietà** come potete vedere nell'immagine 3.2 a pagina 22 che visualizza la vista **Dati**.

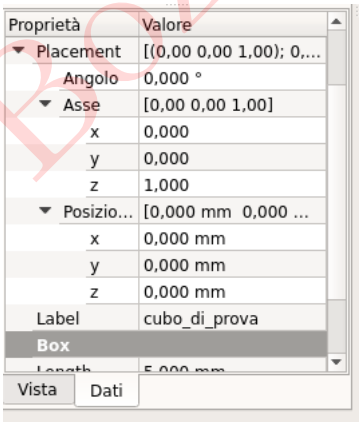


Figura 3.2.: L'editor delle proprietà

L'ordine delle voci varia rispetto a quello del codice, vediamo comunque che **Placement** possiede una freccia sulla sinistra che se premuta espande la vista come nella figura 3.2 e mostra 3 sotto proprietà:

Nome	Descrizione
Angolo	angolo di rotazione in gradi
Asse	asse di riferimento **Vettore** con valori 0 o 1 che selezionano l'asse di riferimento, sono ammessi più valori
Posizione	Posizione X Y e Z

Tabella 3.2.: proprietà Placement

Sorprendentemente questa "scrittura" non è quella che otteniamo aggiungendo questa linea al codice dello **schema base** a pagina 50.

```
print(obj.Placement)
```

subito prima della chiamata alla funzione `setview()`.

Se lanciamo il programma vedremo apparire (se non sono presenti errori) nella **finestra dei rapporti**.

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

Possiamo facilmente riconoscere un **Vettore** chiamato `Pos`, seguito da un secondo **Vettore** chiamato `Yaw-Pitch-Roll`.

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

Nome	Descrizione	Angolo
Yaw (imbardata)	rotazione rispetto a Z	**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 3.3.: Angoli di Eulero

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

Il problema è che raramente in **FreeCAD** un oggetto viene creato con il centro geometrico a X = 0, Y = 0 e Z = 0, o se vogliamo scriverlo come **Vettore** `Vector(0, 0, 0)`, si renderà pertanto necessario modificare il suo posizionamento, modificando la proprietà **Placement**, che però non viene conservata ad esempio quando si sposta un

oggetto in un'altra posizione.

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

Capitolo 4

Modellazione avanzata

Un oggetto può essere creato in molti modi, **FreeCAD** offre molte funzioni per creare oggetti anche molto complessi, il suo arsenale di strumenti è potente e versatile.

In questo capitolo analizzeremo alcune operazioni sugli oggetti, che vanno padroneggiate in quanto sono le tecniche fondamentali per la creazione di oggetti complessi.

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

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

4.1. Le operazioni booleane

Creare oggetti singoli può essere un utile esercizio, ma lo scopo finale è quello di creare oggetti complessi.

Partendo da semplici geometrie e operando sulle stesse utilizzando le **operazioni booleane**, che nonostante il nome strano sono relativamente semplici.

Queste funzioni sono le basi per la modellazione, chi ha studiato gli insiemi, troverà delle analogie, e non è un caso.

Vediamo le **operazioni booleane**:

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

Cominciamo ad estendere lo **schema base** a pagina 50 aggiungendo le righe presenti nel Programma 1 che trovate a pagina 26.

Invochiamo il metodo creato, in questo modo per creare un oggetto **Part::Cylinder**, in questo modo:

```
59  def base_cyl(nome, ang, rad, alt ):
60      obj = DOC.addObject("Part::Cylinder", nome)
61      obj.Angle = ang
62      obj.Radius = rad
63      obj.Height = alt
64
65      DOC.recompute()
```

Programma 1: metodo cilindro

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

Illustriamo i "parametri" che forniamo al metodo:

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

Ora dovremmo avere un programma che assomiglia al listato A.3 a pagina 54

che una volta lanciato dovrebbe mostrare nella **vista combinata**, qualcosa che assomiglia alla figura 4.1a.

Non è proprio una scultura del Canova, ma già possiamo notare alcune cose:

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

Niente di strano in quanto pienamente in accordo a quanto abbiamo elencato nella tabella 3.1 a pagina 20

4.1.1. 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 2 a pagina 27.

Questa porzione di codice la inseriamo alla riga 58 del nostro schema base, subito dopo al metodo `base_cyl`.

Descriviamo brevemente le proprietà:

- **Base** è la geometria a cui dobbiamo aggiungere il secondo oggetto.
- **Tool** è l'oggetto da aggiungere.

```
69  def fuse_obj(nome, obj_0, obj_1):
70      obj = DOC.addObject("Part::Fuse", nome)
71      obj.Base = obj_0
72      obj.Tool = obj_1
73      obj.Refine = True
74      DOC.recompute()
75
76      return obj
```

Programma 2: Unione

- l'uso della proprietà **Refine** settata a `True` che cerca di "affinare" l'oggetto eliminando le facce inutili e le cuciture.

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

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

che posizioniamo "ovviamente" dopo aver creato gli oggetti, quindi subito dopo la riga 71 che crea l'oggetto `primo cilindro` invocando il metodo `base_cyl`.

Lanciando il programma apparentemente non otteniamo nulla, infatti il risultato è la figura 4.1b se non 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 3 a pagina 28.

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

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

Programma 3: Unione multipla

4.1.2. 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 4 a pagina 28.

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

Programma 4: Sottrazione

Inseriamo le righe del Programma 4 alla riga 66 del nostro schema base, subito dopo al metodo `base_cyl`.

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

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

Mettiamo il segno di `#` davanti all'invocazione di `fuse_obj`, copiamo la riga sopra e lanciamo il programma, ottenendo come risultato è la figura 4.1c.

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 per **Part::Cut** l'ordine è importante.

4.1.3. Intersezione

L'Intersezione, estrae la parte comune delle geometrie, creando un oggetto **Part::MultiCommon**.

Creiamo questo metodo come nel Programma 5 a pagina 29.

```
def int_obj(nome, obj_0, obj_1):
    obj = DOC.addObject("Part::MultiCommon", nome)
    obj.Shapes = (obj_0, obj_1)
    obj.Refine = True
    DOC.recompute()

    return obj
```

Programma 5: Intersezione

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

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

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

Mettiamo il segno di `#` davanti all'invocazione di `cut_obj`, copiamo la riga sopra e lanciamo il programma, ottenendo come risultato è la figura 4.1d: .

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

Trovate il listato dell'esempio completo con il nome **Operazioni booleane - esempio completo** nell'Appendice A.4 a pagina 56.

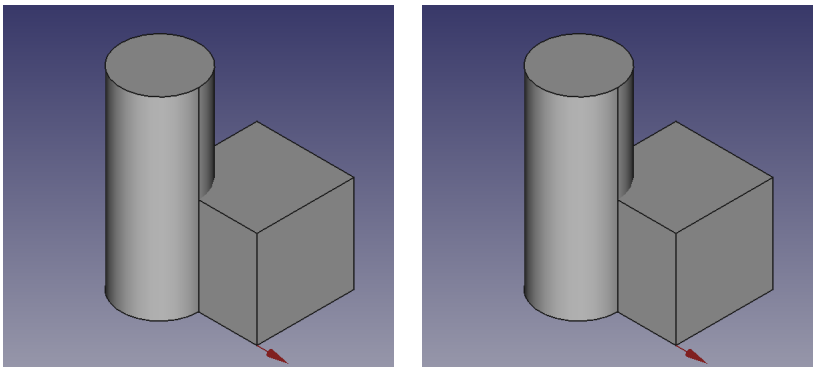
4.2. Estrusione

Finora abbiamo creato direttamente gli oggetti usando le forma base, per forme complesse esiste una tecniche potente chiamata "estrusione", che usa la funzione **extrude**.

Questa tecnica consiste nel creare una forma in 2D e poi estruderla in 3D, più o meno come fa una stampante 3D.

Per utilizzare questa tecnica dobbiamo fare uso di alcuni concetti.

Non ci muoveremo dal modulo Part.



(a) cubo e cilindro. (b) Unione

(c) Sottrazione. (d) Intersezione.

Figura 4.1.: Le operazioni Booleane.

Creiamo questo metodo come nel Programma A.5: Estrusione - esempio completo a pagina 59.

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

```
44  def reg_poly(center=Vector(0, 0, 0), sides=6, dia=6,
45               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 7 alla linea 12 dalla lista completa dei parametri con una spiegazione sommaria di quello che fanno.

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

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

- `outer` = 0 = apotema, 1 diametro esterno, cioè specifica che di che tipo è la misura fornita con il parametro `dia`

Il parametro `outer` merita una spiegazione in merito alla parola apotema, L'apotema è il **raggio della circonferenza interna** del poligono, nel programma è usato in modo improprio. [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, questo elenco deve essere ordinato in senso orario o antiorario, ma comunque ordinato, stiamo passando l'elenco dei vertici di un polygono e le linee non devono "incrociarsi".

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

Trasformiamo l'oggetto poligono in un oggetto di tipo **Polilinea** (**Wire**)

[1] Non 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.

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

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

Questa funzione è stata creata per una funzione eminentemente pratica, creare esagoni, lo si capisce 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.

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.

Internamente sono usate le formule per trasformare l'apotema in raggio del cerchio esterno, e i diametri vengono trasformati in raggi moltiplicando per 0.5. [2]

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.

Ottenuta con questi metodi la parte di piano possiamo finalmente **estruderla**, per farlo scriviamo un metodo:

```
91  def dado(nome, dia, spess):
92      polyg = reg_poly(Vector(0, 0, 0), 6, dia, 0, 0)
93
94      nut = DOC.addObject("Part::Feature", nome + "_dado")
95      nut.Shape = polyg.extrude(Vector(0, 0, spess))
96
97      return nut
```


Analizziamolo in dettaglio:

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

Questa linea di codice si occupa di recuperare la nostra parte di piano, chiamando la funzione VNMreg_poly

[2] Nel programma ci sono 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**, cioè un **numero con la virgola** per 2 che è un **intero** l'interprete o il compilatore convertono l'operazione in una divisione di **interi**, restituendo un **intero** invece che un **float**, facendo diventare poi matti a trovare l'errore nel codice, usando la moltiplicazione si moltiplica un **float** per un altro valore **float** evitando ogni possibile errore di conversione implicita.

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

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

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

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

```
95        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)`, ricordardando che l'esagono era costruito con il centro in (0, 0, 0), in pratica estrudiamo in direzione Z di `spess`.

Dovreste ottenere il risultato mostrato nella figure 4.2a a pagina 34.

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

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

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

Una volta averlo creato noterete che nel'**editore delle proprietà** della **vista combinata**, 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 fate un errore dovete rifare tutta la trafila, a meno di non aver salvato le forme intermedie.

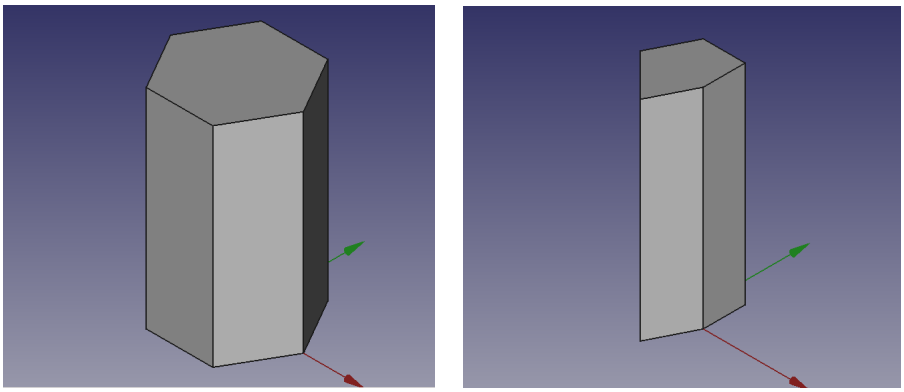
Usando lo scripting, basta editare un paio di parametri e rilanciare il programma, decisamente più comodo.

Nel caso di esempio l'estrusione è stata fatta con un oggetto semplice, se componete profili complessi, ovviamente il risultato sarà più complesso, un esempio semplice semplice è dato dal metodo `estr_comp` presente nel listato citato sotto che permette di ottenere il prisma mostrato nella nella figure 4.2b a pagina 34.

La parte rilevante è la definizione del profilo base da cui viene estruso il prisma, basta creare una lista ordinata di punti, in questo modo:

```
100  def estr_comp(nome, spess):
101      vertex = ((-2,0,0), (-1, 2, 0), (1, 2, 0), (1, 0, 0),
102                (0, -2, 0), (-2,0,0))
```

In realtà abbiamo creato una **tupla** contenente tante **tuple** quante sono i punti del



(a) Dado. (b) Prisma.

Figura 4.2.

poligono.

Ci sono alcune regole da rispettare:

- 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).

Per contenere i punti invece di usare una **lista** abbiamo usato una **tupla** che in pratica è una lista immutabile, nel senso che una volta creata non può essere modificata.

Il suo utilizzo deriva dal fatto che risparmia memoria, notate che la **tupla** è identificata come abbiamo già accennato nella sezione **Le condizioni** a pagina 2.2.4 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

Notate che non accetta una **lista di liste**, per cui comunque i punti vanno racchiusi tra parentesi tonde.

L'utilizzo della funzione 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:

```
104     obj = Part.makePolygon(vertex)
105     wire = Part.Wire(obj)
106     poly_f = Part.Face(wire)
```

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

Trovate il listato dell'esempio completo con il nome **Estrusione - esempio completo** nell'Appendice A.5 a pagina 59.

4.3. Rivoluzione

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

Una funzione molto potente è la 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.

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

Fino a qui nulla di eccezionale, e nemmeno nella parte finale del codice:

```
125     obj = DOC.addObject("Part::Feature", nome)
126     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 comprendere sono i parametri della funzione **revolve**, analizziamoli in dettaglio:

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

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

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

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

```
121         angle = 180 # Rotation angle
```

Questo è ovviamente l'angolo di rotazione 180°

Lanciamo l'esecuzione attraverso questa linea:

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

Il risultato sarà una cosa come quella in figura 4.3.

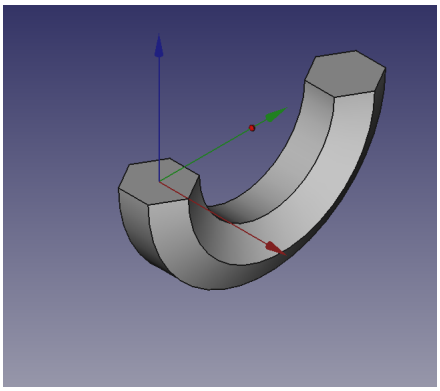


Figura 4.3.: Rivoluzione

Al solito nulla di eccezionale, solo un esempio, l'unico limite è la vostra fantasia.

Trovate il listato dell'esempio completo con il nome **Rivoluzione - esempio completo** nell'Appendice A.6 a pagina 59.

Capitolo 5

La componente Vista

Ai più acuti, occhi di falco non sarà sfuggito nella figura 4.3, la presenza di un puntino rosso.

A chi ha letto il listato non sarà sfuggita la presenza di una riga nel codice del metodo `manico` che non abbiamo citato:

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

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

La funzione è definita come:

```
43  def point(nome, pos, pt_r = 0.25, color = (0.85, 0.0, 0.00),
44            tr = 0):
45      """draw a point for reference"""
46      rot_p = DOC.addObject("Part::Sphere", nome)
47      rot_p.Radius = pt_r
48      rot_p.Placement = FreeCAD.Placement(
49          pos, FreeCAD.Rotation(0,0,0), Vector(0,0,0))
50      rot_p.ViewObject.ShapeColor = color
51      rot_p.ViewObject.Transparency = tr
52
53      DOC.recompute()
```


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, (che possiedono una definizione di default) che lo rendono abbastanza flessibile.

Analizziamoli:

- `pt_r`: che è il raggio della sfera, infatti è passata alla proprietà **Radius** è 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 alle proprietà **ViewObject**, di cui abbiamo già accennato a pagina 19.

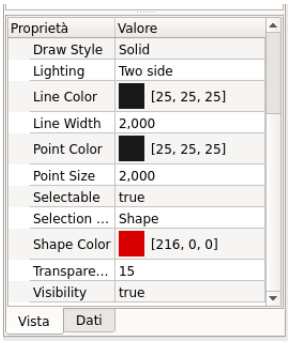


Figura 5.1.: La linguetta Vista

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

Descriviamo sommariamente la parte **ViewObject** che riflette le proprietà che trovate nella linguetta **Vista** di ogni oggetto, nell'immagine 5.1 a pagina 38, della **vista combinata** nella parte **editore delle proprietà**.

Abbiamo già accennato che ogni oggetti comprende un parte relativa alla **Geometria** e una parte **Vista**, ricordiamo che la parte **Geometria** possiamo modificarla nella linguetta **Dati** come abbiamo già visto nell'immagine 3.2 a pagina 22.

Qui abbiamo almeno tre proprietà che potrebbero tornare utili:

- **ShapeColor** contiene il colore dell'oggetto,
- **Trasparency** 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 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 manualli espressi come tripla di valori interi compresi tra $0 \ldots 255$, per ottenere i valori da passare basta semplicemente dividere il valore RGB per 255.

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

Capitolo 6

Tecniche avanzate

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.

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

Alcune tecniche potrebbero sembrare una inutile complicazione, ma se il progetto diventa complicato, dopo aver cominciato a modellare, diventa necessario modificare gran parte del codice e diventa un problema che porta via molto tempo.

Teniamo comunque conto che stiamo modellando per poi stampare in 3D con tutti i problemi della stampa 3D come 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.

6.1. Riferimenti di costruzione

Il riferimento di costruzione è importante ne abbiamo già parlato nella sezione 3.1.6, ma avevamo dovuto solo accennarlo, perché non avevamo gli strumenti per comprendere appieno il suo utilizzo.

Come abbiamo avuto modo di trattare, le geometrie sono costruite prendendo come punto di partenza un punto preciso che purtroppo non è il medesimo per ogni geometria, la cosa diventa complicata quando vogliamo costruire cose complesse, perché non avendo un punto comune dobbiamo giostrarci, tra i vari riferimenti.

Concettualmente parlando, il miglior punto di riferimento è l'origine (0,0,0) per fortuna alcune geometrie hanno già il loro riferimento di costruzione in quel punto.

Introduciamo questa parte di codice:

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

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

    DOC.recompute()

    return obj_b
```

Abbiamo leggermente modificato il codice già visto, introducendo nella definizione del metodo un paramtero opzionale `cent` a cui abbiamo assegnato un valore di default di `False`, in questo modo possiamo riutilizzare tutto il codice precedente ma se ci serve centrare il cubo sull'origine basta aggiungere alla chiamata un `True` alla fine in questo modo:

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

per ritrovarci il cubo centrato in (0,0,0),

se ora costruiamo un cilindro in questo modo:

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

vedremo che stavolta il cubo viene perfettamente in asse con il cilindro.

per cui se scriviamo:

```
obj_f = cut_obj("cubo_cyl", obj1, obj2)
```

vedremo che il nostro parallelepipedo avrà un buco dove c'era il cilindro.

Però, c'è sempre un però, se osserviamo le proprietà **Placement** dell'oggetto scrivendo questa cosa nel file:

```
print("Oggetto finale = ", obj_f.Placement)
```

Leggiamo questo messaggio nella **finestra dei rapporti**:

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

e non ci sembra strano, se però aggiungiamo questa linea:

```
print("Cubo Base = ", obj1.Placement)
```

Leggiamo questo messaggio nella **finestra dei rapporti**:

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

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

Modifichiamo il metodo **cubo** in questo modo

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

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

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

    DOC.recompute()

    return obj_b
```


Abbiamo aggiunto una complicazione, il parametro `off_z` che se presente modifica la posizione Z della geometria.

in questo modo possiamo alzare o abbassare il nostro cubo di quanto vogliamo.

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

Lanciando il programma vedremo che viene visualizzato il cubo, e se lo osserviamo da sotto c'è un buco, in corrispondenza del cilindro che abbiamo creato; Se osserviamo nella

finestra dei rapporti vediamo che vengono stampati le proprietà **Placement** dei tre oggetti.

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

Vediamo che la misura di Z del **Cubo Base** è modificata a 10, ma il riferimento `Pos` della proprietà **Placement** dell **Oggetto finale** non è modificato, ogni operazione che coinvolgerà da ora in poi questa geometria sarà riferita a (0, 0, 0), e ne andrà tenuto conto.

Trovate il listato dell'esempio completo con il nome **Riferimento costruzione - esempio completo** nell'Appendice A.7 a pagina 66.

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.

6.2. Modularizzazione

Avendo bene in mente questo, proviamo a realizzare una cosa particolare, una **scatola di sardine**, è un progetto molto complesso.

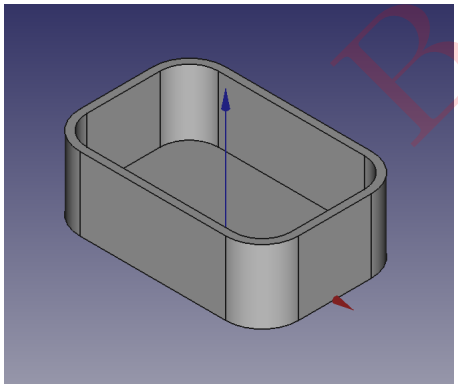


Figura 6.1.: Scatola sardine

Se vogliamo affrontare un modello complesso, dobbiamo usare qualche tecnica, stavolta non per creare le geometrie, ma per gestire 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.

Possiamo anche pensare di trasferire parte di questi metodi in un **Modulo esterno** da importare con una istruzione `import` di Python, e se il progetto è molto complesso renderà la leggibilità del codice maggiore, anche se poi per capire cosa succede dovremo necessariamente andare a leggere il codice nel modulo importato.

Trovate il listato del modulo che porta fuori dal nostro programma alcune routine complesse sotto il nome **Modulo GfcMod - esempio iniziale** nell'Appendice A.8 a pagina 69.

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:

```
def cubo(doc, nome, lung, larg, alt, cent = True, off_z = 0.0):
```

in questo modo viene correttamente usato l'istanza di DOC definita nel programma principale.

Se volete creare un modulo dovete tenere presente alcune cose:

- Il percorso di ricerca dei file comprende la directory in cui c'è il programma chiamante. il file che contiene il modulo deve essere nella stessa directory a meno di non modificare il percorso di ricerca che non è argomento di questa guida.
- il modulo deve avere un nome particolare, il nome non deve contenere caratteri speciali, nemmeno il trattino e il trattino basso, in genere si usa il CamelCase, come nel modulo di esempio.

Nel file chiamante alcune cose devono essere fatte.

Trovate il listato **Sardine - esempio completo** nell'Appendice A.9 a pagina 73.

Notiamo la compattezza del listato, avendo "portato fuori" le routine complesse il listato diventa molto semplice.

Nel programma chiamante, vediamo le righe che compiono l'importazione del modulo

```
import sys
```

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

```
import GfcMod as GFC
```

Queste righe compiono la vera e propria importazione, se ricordate quanto detto, con la riga:

```
import GfcMod as GFC
```

La seconda riga invoca il modulo che abbiamo caricato all'inizio, e sistema le chiamate in modo corretto.

La vera e propria importazione del modulo è nella prima riga, dove il modulo `GfCMod` viene importato con il nome di `GFC`, in questo modo 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, **GFC** non è altro che l'abbreviazione di **Guida di Free CAD**, fantasioso no?

6.3. Modellazione parametrica

Nelle primissime pagine abbiamo detto 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 dobbiamo andare a pescare ogni geometria di base nell'albero di creazione e modificare 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 **Sardine - esempio completo** la parte che crea tutto e lo visualizza sono in pratica le due righe:

```
    obj_f.Tool = obj_int
    obj_f.Refine = True
    DOC.recompute()
```

se commentate con `#` la riga `sardine(...`, non avete più visualizzato nulla e tutti gli oggetti non vengono nemmeno creati, questa è la potenze dello scripting.

Leggiamo la doctring del metodo:

```
def sardine(nome, lung, prof, alt, raggio, spess):
    """sardine
        crea una geometria a forma di scatola di sardine

        Keywords Arguments:
        nome   = nome della geometria finale
        lung   = lunghezza della scatola, lungo l'asse X
        larg   = larghezza della scatole, lungo l'asse Y
        alt    = altezza della scatola, lungo l'asse Z
        raggio = raggio dello "stondamendo" della scatola
        spess  = spessore della scatola
```


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 **Modulo GfcMod - esempio iniziale** quante geometrie vengono modificate vi rendete conto della potenza della modellazione parametrica, ovviamente se avete costruito bene il modello.

In pratica dalla riga 105 alla riga 112 trovate il riferimento al raggio in tutti i "calcoli

preliminari" alla costruzione del modello.

Analizziamo la logica del programma, per creare una "scatola stondata" agli angoli, in soldoni dobbiamo fare due cose, definire due geometrie e sottrarle, una geometria è quella della scatola esterna (`obj`), e una è quella del "buco" interno (`obj_int`).

```
72
73      obj = GFC.cubo_stondato(
74          DOC, nome + "_est", lung, prof, alt, raggio, 0.0)
75
76      lung_int = lung - (spess * 2)
77      prof_int = prof - (spess * 2)
78      alt_int = alt + EPS - spess
79      raggio_int = raggio - spess
80
81      obj_int = GFC.cubo_stondato(
82          DOC, nome + "_int",
```

Come vedete tutte e due sono generate dal metodo `GFC.cubo_stondato(...`, 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 75 a 78 del listato.

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 fine ultimo è quello di riutilizzare buona parte del codice in modo da rendere la "modellazione finale" una cosa veloce.

6.4. Creazione "automatica" di geometrie

Utilizzando l'approccio classico si ricalca in una certa maniera l'abituale modo di procedere 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, ok ripeterlo 100 volte ad esempio per fare 100 fori in una piastra, questo diventa noioso, lungo e complicato.

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


Questa parte di codice presenta una certa ridondanza, sarebbe il candidato perfetto per una semplificazione, del resto ogni geometria differisce solo per il posizionamento e per il nome, ovviamente però avremo necessità di creare geometrie con nomi diversi, possiamo usare la potenza di Python che ci viene in soccorso.

Abbiamo accennato nell'introduzione a Python che per Python tutto è un oggetto, per cui possiamo tranquillamente creare una lista con le posizioni dei centri dei cilindri, ricordiamo che la lista è contenuta tra le parentesi quadre, vediamo il codice come diventa:

```
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         ]
```

Fin qui nulla di eccezionale, sono sparite le variabili `c_c1, c_c2, c_c3, c_c4` ed al suo posto è apparsa una lista chiamata `c_pos`.

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

Questa linea semplicemente sostituisce le vecchie variabili con il riferimento al corrispondente elemento della lista, notare che gli indici delle liste in Python partono da 0, per cui `c_c1` è diventata `c_pos[0]`.

Ora viene la parte che spiega la potenza di Python.

```
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 ci serve perché crea una coppia di dati per ogni elemento della lista, il **numero di posizione** `i` e l'elemento della lista originale `pos` che contiene il nostro vettore di posizione del centro.

Questo ci serve per alimentare correttamente la creazione dell'oggetto che avviene alle linee da 120 a 126.

Notate la composizione del nome dell'oggetto alla linea 121, componiamo una string 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 conca-

tenzazione per le stringhe di Python), ogni oggetto avrà un nome significativo diverso dal precedente.

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

In questo caso erano solo quattro geometrie, ma se fossero state cento, bastava creare una lista contenente le cento posizioni dei centri e non era necessario cambiare una virgola alla parte di creazione geometrie.

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

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

La linea:

```
    obj_int.Shapes = comps
```

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

6.5. Librerie di oggetti

Appendice A

Programmi

Listato A.1.: schema base

```
#
"""sc-base.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2020
   Licence: CC BY-NC-ND 4.0 IT
"""

import FreeCAD
from FreeCAD import Base, Vector
import Part
from math import pi, sin, cos

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)

def setview():
    """Rearrange View"""
    FreeCAD.Gui.SendMsgToActiveView("ViewFit")
    FreeCAD.Gui.activeDocument().activeView().viewAxometric()
```

```
31
32
33 if DOC is None:
34     FreeCAD.newDocument(DOC_NAME)
35     FreeCAD.setActiveDocument(DOC_NAME)
36     DOC = FreeCAD.activeDocument()
37
38 else:
39
40     clear_doc()
41
42
43 # EPS= tolerance to uset to cut the parts
44 EPS = 0.10
45 EPS_C = EPS * -0.5
```

♢ ♢ ♢

Listato A.2.: cubo di prova

```
#
"""cubo-prova.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2020
   Licence: CC BY-NC-ND 4.0 IT
"""

import FreeCAD
from FreeCAD import Base, Vector
import Part
from math import pi, sin, cos

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)

def setview():
    """Rearrange View"""
    FreeCAD.Gui.SendMsgToActiveView("ViewFit")
    FreeCAD.Gui.activeDocument().activeView().viewAxometric()


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

else:

    clear_doc()


```

```
43 # EPS= tolerance to uset to cut the parts
44 EPS = 0.10
45 EPS_C = EPS * -0.5
46
47
48 def cubo(nome, lung, larg, alt):
49     obj_b = DOC.addObject("Part::Box", nome)
50     obj_b.Length = lung
51     obj_b.Width = larg
52     obj_b.Height = alt
53
54     DOC.recompute()
55
56     return obj_b
57
58 obj = cubo("cubo_di_prova", 5, 5, 5)
59
60 setview()
```

♢ ♢ ♢

Listato A.3.: Operazioni booleane - esempio base

```
#
"""ob-ex-tmp.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2020
   Licence: CC BY-NC-ND 4.0 IT
"""

import FreeCAD
from FreeCAD import Base, Vector
import Part
from math import pi, sin, cos

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)

def setview():
    """Rearrange View"""
    FreeCAD.Gui.SendMsgToActiveView("ViewFit")
    FreeCAD.Gui.activeDocument().activeView().viewAxometric()
    FreeCAD.Gui.activeDocument().activeView().setAxisCross(True)


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

else:

    clear_doc()

```

```
43
44  # EPS= tolerance to uset to cut the parts
45  EPS = 0.10
46  EPS_C = EPS * -0.5
47
48
49  def cubo(nome, lung, larg, alt):
50      obj_b = DOC.addObject("Part::Box", nome)
51      obj_b.Length = lung
52      obj_b.Width = larg
53      obj_b.Height = alt
54
55      DOC.recompute()
56
57      return obj_b
58
59  def base_cyl(nome, ang, rad, alt ):
60      obj = DOC.addObject("Part::Cylinder", nome)
61      obj.Angle = ang
62      obj.Radius = rad
63      obj.Height = alt
64
65      DOC.recompute()
66
67      return obj
68
69  # definizione oggetti
70
71  obj = cubo("cubo_di_prova", 5, 5, 5)
72
73  obj1 = base_cyl('primo cilindro', 360, 2, 10)
74
75  setview()
```


♢ ♢ ♢

Listato A.4.: Operazioni booleane - esempio completo

```
#
"""ob-ex-full.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2020
   Licence: CC BY-NC-ND 4.0 IT
"""

import FreeCAD
from FreeCAD import Base, Vector
import Part
from math import pi, sin, cos

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)

def setview():
    """Rearrange View"""
    FreeCAD.Gui.SendMsgToActiveView("ViewFit")
    FreeCAD.Gui.activeDocument().activeView().viewAxometric()
    FreeCAD.Gui.activeDocument().activeView().setAxisCross(True)


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

else:

    clear_doc()

```

```
43
44 # EPS= tolerance to uset to cut the parts
45 EPS = 0.10
46 EPS_C = EPS * -0.5
47
48
49 def cubo(nome, lung, larg, alt):
50     obj_b = DOC.addObject("Part::Box", nome)
51     obj_b.Length = lung
52     obj_b.Width = larg
53     obj_b.Height = alt
54
55     DOC.recompute()
56
57     return obj_b
58
59 def base_cyl(nome, ang, rad, alt ):
60     obj = DOC.addObject("Part::Cylinder", nome)
61     obj.Angle = ang
62     obj.Radius = rad
63     obj.Height = alt
64
65     DOC.recompute()
66
67     return obj
68
69 def fuse_obj(nome, obj_0, obj_1):
70     obj = DOC.addObject("Part::Fuse", nome)
71     obj.Base = obj_0
72     obj.Tool = obj_1
73     obj.Refine = True
74     DOC.recompute()
75
76     return obj
77
78 def mfuse_obj(nome, obj_0, obj_1):
79     obj = DOC.addObject("Part::MultiFuse", nome)
80     obj.Shapes = (obj_0, obj_1)
81     obj.Refine = True
82     DOC.recompute()
83
84     return obj
85
86 def cut_obj(nome, obj_0, obj_1):
87     obj = DOC.addObject("Part::Cut", nome)
88     obj.Base = obj_0
```

```
89      obj.Tool = obj_1
90      obj.Refine = True
91      DOC.recompute()
92
93      return obj
94
95  def int_obj(nome, obj_0, obj_1):
96      obj = DOC.addObject("Part::MultiCommon", nome)
97      obj.Shapes = (obj_0, obj_1)
98      obj.Refine = True
99      DOC.recompute()
100
101     return obj
102
103 # definizione oggetti
104
105 obj = cubo("cubo_di_prova", 5, 5, 5)
106
107 obj1 = base_cyl('primo cilindro', 360,2,10)
108
109 # mfuse_obj("cubo-cyl-fu", obj, obj1)
110
111 #cut_obj("cubo-cyl-cu", obj, obj1)
112
113 int_obj("cubo-cyl-is", obj, obj1)
114
115 setview()
```

♢ ♢ ♢

Listato A.5.: Estrusione - esempio completo

```
#
"""ext-full.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2020
   Licence: CC BY-NC-ND 4.0 IT
"""

import FreeCAD
from FreeCAD import Base, Vector
import Part
from math import pi, sin, cos

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)

def setview():
    """Rearrange View"""
    FreeCAD.Gui.SendMsgToActiveView("ViewFit")
    FreeCAD.Gui.activeDocument().activeView().viewAxometric()
    FreeCAD.Gui.activeDocument().activeView().setAxisCross(True)


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

else:

    clear_doc()

```

```
43
44 def reg_poly(center=Vector(0, 0, 0), sides=6, dia=6,
45              align=0, outer=1):
46     """
47     This return a polygonal shape
48
49     Keywords Arguments:
50         center   - Vector holding the center of the polygon
51         sides    - the number of sides
52         dia      - the diameter of the base circle
53                      (aphotem or externa diameter)
54         align    - 0 or 1 it try to align the base with one axis
55         outer    - 0: aphotem 1: outer diameter (default 1)
56     """
57
58     ang_dist = pi / sides
59
60     if align == 0:
61         theta = 0.0
62     else:
63         theta = ang_dist
64
65     if outer == 1:
66         rad = dia * 0.5
67     else:
68         # dia is the apothem, so calculate the radius
69         # outer radius given the inner diameter
70         rad = (dia / cos(ang_dist)) * 0.5
71
72     vertex = []
73
74     for n_s in range(0, sides+1):
75         vpx = rad * cos((2 * ang_dist * n_s) + theta) + center[0]
76         vpy = rad * sin((2 * ang_dist * n_s) + theta) + center[1]
77         vertex.append(Vector(vpx, vpy, 0))
78
79     obj = Part.makePolygon(vertex)
80     wire = Part.Wire(obj)
81     poly_f = Part.Face(wire)
82
83     return poly_f
84
85
86 # EPS= tolerance to uset to cut the parts
87 EPS = 0.10
88 EPS_C = EPS * -0.5
```

```
89
90
91  def dado(nome, dia, spess):
92      polyg = reg_poly(Vector(0, 0, 0), 6, dia, 0, 0)
93
94      nut = DOC.addObject("Part::Feature", nome + "_dado")
95      nut.Shape = polyg.extrude(Vector(0, 0, spess))
96
97      return nut
98
99
100 def estr_comp(nome, spess):
101     vertex = ((-2,0,0), (-1, 2, 0), (1, 2, 0), (1, 0, 0),
102               (0, -2, 0), (-2,0,0))
103
104     obj = Part.makePolygon(vertex)
105     wire = Part.Wire(obj)
106     poly_f = Part.Face(wire)
107
108     cexsh = DOC.addObject("Part::Feature", nome)
109     cexsh.Shape = poly_f.extrude(Vector(0, 0, spess))
110
111     return cexsh
112
113
114 # definizione oggetti
115
116 dado("Dado", 5.5, 10)
117
118 #estr_comp("complesso", 10)
119
120 setview()
```


♢ ♢ ♢

Listato A.6.: Rivoluzione - esempio completo

```
#
"""rev-full.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2020
   Licence: CC BY-NC-ND 4.0 IT
"""

import FreeCAD
from FreeCAD import Base, Vector
import Part
from math import pi, sin, cos

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)

def setview():
    """Rearrange View"""
    FreeCAD.Gui.SendMsgToActiveView("ViewFit")
    FreeCAD.Gui.activeDocument().activeView().viewAxometric()
    FreeCAD.Gui.activeDocument().activeView().setAxisCross(True)


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

else:

    clear_doc()

```

```
43 def point(nome, pos, pt_r = 0.25, color = (0.85, 0.0, 0.00),
44         tr = 0):
45     """draw a point for reference"""
46     rot_p = DOC.addObject("Part::Sphere", nome)
47     rot_p.Radius = pt_r
48     rot_p.Placement = FreeCAD.Placement(
49         pos, FreeCAD.Rotation(0,0,0), Vector(0,0,0))
50     rot_p.ViewObject.ShapeColor = color
51     rot_p.ViewObject.Transparency = tr
52 
53     DOC.recompute()
54 
55     return rot_p
56 
57 def reg_poly(center=Vector(0, 0, 0), sides=6, dia=6,
58              align=0, outer=1):
59     """
60     This return a polygonal face
61 
62     Keywords Arguments:
63         center   - Vector holding the center of the polygon
64         sides    - the number of sides
65         dia      - the diameter of the base circle
66                      (aphotem or externa diameter)
67         align    - 0 or 1 it try to align the base with one axis
68         outer    - 0: aphotem 1: outer diameter (default 1)
69 
70     """
71 
72     ang_dist = pi / sides
73 
74     if align == 0:
75         theta = 0.0
76     else:
77         theta = ang_dist
78 
79     if outer == 1:
80         rad = dia * 0.5
81     else:
82         # dia is the apothem, so calculate the radius
83         # outer radius given the inner diameter
84         rad = (dia / cos(ang_dist)) * 0.5
85 
86     vertex = []
87 
88     for n_s in range(0, sides+1):
```

```
89          vpx = rad * cos((2 * ang_dist * n_s) + theta) + center[0]
90          vpy = rad * sin((2 * ang_dist * n_s) + theta) + center[1]
91          vertex.append(Vector(vpx, vpy, 0))
92
93      obj = Part.makePolygon(vertex)
94      wire = Part.Wire(obj)
95      poly_f = Part.Face(wire)
96
97      return poly_f
98
99
100 # EPS= tolerance to use to cut the parts
101 EPS = 0.10
102 EPS_C = EPS * -0.5
103
104
105 def dado(nome, dia, spess):
106     poly_f = reg_poly(Vector(0, 0, 0), 6, dia, 0, 0)
107
108     nut = DOC.addObject("Part::Feature", nome + "_dado")
109     nut.Shape = poly_f.extrude(Vector(10, 5, spess))
110
111     return nut
112
113
114 def manico(nome):
115     """Revolve a face"""
116     face = reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0)
117     # base point of the rotation axis
118     pos = Vector(0,10,0)
119     # direction of the rotation axis
120     vec = Vector(1,0,0)
121     angle = 180 # Rotation angle
122
123     point("punto_rot", pos)
124
125     obj = DOC.addObject("Part::Feature", nome)
126     obj.Shape = face.revolve(pos, vec, angle)
127
128     DOC.recompute()
129
130     return obj
131
132 # definizione oggetti
133
134 manico("Manico")
```

```
135
136 setview()
```

♢ ♢ ♢

Listato A.7.: Riferimento costruzione - esempio completo

```
#
"""rif-cost2.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2020
   Licence: CC BY-NC-ND 4.0 IT
"""

import FreeCAD
from FreeCAD import Base, Vector
import Part
from math import pi, sin, cos

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)

def setview():
    """Rearrange View"""

    FreeCAD.Gui.activeDocument().activeView().viewAxometric()
    FreeCAD.Gui.activeDocument().activeView().setAxisCross(True)
    FreeCAD.Gui.SendMsgToActiveView("ViewFit")


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

else:

    clear_doc()
```

```
43
44
45  # EPS= tolerance to uset to cut the parts
46  EPS = 0.10
47  EPS_C = EPS * -0.5
48
49
50  def cubo(nome, lung, larg, alt, cent = False, off_z = 0):
51      obj_b = DOC.addObject("Part::Box", nome)
52      obj_b.Length = lung
53      obj_b.Width = larg
54      obj_b.Height = alt
55
56      if cent == True:
57          posiz = Vector(lung * -0.5, larg * -0.5, off_z)
58      else:
59          posiz = Vector(0, 0, off_z)
60
61      obj_b.Placement = FreeCAD.Placement(
62          posiz,
63          FreeCAD.Rotation(0, 0, 0),
64          FreeCAD.Vector(0,0,0)
65          )
66
67      DOC.recompute()
68
69      return obj_b
70
71
72  def base_cyl(nome, ang, rad, alt ):
73      obj = DOC.addObject("Part::Cylinder", nome)
74      obj.Angle = ang
75      obj.Radius = rad
76      obj.Height = alt
77
78      DOC.recompute()
79
80      return obj
81
82
83  def cut_obj(nome, obj_0, obj_1):
84      obj = DOC.addObject("Part::Cut", nome)
85      obj.Base = obj_0
86      obj.Tool = obj_1
87      obj.Refine = True
88      DOC.recompute()
```

```
89
90      return obj
91
92
93  obj1 = cubo("cubo_cyl", 10, 20, 10, True, 10)
94  obj2 = base_cyl("cilindro", 360, 2.5, 15 )
95
96  obj_f = cut_obj("cubo_cyl", obj1, obj2)
97
98  print("Oggetto finale = ", obj_f.Placement)
99  print("Cubo Base = ", obj1.Placement)
100 print("Cilindro = ", obj2.Placement)
101
102
103 setview()
```

♢ ♢ ♢

Listato A.8.: Modulo GfcMod - esempio iniziale

```
"""GfcMod.py
   module

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2020
   Licence: CC BY-NC-ND 4.0 IT
"""

import FreeCAD
from FreeCAD import Base, Vector
import Part
from math import pi, sin, cos


# EPS= tolerance to uset to cut the parts
EPS = 0.10
EPS_C = EPS * -0.5

def cubo(doc, nome, lung, larg, alt, cent = True, off_z = 0.0):
    obj_b = doc.addObject("Part::Box", nome)
    obj_b.Length = lung
    obj_b.Width = larg
    obj_b.Height = alt

    doc.recompute()

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

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

    return obj_b

```

```
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 axis
72         outer     - 0: aphotem 1: outer diameter (default 1)
73     """
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) + center[0]
93          vpy = rad * sin((2 * ang_dist * n_s) + theta) + center[1]
94          vertex.append(Vector(vpx, vpy, 0))
95
96      obj = Part.makePolygon(vertex)
97      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     c_c1 = Vector((lung * -0.5) + raggio, (larg * -0.5) + raggio, 0)
106     c_c2 = Vector((lung * -0.5) + raggio, (larg * 0.5) - raggio, 0)
107     c_c3 = Vector((lung * 0.5) - raggio, (larg * 0.5) - raggio,  0)
108     c_c4 = Vector((lung * 0.5) - raggio, (larg * -0.5) + raggio, 0)
109
110     obj_dim = c_c3 - c_c1
111     fi_lung = obj_dim[0] + raggio * 2
112     fi_larg = obj_dim[1] + raggio * 2
113
114     obj_c1 = base_cyl(doc, nome +'_cil1', 360, raggio, alt)
115     obj_c1.Placement = FreeCAD.Placement(
116             c_c1,
117             FreeCAD.Rotation(0, 0, 0),
118             FreeCAD.Vector(0,0,0))
119
120     obj_c2 = base_cyl(doc, nome +'_cil2', 360, raggio, alt)
121     obj_c2.Placement = FreeCAD.Placement(
122             c_c2,
123             FreeCAD.Rotation(0, 0, 0),
124             FreeCAD.Vector(0,0,0))
125
126     obj_c3 = base_cyl(doc, nome +'_cil3', 360, raggio, alt)
127     obj_c3.Placement = FreeCAD.Placement(
128             c_c3,
129             FreeCAD.Rotation(0, 0, 0),
130             FreeCAD.Vector(0,0,0))
131
132     obj_c4 = base_cyl(doc, nome +'_cil4', 360, raggio, alt)
133     obj_c4.Placement = FreeCAD.Placement(
134             c_c4,
```

```
135             FreeCAD.Rotation(0, 0, 0),
136             FreeCAD.Vector(0,0,0))
137
138
139     obj1 = cubo(doc, nome + "_int_lu", fi_lung , obj_dim[1], alt, False)
140     obj1.Placement = FreeCAD.Placement(Vector(fi_lung * -0.5, obj_dim[1] * -0.5, 0), Fi
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), Fi
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
```

♢ ♢ ♢

Listato A.9.: Sardine - esempio completo

```
#
"""sardine.py

    This code was written as an sample code
    for "FreeCAD Scripting Guide"

    Author: Carlo Dormeletti
    Copyright: 2020
    Licence: CC BY-NC-ND 4.0 IT
"""

import os
import sys
import importlib

import FreeCAD
from FreeCAD import Base, Vector
import Part
from math import pi, sin, cos

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

import GfcMod as GFC
importlib.reload(GFC)

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)

def setview():
    """Rearrange View"""

    FreeCAD.Gui.activeDocument().activeView().viewAxometric()
    FreeCAD.Gui.activeDocument().activeView().setAxisCross(True)
    FreeCAD.Gui.SendMsgToActiveView("ViewFit")
```

```
43
44
45  if DOC is None:
46      FreeCAD.newDocument(DOC_NAME)
47      FreeCAD.setActiveDocument(DOC_NAME)
48      DOC = FreeCAD.activeDocument()
49
50  else:
51
52      clear_doc()
53
54
55  # EPS= tolerance to uset to cut the parts
56  EPS = 0.10
57  EPS_C = EPS * -0.5
58
59
60  def sardine(nome, lung, prof, alt, raggio, spess):
61      """sardine
62          crea una geometria a forma di scatola di sardine
63
64          Keywords Arguments:
65          nome   = nome della geometria finale
66          lung   = lunghezza della scatola, lungo l'asse X
67          larg   = larghezza della scatole, lungo l'asse Y
68          alt    = altezza della scatola, lungo l'asse Z
69          raggio = raggio dello "stondamendo" della scatola
70          spess  = spessore della scatola
71      """
72
73      obj = GFC.cubo_stondato(
74          DOC, nome + "_est", lung, prof, alt, raggio, 0.0)
75
76      lung_int = lung - (spess * 2)
77      prof_int = prof - (spess * 2)
78      alt_int = alt + EPS - spess
79      raggio_int = raggio - spess
80
81      obj_int = GFC.cubo_stondato(
82          DOC, nome + "_int",
83          lung_int, prof_int, alt_int, raggio_int, spess)
84
85      obj_f = DOC.addObject("Part::Cut", nome)
86      obj_f.Base = obj
87      obj_f.Tool = obj_int
88      obj_f.Refine = True
```

```
89      DOC.recompute()
90
91      return obj_f
92
93  sardine("scatola", 30, 20, 10, 5, 1)
94
95  setview()
```

♢ ♢ ♢

Appendice B

Elementi Interfaccia Utente

Appendice C

Glossario

angoli di Eulero o di Tait-Bryan - p. 23

operazioni booleane - p. 25
- **intersezione** - p. 25
- **sottrazione** - p. 25
- **unione** - p. 25

Appendice D

Voci di menù

File Voce di menù - p. 6
  Apri - p. 6

Modifica Voce di menù (*Orig:* **Edit**) - p. 4, 6, 7
  Preferenze (*Orig:* **Preferences**) - p. 4, 6, 7
    Generale (*Orig:* **General**) - p. 4, 6, 7
      Editor Linguetta - p. 7
      Finestra di Output Linguetta - p. 6
      Lingua Linguetta (*Orig:* **Language**) - p. 4

Visualizza Voce di menù - p. 4, 5, 18, 21
  Barre degli Strumenti - p. 5
    Axonometric - p. 19
    Isometrica - p. 19
    Macro - p. 5, 6
    Vista Ortografica - p. 18
    Vista Prospettica - p. 18
    Viste Standard - p. 18
  Origine degli assi - p. 21
  Panelli - p. 4
    Console Python - p. 5
    Report - p. 5

Appendice E

Oggetti di FreeCAD