

Programmare script e Add-Ons per Blender 3D 2.5

Volume 1

Francesco *RedBaron85* Milanese

www.redbaron85.com

SOMMARIO

Premessa	5
Le basi del linguaggio Python	6
Il linguaggio Python	6
Cosa serve per programmare in Python	7
Il prompt dell'IDLE	8
Importazione di moduli	9
Commenti	9
L'indentazione	10
Tipi di dati primitivi e definizione delle variabili	10
Lo slicing	12
Le liste	13
Gli insiemi	14
Le tuple	15
I dizionari	16
I comandi dir e type	17
Il blocco IF	18
Il ciclo FOR	19
Il ciclo WHILE e l'istruzione BREAK	20
Le funzioni	21
Classi ed ereditarietà	23
Gestione delle eccezioni	24
Altri argomenti	26
Informazioni e operazioni preliminari	27
La documentazione delle API Blender Python	27
Creazione degli script e degli Add-Ons: strumenti disponibili	29
Descrizione dei principali sottomoduli di bpy	34
Primi passi con le Blender Python API	36
Trasformazioni in Object Mode	44

Animazioni Object Mode: un case study	53
Creazione di un Add-On per Blender, informazioni preliminari.....	61
Operatori e modificatori in Object ed Edit Mode.....	66
ESEMPIO 1	70
ESEMPIO 2	71
Moduli di Input / Output su file: un Case Study.....	73
Case Study: il plugin “Add Point”	81
Case Study: il plugin “Mixer Luci”	90

Premessa

La possibilità di estendere le funzionalità di base di Blender 3D mediante la definizione di script e Add-Ons in linguaggio Python è uno dei punti di forza di questo meraviglioso programma.

Le API di programmazione e l'intero sistema di gestione dei plugin hanno subito notevoli cambiamenti rispetto alla versione 2.4 del programma, rendendo inutilizzabili gli script sviluppati per tali release ma più facile la scrittura dei plugin destinati alla versione 2.5.

In questo libro, realizzato facendo riferimento alla versione 2.59 stabile delle API Blender Python, dopo un capitolo introduttivo riguardante il linguaggio Python puro si procederà alla definizione di script sempre più articolati fino alla creazione di veri e propri plugin, detti Add-Ons, integrabili in maniera stabile nell'applicazione.

Il libro è rivolto a chi ha una buona conoscenza degli elementi principali di Blender 3D 2.5 e possibilmente un po' di familiarità, se non con il linguaggio Python puro, con i concetti fondamentali della programmazione orientata agli oggetti.

Francesco *RedBaron85* Milanese è un Blender Foundation Certified Trainer.

Ha realizzato e gestisce il sito web **RedBaron85.com** (www.redbaron85.com), dove pubblica periodicamente videotutorials, ebooks e guide sull'utilizzo di Blender 3D e altri programmi di CG.

Tiene inoltre corsi, seminari e consulenze, sia per enti pubblici che privati, su vari software e linguaggi di programmazione riguardanti la Computer Grafica 3D.

CAPITOLO 1

Le basi del linguaggio Python

Questo capitolo non ha la pretesa di essere un manuale di programmazione in linguaggio **Python**, né di insegnare le basi della programmazione *tout court*. Il libro, infatti, è rivolto a chi ha già una certa dimestichezza con la programmazione (anche in altri linguaggi) e, quindi, conosce i costrutti fondamentali quali i blocchi *if*, i cicli *for*, ecc... ma non conosce la loro sintassi in linguaggio **Python**.

Se si ha esperienza di programmazione in linguaggio **Python**, si può saltare questo capitolo introduttivo, ed eventualmente usarlo per ripassare la forma di alcuni costrutti, la definizione di funzioni o classi e altri elementi di base di programmazione.

Il linguaggio Python

Python è un linguaggio di programmazione di alto livello.

Si tratta di un linguaggio:

- **interpretato**: gli script non vengono compilati, non c'è bisogno di linking, l'interprete **Python** esegue direttamente i file “in chiaro” degli **script**;
- **interattivo**: gli script possono essere eseguiti “al volo”, un'istruzione o un blocco di istruzioni per volta, in una console interattiva;
- **orientato agli oggetti**: supporta la definizione di **classi** con campi, metodi e caratteristiche della programmazione **OO**, come l'**ereditarietà**;
- **portabile**: poiché è interpretato, può essere scritto una sola volta ed eseguito su tutti i sistemi operativi dotati di **ambiente di runtime Python**; non c'è bisogno, quindi, di **compilarlo** per le varie piattaforme, come avviene ad esempio con C e C++.

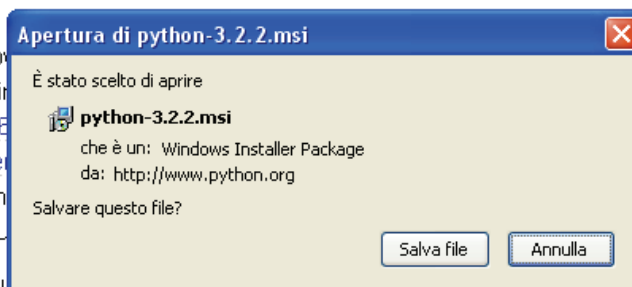
Cosa serve per programmare in Python

L'unica cosa necessaria per programmare in **Python** è il **kit di sviluppo** del linguaggio: un'applicazione, scaricabile gratuitamente dal sito ufficiale del programma (www.python.org , sezione **Download**) che installa le librerie necessarie per l'esecuzione degli script e il **Python IDLE**, una **console interattiva** che consente di eseguire “al volo” le istruzioni **Python**.

Also look at the [detailed Python 3.2.2 page](#):

- [Python 3.2.2 Windows x86 MSI Installer](#) (Windows)
- [Python 3.2.2 Windows X86-64 MSI Installer](#) (Windows)
- [Python 3.2.2 Mac OS X 64-bit/32-bit x86-64/i386](#) (Mac OS X)
- [Python 3.2.2 Mac OS X 32-bit i386/PPC Installer](#) (Mac OS X)
- [Python 3.2.2 compressed source tarball](#) (for Linux, Unix)
- [Python 3.2.2 bzipipped source tarball](#) (for Linux, Unix)

A [comprehensive list of all released versions](#) is available if you need source code for an older version of Python.



L'installazione del kit varia ovviamente da sistema operativo a sistema operativo ma segue le modalità “classiche” di installazione di un programma per ciascun ambiente; ad esempio, su sistemi **Win** è sufficiente fare doppio click sul file di installazione del programma e proseguire col wizard di installazione guidata, avendo cura di installare il kit “per tutti gli utenti”.



A seconda della versione di **Blender** in uso, è necessario installare una determinata versione di **Python**; per la versione **2.59 di Blender**, la versione di **Python** da installare è la **3.2**.

Alcune distribuzioni di **Blender** forniscono un interprete **Python** preinstallato al loro interno, tuttavia l'installazione del kit vero e proprio è indispensabile per poter utilizzare, ad esempio, **IDE** di sviluppo “esterni”, come **Eclipse**.

I file degli script, che possono essere definiti sia mediante un semplice **editor di testo** che un ambiente di sviluppo complesso, vanno salvati in formato **.py**; se le variabili d'ambiente del sistema sono state configurate in modo da poter eseguire gli script **Python** da console, sarà sufficiente scrivere

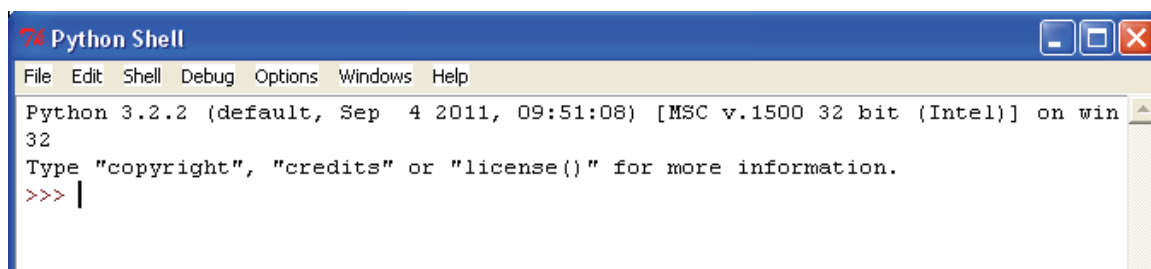
Python [nomeScript].py

nel prompt di sistema per eseguire lo script salvato nel file **.py** presente nella cartella corrente.

In questo capitolo faremo uso dell'**IDLE di Python** per mostrare l'esecuzione di alcuni comandi ma già dal prossimo capitolo abbandoneremo questo ambiente in favore della **Python Console di Blender 3D**.

Il prompt dell'IDLE

L'**IDLE** si presenta come una finestra con titolo **"Python Shell"** e il cursore che lampeggia subito dopo la prima riga di **prompt** dell'ambiente; il prompt è specificato mediante i simboli **">>>"**.



Avviato l'**IDLE**, tutto quello che bisogna fare per eseguire uno script è... scrivere e premere **Invio**. Scrivendo ad esempio:

```
>>> print (5)
5
>>> n = 5
>>> print (n)
5
>>> print (n*2)
10
>>> |
```

si utilizzerà la funzione primitiva di **Python** **"print([variabile])"**, che stampa a video il valore della variabile passata come argomento (o il valore stesso, se si passa un tipo primitivo come un numero o una stringa).

La definizione di classi, funzioni, e costanti primitive di Python si trova nella **Documentazione ufficiale** del linguaggio, che contiene le informazioni sulle **API** (Application Programming Interface) dello stesso; tale Documentazione è reperibile sul **sito web ufficiale** del linguaggio (www.python.org).

Importazione di moduli

E' possibile estendere le funzionalità fornite di base dalle **API Python** mediante l'importazione di **moduli esterni**, ossia librerie di classi e funzioni dedicate a scopi particolari; ad esempio, il modulo **random** (fornito con le **API Python** di base, ma le cui istruzioni non sono richiamabili di default nell'IDLE se non dopo un'importazione esplicita) mette a disposizione funzioni per la generazione di numeri casuali.

L'**importazione dei moduli** aggiuntivi avviene mediante l'istruzione:

```
>>> import [nomeModulo]
```

per cui, per importare il modulo **random** ed utilizzare la sua funzione **randrange(s,e)**, che genera un numero casuale tra *s* ed *e*, sarà sufficiente scrivere:

```
>>> import random
>>> n = random.randrange(1,100)
>>> print(n)
80
>>> n = random.randrange(|
                        (start, stop=None, step=1, int=<class 'int'>)
                        Choose a random item from range(start, stop[, step]).
```

Nell'immagine precedente viene mostrato il **Tooltip** che appare nell'IDLE quando si fa uso di una funzione **Python**: tale **Tooltip** contiene un "suggerimento" sull'utilizzo della funzione, in questo caso **randrange**, indicando quali sono i parametri da utilizzare.

Commenti

Le righe di **commento** devono iniziare col simbolo #:

```
>>>
>>> # Questo è un commento
>>> print(5) # Commento in linea, dopo un'istruzione
5
>>> |
```

L'indentazione

In **Python**, la delimitazione dei **blocchi di istruzioni** non avviene mediante le parentesi graffe o con istruzioni di apertura e chiusura (es.: “*begin/end*”), ma semplicemente rientrando il testo delle istruzioni (incluso il corpo di una classe o una funzione), ossia mediante **l'indentazione**, con una o più pressioni del tasto **TAB** (più pressioni nel caso di blocchi annidati, ad esempio un *if* dentro uno o due cicli *for*).

```
>>> for i in range(1,10):  
    for j in range(1,10):  
        print(i*j)      # Dentro il ciclo annidato j  
    # Dentro il ciclo i (un solo TAB)  
# Fuori dal ciclo i; per chiudere il blocco, premere Invio  
  
1  
2  
3  
4  
5
```

L'uso dell'indentazione è **obbligatorio**: Python valuterà come **errori di sintassi** gli script privi di indentazione (dove richiesta).

Tipi di dati primitivi e definizione delle variabili

In **Python**, una variabile viene creata specificandone il nome ed assegnandole un valore, senza dover specificare il tipo: **Python** provvederà a riconoscerlo automaticamente.

Nel caso dei tipi primitivi, **Python** si affiderà alle convenzioni di definizione di una variabile per riconoscerne il tipo, in quanto:

- **i dati numerici**, sia interi che in virgola mobili, sono semplici numeri, non rinchiusi in apici o doppi apici, con il . per identificare i decimali;
- **i booleani** sono solo **True** e **False**, senza apici o doppi apici a racchiudere tali parole;
- **stringhe** e **caratteri** (che poi sono stringhe di un solo elemento) vanno racchiusi in apici o doppi apici;
- **None** è un tipo a parte, il tipo nullo.

```
>>> numero = 2 # Variabile intera
>>> print(numero)
2
>>> numero = float(numero) # Conversione in variabile in virgola mobile (float)
>>> print(numero)
2.0
>>> stringa = "True" # Questa è una STRINGA di valore "True"
>>> print(stringa)
True
>>> booleano = True # Questo è un BOOLEAN "vero" (True, 1), non una stringa
>>> print(booleano)
True
>>> # Posso concatenare due stringhe...
>>> stringa = stringa + stringa
>>> print(stringa)
TrueTrue
>>> # ... ma non una stringa con un altro tipo senza casting esplicito: si avrà errore:
>>> stringa = stringa + numero
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    stringa = stringa + numero
TypeError: Can't convert 'float' object to str implicitly
>>> # Casting di "numero" in stringa:
>>> numero = str(numero)
>>> print(numero)
2.0
>>> # Ora posso concatenare:
>>> stringa = stringa + numero
>>> print(stringa)
TrueTrue2.0
>>> |
```

Le **API Python** mettono a disposizione vari metodi per **convertire** (*casting*) un tipo in un altro; ad esempio, per cambiare la variabile numerica *n* in **stringa**, scriveremo:

n = *str*(*n*)

In generale, le **stringhe** sono **oggetti** non banali, forniti di **metodi** utili per la loro elaborazione; esempi:

- *find*("[carattere]") restituisce l'indice (posizione da sinistra, partendo da 0) della prima occorrenza del carattere passato come parametro all'interno della stringa che ha invocato il metodo, -1 se non presente;
- *strip*() rimuove gli spazi all'inizio e alla fine della stringa;
- *replace*(*sottostringa1*, *sottostringa2*) sostituisce la porzione di stringa "sottostringa1" con la stringa "sottostringa2";
- *split*("[stringa di splitting]") suddivide la stringa in più parti ad ogni occorrenza di "stringa di splitting", inserendo le varie stringhe così generate in un oggetto di tipo *list* (lista), che tratteremo a breve;
- *join*(*[listaElementi]*) unisce gli elementi presenti in una lista di stringhe in un'unica, nuova stringa.

```
>>> stringa = "    Ciao    "
>>> stringa.find("i")
4
>>> stringa.find("n")
-1
>>> stringa.strip()
'Ciao'
>>> stringa.replace("Ciao", "CiaoCiao")
'    CiaoCiao    '
>>> nuovaStringa = stringa.split("oC")
>>> print(stringa)
    Ciao
>>> print(nuovaStringa)
['    Ciao    ']
>>> lista = ["Ciao", " come ", "stai?"]
>>> terzaStringa = "".join(lista)
>>> print(terzaStringa)
Ciao come stai?
>>> |
```

Fanno parte dei **tipi primitivi di Python** anche tipi più “complessi”:

- liste;
- insiemi;
- tuple;
- dizionari.

Li prenderemo presto in esame.

Lo slicing

Una tecnica utilizzabile sulle stringhe e su altre sequenze di dati o “oggetti iterabili” è quella dello *slicing* (lett.: “tagliare a fette”), che consente di **estrapolare una parte di un elemento**, ed in particolare la parte racchiusa tra un indice iniziale e uno finale passati come parametro e separati dal carattere “:” ; se non specificheremo uno dei due indici, **Python** interpreterà come “*dall’inizio*” (se l’indice non specificato è quello a sinistra di “:”) o “*fino alla fine*” (se non specificheremo il secondo indice).

NOTA: è possibile recuperare l’**ultimo elemento** di una stringa, una list o un oggetto iterabile in generale con *[-1]*.

Esempio di utilizzo dello **slicing** in figura nella pagina seguente.

```
>>> stringa = "Ciao come stai?"
>>> stringa1 = stringa[:2]
>>> print(stringa1)
Ci
>>> stringa2 = stringa[3:5]
>>> print(stringa2)
o
>>> stringa3 = stringa[5:]
>>> print(stringa3)
come stai?
>>> print(stringa[-1])
?
>>> print(stringa[-4])
t
>>>
```

Le liste

Una **list** è un insieme *ordinato* di elementi *eterogenei*; ciascun elemento ha una sua posizione all'interno della lista ed è possibile aggiungere o rimuovere elementi dinamicamente (una **list** non è, quindi, un oggetto immutabile).

Le liste vengono create ed identificate mediante le **parentesi quadre []**; è possibile creare una lista vuota ed aggiungere un seguito elementi o specificarli in fase di creazione, separandoli con delle virgole; è possibile riferirsi ad uno o più elementi di una lista con gli indici e lo **slicing** (in questo caso, verrà restituita una nuova **list**).

```
>>> lista1 = []
>>> lista1.append("Ciao")
>>> print(lista1)
['Ciao']
>>>
>>> lista2 = ["Stringa", True, 5, 24.2, None]
>>> lista2[2]
5
>>> lista2[-1]
None
>>> lista2[-2]
24.2
>>> lista2[1:3]
[True, 5]
>>>
```

Qualsiasi cosa può far parte di una lista, anche un'altra lista.

```
>>>
>>> listaA = ["Ciao", 3]
>>> listaB = ["Stringa", listaA, 20.2]
>>> print(listaB)
['Stringa', ['Ciao', 3], 20.2]
>>>
```

Ecco alcuni **metodi utili** messi a disposizione da **list**:

- ***append(elemento)***
Consente di aggiungere l'elemento passato come parametro in coda alla lista che sta invocando il metodo.
- ***insert(indice, elemento)***
Inserisce un elemento nella posizione data da **"indice"** (il primo elemento ha indice 0), spostando gli altri elementi (non sovrascrive elementi esistenti).
- ***extend(list)***
Come ***append***, ma per le liste: concatena la lista passata come parametro a quella che ha invocato il metodo.
- ***remove(indice)***
Rimuove, dalla lista che lo invoca, l'elemento di posizione **"indice"**.
- ***index(elemento)***
Restituisce la posizione dell'elemento passato come parametro, genera errore (cfr. sezione sulle **Eccezioni**) se l'elemento non è presente.
- ***sort()***
Ordina, con criterio alfabetico o numerico crescente, gli elementi presenti in una lista; eventuali elementi di tipo **None** verranno messi per primi.
- ***reverse()***
Inverte l'ordine degli elementi contenuti nella lista che lo invoca.
- ***len(lista)***
Restituisce, sotto forma di numero intero, il numero di elementi che compongono una lista.
- ***count(elemento)***
Conta il numero di occorrenze dell'elemento passato come parametro all'interno della lista che lo invoca.

Gli insiemi

In **Python**, gli insiemi (**"set"**) sono gruppi di elementi indicizzati che non contengono duplicati (le liste, invece, possono contenere elementi duplicati); vengono creati con la funzione **"set(elemento)"**:

```
>>> insieme1 = set(['ciao', 'come', 'stai']) # Il parametro è una list di 3 elementi stringa
>>> insieme2 = set(['ciao', 'bene', 'grazie'])
>>> |
```

Le operazioni possibili sono quelle effettuabili sugli insiemi matematici, ossia unione, differenza, intersezione, xor, ...; esempi:

- **l'unione** viene implementata con il simbolo `|` (corrisponde ad un OR);
- **l'intersezione** viene implementata con il simbolo `&` (corrisponde ad un AND);
- **la differenza** viene implementata mediante il simbolo `-`;
- **lo XOR** viene implementato mediante il simbolo `^`.

```
>>> insiemeUnione = insieme1 | insieme2
>>> print(insiemeUnione) # Notare l'assenza del "ciao" duplicato
{'grazie', 'bene', 'ciao', 'come', 'stai'}
>>>
>>> insiemeDifferenza = insieme1 - insieme2
>>> print(insiemeDifferenza)
{'come', 'stai'}
>>>
>>> insiemeIntersezione = insieme1 & insieme2
>>> print(insiemeIntersezione)
{'ciao'}
>>>
>>> insiemeXOR = insieme1 ^ insieme2
>>> print(insiemeXOR)
{'come', 'grazie', 'bene', 'stai'}
>>>
```

NOTA: passando una stringa come argomento in fase di creazione dell'insieme, detta stringa verrà scomposta nelle lettere che la formano, che verranno trattate come singoli elementi da inserire nell'insieme, ma poiché *set* non considera gli elementi duplicati potrebbero verificarsi effetti indesiderati:

```
>>> insiemeDaStringa = set("ciao come stai")
>>> print(insiemeDaStringa) # Notare l'assenza degli elementi duplicati
{'a', ' ', 'c', 'e', 'i', 'm', 'o', 's', 't'}
>>> |
```

Le tuple

Le *tuple* sono sequenze **immutabili** di oggetti di vario tipo.

Vengono create specificando i parametri da associare alla variabile tupla tra **parentesi tonde**; una tupla con un solo elemento va creata facendo seguire l'elemento da una virgola.

```
>>>
>>> esempioTupla = (1, 'c', 'parola', 5)
>>> tuplaUnElemento = ('x',)
>>> .
```

Le tuple mettono a disposizione il metodo **in** che restituisce un booleano (*True / False*) per verificare l'appartenenza di un elemento alla tupla; come con le liste, è possibile recuperare un elemento specificandone la posizione all'interno delle **parentesi quadre**, recuperare l'ultimo elemento con **[-1]** ed effettuare lo **slicing** (in questo caso, elementi multipli verranno restituiti sotto forma di una nuova tupla).

```
>>>
>>> risposta = 'parola' in esempioTupla
>>> print(risposta)
True
>>>
>>> esempioTupla[2]
'parola'
>>>
>>> esempioTupla[-1]
5
>>> esempioTupla[:2]
(1, 'c')
>>> |
```

I dizionari

Un **dizionario** è una struttura dati **associativa**, sostanzialmente un **array di coppie chiave-valore**.

Un dizionario vuoto, da popolare in seguito, può essere creato con la funzione **dict()** oppure assegnando **{}** ad una variabile, mentre un dizionario con alcuni elementi iniziali va creato specificando le coppie chiave-valore all'interno di parentesi graffe, separando la chiave il valore di ciascuna coppia con i due punti e le coppie tra loro mediante virgole:

```
>>>
>>> dizionario1 = dict()
>>> dizionario2 = {'chiave0' : 0, 'chiave1' : 'stringa', 'chiave2' : True}
>>>
>>> print(dizionario1)
{}
>>> print(dizionario2)
{'chiave2': True, 'chiave1': 'stringa', 'chiave0': 0}
>>>
>>> altroDizionarioVuoto = {}
>>> |
```

E' possibile inserire una nuova coppia specificando una chiave tra parentesi quadre e, a destra dell'uguale, il valore da associarle; per recuperare il valore associato ad una chiave è sufficiente specificare la chiave tra parentesi quadre subito dopo il nome del dizionario; infine, per rimuovere un elemento dal dizionario va utilizzato il comando **“del nomeDizionario[nomeChiave]”**.

```
>>>
>>> dizionario1['chiave'] = True
>>> print(dizionario1)
{'chiave': True}
>>>
>>> print(dizionario1['chiave'])
True
>>>
>>> del dizionario1['chiave']
>>>
>>> print(dizionario1)
{}
>>>
```

Tra i metodi messi a disposizione dai dict abbiamo:

- ***values()***
Restituisce, in una **list**, i valori presenti nel dizionario.
- ***keys()***
Restituisce, in una **list**, le chiavi presenti nel dizionario.
- ***items()***
Restituisce, in una **lista di tuple**, le coppie chiave-valore presenti nel dizionario; intuitivamente, ogni tupla della lista è composta da due elementi, ossia la chiave e il valore.
- ***clear()***
Svuota il dizionario, eliminando tutte le coppie chiave-elemento presenti.
- ***has_key('chiave')***
Restituisce il **booleano True** se la chiave passata come parametro è presente nel dizionario, **False** altrimenti.

I comandi ***dir*** e ***type***

I comandi ***dir*** e ***type*** consentono di conoscere gli attributi (campi e metodi) e la tipologia dell'elemento passato come parametro; si tratta di due funzioni utilissime per “scoprire” qual è il tipo di elemento che si deve trattare e recuperare velocemente i suoi attributi, ad esempio perché non ci si ricorda il nome esatto di una funzione o di un campo.

Esempio di utilizzo di ***dir*** e ***type*** sul tipo primitivo ***list*** in figura nella pagina seguente.

```
>>>
>>> lista = ['stringa', 3]
>>>
>>> type(lista)
<class 'list'>
>>>
>>> dir(lista)
['_add_', '__class__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

Il blocco IF

Il costrutto **if** consente di **verificare una determinata condizione** ed intraprendere, in base all'esito, delle **azioni**; la sintassi è la seguente (attenzione all'indentazione):

if condizione1:

azione1

elif condizione2:

azione2

else:

azione3

Attenzione anche ai due punti posti alla fine delle espressioni **if**, **elif** ("else if") ed **else**.

E' possibile utilizzare tutti gli **elif** che servono ma la presenza di blocchi **elif** o di un blocco **else** non è obbligatoria: tali istruzioni servono solo per indicare azioni da intraprendere se la condizione principale (**if**) non è verificata (**else**) o se se ne verifica un'altra specifica (**elif**).

```
>>>
>>> valore = 5
>>> if valore<3:
>>>     print("minore")
elif valore==3:
>>>     print("uguale")
else:
>>>     print("maggiore")

maggiore
>>> |
```

Gli **operatori relazionali** per verificare le condizioni sono:

<	Minore; es.: 3<5
<=	Minore o uguale
==	Uguale; es.: 5==5 è True
>=	Maggiore o uguale
>	Maggiore
!=	Diverso da; es.: 3!=5 è True
<>	Diverso da; es.: 3<>3 è False

Il ciclo FOR

Il costrutto **for** consente di effettuare **cicli di operazioni**, ossia di **ripetere** un insieme di istruzioni per un determinato numero di volte.

La sintassi è:

```
for [indice] in [sequenza]:  
    # istruzioni
```

Nella sua forma base, **for** viene creato con il costrutto **range**, disponibile in tre versioni:

- **range(n);**
esegue n iterazioni partendo dall'indice 0 fino a n-1;
- **range(inizio, fine);**
esegue il ciclo per (fine-inizio) volte; ad ogni iterazione, il valore di [indice] sarà pari a (inizio + numIterazioneCorrente);
- **range(inizio, fine, passo);**
esegue il ciclo per ((fine-inizio) / passo) volte; ad ogni iterazione, il valore di [indice] sarà pari a (inizio + (numIterazioneCorrente * passo)).

Esempio pratico di utilizzo di **for** con due versioni di **range** in figura nella pagina seguente.

```
>>>
>>> for i in range(5):
    print(i)

0
1
2
3
4
>>> for n in range(1, 10, 2):
    print(n)

1
3
5
7
9
>>>
```

E' possibile generare un **ciclo *for*** scorrendo un qualsiasi dato iterabile, ad esempio una *list*, in questo caso la sua sintassi sarà:

```
for [indice] in [oggetto iterabile]:
    # istruzioni
```

```
>>>
>>> stringa = "Ciao"
>>>
>>> for s in stringa:
    print(s)

C
i
a
o
>>>
```

Il ciclo WHILE e l'istruzione BREAK

Il costrutto *while* effettua le operazioni specificate fintantoché si verifica una condizione; a differenza del ciclo *for*, non prevede un meccanismo di incremento di **contatore** e, se non si specifica un contatore da incrementare o una **condizione di uscita** (da effettuare ad esempio con *break*, descritto in seguito), il ciclo non terminerà mai.

La sintassi è la seguente:

```
while condizione:  
    operazioni
```

```
>>>  
>>> a = 0  
>>> b = 3  
>>>  
>>> while a<b:  
    print(a)  
    a = a+1  
  
0  
1  
2  
>>> .
```

L'istruzione **break**, valida sia per cicli **for** che per cicli **while**, consente di uscire bruscamente da un ciclo; è utile in combinazione con un costrutto **if**, in quanto consente di terminare bruscamente il ciclo se si verifica una determinata condizione.

```
>>>  
>>> c = 0  
>>> for c in range(10):  
    print(c)  
    if (c*2 > 6):  
        break  
  
0  
1  
2  
3  
4  
>>> .
```

Le funzioni

Le **funzioni** (porzioni di codice dotate di nome, parametri di input e valori restituiti in output, richiamabili da altre parti del programma), vengono definite mediante la parola chiave **def** secondo la seguente sintassi (attenzione, come sempre, all'indentazione, da rimuovere quando si vuole chiudere il blocco di definizione di una funzione):

```
def nomeFunzione([0, 1 o più parametri di input]):  
    # Corpo della funzione  
    return [1 o più valori da restituire in output, opzionali]
```

Per richiamarle, è sufficiente scrivere il loro nome seguito da parentesi tonde contenenti 0 o più parametri (dipende ovviamente dalla funzione).

```
>>>
>>> def confronto(a, b):
    if a>b:
        return -1
    elif a==b:
        return 0
    else:
        return 1

>>> print(confronto(3,6))
1
>>>
```

Se nella definizione di una funzione sono presenti delle associazioni “*parametro=valore*”, allora non è obbligatorio specificare quel parametro quando si richiama la funzione; in assenza di tale parametro, la funzione assumerà il valore di default, cioè quello specificato nella definizione.

Eventuali argomenti con valore di default non possono essere seguiti, nella definizione della funzione, da altri argomenti privi di valore di default.

```
>>>
>>> def confronto(a=3, b=5):
    if (a>b):
        return -1
    elif (a==b):
        return 0
    else:
        return 1

>>> print(confronto())
1
>>> print(confronto(6,4))
-1
>>> print(confronto(b=3))
0
>>>
```

Le funzioni possono **restituire** 0, 1 o più valori; in questo caso, bisognerà assegnare la funzione a due o più variabili, separate da virgole, come mostra l’esempio nella figura seguente.

```
>>>
>>> import random
>>> def dueRandom():
    a = random.randrange(0,100)
    b = random.randrange(0,100)
    return a, b

>>> var1, var2 = dueRandom()
>>>
>>> print(var1)
96
>>> print(var2)
76
>>>
```

Classi ed ereditarietà

Le **classi di oggetti** vanno definite utilizzando la parola chiave **class**, seguita dal nome della classe e dai due punti che identificano l’apertura del blocco di codice appartenente alla classe; tale codice deve essere provvisto, ovviamente, della corretta indentazione.

```
class nomeClasse:  
    # corpo della classe
```

Si crea una **istanza** di una classe mediante l’assegnazione ad una variabile:

```
variabileIstanza = nomeClasse([eventuali parametri al costruttore])
```

All’interno della classe ci si potrà riferire all’oggetto istanziato (“se stesso”) mediante la parola chiave **self**, che dovrà essere il primo parametro di tutti i metodi della classe, costruttore incluso, ma che *non va passato come parametro quando tali metodi vengono invocati*.

self va utilizzato anche per definire gli attributi (campi) della classe.

Il **costruttore** è il metodo principale della classe ed è definito sempre così:

```
__init__(self, [eventuali parametri di input])
```

(due tratti di underscore _ prima e dopo la parola **init**).

Per definire un **metodo privato**, far precedere il suo nome da due caratteri underscore:

metodo.

Python consente l’**overloading degli operatori**, ossia la possibilità di riadattare alle nostre classi alcuni operatori di base del linguaggio **Python**, ad esempio quelli relazionali (>, >=, <, ...), in modo utilizzarli per confrontare due oggetti secondo un criterio da noi definito (a breve un esempio completo di definizione di una **class** con **overloading** degli operatori).

I metodi per i quali è consentito l’**overloading** vengono indicati con due tratti di underscore sia prima che dopo il nome del metodo; sono metodi soggetti ad **overloading**, ad esempio, i seguenti:

__len__	“Length”, lunghezza o numero di elementi
__str__	“To String”, informazioni sull’oggetto, in forma di stringa
__gt__	“Greater than”, cioè “maggiore di” (>)
__lt__	“Less than”, cioè “minore di” (<)

<code>__eq__</code>	“Equal”, cioè “uguale a” (==)
<code>__le__</code>	“Less equal” (<=)
<code>__ge__</code>	“Greater equal” (>=)

Nel caso di operatori come quelli relazionali ci si riferirà all’istanza “di sinistra” con *self* e all’istanza “di destra” con *other*.

```
>>> class persona:
    nome = ""
    cognome = ""
    eta = 0
    def __init__(self, n, c, e):
        self.nome = n
        self.cognome = c
        self.eta = e
    def __str__(self):
        print("Persona di nome: " + self.nome + ", cognome: " + self.cognome +
", anni: " + str(self.eta))
    def __lt__(self, other):
        return (self.eta < other.eta)

>>> persona1 = persona("Mario", "Rossi", 50)
>>> persona1.__str__()
Persona di nome: Mario, cognome: Rossi, anni: 50
>>> persona2 = persona("Giorgio", "Bianchi", 40)
>>> persona1 < persona2
False
>>>
```

Per implementare l’ereditarietà delle classi in **Python** è sufficiente passare come **parametro**, nella definizione della classe figlia, il **nome della classe genitore** (senza apici o doppi apici):

```
class classeFiglia(nomeClassePadre):
    # Corpo della classe figlia
```

La classe figlia erediterà **campi e metodi degli antenati**, eventualmente **sovrascrivendoli**.

Gestione delle eccezioni

Le **eccezioni** sono eventi “anomali” che, se non opportunamente trattati, possono portare alla terminazione improvvisa dell’intero programma o script; l’esempio tipico è la **divisione per 0**, operazione che in **Python** restituirà appunto un errore “**ZeroDivisionError**” (un vero e proprio oggetto, infatti le eccezioni standard di **Python** sono **classi** ed è possibile definire eccezioni personalizzate, richiamabili con il comando “*raise NomeEccezione*”), come visibile nell’immagine seguente:

```
>>>  
>>> risultato = 5/0  
Traceback (most recent call last):  
  File "<pyshell#334>", line 1, in <module>  
    risultato = 5/0  
ZeroDivisionError: division by zero  
>>>  
>>>
```

Le parti di codice sensibile (operazioni matematiche, operazioni su liste o elementi iterabili, operazioni di I/O o di rete, ecc..) dovrebbero essere sempre inserite in un blocco particolare, detto **“try”**, in grado di intercettare questi errori e passare il controllo del programma ad altri blocchi, detti **“except”**, in grado di gestire l’errore ed intraprendere altre azioni senza terminare l’applicazione.

La sintassi di **try-except** e dell’eventuale blocco **finally**, che indica operazioni da effettuare in ogni caso, è la seguente:

```
try:  
    # Porzione di codice da controllare  
except [eccezione]:  
    # Operazioni da effettuare se si verifica l'eccezione1; è possibile specificare  
    # più blocchi Except, da associare a diversi tipi di eccezioni verificabili  
else:  
    # Blocco, opzionale, da eseguire se non sono state lanciate eccezioni  
finally:  
    # Blocco da eseguire comunque, sia che siano state lanciate eccezioni che non
```

L’immagine seguente si riferisce al problema della **divisione per 0** trattato con un blocco **try-except**.

```
>>>  
>>> try:  
    risultato = 5/0  
except ZeroDivisionError:  
    print("Tentata divisione per zero, errore!!!")  
  
Tentata divisione per zero, errore!!!  
>>>
```

Altri argomenti

Come anticipato all'inizio del capitolo, quella appena mostrata è una trattazione superficiale delle **API** di programmazione **Python**; non sono stati trattati, infatti, argomenti come la gestione dell'I/O su file system, la gestione delle comunicazioni in rete, le operazioni con i database e l'utilizzo di moduli per la creazione di interfacce grafiche, argomenti comunque non richiesti per la creazione degli **script per Blender** che prenderemo in esame in questo libro.

Quello appena visto è quindi un capitolo utile per un richiamo veloce e per comprendere i comandi che verranno mostrati negli **script Python** per la creazione di **Add-Ons per Blender**, a partire dal prossimo capitolo.

CAPITOLO 2

Informazioni e operazioni preliminari

In questo capitolo esamineremo alcuni concetti preliminari prima di passare alla definizione ed esecuzione degli script all'interno della **Python Console** del programma; tratteremo, in particolare, del recupero delle informazioni sulle **API Blender Python** (la cosiddetta “**Documentazione**”), dei possibili metodi di definizione di **script Python per Blender** e dei principali sottomoduli di *bpy*, il modulo principale delle **API Blender Python**.

La documentazione delle API Blender Python

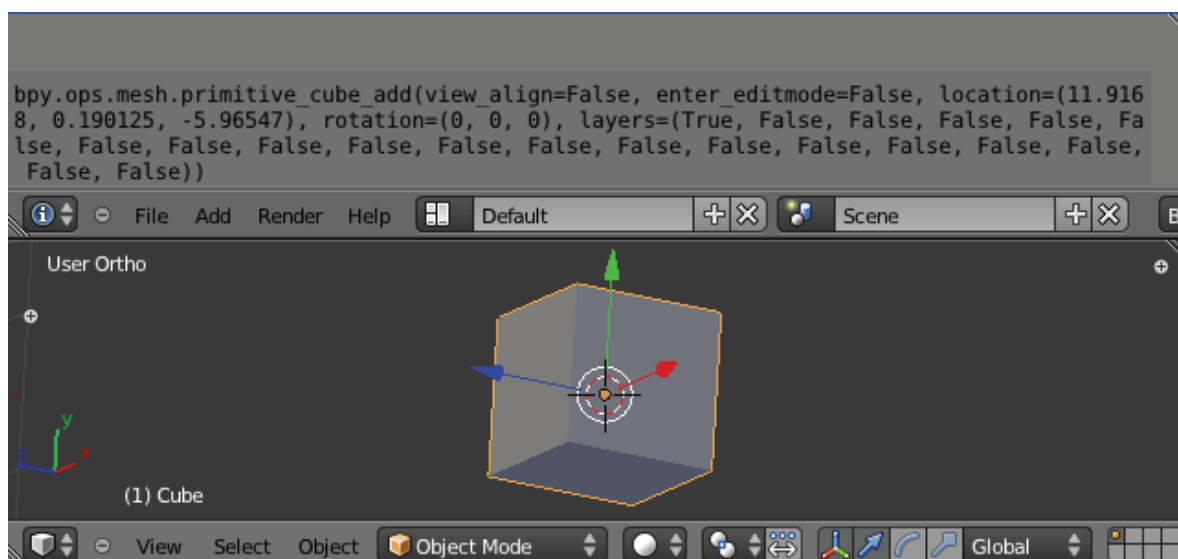
Le informazioni dettagliate su moduli, classi, campi e metodi delle **API Blender Python** si trovano nella relativa “**Documentazione**”: un file (tipicamente un **pdf** con segnalibri interni o un insieme di file **html** con link tra le pagine) dove vengono descritti i vari elementi, mostrando anche i parametri di ingresso e i valori restituiti dalle varie funzioni o i tipi di variabili, le costanti predefinite, ecc...

E' impensabile (e probabilmente anche inutile) cercare di conoscere *tutte* le voci delle **API Blender Python**; è importantissimo, invece, conoscere la **struttura dei moduli**, per “sapere cosa cercare e dove” quando si deve risolvere un problema; ad esempio, il nome dell'oggetto selezionato è, come vedremo, prerogativa del modulo “**context**”, per cui anziché cercare a caso nel file della documentazione potremo iniziare subito dalla sezione “**bpy.context**” ed esaminare le voci disponibili, trovare il campo che ci interessa ed analizzarne il tipo, per capire come trattarlo.

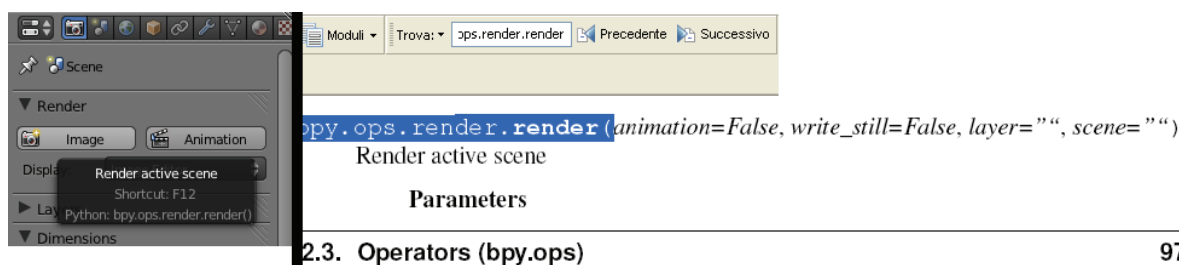
Il file **.pdf** della documentazione è in Inglese ed è scaricabile gratuitamente; una copia di tale file è presente all'indirizzo <http://www.redbaron85.com/BPY259.pdf>.

Per ricavare almeno i **nomi** di alcuni comandi, in modo da cercarne poi le specifiche dettagliate nel file di documentazione, è possibile ricorrere a due strumenti: l'**area di testo della finestra Info** e il **Tooltip** visualizzato quando il cursore del mouse si ferma su un elemento.

Il corpo della finestra **Info**, visualizzata nell'interfaccia di default di Blender in forma “ristretta” (solo l'header), è infatti **un'area di testo non scrivibile** che riporta alcune informazioni sulle operazioni svolte interattivamente dall'utente e che può tornare utile per ricavare i nomi di alcune **funzioni delle API Blender Python**; se, ad esempio, si inserisce una mesh **Box** nella scena mediante “**Add → Box**”, nella finestra **Info** comparirà una scritta che informerà che si è fatto uso della **funzione “primitive_cube_add”** di “**bpy.ops.mesh**” (operazioni, “**ops**”, riguardanti le mesh) delle **API Blender Python** per aggiungere una mesh primitiva di tipo “**box**” nella posizione del **3D Cursor**:



Quando si passa il cursore del mouse su un elemento della finestra del programma, spesso viene restituita nel **Tooltip** una stringa che indica il **nome** (o perlomeno la “parte finale” dello stesso, utilizzabile per una ricerca nel **pdf della documentazione**, mediante lo strumento di ricerca del programma o, in certi casi, l'indice) dell'elemento nelle **API Blender Python**; ad esempio, passando il mouse sul pulsante “**Image**” della scheda **Render**, all'interno della **Properties Window**, si avrà “**bpy.ops.render.render()**”, per cui per avere maggiori informazioni su questa funzione sarà sufficiente cercarla nella sezione “**Operator (bpy.ops)**” delle **API** (pagina 97 del file **.pdf**, in questo particolare caso):



2.3. Operators (bpy.ops)

97

Va suggerito inoltre l'utilizzo dei comandi *dir* e *type* di **Python** per capire che tipo di dato è un certo elemento e, in caso di classi o struct, quali sono i suoi campi e metodi. Negli esempi presenti nel libro si farà spesso uso di questi due comandi a fini didattici.

Creazione degli script e degli Add-Ons: strumenti disponibili

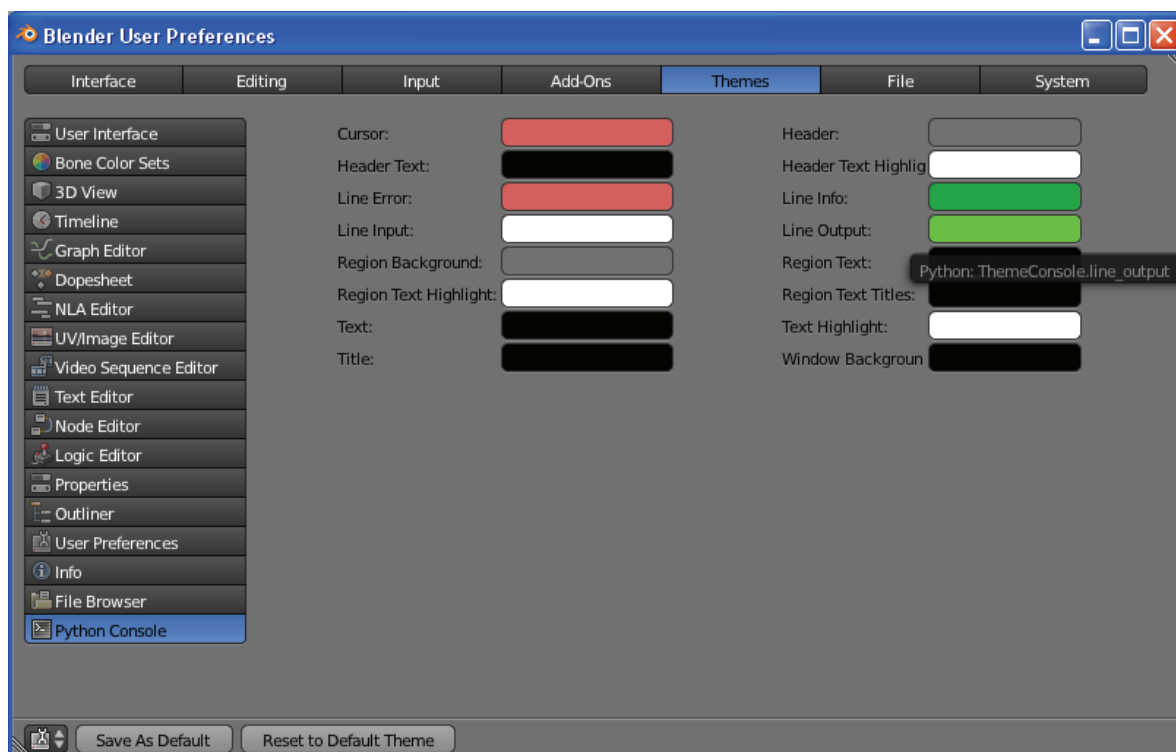
La **Console Python di Blender** consente di digitare ed eseguire singole istruzioni o script “al volo”: per utilizzarla, è sufficiente cambiare una finestra del programma in **Python Console**, iniziare a digitare le istruzioni nel **prompt** (i simboli `>>>`) e premere Invio per confermare; in caso di apertura di blocchi, come per **cicli for** o **comandi if**, la riga successiva mostrerà tre punti (...) al posto del prompt di default, ma non implementerà automaticamente l'**indentazione**, lasciando a noi il compito di premere una o più volte il tasto **TAB**.

Per uscire da un blocco, sarà sufficiente tornare a capo una volta con il prompt definito da “...”, lasciando **una riga vuota**; l'immagine seguente mostra come definire una variabile stringa e stampare tale stringa 5 volte mediante **ciclo for**:

```
>>> stringa="Ciao"
>>> for i in range(5):
...     print(stringa)
...
Ciao
Ciao
Ciao
Ciao
Ciao
>>> |
```

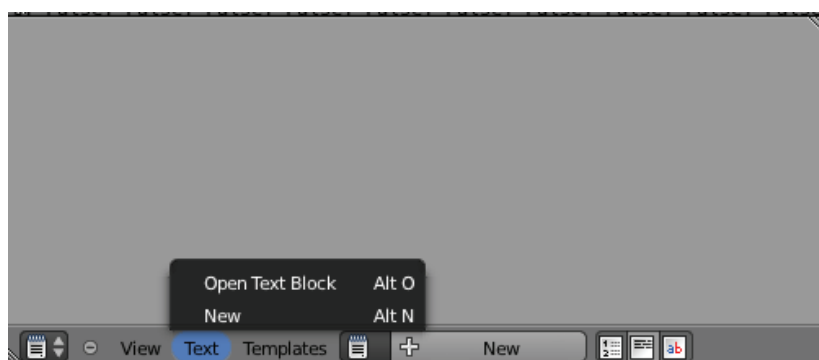
E' possibile cambiare il **tema dei colori** della finestra **Python Console** mediante le voci presenti nella sezione “**Themes → Python Console**” all'interno della finestra **User Preferences**,

mostrata in figura; ad esempio, per cambiare il colore dell'output della **Console** in verde, impostare questo colore nella casella **“Line Output”**.



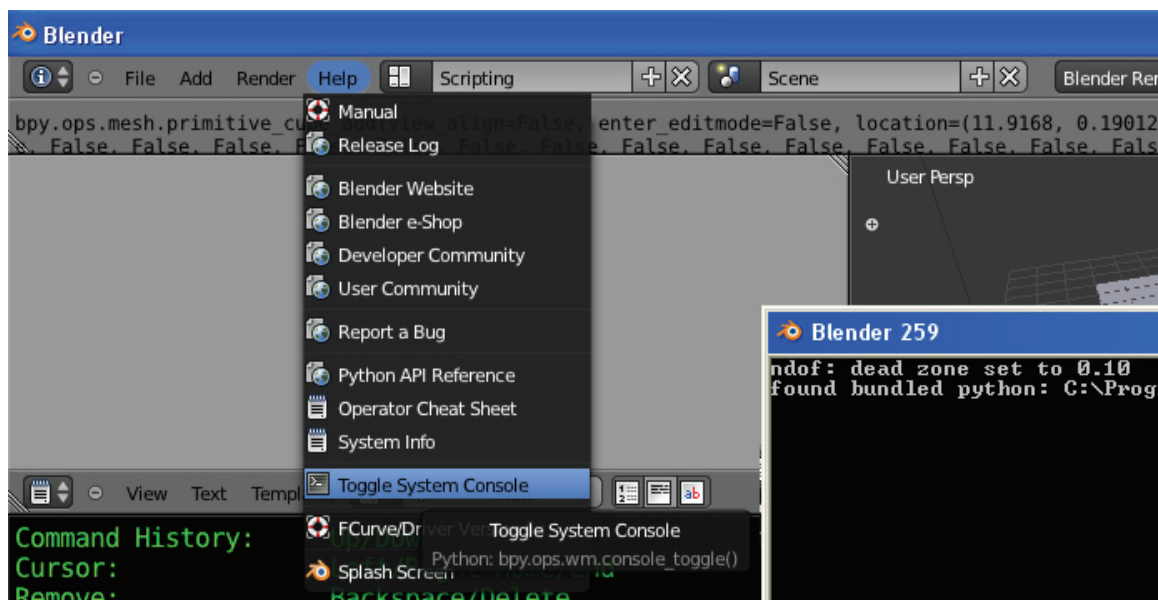
Per ingrandire o rimpicciolire la **dimensione del testo** nelle finestre **Python Console** o **Text Editor**, invece, utilizzare **“CTRL +”** e **“CTRL –”** (+ e – del tastierino numerico) oppure ruotare la **rotellina del mouse** mentre si tiene premuto il tasto CTRL.

Ovviamente, non si può pensare di dover ricopiare ogni volta le istruzioni nella **Console** ed eseguirle in questo modo, anche perché gli **Add-Ons** devono essere memorizzati in maniera permanente nel sistema per poter essere riutilizzati in seguito; un passo in avanti in tal senso è costituito dalla finestra **Text Editor**, che consente di creare, salvare o caricare un qualsiasi file di testo (**.txt**) o script (**.py**) all'interno di **Blender** e, nel caso si tratti di uno script, eseguirlo al volo cliccando sul pulsante **“Run Script”** nell'header della finestra.

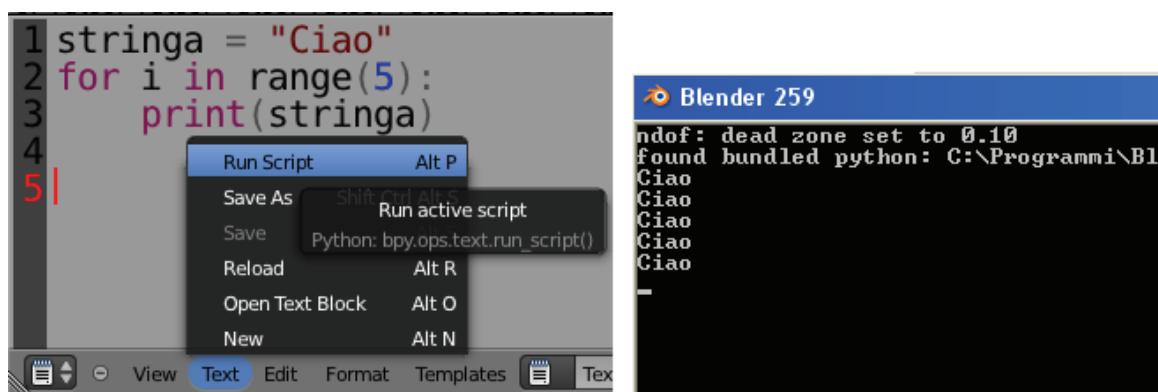


Per creare un nuovo file, ad esempio, è sufficiente cliccare su **“Text → New”** nell’header, iniziare a scrivere il testo o codice, salvare il file su disco con **“Text → Save”** o **“Text → Save As”** ed eventualmente (nel caso si tratti di uno script) eseguirlo cliccando su **“Run Script”** o **“Text → Run Script”**; ovviamente, è possibile caricare file precedentemente creati con **“Text → Open”**.

L’esecuzione di uno script da **Text Editor** o il test di un **Add-On** riporterà gli **errori a runtime** ed altri messaggi nella **Console Window** di **Blender**, a differenza di quanto avviene nella **Python Console** che riporta gli errori al suo interno; in **Blender 2.59**, la **Console Window** del programma *non* viene più visualizzata di default all’avvio dello stesso, ma per attivarla (ed osservare quindi l’output degli script o altri messaggi del programma) è sufficiente cliccare su **“Help → Toggle System Console”** nella barra del menù, all’interno della finestra **Info**.



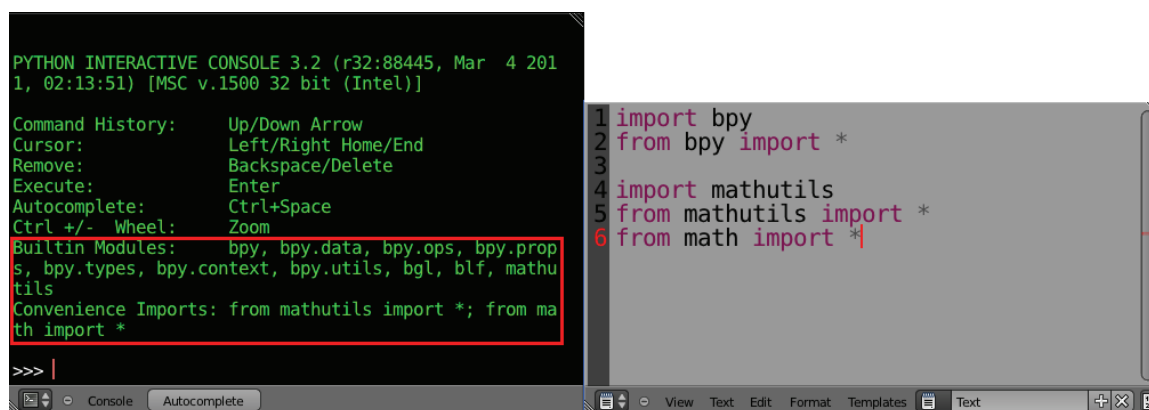
Inserendo nella finestra **Text Editor** lo stesso script visto poco fa trattando la **Python Console** ed eseguendolo con **“Run Script”**, ad esempio, l’output verrà visualizzato nella **Console Window**.



NOTA: la **Python Console** importa automaticamente alcuni **moduli** di base delle **API Blender Python** (*bpy*) e moduli accessori come *mathutils*; nella finestra **Text Editor**, questa operazione non viene effettuata quando si esegue uno script con **“Run Script”**, per cui sarà necessario **importare esplicitamente tutti i moduli aggiuntivi** necessari per eseguire lo script all’inizio del file.

L’importazione avviene mediante l’istruzione **“import”** seguita dal **nome del modulo**; i moduli di base di **Blender**, importati di default nella **Python Console**, sono:

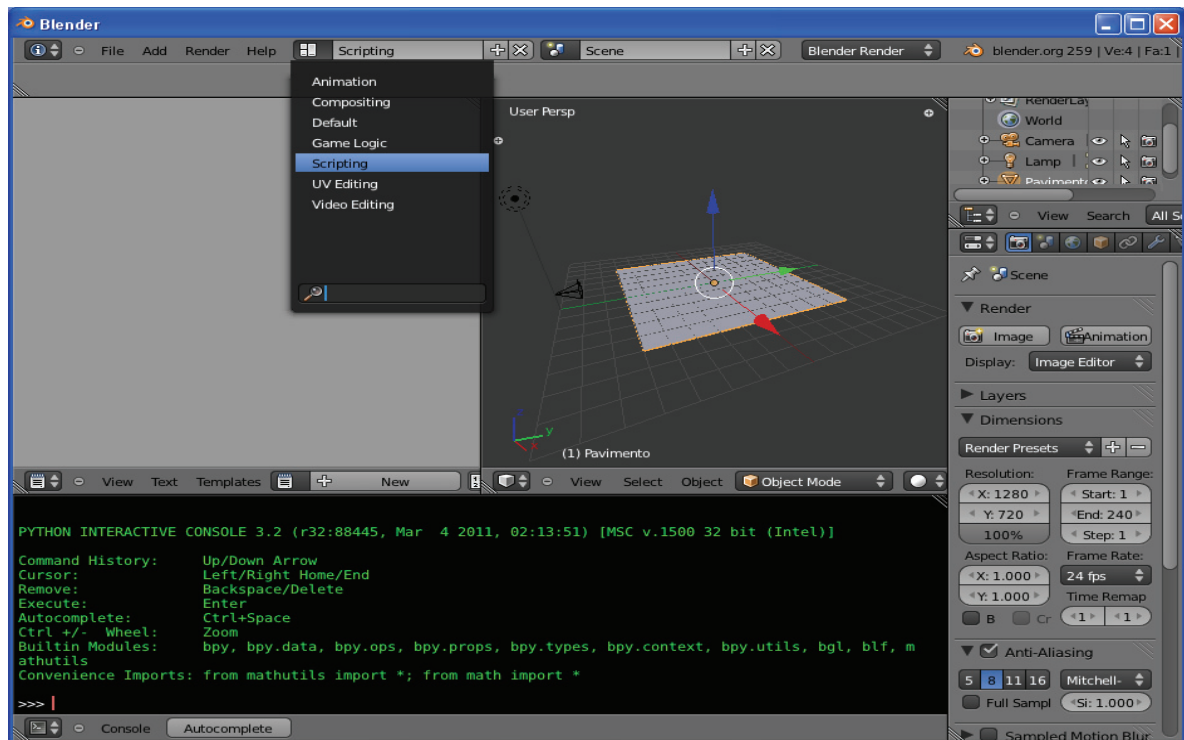
- *bpy* e i suoi sottomoduli *data*, *ops*, *props*, *types*, *context*, *utils*;
- *bgl*, per l’interfaccia (modulo **Blender OpenGL**);
- *blf*, per la resa del testo (font);
- *mathutils* e *math*, per funzioni e costanti matematiche.



Non fanno parte dei moduli di base e non verranno trattati in primo volume sottomoduli come *aud* (audio), *bge* e i suoi sottomoduli (**Game Engine**, il motore di gioco di **Blender**) ed altri elementi.

Tra le **Screens** messe a disposizione di default da **Blender**, una particolarmente utile per gli sviluppatori è **“Scripting”**, che suddivide la finestra del programma in modo da mostrare la finestra **Info** con un po’ della sua area di testo (che mostra le funzioni **Python** invocate interattivamente quando si utilizza il programma, ad esempio inserendo o trasformando un oggetto nella **3D View**), una finestra **Text Editor**, una finestra **Python Console** e una **3D View**, oltre ovviamente all’**Outliner** e alla **Properties Window**.

Tale layout è riportato in figura nella pagina seguente.



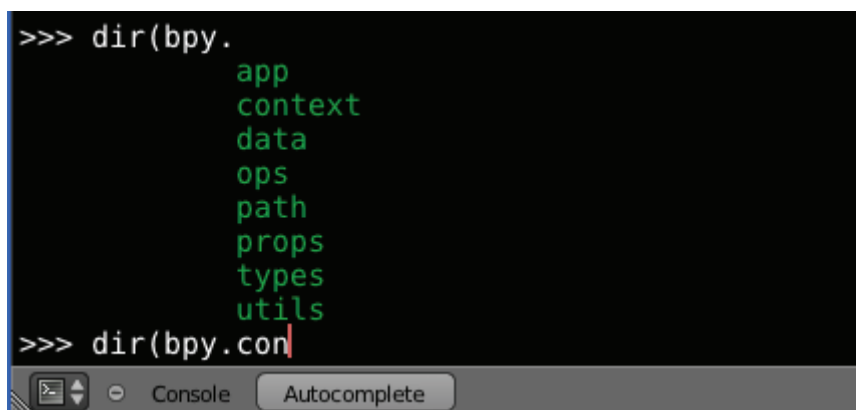
L'utilizzo del **Text Editor**, che riconosce i blocchi delle istruzioni **Python** e suggerisce automaticamente l'indentazione, è di fondamentale importanza per creare e salvare script e Add-Ons, tuttavia per progetti più complessi e per sfruttare funzioni come il *completamento automatico* o *Tooltip* con la documentazione riguardante un particolare elemento è necessario affidarsi ad un **IDE** provvisto delle **API** e del compilatore **Python**, ad esempio *Eclipse*, per il quale esistono anche plugin per programmare script con le **API Blender Python**.

A proposito del *completamento automatico*, prima di chiudere questo capitolo va detto che una forma di base di questa funzione è presente all'interno della **Python Console**, dov'è raffigurata mediante il pulsante "**Autocomplete**"; in particolare, se si fa click su tale pulsante mentre si digita del testo nella **Python Console**, il programma:

- suggerirà alcune opzioni possibili per completare il testo;
- completerà automaticamente il testo qualora ci sia solo un'opzione disponibile.

Anche se le voci trattate saranno chiare solo a partire dal prossimo capitolo, facciamo subito un esempio: digitando "*dir(bpy.*" e cliccando su **Autocomplete**, **Blender** elencherà i nomi dei sottomoduli di *bpy*, in modo da suggerirci le opzioni disponibili, mentre digitando "*dir(bpy.con*" e cliccando su **Autocomplete**, **Blender** completerà automaticamente la scritta rilasciando il cursore

subito dopo `dir(bpy.context)`, visto che `context` è l'unica opzione disponibile per `bpy` con prefisso `con`.



```
>>> dir(bpy.  
      app  
      context  
      data  
      ops  
      path  
      props  
      types  
      utils  
>>> dir(bpy.con
```

Le prime operazioni che faremo per interagire con **Blender** mediante le sue **API Python** verranno effettuate tramite **Python Console**, ma ovviamente è possibile inserirle in un file di testo, salvarle ed eseguirle mediante **Text Editor**, a patto di scrivere all'inizio del file:

```
import bpy  
from bpy import *           # importa tutti gli elementi di bpy; * sta per "tutto"  
import mathutils  
from mathutils import *  
from math import *
```

in modo da importare `bpy`, `mathutils` (per funzioni e costanti matematiche) e i loro sottomoduli.

Descrizione dei principali sottomoduli di bpy

Come anticipato, il modulo principale di accesso alle funzioni delle **API Blender Python** è `bpy` (abbr. di **Blender PYthon**, appunto); i suoi sottomoduli principali, alcuni dei quali verranno trattati in questo volume, raggruppano classi, campi e metodi relativi al contesto di esecuzione del programma, ai tipi di elementi utilizzabili in **Blender** (non solo oggetti della scena 3D come `Material`, `Mesh`, `Lattice` o altro, ma anche elementi puramente informatici come collezioni, dizionari, ecc...) ed altro ancora:

- `context`

Fornisce indicazioni sul `contesto` attuale del file **Blender** corrente, ossia qual è l'oggetto o quali sono gli oggetti selezionati, qual è l'oggetto attivo, ecc...

- **data**

Consente di accedere a tutti gli **oggetti presenti all'interno del file Blender corrente**; in realtà, ciascun elemento di data è una collezione di oggetti di un certo tipo: ad esempio, *bpy.data.objects* è una **collection** (sostanzialmente, una lista) di tutti gli oggetti presenti, mentre *bpy.data.meshes* è una **collection** delle sole mesh presenti, ecc... E' possibile quindi ricavare un particolare elemento della scena e associarlo ad una variabile per poterlo modificare in seguito.

- **ops**

Contiene tutti gli **“operatori”** di Blender, ossia le operazioni effettuabili; ad esempio, *“bpy.ops.subdivide(number_cuts=3, smoothness=0.5)”* applica il modificatore **Subdivide** all'oggetto attualmente selezionato (ossia: l'oggetto con proprietà *“select”* a **True**, come vedremo in seguito) con 3 **iterazioni** e **smoothness** 0.5.

- **types**

Contiene le definizioni delle **classi** utilizzate nei moduli *bpy.data*, *bpy.context* e *bpy.ops*; esempi: **Action** (generica azione di animazione, nel **Dope Sheet**), **ColorRamp** (ramp che associa un valore numerico ad un colore, come nel caso degli ombreggiatori diffuso e speculare); ogni elemento mette a disposizione campi e metodi, ciascuno con i propri parametri di input e output, elencati dettagliatamente nella Documentazione ufficiale delle **Blender Python API** (es.: *evaluate(position)* di **ColorRamp** prende in input un float tra 0.0 e 1.0 che identifica un punto nella **Ramp** e restituisce il valore numerico associato al colore presente in quel punto della **Color Ramp**).

- **utils**

Contiene alcune funzioni “generiche”, utili ma non associate ad un particolare contesto.

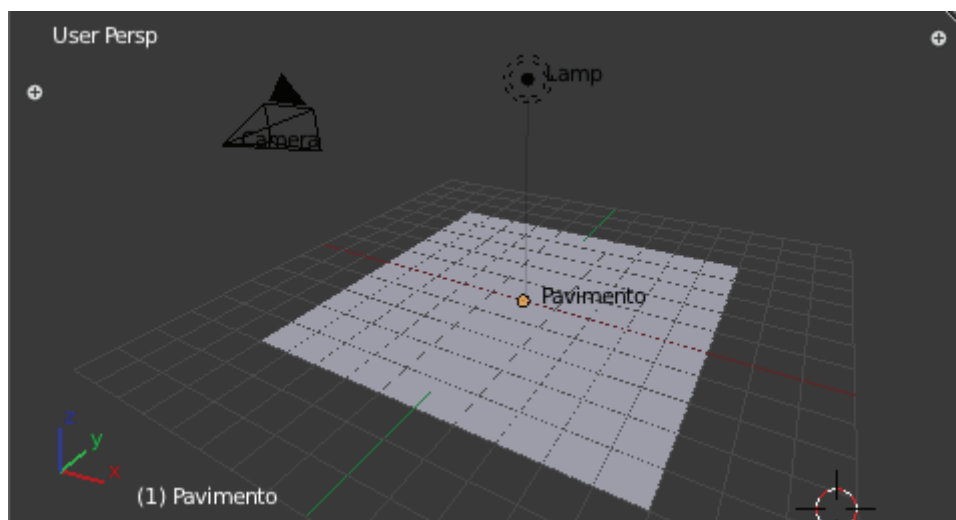
Le caratteristiche e le funzioni dei vari moduli diventeranno sempre più chiare e familiari con la pratica, nel corso dei prossimi capitoli.

* * *

CAPITOLO 3

Primi passi con le Blender Python API

***PREMESSA:** iniziamo con una scena semplicissima, contenente una fonte di luce di nome “Lamp”, una telecamera “Camera” e un plane di nome “Pavimento”.*



La prima cosa che, istintivamente, viene da chiedersi guardando la scena corrente e la finestra Python Console, con il prompt `>>>` pronto a ricevere i nostri comandi, è: “come faccio a selezionare un oggetto nella scena [e, in seguito, modificare qualche sua proprietà]?”

Ecco quindi il primo oggetto delle Blender Python API che dobbiamo conoscere: “*bpy.data.objects*”, di tipo *bpy_prop_collection*, come è possibile verificare digitando:

```
>>> type(bpy.data.objects)
<class 'bpy_prop_collection'>
```

Il fatto di trovarci in presenza di una *“collection”*, una collezione di elementi, ci suggerisce la possibilità di poter scorrere tale insieme come se si trattasse di un elenco, o meglio, una *list* di elementi; la nostra supposizione è corretta, per cui scriviamo:

```
>>> for elemento in bpy.data.objects:
...     print(elemento)
...
<bpy_struct, Object("Camera")>
<bpy_struct, Object("Lamp")>
<bpy_struct, Object("Pavimento")>
>>> |
```

Attenzione, nello scrivere la seconda riga, a premere una volta il tasto TAB per ottenere la corretta **indentazione**, ponendo cioè il comando *print* (che stampa a video le informazioni sull'elemento corrente preso dalla lista) all'interno del blocco dell'istruzione *for*; alla terza riga, premere Invio per uscire dal blocco *for* e concludere l'operazione.

All'interno di *bpy.data.objects*, gli oggetti vengono identificati mediante **stringhe**, in base al loro nome, per cui un altro modo per richiamare un particolare elemento è quello di passare il suo nome tra [] dopo *objects*, ad esempio in questo modo:

```
>>> pavimento = bpy.data.objects['Pavimento']
>>> |
```

La possibilità di assegnare un oggetto ad una variabile ci induce ad effettuare tale operazione, ad esempio, con *bpy.data.objects*, per evitare di dover scrivere questa stringa ogni volta che dobbiamo identificare uno o più oggetti della scena (questa ipotesi non è proprio corretta, come vedremo al termine di questo capitolo); scriviamo quindi:

```
>>> elementiScena = bpy.data.objects
>>> |
```

“elementiScena” avrà le stesse caratteristiche di *bpy.data.objects*, per cui potremo scorrere i suoi elementi con un ciclo *for*, richiamarne uno direttamente con *[“”]*, ecc...

Una precisazione: l'aver richiamato un oggetto dall'elenco di oggetti presenti nella scena, assegnandolo eventualmente ad una variabile, **non implica la sua selezione** (via script) nella scena 3D, operazione che vedremo comunque tra poco.

Ora che sappiamo selezionare un oggetto, come ricavarne e modificarne le proprietà?

Campi e metodi vengono descritti dettagliatamente nelle Blender Python API ma fortunatamente, almeno per le operazioni elementari, i loro nomi dicono in maniera abbastanza chiara a cosa servono.

Per ricavare le caratteristiche di un elemento, ricorriamo quindi ad uno dei nostri migliori amici: la funzione Python nativa *dir*.

Ricordando che *pavimento* è la variabile associata ad una mesh di nome “Pavimento” presente nella scena, scriviamo:

```
>>> dir(pavimento)
```

premiamo Invio ed analizziamo il risultato:

```
[ '_doc', '_module', '_slots', 'active_material', 'active_material_index', 'active_shape_key', 'active_shape_key_index', 'animation_data', 'animation_data_clear', 'animation_data_create', 'animation_visualisation', 'bl_rna', 'bound_box', 'children', 'closest_point_on_mesh', 'collision', 'color', 'constraints', 'copy', 'data', 'delta_location', 'delta_rotation_euler', 'delta_rotation_quaternion', 'delta_scale', 'dimensions', 'draw_bounds_type', 'draw_type', 'dupli_faces_scale', 'dupli_frames_end', 'dupli_frames_off', 'dupli_frames_on', 'dupli_frames_start', 'dupli_group', 'dupli_list', 'dupli_list_clear', 'dupli_list_create', 'dupli_type', 'empty_draw_size', 'empty_draw_type', 'empty_image_offset', 'field', 'find_armature', 'game', 'grease_pencil', 'hide', 'hide_render', 'hide_select', 'is_duplicator', 'is_modified', 'is_visible', 'layers', 'library', 'location', 'lock_location', 'lock_rotation', 'lock_rotation_w', 'lock_rotations_4d', 'lock_scale', 'material_slots', 'matrix_basis', 'matrix_local', 'matrix_world', 'mode', 'modifiers', 'motion_path', 'name', 'parent', 'parent_bone', 'parent_type', 'parent_vertices', 'particle_systems', 'pass_index', 'pose', 'pose_library', 'proxy', 'proxy_group', 'ray_cast', 'rna_type', 'rotation_axis_angle', 'rotation_euler', 'rotation_mode', 'rotation_quaternion', 'scale', 'select', 'shape_key_add', 'show_axis', 'show_bounds', 'show_name', 'show_only_shape_key', 'show_texture_space', 'show_transparent', 'show_wire', 'show_x_ray', 'soft_body', 'tag', 'time_offset', 'to_mesh', 'track_axis', 'type', 'up_axis', 'update_tag', 'use_dupli_faces_scale', 'use_dupli_frames_speed', 'use_dupli_vertices_rotation', 'use_fake_user', 'use_shape_key_edit_mode', 'use_slow_parent', 'use_time_offset_add_parent', 'use_time_offset_edit', 'use_time_offset_parent', 'use_time_offset_particle', 'user_clear', 'users', 'users_group', 'users_scene', 'vertex_groups']
```

C'è solo l'imbarazzo della scelta!

Utilizziamo, ad esempio, “*type*”, analizzandone per prima cosa il tipo restituito, informazione che possiamo ottenere digitando:

```
>>> type(pavimento.type)
<class 'str'>
```

che restituisce “*str*”, ossia una stringa. Digitiamo quindi:

```
>>> pavimento.type
'MESH'
```

per far stampare a video, sotto forma di stringa, il **tipo** dell'oggetto identificato da *pavimento*: “**MESH**”.

Proviamo quindi ad effettuare un'operazione di “filtraggio” dell'elenco di elementi presenti nella scena decidendo di stampare i nomi di quelli di tipo mesh... beh, prima dobbiamo sapere come ricavare il **nome** di un oggetto, ma dando un'occhiata all'output di *dir(pavimento)* notiamo subito “**name**”, dal significato ovvio, quindi procediamo scrivendo direttamente:

```
>>> for elemento in elementiScena:
...     if(elemento.type=="MESH"):
...         print(elemento.name)
...
Pavimento
```

Possiamo fare di più: possiamo creare **una nuova lista** (o collezione) di elementi scegliendo, a partire da quelli presenti nella scena, solo quelli di tipo mesh (nel nostro caso, solo il plane “**Pavimento**”); scriviamo quindi:

```
>>> listaMesh = []
>>>
>>> for elemento in elementiScena:
...     if(elemento.type == "MESH"):
...         listaMesh.append(elemento)
...
>>> |
```

Con la prima istruzione definiamo una nuova **list** vuota di nome “**listaMesh**”, mentre con l'ultima aggiungiamo un elemento in coda a tale lista.

NOTA IMPORTANTE: nel secondo esempio di codice riguardante questa iterazione, abbiamo aggiunto alla lista *l'elemento*, non il valore del suo attributo **name**, stampato invece nel primo blocco di codice; la differenza è notevole: nel primo caso passiamo solo una stringa rappresentante il valore di un campo dell'oggetto, nel secondo caso tutto l'oggetto (con tipo, metodi, campi).

A loro volta, tutti gli elementi presenti nella nuova list avranno campi e metodi degli oggetti associati, per cui scrivendo **listaMesh[0].name** (utilizzando quindi l'indice numerico per prelevare un elemento dalla lista di mesh) otterremo:

```
>>> listaMesh[0].name
'Pavimento'
>>> |
```

Esempi di “**types**” di oggetti della scena: *MESH*, *LAMP*, *CAMERA*, *EMPTY*, *CURVE*, *SURFACE*, *LATTICE*, ...

Ora che abbiamo preso familiarità con l’operazione di assegnazione di uno o più oggetti ad una variabile e recupero di tipo, campi e metodi di un oggetto, possiamo approfondire la nostra conoscenza delle **Blender Python API**.

Come anticipato, il recupero di un elemento dalla lista di elementi presenti nella scena non implica la sua *selezione*, all’interno della stessa, nel modo in cui siamo abituati a selezionare un elemento nella 3D View e a considerarlo come “oggetto selezionato” e, in certi casi, “oggetto attivo”.

Nella 3D View selezioniamo, ad esempio, sia il plane **“Pavimento”** che l’oggetto telecamera (selezione multipla, con la telecamera oggetto attivo in quanto oggetto selezionato per ultimo).

Per identificare gli **oggetti selezionati nella scena**, Blender utilizza una particolare *collection*: `bpy.context.selected_objects`, per cui scriviamo:

```
>>> oggettiSelezionati = bpy.context.selected_objects  
>>> |
```

per ottenere la lista di elementi attualmente selezionati nella scena e, contestualmente, assegnarla ad una variabile di nome **“oggettiSelezionati”** (per comodità d’uso in seguito).

Da notare che siamo passati dal **modulo DATA** al **modulo CONTEXT** per recuperare questa lista.

Digitiamo **oggettiSelezionati** nella **Console Python** ed osserviamo l’output:

```
>>> oggettiSelezionati  
[bpy.data.objects["Pavimento"], bpy.data.objects["Camera"]]
```

E’ possibile capire mediante il codice Python se un oggetto è selezionato, all’interno della scena, ed eventualmente deselectionarlo o, al contrario, inserirlo tra gli oggetti selezionati?

Ovviamente sì e in effetti basta esaminare l’output di *dir([elemento])*, già mostrato nel caso della variabile **pavimento** (associata all’oggetto “Pavimento” della scena), per notare la presenza di **“select”**...

...ma cos’è **“select”**? E’ una funzione, per cui dobbiamo invocarla per selezionare o deselectionare un elemento, o è una variabile (un “flag”, come si dice in questi casi)?

Per scoprirlo, scriviamo:

```
>>> type(pavimento.select)
<class 'bool'>
>>> |
```

L'output, `<class 'bool'>`, ci dice che siamo in presenza di una **variabile booleana**, che può assumere i valori **True** o **False** (senza "").

Per capire se un oggetto è selezionato nella **3D View**, quindi, è sufficiente scrivere:

[identificatoreElemento].select

ad esempio tra le forme possibili abbiamo:

```
>>> pavimento.select
True

>>> bpy.data.objects['Pavimento'].select
True

>>> bpy.data.objects[0].select
True

>>> listaMesh[0].select
True
```

ecc... ed analizzare il risultato (nel nostro caso, **True** perché abbiamo selezionato sia la mesh **“Pavimento”** che la telecamera).

Volendo aggiungere, alla lista degli elementi selezionati nella scena, anche la fonte di luce, possiamo scrivere:

```
>>> elementiScena['Lamp'].select = True
>>> |
```

Stiamo cioè cambiando la proprietà **“select”** dell'oggetto fonte di luce **“Lamp”** in vera, selezionando di fatto l'oggetto, per cui digitando:

```
>>> bpy.context.selected_objects
[bpy.data.objects["Pavimento"], bpy.data.objects["Lamp"], b
py.data.objects["Camera"]]
>>> |
```

avremo in output l'elenco degli oggetti selezionati, tra i quali figura ora anche la Lamp.

Va da sé che per *deselezionare* un oggetto nella scena sarà sufficiente scrivere:

[identificativoElemento].select = False

utilizzando una delle varie forme possibili per identificare un elemento (nome, per indice numerico, ecc...).

Per deselezionare tutti gli elementi presenti nella scena è sufficiente scrivere:

```
>>> for obj in bpy.data.objects:
...     obj.select = False
...
>>> |
```

mentre per selezionarli tutti dovremo scrivere:

```
>>> for obj in bpy.data.objects:
...     obj.select = True
...
>>> |
```

Con tutti gli elementi della scena selezionati, dopo l'ultima operazione, approfittiamone per mostrare **una brutta sorpresa**: digitando

```
>>> oggettiSelezionati
[bpy.data.objects["Pavimento"], bpy.data.objects["Camera"]]
>>> |
```

non otterremo, in output, la lista *aggiornata* degli elementi attualmente selezionati!

Questa considerazione vale per tutte le variabili create *a partire da elementi context*, in quanto il contesto cambia continuamente, e nelle variabili create *a partire da altre variabili definite in “stati precedenti”*, perché in questi casi non c'è un aggiornamento a catena; ad esempio, aggiungendo un cubo nella scena e digitando:

```
>>> for obj in elementiScena:
...     print(obj.name)
...
Camera
Cube
Lamp
Pavimento
>>> |
```

avremo tale oggetto nell'elenco restituito, ma non lo troveremo nella lista *“listaMesh”*, definita da un altro oggetto prima dell'inserimento del cubo:

```
>>> for elemento in listaMesh:
...     print(elemento.name)
...
Pavimento
>>> |
```

Il bicchiere mezzo vuoto è che non possiamo affidarci ad una variabile di comodo, più facile da ricordare, per identificare certe liste “native” delle API, perché tale variabile non viene aggiornata automaticamente; di contro, il bicchiere mezzo pieno è che possiamo creare liste di particolari elementi ed essere sicuri che non muteranno, “congelando” così una determinata configurazione della scena, per utilizzarla anche dopo varie operazioni sugli elementi in essa presenti.

Ricapitolando, in questo capitolo abbiamo visto:

- come ottenere la lista degli elementi presenti nella scena;
- come identificare un particolare elemento tra quelli nella scena e assegnarlo ad una variabile;
- come ricavare tipo, campi e metodi di un elemento, analizzando poi tali proprietà;
- come “filtrare” la lista degli elementi della scena isolando, ad esempio, solo le mesh;
- come selezionare o deselectare uno o più elementi.

* * *

Capitolo 4

Trasformazioni in Object Mode

Nel capitolo precedente abbiamo visto come elencare e selezionare gli elementi della scena ed accedere all’elenco dei loro campi e metodi; abbiamo visto, inoltre, come cambiare una proprietà, ed in particolare l’attributo “*select*”, di tipo **booleano**.

Esistono varie proprietà, decisamente più interessanti, molto diverse dal tipo **boolean**; in questo capitolo, daremo un’occhiata ad alcune di queste proprietà, come ad esempio *location* e *scale*, approfittandone per parlare del tipo *Vector* e di altri argomenti.

Utilizziamo la scena descritta nel capitolo precedente, composta da una fonte di luce “*Lamp*”, una telecamera “*Camera*” e un plane chiamato “*Pavimento*”.

Abbiamo visto come identificare questi elementi mediante il loro nome, associato all’attributo “*name*”, di tipo **Stringa**; va da sé che è possibile rinominare un oggetto, anche parzialmente (ossia: aggiungendo un prefisso o un suffisso o cambiando parte del nome) con i metodi propri di **String**; esempi di manipolazioni (prefisso, suffisso, concatenazione, sostituzione) vengono mostrati in figura nella pagina seguente.

```
>>> pavimento.name
'Pavimento'

>>> pavimento.name = "___" + pavimento.name
>>> pavimento.name
'___Pavimento'

>>> pavimento.name = pavimento.name + ".001"
>>> pavimento.name
'___Pavimento.001'

>>> pavimento.name = pavimento.name + bpy.data.objects['Lamp'].name
>>> pavimento.name
'___Pavimento.001Lamp'

>>> pavimento.name.replace(".001", "_")
'___Pavimento_Lamp'

>>> |
```

Tra le proprietà degli oggetti elencate nell'output di *dir([identificativoOggetto])* notiamo quelle che si riferiscono alle caratteristiche **Object Mode** fondamentali degli elementi, ossia **Location**, **Dimensions**, **Scale** e alcune voci di tipo **Rotation**.

Tralasciamo per il momento le operazioni riguardanti l'orientamento (**Rotation**) degli oggetti ed esaminiamo **Location**, **Dimensions** e **Scale**.

Come fare per ricavare le coordinate di posizione X, Y e Z di un oggetto ed eventualmente trasformarle? In pratica: con che tipo di dato sono immagazzinate queste informazioni?

Per scoprirlo, è sufficiente digitare:

```
>>> type(pavimento.location)
<class 'mathutils.Vector'>
```

che ci dà informazioni sul *tipo* (un *Vector* del modulo *mathutils*), mentre per scoprire il *contenuto* è sufficiente digitare:

```
>>> pavimento.location
Vector((0.0, 0.0, 0.0))
```

Le stesse operazioni, se effettuate su **Scale** e **Dimensions**, restituiscono risultati molto simili a quelli visti per **Location**, come mostrato in figura nella pagina seguente.

```
>>> type(pavimento.scale)
<class 'mathutils.Vector'>

>>> pavimento.scale
Vector((1.0, 1.0, 1.0))

>>> type(pavimento.dimensions)
<class 'mathutils.Vector'>

>>> pavimento.dimensions
Vector((10.0, 10.0, 0.0))

>>> |
```

Resta quindi da vedere come trattare questi oggetti **Vector**, che ad una prima occhiata sembrano liste di valori (es.: le coordinate X, Y e Z dell'Origin dell'oggetto, nel caso di Location)... beh, in effetti i **Vector** sono liste di 2, 3 o 4 elementi, di tipo **float** o **interi**, che in più mettono a disposizione funzioni per implementare il prodotto scalare, il prodotto vettoriale e altre operazioni.

Per accedere ad un valore di un **Vector** è necessario utilizzare l'indice, ossia la posizione che tale valore ha nell'insieme di valori del vettore, specificandolo tra **[]**; ad esempio, considerando che **Location** contiene le coordinate X, Y e Z di un oggetto in questo ordine, per ottenere la coordinata Y del plane **"Pavimento"** (identificato dalla variabile pavimento) sarà sufficiente scrivere:

```
>>> pavimento.location[1]
0.0
```

oppure utilizzare i nomi associati a tali campi: agli indici 0, 1, 2 e 3, **Vector** associa infatti i campi **'x'**, **'y'**, **'z'**, **'w'** (queste ultime due non sono sempre presenti, dipende dal vettore), per cui per ottenere lo stesso scopo di prima possiamo scrivere:

```
>>> pavimento.location.y
0.0
```

Non sarà sfuggita la forma della "chiamata", diversa dal solito: in effetti, qui non c'è una vera e propria chiamata (**'y'** non è una funzione, altrimenti avremmo scritto **.y()**), né ci si sta riferendo al campo con la **chiave** **['y']**, ma con **.y**: **'y'** è infatti un campo (field) di **Vector** e si accede ai campi di un oggetto direttamente con **.[nome]**.

Un altro campo di **Vector** è **length**, che restituisce il modulo del vettore.

Il valore di un campo di *Vector* può essere assegnato ad una variabile, in modo da memorizzarlo, trattarlo a parte ed eventualmente ri-assegnarlo al *Vector* originale:

```
>>> pavimento.location.y
0.0

>>> yDiPavimento = pavimento.location.y
>>> yDiPavimento = 2
>>> pavimento.location.y = yDiPavimento
>>> pavimento.location
Vector((0.0, 2.0, 0.0))

>>> |
```

(**nota:** l'operazione appena effettuata è del tutto inutile, serviva solo a mostrare la possibilità di estrapolare o riassegnare una componente di *Vector*; in realtà, per ottenere lo stesso scopo, sarebbe bastato scrivere: “*pavimento.location.y=2.0*”).

Diamo un'occhiata alla scena 3D: il **plane** non si trova più nella posizione originale, ma è stato spostato di 2 unità lungo l'asse Y!

Abbiamo appena effettuato la nostra prima trasformazione Object Mode mediante script Python.

E' possibile effettuare operazioni sull'intero *Vector*, ad esempio:

```
>>> mioVettore = Vector((2.0, 3.0, 4.0))
>>> mioVettore
Vector((2.0, 3.0, 4.0))

>>> pavimento.location
Vector((0.0, 2.0, 0.0))

>>> pavimento.location = pavimento.location + mioVettore
>>> pavimento.location
Vector((2.0, 5.0, 4.0))

>>> |
```

definisce un nuovo vettore (mediante la funzione costruttore di *Vector*, che crea un nuovo *Vector* assegnandogli al volo i valori passati) ed effettua la somma vettoriale (membro a membro) delle componenti di tale vettore e di Location, implementando una traslazione, mentre con:

```
>>> pavimento.dimensions = pavimento.dimensions * 3.0
>>> |
```

triplicheremo le dimensioni dell'oggetto (prodotto di uno scalare per le componenti di un vettore, in questo caso *Dimensions*).

Ecco alcuni metodi utili forniti da questa classe:

- **copy()** --- Restituisce una copia del vettore che lo invoca.
- **zero()** --- Imposta a 0 tutti i campi del vettore che lo invoca.
- **normalize()** --- Normalizza il vettore, facendo sì che il modulo del vettore così ottenuto sia uguale a 1.
- **negate()** --- Cambia il valore di ogni campo del vettore nel suo opposto.
- **resize2D()** --- Cambia il vettore in 2D.
- **resize3D()** --- Cambia il vettore in 3D, con nuovo valore 0.0.
- **resize4D()** --- Cambia il vettore in 4D, con valore z = 0.0 e valore w = 1.0.
- **reflect(simmetria)** --- Restituisce il vettore "simmetrico" di quello che lo ha invocato, con asse di simmetria rappresentato dal vettore "simmetria" passato come argomento.
- **cross(vettore2)** --- Restituisce, sotto forma di Vector, il prodotto vettoriale tra il vettore che lo invoca e un vettore passato come parametro ("vettore2"). Entrambi i vettori devono avere tre dimensioni.
- **dot(vettore2)** --- Restituisce, sotto forma di valore float (numero in virgola mobile), il prodotto scalare tra il vettore che lo invoca e un vettore passato come parametro ("vettore2"). I vettori devono avere la stessa dimensione.

Parliamo a questo punto di **Rotation**, o meglio delle voci riguardanti l'orientamento degli oggetti.

Si tratta, infatti, di tre elementi:

- **rotation_axis_angle**
- **rotation_euler**
- **rotation_quaternion**

ai quali bisogna affiancare anche **rotation_mode**, che può assumere uno dei seguenti valori: 'QUATERNION', 'XYZ', 'XZY', 'YXZ', 'YZX', 'ZXY', 'ZYX', 'AXIS_ANGLE' (default: XYZ).

I primi tre **rotation_** sono presenti in quanto in Computer Grafica, a livello implementativo, le rotazioni vengono definite generalmente con la **matrice di trasformazione di un elemento** (che integra informazioni su traslazione, rotazione e scaling in un unico oggetto) oppure mediante la definizione di **quaternioni** (dal punto di vista pratico, si tratta di vettori di 4 elementi, detti XYZW).

Poiché l'utilizzo della matrice di trasformazione o dei quaternioni non è intuitivo, le API di programmazione 3D mettono spesso a disposizione le modalità **Axis Angle** ed **Euler**; in questa trattazione, parleremo di *rotation_euler*.

La classe **Euler** appartiene al modulo *mathutils* e permette di *impostare l'orientamento* (non di “ruotare”, quindi!!) di un elemento intorno ad un determinato asse di rotazione per un certo numero di radianti; in particolare, mette a disposizione i campi:

- **x** --- Angolo di rotazione, in radianti, intorno all'asse x (**heading**).
- **y** --- Angolo di rotazione, in radianti, intorno all'asse y (**pitch**).
- **z** --- Angolo di rotazione, in radianti, intorno all'asse z (**roll**).

e i metodi:

- **zero()** --- Imposta tutti i campi a 0.
- **copy()** --- Restituisce una copia dell'oggetto **Euler**.
- **toMatrix()** --- Restituisce una matrice (oggetto *Matrix* delle *Mathutils*) rappresentante l'oggetto **Euler** che ha invocato il metodo.
- **toQuat()** --- Restituisce un quaternione (oggetto *Quaternion* delle *Mathutils*) rappresentante l'oggetto **Euler** che ha invocato il metodo.

per cui, se ad esempio si desidera *porre pavimento a 30° intorno all'asse X*, sarà sufficiente scrivere:

```
>>> pavimento.rotation_euler.x = pi/6.0
>>> pavimento.rotation_euler
Euler((0.5235987901687622, 0.0, 0.0), 'XYZ')
```

(da notare che “pi” è riconosciuta come costante primitiva Python che identifica il valore π) mentre per *ruotare ulteriormente pavimento di 30° intorno all'asse X*, dovremo scrivere:

```
>>> pavimento.rotation_euler
Euler((0.5235987901687622, 0.0, 0.0), 'XYZ')
>>> pavimento.rotation_euler.x = pavimento.rotation_euler.x
+ (pi/6.0)
>>> pavimento.rotation_euler
Euler((1.0471975803375244, 0.0, 0.0), 'XYZ')
>>> |
```

La classe **Euler** permette di sommare due oggetti di questo tipo mediante operatore “+”, per cui non possiamo definire un **Euler** rappresentante le entità delle rotazioni intorno ai vari assi e sommarlo a *rotation_euler*.

Grazie a Python, la modellazione “parametrica”, soprattutto nell’allestimento della scena, può essere sfruttata al massimo; supponiamo, ad esempio, di avere una mesh rappresentante una colonna nella nostra scena 3D, con tale oggetto chiamato appunto “**Colonna**”, e di dover disporre alcune copie di tale oggetto in punti e con orientamenti ben determinati nella scena 3D, ad esempio 5 copie (oltre all’originale, da lasciare nella sua posizione e col suo orientamento) nelle posizioni:

5.0, 3.0, 0.0
10.0, 3.0, 0.0
7.0, 6.0, 0.0
2.0, 10.0, 0.0
8.0, 8.0, 0.0

e con orientamenti Z: 30.0°, 45.0°, 20.0°, 30.0°, 0.0°; anziché copiare e disporre gli oggetti nella 3D View, possiamo scrivere nella **Python Console** lo script riportato qui di seguito.

```
>>> listaPosizioni = [Vector((5.0, 3.0, 0.0)), Vector((10.0, 3.0, 0.0)), Vector((7.0, 6.0, 0.0)),  
... Vector((2.0, 10.0, 0.0)), Vector((8.0, 8.0, 0.0))]  
>>>  
>>> listaOrientamentiZ = [pi/6.0, pi/4.0, pi/9.0, pi/6.0, 0.0]  
>>>  
>>> listaCopie = ["Colonna2", "Colonna3", "Colonna4", "Colonna5", "Colonna6"]  
>>>  
>>> for obj in bpy.data.objects:  
...     obj.select = False  
...  
>>> bpy.data.objects['Colonna'].select = True  
>>>  
>>> for i in range(5):  
...     bpy.ops.object.duplicate()  
...     nuovaColonna = bpy.context.selected_objects[-1]  
...     nuovaColonna.location = listaPosizioni[i]  
...     nuovaColonna.rotation_euler.z = listaOrientamentiZ[i]  
...     nuovaColonna.name = listaCopie[i]  
...  
{'FINISHED'}  
{'FINISHED'}  
{'FINISHED'}  
{'FINISHED'}  
{'FINISHED'}  
>>> |
```

Analizziamo questo script passo per passo.

Le prime tre istruzioni servono a popolare altrettante **liste** di elementi di vario tipo, in particolare una lista di 5 **vettori** (le posizioni da assegnare alle colonne), una lista di 5 valori **float** (gli angoli di orientamento Z delle colonne) e una lista di 5 **stringhe** (i nomi da assegnare alle varie copie dell’oggetto iniziale).

```
>>> listaPosizioni = [Vector((5.0, 3.0, 0.0)), Vector((10.0, 3.0, 0.0)), Vector((7.0, 6.0, 0.0)),  
... Vector((2.0, 10.0, 0.0)), Vector((8.0, 8.0, 0.0))]  
>>>  
>>> listaOrientamentiZ = [pi/6.0, pi/4.0, pi/9.0, pi/6.0, 0.0]  
>>>  
>>> listaCopie = ["Colonna2", "Colonna3", "Colonna4", "Colonna5", "Colonna6"]
```

Seguono un ciclo *for* di **deselezione** di tutti gli elementi della scena e un'istruzione che ci consente di selezionare solo l'oggetto che ha nome **“Colonna”**; queste operazioni si rendono necessarie perché l'operatore di copia delle API Blender Python duplica *gli elementi attualmente selezionati nella scena*, per cui dobbiamo limitare questa operazione all'oggetto **“Colonna”**.

```
>>> for obj in bpy.data.objects:  
...     obj.select = False  
...  
>>> bpy.data.objects['Colonna'].select = True
```

Il ciclo *for* effettua **5 volte** un determinato set di operazioni:

- duplica l'oggetto selezionato, selezionando questa volta solo il nuovo oggetto copia; da notare che la funzione **“duplicate”** appartiene al **modulo ops**, contenente in generale le funzioni associate agli *operatori di modifica e trasformazione di Blender*;
- assegna l'oggetto copia (oggetto selezionato nella scena, quindi ultima voce nella lista **“selected_objects”**, che conosciamo bene) alla variabile temporanea **“nuovaColonna”**;
- imposta la posizione di **“nuovaColonna”** recuperando il vettore di elementi col relativo indice da *listaPosizioni*;
- imposta l'orientamento Z di **“nuovaColonna”** recuperando l'angolo di rotazione dalla lista *listaOrientamentiZ*;
- rinomina **“nuovaColonna”** prendendo il nome dalla lista **“listaCopie”**, utilizzando anche in questo caso l'indice **“i”** del ciclo *for* (da 0 a 4, estremi inclusi) per recuperare il valore corrente nella lista.

```
>>> for i in range(5):  
...     bpy.ops.object.duplicate()  
...     nuovaColonna = bpy.context.selected_objects[-1]  
...     nuovaColonna.location = listaPosizioni[i]  
...     nuovaColonna.rotation_euler.z = listaOrientamentiZ[i]  
...     nuovaColonna.name = listaCopie[i]  
...  
{'FINISHED'}  
{'FINISHED'}  
{'FINISHED'}  
{'FINISHED'}  
{'FINISHED'}  
>>> |
```

L'utilità di questo script può sembrare scarsa, ma con molti elementi da disporre è da preferire rispetto al posizionamento manuale nella 3D View e in presenza di **dati calcolati** (es.: posizioni e orientamenti degli oggetti derivanti da simulazioni “fisiche” come collisioni, gravità, ecc...) risulta addirittura indispensabile.

Prima di concludere il capitolo, una piccola nota: l'utilizzo dei **radianti** non è proprio intuitivo, ma fortunatamente è sufficiente **importare**, all'inizio dello script, il modulo **math** di Python ed utilizzare la sua funzione **radians(angolo)** per convertire in radianti il valore in gradi di un angolo; allo stesso modo, con la funzione **degrees(angolo)** è possibile convertire in gradi il valore in radianti di un angolo, come mostrato nella figura seguente.

```
>>> import math
>>>
>>> angolo = math.degrees(pi)
>>> angolo
180.0

>>> angolo = math.radians(180.0)
>>> angolo
3.141592653589793

>>> |
```

Abbiamo quindi visto come poter operare delle trasformazioni sulle proprietà fondamentali degli oggetti; ovviamente, è possibile implementare algoritmi che trasformino gli oggetti nel tempo, generando animazioni “calcolate”, dobbiamo solo sapere come spostarci tra i **frames** di un videoclip e inserire dei **keyframes** per gli oggetti... ne parleremo nel prossimo capitolo, esaminando un particolare “Case Study”.

* * *

Capitolo 5

Animazioni Object Mode: un case study

In questo capitolo prenderemo alcuni elementi delle *animation_data*, ossia le informazioni riguardanti le animazioni degli oggetti in Blender 3D 2.5; vedremo infatti come passare da un frame ad un altro nella **Timeline**, inserire dei keyframes per gli oggetti e persino come rendere ciclica una **Action** e cambiare l'interpolazione dei keyframes, il tutto **via codice**, senza effettuare operazioni interattive nella 3D View.

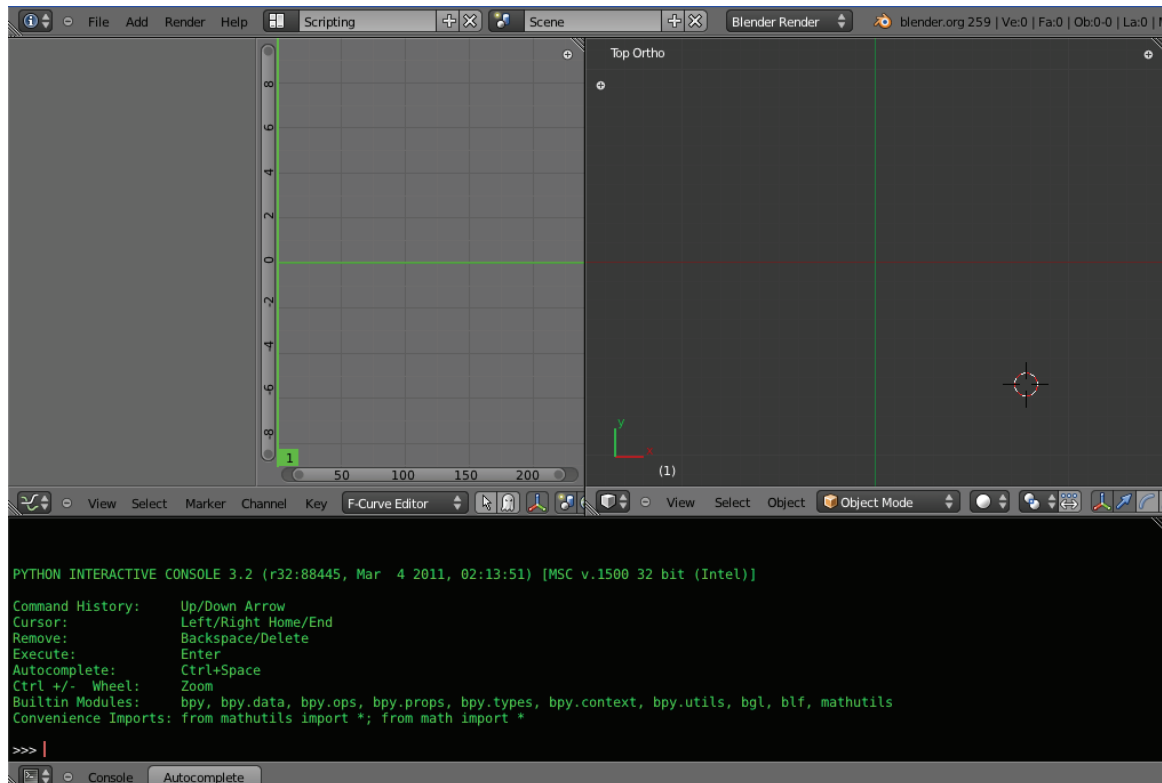
Anziché limitarci a elencare le API che riguardano tali elementi (che sono tante e non ci servono tutte), procederemo passo per passo nella realizzazione di uno script riguardante una simulazione “fisica”: l'orbita di un pianeta (anche se con qualche licenza rispetto alla realtà, come vedremo).

Per i dati, traiamo ispirazione dal pianeta **Mercurio**, che ha la particolarità di completare il moto di rotazione in 56 giorni terrestri e quello di rivoluzione in 88 giorni terrestri, per cui (effetto “spin/orbita”) su tale pianeta il “giorno” finisce con l'essere più lungo dell’“anno”.

Per realizzare questa animazione, dove effettueremo una corrispondenza tra i “giorni terrestri” e i “**frames**” del videoclip di Blender 3D, utilizzeremo una sfera per Mercurio, una per il Sole e una Empty, che servirà da Pivot Point per il moto di rivoluzione del pianeta, tuttavia **non dobbiamo aggiungere questi elementi alla scena**, anzi: cancelliamo tutto quello che c'è nella scena di default e iniziamo completamente da zero, in modo da inserire, posizionare e modificare gli elementi completamente **via codice**!

Iniziamo quindi da una **scena completamente vuota**.

Vi consiglio di impostare la **finestra dell'applicazione** come visibile nella figura seguente, con la **3D View** in vista **Top Ortho** in alto, in modo da tenere sott'occhio, la scena 3D (notando la creazione, il posizionamento e l'animazione degli elementi), la finestra **Console** e il **Graph Editor**, dove vengono visualizzate le **curve IPO** degli oggetti, curve che in seguito modificheremo mediante lo **script**.



Per prima cosa, aggiungiamo una **Empty** alla scena in posizione (0.0; 0.0; 0.0); tale Empty rappresenterà il Pivot Point di rotazione del pianeta Mercurio (per comodità, realizzeremo un'orbita circolare sul piano XY intorno al centro dell'universo virtuale... è una simulazione fisicamente errata sotto molti punti di vista, ma per i nostri scopi ciò non rappresenta un problema).

```
>>> bpy.ops.object.add(type="EMPTY", location=(0.0, 0.0, 0.0))
{'FINISHED'}
>>>
```

Si aggiunge una Empty ricorrendo all'operazione **bpy.ops.object.add**, come verificabile inserendo un qualsiasi oggetto nella scena ed esaminando la finestra Info di Blender.

Abbiamo posizionato la **Empty** durante la sua definizione; ciò è stato possibile passando come parametro per *add*, oltre al **tipo** di oggetto, il **vettore** di coordinate per l'attributo *location*.

Per inserire e posizionare la **Sphere** rappresentante Mercurio, ponendola in posizione iniziale (10.0; 0.0; 0.0), dobbiamo invece utilizzare la funzione *primitive_uv_sphere_add* di *mesh*:

```
>>> bpy.ops.mesh.primitive_uv_sphere_add(location=(10.0, 0.0, 0.0))
{'FINISHED'}

>>> Mercurio = bpy.data.objects['Sphere']
>>> |
```

che crea una **Sfera UV** con le impostazioni di default (32 segments, 16 rings, size 1).

Dopo aver effettuato questa operazione, assegniamo la sfera appena creata alla variabile *Mercurio*.

L'oggetto selezionato nella scena sarà proprio la sfera (il cui **nome** è sempre “*Sphere*”: *Mercurio* è una **variabile** alla quale abbiamo associato l'oggetto!), in quanto ultimo oggetto creato.

Al centro dell'universo virtuale, poniamo una **sfera** raffigurante il Sole; non dovremo modificare questa sfera in seguito, serve solo per ragioni “sceniche”:

```
>>> bpy.ops.mesh.primitive_uv_sphere_add(location=(0.0, 0.0, 0.0),
size=(2.0))
{'FINISHED'}
```

Selezioniamo la **Empty** e poniamoci al **frame 1** dell'animazione mediante la funzione *change_frame* di *anim*:

```
>>> for obj in bpy.data.objects:
...     obj.select = False
...
>>> bpy.data.objects['Empty'].select = True
>>>
>>> bpy.ops.anim.change_frame(frame=1)
{'FINISHED'}

>>> |
```

A questo punto, per poter **inserire un keyframe** su uno dei canali dell'oggetto selezionato, utilizziamo la funzione *keyframe_insert_menu* di *anim*:

```
>>> bpy.ops.anim.keyframe_insert_menu(type="Rotation")
{'FINISHED'}
>>> |
```

Ovviamente, la prossima cosa da fare è andare al **frame 88** (la rivoluzione di Mercurio avviene infatti in 88 giorni, nel nostro caso 88 frames), **ruotare di 360° (2π radianti) la Empty** ed inserire un nuovo **keyframe *Rotation***, per cui scriviamo:

```
>>> bpy.ops.anim.change_frame(frame=88)
{'FINISHED'}

>>> bpy.data.objects['Empty'].rotation_euler.z = 2*pi
>>> bpy.ops.anim.keyframe_insert_menu(type="Rotation")
{'FINISHED'}
>>> |
```

Con l'oggetto **Mercurio** dovremo effettuare delle operazioni molto simili a quelle appena viste per la Empty: selezionarlo, inserire un keyframe al frame 1 in posizione iniziale, andare al frame 56 (durata del suo periodo di rotazione), ruotare l'oggetto di 360° e inserire un nuovo keyframe...

```
>>> for obj in bpy.data.objects:
...     obj.select = False
...
>>> Mercurio.select = True
>>> bpy.ops.anim.change_frame(frame=1)
{'FINISHED'}

>>> bpy.ops.anim.keyframe_insert_menu(type="Rotation")
{'FINISHED'}

>>> bpy.ops.anim.change_frame(frame=56)
{'FINISHED'}

>>> Mercurio.rotation_euler.z = 2*pi
>>> bpy.ops.anim.keyframe_insert_menu(type="Rotation")
{'FINISHED'}
>>> |
```

Per poter implementare **una relazione di parentela**, è sufficiente impostare la **Empty** nel campo **parent** dell'oggetto **Mercurio** (probabilmente avrete notato la presenza di questo campo, insieme a molti altri, già dal capitolo 3, dall'output di *dir([mesh])*):

```
>>> Mercurio.parent = bpy.data.objects['Empty']
>>> |
```

Premendo **ALT-A** mentre il focus del mouse si trova nella **3D View**, sarà possibile osservare l’animazione generata fino a questo punto; tuttavia, le rotazioni si fermeranno ai **frame 56 e 88**: è giusto che sia così, ma per completare il lavoro sarebbe più opportuno **estendere ciclicamente le rotazioni** per tutta la durata del videoclip (di default, 250 frames).

Tale operazione viene effettuata applicando un **modificatore Cycles alle curve IPO** di rotazione delle mesh, tuttavia per mettere in pratica quanto detto dobbiamo prima capire dove e come sono memorizzate le **informazioni sulle animazioni degli oggetti** presenti nella scena.

Queste informazioni si trovano all’interno dell’oggetto **“animation_data”** associato all’elemento in questione, sia esso una mesh, un Lattice, una telecamera, ecc...

All’interno di **“animation_data”** possiamo poi individuare **“actions”**, associato alle **azioni** create automaticamente da Blender per un oggetto non appena si inserisce almeno un **keyframe** su qualche canale; all’intero delle **“actions”**, poi, abbiamo le **“fcurves”**, che rappresentano le **curve IPO** associate all’oggetto (per intenderci, quelle visibili nel **Graph Editor**).

Le **curve IPO** possono essere molteplici e per identificarle si fa uso di particolari **stringhe** per il campo **data_path**; tali stringhe sono indicate nelle **API Blender Python**; nel nostro particolare caso, visto che ci interessano le **curve IPO di rotazione** della Empty e di Mercurio, cercheremo le **fcurves “rotation_euler”**.

Per prima cosa, quindi, depositiamo in **“fcurvesMercurio”** e **“fcurvesEmpty”** le curve IPO di rotazione dei due oggetti **Mercurio** ed **Empty** (per comodità, definiamo una variabile **Empty** associandole il relativo oggetto della scena).

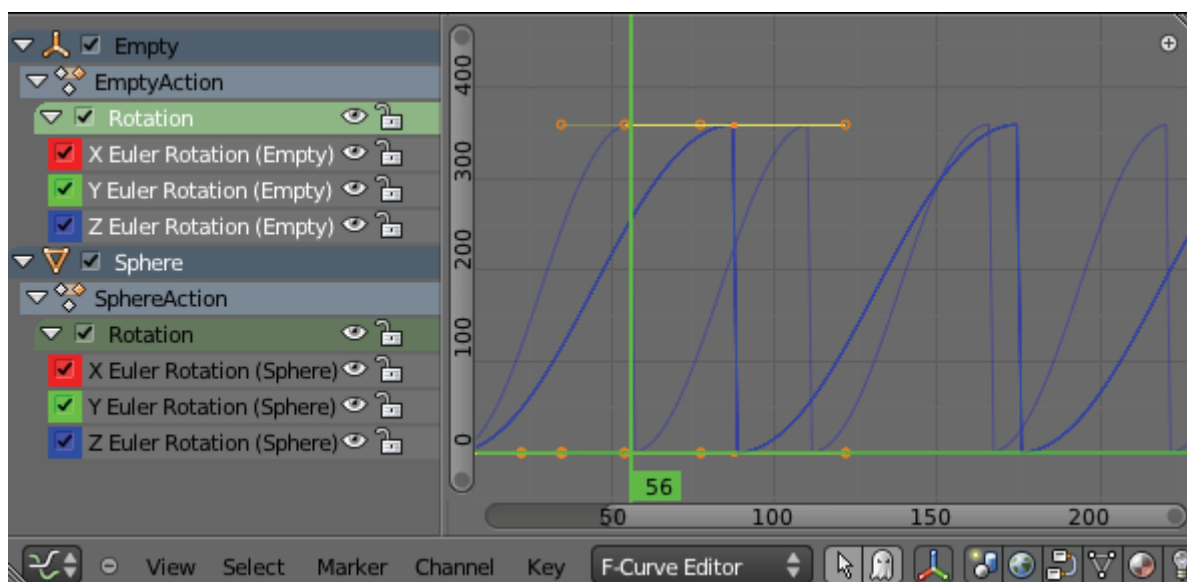
```
>>> fcurvesMercurio = [fmer for fmer in Mercurio.animation_data.action.fcurves if fmer.data_path == 'rotation_euler' and fmer.array_index == 2]
>>>
>>> Empty = bpy.data.objects['Empty']
>>>
>>> fcurvesEmpty = [femp for femp in Empty.animation_data.action.fcurves if femp.data_path == 'rotation_euler' and femp.array_index==2]
>>> |
```

La lista **fcurvesMercurio** viene quindi definita con questo criterio: inserire nella lista le **fcurves** associate a **Mercurio** se tali **fcurves** sono relative alla rotazione (**data_path = ‘rotation_euler’**) e **sull’asse Z** (l’array contiene tre campi, uno per ciascun asse di rotazione, nell’ordine X,Y e Z associati agli indici 0, 1 e 2); discorso analogo per **fcurvesEmpty**.

A questo punto si prelevano, dalle liste create, solo i primi elementi (in realtà, ciascuna lista contiene solo un elemento, ma per evitare di doverlo richiamare con `[0]` in seguito lo isoliamo ponendolo in una variabile temporanea) e si creano due oggetti *FModifier* di tipo “CYCLES”, ossia appunto due modificatori *CYCLES* per *F-Curves*, associandoli al volo alle due curve *IPO*.

```
>>>
>>> fcM = fcurvesMercurio[0]
>>> fcE = fcurvesEmpty[0]
>>>
>>> modM = fcM.modifiers.new(type="CYCLES")
>>> modE = fcE.modifiers.new(type="CYCLES")
>>>
```

Il risultato, oltre che nella **3D View** (avviando l’animazione) può essere verificato anche aprendo una finestra **Graph Editor** ed osservando le curve *IPO* dei canali *Z* della *Empty* e della sfera *Mercurio*: sono cicliche!



Osservando il **Graph Editor** ci rendiamo conto anche di un’altra cosa: l’**interpolazione** ai due keyframes è quella implementata di default da Blender, con le maniglie *Bezier*, ossia un’interpolazione “dolce”, mentre noi preferiremmo un’interpolazione lineare...

Dalle **API Blender Python** leggiamo che una *FCurve* mette a disposizione il campo *keyframes_points*, una collection di *FCurveKeyframePoints* ciascuno dei quali rappresenta uno dei *keyframes* della curva (i “punti” sulla curva nel **Graph Editor**); poniamo tali liste in due variabili d’appoggio, per comodità:

```
>>> Mkeyframes = fcM.keyframe_points
>>> Ekeyframes = fcE.keyframe_points
>>> |
```

Ciascun oggetto appartenente alle due liste è un oggetto di tipo **Keyframe**, il quale a sua volta mette a disposizione il campo **interpolation**, dal significato intuitivo e valore di default **“BEZIER”**; con un ciclo **for**, quindi, scorriamo la **lista di Keyframes** e cambiamo l’interpolazione degli elementi in **“LINEAR”**:

```
>>> for mkf in Mkeyframes:
...     mkf.interpolation="LINEAR"
...
>>> for ekf in Ekeyframes:
...     ekf.interpolation="LINEAR"
...
>>> |
```

Nella pagina seguente è riportato il **codice sorgente completo** dell’esempio.

* * *

Animazione "Mercurio"; Francesco Milanese, 2011

www.redbaron85.com; redbaron85@redbaron85.com

```
bpy.ops.object.add(type="EMPTY", location=(0.0,0.0,0.0))
bpy.ops.mesh.primitive_uv_sphere_add(location=(10.0, 0.0, 0.0))
Mercurio = bpy.data.objects['Sphere']
bpy.ops.mesh.primitive_uv_sphere_add(location=(0.0,0.0,0.0), size=(2.0))
for obj in bpy.data.objects:
    obj.select = False
```

```
bpy.data.objects['Empty'].select = True
bpy.ops.anim.change_frame(frame=1)
bpy.ops.anim.keyframe_insert_menu(type='Rotation')
bpy.ops.anim.change_frame(frame=88)
bpy.data.objects['Empty'].rotation_euler.z = 2*pi
bpy.ops.anim.keyframe_insert_menu(type='Rotation')
for obj in bpy.data.objects:
    obj.select = False
```

```
Mercurio.select = True
bpy.ops.anim.change_frame(frame=1)
bpy.ops.anim.keyframe_insert_menu(type='Rotation')
bpy.ops.anim.change_frame(frame=56)
Mercurio.rotation_euler.z = 2*pi
bpy.ops.anim.keyframe_insert_menu(type='Rotation')
Mercurio.parent = bpy.data.objects['Empty']
```

```
fcurvesMercurio = [fmer for fmer in Mercurio.animation_data.action.fcurves if fmer.data_path ==
'rotation_euler' and fmer.array_index == 2]
```

```
Empty = bpy.data.objects['Empty']
fcurvesEmpty = [femp for femp in Empty.animation_data.action.fcurves if femp.data_path ==
'rotation_euler' and femp.array_index == 2]
```

```
fcM = fcurvesMercurio[0]
fcE = fcurvesEmpty[0]
```

```
modM = fcM.modifiers.new(type='CYCLES')
modE = fcE.modifiers.new(type='CYCLES')
```

```
Mkeyframes = fcM.keyframe_points
Ekeyframes = fcE.keyframe_points
```

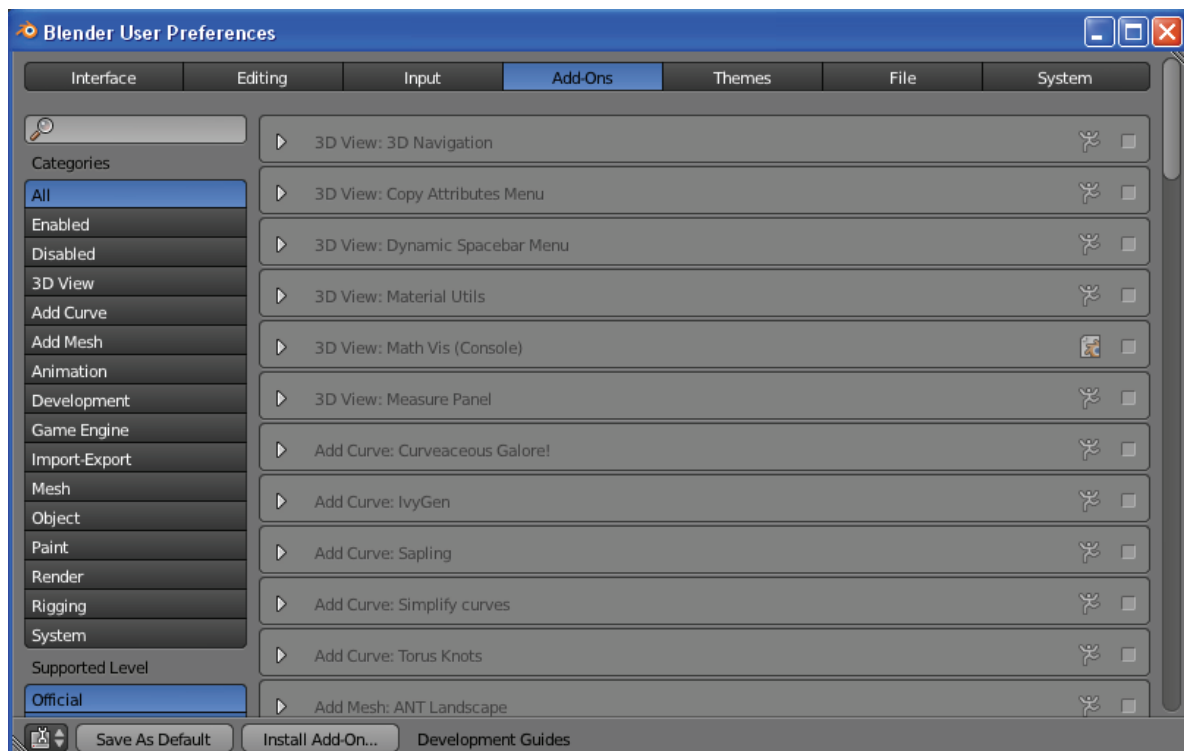
```
for mkf in Mkeyframes:
    mkf.interpolation="LINEAR"
```

```
for ekf in Ekeyframes:
    ekf.interpolation="LINEAR"
```

Capitolo 6

Creazione di un Add-On per Blender, informazioni preliminari

Finora abbiamo trattato gli **script Python per Blender** come singole o poche istruzioni da digitare nella **Console** del programma o da caricare da file mediante finestra **Text Editor** e da eseguire “al volo”; tutti gli esempi presentati, comunque, non costituivano dei **plugin**, in quanto non integrati all’interno del programma mediante il sistema degli **Add-Ons**, installabili dall’omonima scheda all’interno della finestra **User Preferences** e riutilizzabili in seguito senza dover ricopiare lo script o ricaricare i file **.py** nella finestra **Text Editor** ed eseguirli.



La descrizione degli elementi “standard” di un **plugin** e la definizione di un Case Study renderebbero un solo capitolo eccessivamente corposo e, quindi, confusionario; per questo motivo, si è deciso di dedicare questa sezione alla descrizione dei **moduli** necessari per implementare un plugin e di rimandare gli esempi pratici ad un paio di **Case Study** in appendice: “*Mixer Luci*”, un modulo inserito sotto forma di pannello (*Panel*) nella Tool Shelf, e “*Add Point*”, operatore inserito come voce nel menù “*Add → Mesh*” della finestra *Info*, in modo da esaminare anche i due tipi di interfaccia grafica.

Un generico **Add-On** di **Blender** deve includere, nel file o nei file **Python** che lo definiscono:

- la sezione “*bl_info*”;
- una o più **classi** relative all’operazione da svolgere e alla **GUI** (interfaccia grafica, cioè voci di menù, box nella Tool Shelf, ecc...) da associare all’**Add-On**;
- i metodi *register* e *unregister*.

Le **classi** da implementare dovranno specificare, nella loro **definizione**, la loro “natura” e il loro **scopo**, ossia se si tratta di una voce di menù, di un modifier, di un operatore generico, ecc... In questo modo, **erediteranno dai tipi base di Blender** alcune funzioni relative al “contesto” nel quale vengono eseguite, per cui il programmatore potrà richiamare facilmente metodi e campi nativi di **Blender** per effettuare alcune operazioni; ad esempio, se si desidera creare una classe chiamata “*MIO_ADDON_MENU*” per definire un **menù** e le sue voci, bisognerà definire la classe come:

```
class MIO_ADDON_MENU(bpy.types.Menu):
```

passando cioè il **tipo “Menu”**, nativo di **Blender**, insieme ai suoi campi, metodi e al suo “contesto”, al nuovo oggetto creato.

Nel caso di classe per la **GUI dell’Add-On**, si renderà praticamente obbligatoria la presenza della funzione “*draw*”, contenente le informazioni sugli elementi da visualizzare a video (“*draw*” significa appunto “disegna, traccia”) e la loro interazione col codice che effettuerà le operazioni.

La funzione *draw* prende in input l’oggetto che l’ha richiamata (*self*) e il contesto di esecuzione (*context*), per permettere al programmatore di recuperare velocemente il **layout grafico** di base dell’oggetto ed inserire e personalizzare **elementi** (etichette in un menù, sottomenù, pulsanti numerici, checkbox, ...).

Il primo elemento da inserire nel file **Python** di un **Add-On** deve essere, comunque, l'oggetto **"bl_info"**: si tratta di un elemento di tipo **dizionario** (una struttura dati primitiva delle **API Python**, come descritto nel capitolo dedicato alle basi di tale linguaggio) contenente **associazioni chiave-valore** dove le chiavi sono caratteristiche del **plugin** (l'autore, la versione, il contesto di esecuzione, ecc...) e i valori sono le relative informazioni.

Le voci di **bl_info** sono:

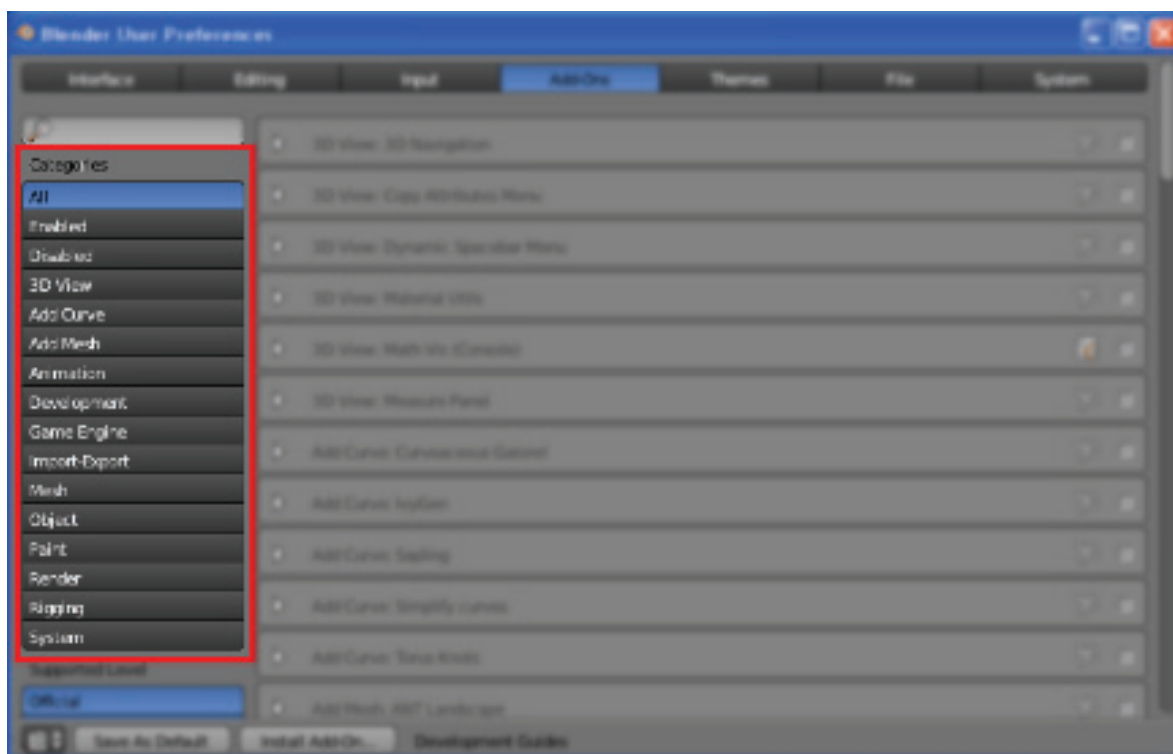
- **"name"** = una stringa contenente il nome del plugin, da visualizzare nella finestra Add-On;
- **"author"** = una stringa contenente il nome dell'autore del plugin;
- **"version"** = la versione, da specificare tra parentesi tonde, separando i valori mediante virgole;
- **"blender"** = la versione di Blender alla quale è destinato lo script, da specificare secondo il formato visto per "version";
- **"api"** = un numero che identifica la versione delle API utilizzate;
- **"location"** = una stringa che indica all'utente dove potrà recuperare l'interfaccia del plugin;
- **"description"** = una stringa contenente una descrizione dell'Add-On e del suo funzionamento;
- **"warning"** = eventuali messaggi di avvertimento (es.: "versione beta, non stabile"), che verranno visualizzati nella finestra User Preferences accanto all'icona "pericolo" (triangolo giallo con punto esclamativo);
- **"wiki_url"** = stringa contenente il percorso URL della documentazione associata al plugin (se presente);
- **"tracker_url"** = stringa contenente il percorso URL da visualizzare per segnalare un bug;
- **"category"** = una stringa relativa al campo di applicazione del plugin (maggiori informazioni su questo campo in seguito).

bl_info e le sue voci vengono inseriti nel file **Python** nella modalità indicata in figura nella pagina seguente, in modo da facilitare la lettura delle informazioni da parte di utenti e sviluppatori.

```
import bpy
```

```
bl_info = {  
    "name" : "Esempio AddOn",  
    "author" : "RedBaron85",  
    "version" : (1,0),  
    "blender" : (2,5,9),  
    "api" : 35853,  
    "location" : "View 3D > Add > Mesh > Esempio AddOn",  
    "description" : "Esempio di blocco bl_info per un AddOn di Blender",  
    "warning" : "",  
    "wiki_url" : "",  
    "tracker_url" : "http://www.redbaron85.com",  
    "category" : "Add Mesh"  
}
```

La voce **“category”** fa riferimento al **tipo di Add-On** da creare ed è l’unica che richiede una certa precisione: i valori possibili per la stringa, infatti, sono quelli presenti nell’elenco a sinistra nella finestra **Add-On**¹ (sezione **“Category”**, appunto) e il valore dato consente di **inserire l’Add-On in uno dei sottomenù**, oltre che nella sezione **“All”**; un valore errato (ossia: non presente nell’elenco) inserirà l’Add-On solo nella sezione **“All”** della scheda.



¹ In dettaglio: “3D View”, “Add Curve”, “Add Mesh”, “Animation”, “Development”, “Game Engine”, “Import-Export”, “Mesh”, “Object”, “Paint”, “Render”, “Rigging”, “System”; significati intuitivi.

I metodi *register* e *unregister* servono invece ad inserire o, nel secondo caso, a rimuovere una classe (detta in Blender “operator”, operatore) dall’elenco di funzioni presenti all’interno del programma; in pratica, quando si richiama una funzione dello script, **Blender** la cerca tra le funzioni presenti in un elenco dove le classi di uno script (con i relativi campi e metodi) vengono aggiunti mediante *register_module* di *bpy.utils* (e dal quale vengono rimossi con *unregister_module* di *bpy.utils*), per cui non basta definire classi e metodi dello script e installarlo con “Install Add-Ons” di User Preferences per renderlo utilizzabile: è necessario “registrarne gli elementi” nel programma con *register_module*.

Questi metodi, eventuali metodi accessori e il controllo:

```
if __name__ == "__main__":  
    register()
```

(che ha praticamente la stessa forma per qualsiasi Add-On) vanno definiti alla fine del file .py; ad esempio, per registrare una classe chiamata “MIO_MENU” che definisce una voce di menù (prendendo quindi in input *bpy.types.Menu* nella sua dichiarazione), bisognerà scrivere:

```
def menu_func(self, content):  
    self.layout.menu("MIO_MENU", icon="PLUGIN") # La voce da inserire in Add --> Menu  
  
def register():  
    bpy.utils.register_module(__name__)  
    # Aggiunge il menu' MIO_MENU al menu' INFO_MT_mesh_add di Blender (il menu' Add --> Mesh)  
    bpy.types.INFO_MT_mesh_add.append(menu_func) # vd. menu_func(self, content), appena sopra  
  
def unregister():  
    bpy.utils.unregister_module(__name__)  
    # Toglie il nuovo menu' MIO_MENU da Add --> Mesh (quando si disattiva lo script)  
    bpy.types.INFO_MT_mesh_add.remove(menu_func)  
  
if __name__ == "__main__":  
    register()
```

Dopo aver definito l’Add-On mediante uno o più file **Python**, per installarlo sarà sufficiente cliccare su “Install Add-On”, in basso a sinistra nella scheda **Add-Ons** nel pannello **User Preferences**, scegliere dalla finestra stile esplora risorse il file .py principale dello script (che importerà gli altri file .py eventualmente presenti, necessari per far funzionare lo script) e confermare l’operazione cliccando su “Install Add-On...”; a questo punto, l’Add-On sarà presente all’interno della scheda **Add-Ons**, sia in “All” che nella sua *category*, se indicata in maniera appropriata nel blocco *bl_info*.

* * *

Capitolo 7

Operatori e modificatori in Object ed Edit Mode

In questo capitolo intendiamo dare una panoramica delle principali funzioni riguardanti la modifica delle **Mesh** in **Object** ed **Edit Mode** mediante le **API Blender Python**.

Per ovvi motivi, non è possibile trattare dettagliatamente *tutte* le **API** dei vari tipi di oggetti presenti in **Blender**, operazione comunque poco utile dal punto di vista didattico; è importante, invece, capire *come* relazionarsi con i vari elementi, qual è la logica che sta dietro alla definizione dei moduli e il loro utilizzo: compreso questo, basterà utilizzare la **Documentazione** per conoscere le **API** che ci interessano volta per volta e realizzare gli script.

Nell'esaminare i campi e le caratteristiche di oggetti della scena di **Blender** prenderemo in esame, in questo capitolo, le **mesh**, in quanto la maggior parte delle operazioni realizzate mediante script e **Add-Ons** riguarda proprio questi oggetti e la definizione di nuovi **Modifiers**; inoltre, le mesh possiedono vari campi, di vario tipo, per rappresentare vertici, spigoli, facce ed altri elementi, quindi si tratta di un buon punto di partenza per trattare vari moduli delle **API Blender Python**.

Per ogni oggetto di *tipo mesh*, le informazioni più importanti sulla struttura si trovano nelle **collections Vertices, Edges, Faces**; abbiamo evidenziato la parola *tipo* per indicare che, quando si cercano informazioni sulla natura di questi elementi, non si deve procedere da **bpy.data.meshes**, che è la collezione delle **mesh** presenti nella scena, ma da **bpy.types.Mesh**, che definisce invece il prototipo dell'oggetto **Mesh**.

In **bpy.types** esistono diversi prototipi di oggetti e tra questi abbiamo, ad esempio, *MeshVertex*, *MeshVertices*, *MeshEdge*, *MeshEdges*, *MeshFace*, *MeshFaces*, ...

Dal punto di vista pratico, invece, dovremo accedere al blocco **data** di un oggetto della scena per poter esaminare la sua struttura e modificare le sue sottoparti. Si accede al blocco **data** di un particolare elemento proprio con **[oggetto].data**.

Scelto un oggetto della scena, ad esempio una mesh **Cube**, in **bpy.data.objects['Cube'].data** avremo campi come *vertices*, *edges*, *faces*; si tratta di **collections**, che come sappiamo sono oggetti iterabili, per cui è possibile scorrere tutti gli elementi, uno per uno, ed effettuare delle operazioni...
...ma di che tipo sono gli elementi contenuti in queste **collections**? Nel caso di *vertices*, saranno dei **bpy.types.MeshVertex**, nel caso di *edges* saranno dei **bpy.types.MeshEdge**, eccetera.

A questo punto, si consulta la **Documentazione** per esaminare le proprietà dei vari oggetti (non affidarsi a **dir** e **type**, in questa fase).

Ricapitolando:

- per recuperare **istanze** di vertici, spigoli e facce di una mesh (e, in generale, parti di un oggetto della scena **Blender**), si procede da **[oggetto].data**;
- il **tipo** di oggetto recuperato e le informazioni su campi e metodi messi a disposizione dal suo prototipo si trovano invece in **bpy.types.[tipoOggetto]**.

Vediamo un esempio pratico.

Supponiamo di voler spostare tutti i vertici dell'oggetto attivo nella **3D View** alla coordinata Z 0.0 se questi hanno coordinate Z maggiori di tale valore.

Quello che bisogna fare è:

- recuperare l'oggetto attivo;
- accedere al suo blocco **data.vertices**;
- esaminare uno per uno i vertici (mediante ciclo **for** sulla *collection*) e, per ogni vertice, cambiare la coordinata Z in 0.0 se questa era maggiore di tale valore.

La **Documentazione** di **MeshVertex** ci dice che le informazioni sulla posizione di un vertice si trovano nel campo **.co** (coordinates), un vettore di tre elementi **float** (XYZ).

Lo script riportato di seguito aggiunge una mesh alla scena (una sfera di dimensione 2 centrata nell'origine) ed esegue le operazioni descritte.

```
>>>
>>> bpy.ops.mesh.primitive_uv_sphere_add(location=(0.0,0.0,0.0), size=(2.0))
{'FINISHED'}

>>> sfera = bpy.context.active_object
>>>
>>> datiSfera = sfera.data
>>>
>>> verticiSfera = datiSfera.vertices
>>>
>>> for v in verticiSfera:
...     if v.co[2] > 0.0:
...         v.co[2] = 0.0
...
>>> |
```

* * *

ATTENZIONE: le operazioni appena descritte vanno effettuate quando l'oggetto attivo è in **OBJECT MODE**, altrimenti gli script non avranno effetto (ma non genereranno un errore).

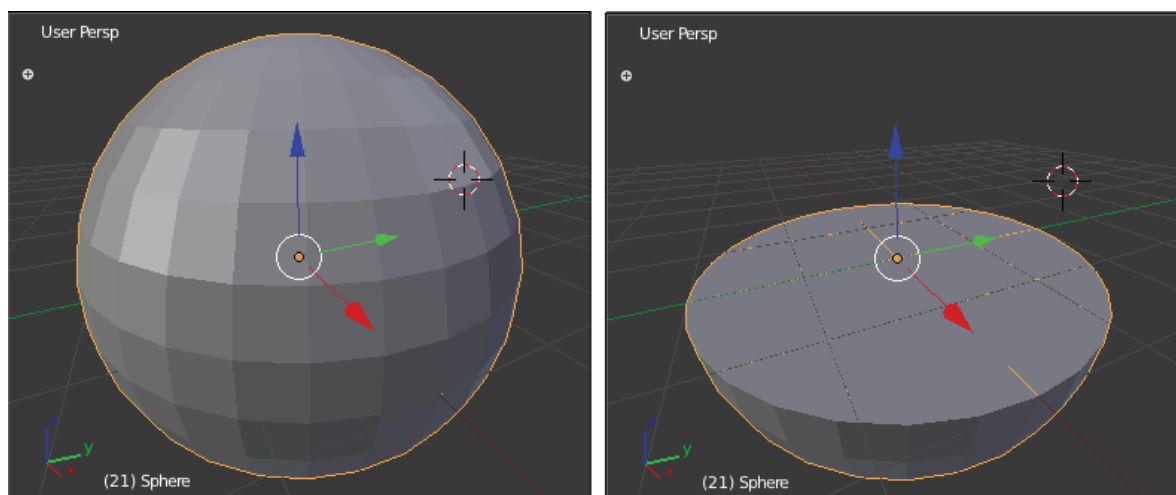
Il **contesto di esecuzione** (OBJECT, EDIT, ...) può essere impostato esplicitamente via script; in particolare, si tratta di chiamare la funzione `mode_set(mode= '[MODE]')` di `bpy.ops.object`, ad es.: `bpy.ops.object.mode_set(mode='EDIT')`.

Valori possibili per mode: *OBJECT*, *EDIT*, *SCULP*, *VERTEX_PAINT*, *WEIGHT_PAINT*, *TEXTURE_PAINT*, *PARTICLE_EDIT*, *POSE*.

Altri modificatori, come ad esempio **Extrude**, vanno invece applicati quando l'oggetto è in **Edit Mode**, altrimenti genereranno un errore *"[nomeMetodo].poll() failed, context is incorrect"*. Le **API** dei vari modificatori indicano, in genere, qual è il giusto contesto (*MODE*) di esecuzione.

* * *

L'immagine seguente mostra lo stato della 3D View prima e dopo l'esecuzione del ciclo for nello script precedentemente descritto.



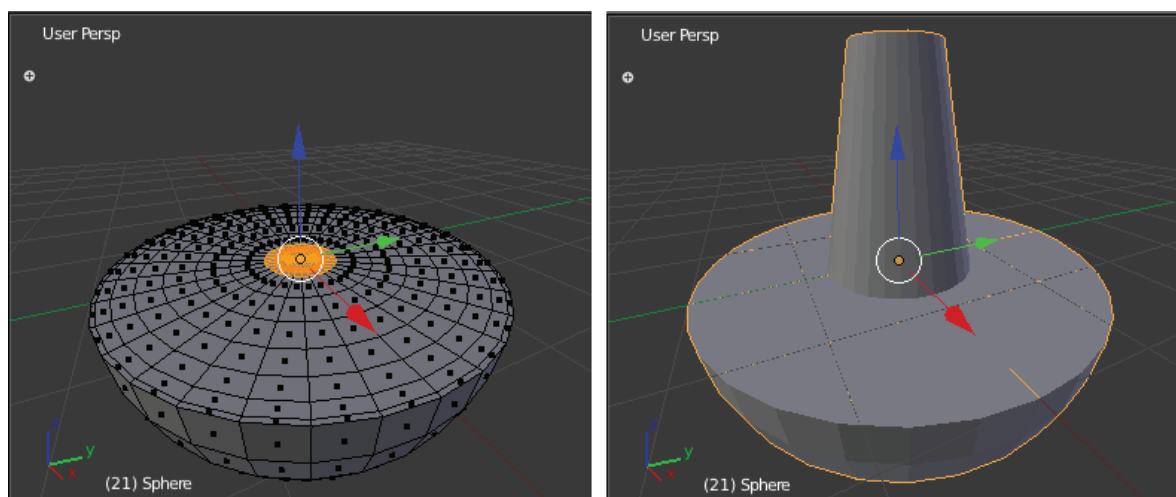
Quanto detto non implica l'impossibilità di applicare delle operazioni a vertici, spigoli o facce selezionate interattivamente in **Edit Mode** nella **3D View**; quello che bisogna fare, in questo caso, è:

- effettuare la selezione degli elementi da trasformare nella **3D View** in **Edit Mode**;
- passare in **Object Mode** (Blender memorizzerà comunque la selezione effettuata in Edit Mode);
- applicare lo script agli elementi (vertici, spigoli, facce, ...) con campo "*select*" a **True**.

Riprendendo la sfera editata con lo script visto precedentemente, selezioniamo qualche vertice in **Edit Mode** in una **3D View**, poi passiamo in **Object Mode** ed eseguiamo lo script definito qui di seguito nella finestra **Python Console**, osservando poi il risultato a video nella **3D View**.

```
>>>
>>> for v in verticiSfera:
...     if v.select == True:
...         v.co[2] = 2.0
...
>>> |
```

L'immagine seguente mostra la selezione effettuata in **Edit Mode** e il risultato dell'esecuzione dello script (eseguito *dopo* il passaggio in **Object Mode**).



Diverso è il caso delle **trasformazioni** da applicare ad una sottoparte di un oggetto; con **.co**, infatti, abbiamo modificato dei *valori* propri dell'elemento, ma a volte è necessario effettuare *operazioni* quali traslazioni, suddivisioni, beveling, ecc...

Queste *operazioni* sono definite in **bpy.ops**, ossia appunto le **operations** messe a disposizione da Blender (e che possiamo estendere, definendo *ops* personalizzate, come vedremo nel Case Study in **Appendice A**).

L'utilizzo di questi operatori è semplicissimo: è sufficiente selezionare gli oggetti, sia nella 3D View (con i classici strumenti di selezione interattiva) che via **Python** (impostando *select* a *True*, sia nel caso di un intero oggetto che di una sua sottoparte), e richiamare la **funzione** *bpy.ops* che effettua l'operazione desiderata.

ESEMPIO 1

Lo script riportato qui di seguito inserisce un oggetto **Monkey** nella scena, gli associa un oggetto **Modifier** di tipo **Subdivision Surface**, personalizza tale modificatore e lo applica.

```
>>>
>>> bpy.ops.mesh.primitive_monkey_add(location=(0.0, 0.0, 0.0))
{'FINISHED'}

>>> bpy.ops.object.modifier_add(type = 'SUBSURF')
{'FINISHED'}

>>> modificatore = bpy.context.active_object.modifiers[0]
>>> type(modificatore) # Di che tipo e' questo modificatore?
<class 'bpy.types.SubsurfModifier'>

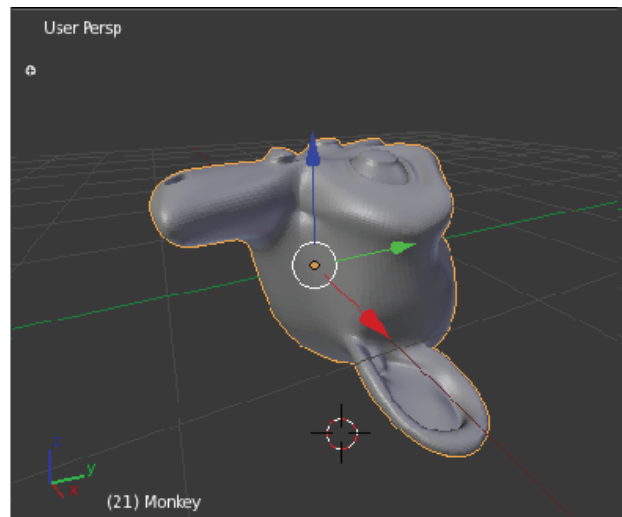
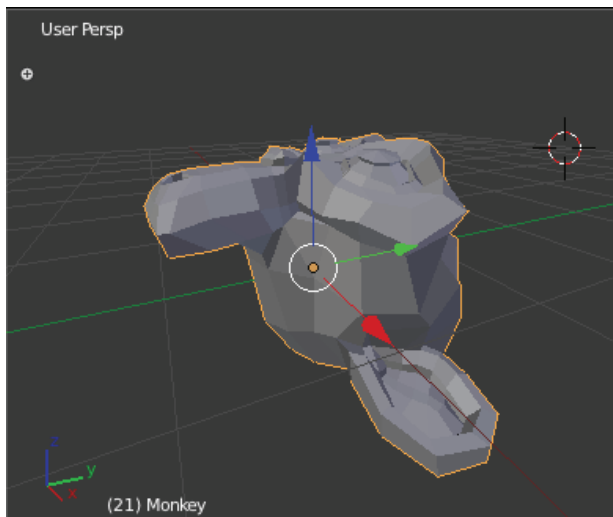
>>> dir(modificatore) # Quali sono le sue caratteristiche?
['__doc__', '__module__', '__slots__', 'bl_rna', 'levels', 'name',
'render_levels', 'rna_type', 'show_expanded', 'show_in_editmode',
'show_on_cage', 'show_only_control_edges', 'show_render', 'show_view
port', 'subdivision_type', 'type', 'use_apply_on_spline', 'use_subs
urf_uv']

>>> modificatore.levels = 3
>>> modificatore.render_levels = 3
>>> modificatore.name = "SubSurf3"
>>>
>>> bpy.ops.object.modifier_apply(modifier="SubSurf3")
{'FINISHED'}

>>> |
```

In questo caso il modificatore viene applicato a tutto l'oggetto, in **Object Mode**.

Quando non si conosce il metodo che applica un modificatore “nativo” di **Blender**, si può sempre applicarlo in maniera interattiva nella **3D View** ed esaminare la stringa che apparirà nell'area di testo della finestra **Info**, in modo da recuperare il nome del **Modifier** e cercarlo nella **Documentazione**.



L'esempio appena descritto ci informa di un altro aspetto delle operazioni in **Blender**: i **Modifiers** vengono associati agli oggetti presenti nella scena inserendoli in una particolare *collection* dell'oggetto, detta “**modifiers**”, accessibile con:

`[oggetto].modifiers`

dove ciascun elemento è, appunto, di tipo **bpy.types.Modifier**.

Sono, ad esempio, modificatori nativi di **Blender**: *ArmatureModifier*, *CurveModifier*, *DecimateModifier*, *DisplaceModifier*, *SubsurfModifier*, ecc...

Da notare infine che per applicare il modificatore non si utilizza un metodo di **Modifier**, ma **modifier_apply()** di **bpy.ops.object**, specificando ovviamente il nome (campo ‘**name**’ del **Modifier**) del modificatore da applicare e rimuovere automaticamente dalla pila di **Modifiers** dell'oggetto.

ESEMPIO 2

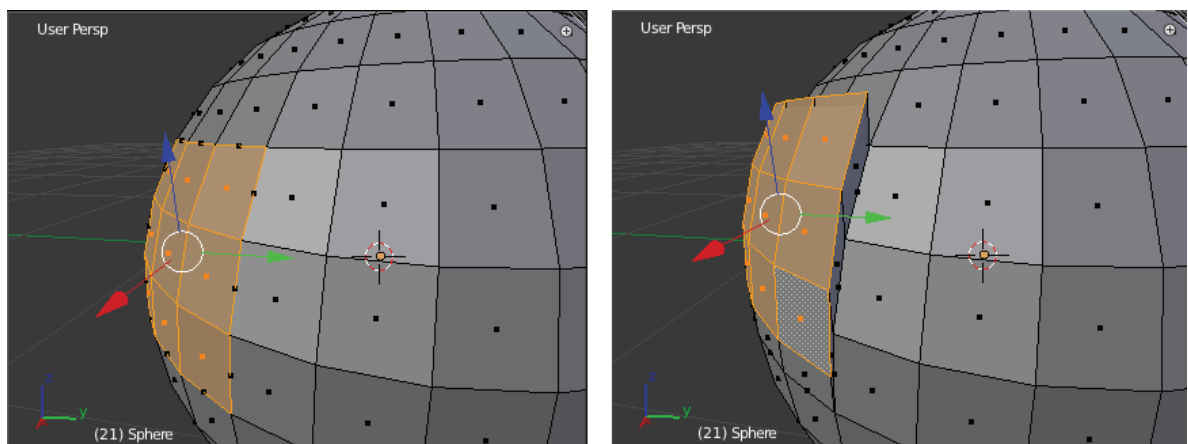
Lo script riportato qui di seguito estrude le facce selezionate in maniera interattiva nella **3D View** (l'oggetto è, quindi, quello attivo) e le **sposta** con valore 0.1 su Z con **orientation** “**NORMAL**”.

Lo script deve essere eseguito quando l'oggetto è in **Edit Mode**, perché questo è il corretto *contesto* di esecuzione dell'operatore **Extrude**; utilizzando una diversa modalità, si avrà un errore “**poll() failed, context is incorrect**”.

L'operazione viene effettuata in due fasi perché l'estrusione, di per sé, non sposta le facce: si limita a creare quelle nuove e a selezionarle, mentre per spostarle, ruotarle, scalarle, ecc.. è necessario ricorrere ai metodi di *bpy.ops.transform*, come ad esempio *translate*.

```
>>> bpy.ops.mesh.extrude()  
{ 'FINISHED' }  
  
>>> bpy.ops.transform.translate(value=(0.0, 0.0, 0.1), constraint_orientation="NORMAL")  
{ 'FINISHED' }  
  
>>> |
```

L'immagine seguente si riferisce all'oggetto nella **3D View** prima e dopo l'esecuzione dello script (da notare che si è sempre in **Edit Mode**).



* * *

Negli script mostrati finora si è fatto uso più volte di funzioni di tipo “*primitive_[tipo]_add*”; tali funzioni consentono appunto di aggiungere elementi nella scena 3D ed appartengono tutte al modulo *bpy.ops* (e le informazioni sulle loro **API** si trovano nella relativa sezione della **Documentazione**).

* * *

Capitolo 8

Moduli di Input / Output su file: un Case Study

Le **API Python** forniscono moduli e funzioni native per effettuare operazioni di **Input / Output** sui file; in **Blender**, queste funzioni possono essere utilizzate, ad esempio, per importare file immagini, modelli 3D, informazioni sull'animazione di Armature, ecc...

In questo capitolo, vedremo come scrivere dei dati su file approfittandone per esaminare alcune **API** relative ai **sistemi particellari** in **Blender 3D**.

Supponiamo di voler ricavare la posizione di una decina di **particelle** immesse nella scena da una mesh e soggette all'influenza di due **Force Field** in grado di modificarne la traiettoria; dobbiamo inoltre scrivere i dati ottenuti frame dopo frame su un file di testo (in chiaro) perché, ad esempio, il nostro professore userà quel file con un altro programma per interpretare i dati ricavati dalla simulazione.

Per prima cosa allestiamo la **scena** (la configurazione qui descritta è, ovviamente, solo un esempio, l'importante è concentrarsi sulle **API Blender Python**), partendo da una **scena completamente vuota** (cancelliamo quindi telecamera, fonte di luce e mesh presenti di default nelle scene appena create) ed inserendo tutti gli elementi **via codice**, approfittandone per esaminare nuovi metodi e campi.

Posizioniamo per prima cosa due **Empty**, che forniremo di **Force Fields**, in (-10, 0, 0) e (10, 0, 0), mediante le istruzioni viste nel **capitolo 5**, rinominandole nel mentre in “**Empty1**” ed “**Empty2**”:

```
>>> bpy.ops.object.add(type="EMPTY", location=(-10.0,0.0,0.0))
{'FINISHED'}

>>> Empty1 = bpy.context.selected_objects[-1]
>>> Empty1.name = "Empty1"
>>> Empty1.select = False
>>>

>>> bpy.ops.object.add(type="EMPTY", location=(10.0,0.0,0.0))
{'FINISHED'}

>>> Empty2 = bpy.context.selected_objects[-1]
>>> Empty2.name = "Empty2"
>>> Empty2.select = False
```

Le informazioni che ci consentono di utilizzare un oggetto, nel nostro caso la **Empty**, come **Force Field**, e di impostarne contestualmente i parametri, si trovano in `[oggetto].field`, elemento di tipo “**FieldSettings**” (lett.: “impostazioni campo”, appunto).

```
>>> dir(Empty1)
['_doc_', '_module_', '_slots_', 'active_material', 'active_material_index', 'active_shape_key', 'active_shape_key_index', 'animation_data', 'animation_data_clear', 'animation_data_create', 'animation_visualisation', 'bl_rna', 'bound_box', 'children', 'closest_point_on_mesh', 'collision', 'color', 'constraints', 'copy', 'data', 'delta_location', 'delta_rotation_euler', 'delta_rotation_quaternion', 'delta_scale', 'dimensions', 'draw_bounds_type', 'draw_type', 'dupli_faces_scale', 'dupli_frames_end', 'dupli_frames_off', 'dupli_frames_on', 'dupli_frames_start', 'dupli_group', 'dupli_list', 'dupli_list_clear', 'dupli_list_create', 'dupli_type', 'empty_draw_size', 'empty_draw_type', 'empty_image_offset', 'field', 'find_armature', 'game', 'grease_pencil', 'hide', 'hide_render', 'hide_select', 'is_duplicator', 'is_modified', 'is_visible', 'layers', 'library', 'location', 'lock_location', 'lock_rotation', 'lock_rotation_w', 'lock_rotations_4d', 'lock_scale', 'material_slots', 'matrix_basis', 'matrix_local', 'matrix_world', 'mode', 'modifiers', 'motion_path', 'name', 'parent', 'parent_bone', 'parent_type', 'parent_vertices', 'particle_systems', 'pass_index', 'pose', 'pose_library', 'proxy', 'proxy_group', 'ray_cast', 'rna_type', 'rotation_axis_angle', 'rotation_euler', 'rotation_mode', 'rotation_quaternion', 'scale', 'select', 'shape_key_add', 'show_axis', 'show_bounds', 'show_name', 'show_only_shape_key', 'show_texture_space', 'show_transparent', 'show_wire', 'show_x_ray', 'soft_body', 'tag', 'time_offset', 'to_mesh', 'track_axis', 'type', 'up_axis', 'update_tag', 'use_dupli_faces_scale', 'use_dupli_faces_speed', 'use_dupli_vertices_rotation', 'use_fake_user', 'use_shape_key_edit_mode', 'use_slow_parent', 'use_time_offset_add_parent', 'use_time_offset_edit', 'use_time_offset_parent', 'use_time_offset_particle', 'user_clear', 'users', 'users_group', 'users_scene', 'vertex_groups']

>>> type(Empty1.field)
<class 'bpy.types.FieldSettings'>
```

Le **API Blender Python** di **FieldSettings** indicano che, tra le altre cose, possiamo:

- definire il **tipo** di forza, con *type* (il valore di default è **NONE**, infatti un oggetto appena creato non è un campo di forza; possibili valori per questo campo: [“NONE”, “FORCE”, “WIND”, “VORTEX”, “MAGNET”, “HARMONIC”, “CHARGE”, “LENNARDJ”, “TEXTURE”, “GUIDE”, “BOID”, “TURBULENCE”, “DRAG”]);
- impostare l’**intensità** del campo di forza, con *strength*;
- impostare l’intensità della **turbolenza**, specificando un valore *float* tra 0 e 10 in *size*;
- impostare la **forma** del campo di forza (default: “**POINT**”, omnidirezionale) con *shape* (valori possibili: “POINT”, “PLANE”, “SURFACE”, “POINTS”).

Per il primo campo di forza, impostiamo il tipo a **“FORCE”**, con intensità -10.0 e gli altri valori come di default, mentre per il secondo impostiamo **“TURBULENCE”** con intensità 20.0 e gli altri valori come di default:

```
>>> ForceField1 = Empty1.field
>>> ForceField1.type = "FORCE"
>>> ForceField1.strength = -10.0
>>>
>>> ForceField2 = Empty2.field
>>> ForceField2.type = "TURBULENCE"
>>> ForceField2.strength = 20.0
>>> |
```

I **campi di forza** sono stati impostati, adesso passiamo alla **mesh emettitrice**: va bene anche una **UV Sphere**, l'importante in questo caso sono i **metodi** riguardanti il **sistema particellare**.

Poniamo la sfera al centro dell'universo virtuale, con tutte le impostazioni di default (size, rings, location, segments, ...), rinominandola in **“Emettitore”** ed associandole una **variabile** omonima:

```
>>> bpy.ops.mesh.primitive_uv_sphere_add()
{'FINISHED'}

>>> Emettitore = bpy.context.selected_objects[-1]
>>> Emettitore.name = "Emettitore"
>>> |
```

L'accesso alle informazioni dei **sistemi particellari** associabili ad una mesh è dato da **“particle_systems”**, una collezione di **“ParticleSystem”**, infatti ad una mesh è possibile associare più sistemi particellari, inseriti in questa collezione nelle **API Blender Python**.

Di default, una mesh appena creata **non ha alcun sistema particellare** associato, per cui la prima cosa da fare è creare un **“particle_system”** e aggiungerlo alla lista dei sistemi della mesh, nel nostro caso la sfera **“Emettitore”**; la creazione di un sistema particellare via codice avviene selezionando l'oggetto al quale bisogna assegnare il sistema particellare ed invocando il metodo **particle_system_add()** di **bpy.ops.object**, che crea un sistema particellare con valori di default per l'oggetto selezionato:

```
>>> # In questo momento, l'unico oggetto selezionato e' proprio Emettitore
>>> bpy.ops.object.particle_system_add()
{'FINISHED'}
```

(nota: l'utilizzo, anche nei commenti, di caratteri accentati quali **“è”**, **“è”**, **“ì”**, ecc... genererà errore; utilizzare l'apostrofo **“ ’ ”**).

Assegniamo il sistema particellare alla variabile “**SistemaParticellare**” e personalizziamolo sfruttando i metodi di “**Particle_System.settings**” (un oggetto di tipo **ParticleSettings**), elencati nelle **API Blender Python**²:

```
>>> SistemaParticellare = Emettitore.particle_systems[0]
>>> SistemaParticellare.name = "Emissione"
>>> SistemaParticellare.settings.frame_start = 1
>>> SistemaParticellare.settings.frame_end = 10
>>> SistemaParticellare.settings.lifetime = 250
>>> SistemaParticellare.settings.count = 10
>>> SistemaParticellare.settings.normal_factor = 0.0
>>>
```

In questo modo, l’**emissione** di particelle inizierà al frame 1, terminerà al frame 10 e genererà 10 particelle con **Lifetime** 250 (la durata dell’animazione di default di **Blender**), con velocità iniziale (**Normal**) 0.0.

La **forza di gravità** “disturba” la nostra animazione, ma anziché toglierla solo per le particelle del sistema particellare in uso togliamola per tutta la **scena**, approfittandone per menzionare le impostazioni di **Scene**:

```
>>>
>>> bpy.context.scene.gravity = Vector((0.0, 0.0, 0.0))
>>>
```

Avviando ora l’animazione in una **3D View**, vedremo le particelle partire dalla sfera e muoversi, nel corso dei **250 frames “standard”**, all’interno della scena 3D.

Il nostro professore, nel mentre, ci informa che gli interessano i dati relativi ai **primi 100 frames**, in un file **.txt**, con questo formato:

- ogni **riga** corrisponde ad un **frame** dell’animazione, quindi serviranno 100 righe;
- in ogni **riga**, la posizione di una particella deve essere indicata mediante **tre valori** (coordinate XYZ) con due cifre decimali, separati da “,” e posti tra “()”;
- il carattere di separazione tra le coordinate delle varie particelle è “#”.

² Questo è uno dei casi in cui la presenza della funzione Blender Python nel TOOLTIP torna particolarmente utile: la corrispondenza tra le voci di un ParticleSystem nella GUI e nelle API non è proprio perfetta ma basta porre il mouse su un elemento per ottenere il suggerimento della voce API da utilizzare; ad esempio, ponendo il mouse sul campo “Amount” nella scheda Particle Systems, nella Properties Window, il Tooltip indicherà “ParticleSettings.count”, indicando il campo da modificare per cambiare tale parametro via codice (per informazioni sul tipo di parametro e sui valori ammissibili, invece, è necessario ricorrere alla documentazione vera e propria).

Quello che faremo sarà quindi:

1. creare una **lista di stringhe**, ciascuna delle quali conterrà i **dati** di un **frame**;
2. in ogni **stringa**, inserire i valori recuperati dalle **particelle** ai vari **frame**, formattandoli secondo le modalità espresse precedentemente;
3. scrivere questa **lista** su **file**, un elemento per riga.

Nel sistema particellare, una **singola particella** è identificata mediante *“particle”*; intuitivamente, l’oggetto *“particles”* di un *“particle_system”* è una **collection**, una collezione ordinata di oggetti *“particle”*.

Per recuperare le **informazioni di posizione di una particella** ad un dato frame, quindi, è sufficiente scrivere:

```
>>> SistemaParticellare.particles[0].location
Vector((-0.3589247763156891, 0.9041329026222229, 0.21718668937683105))
>>> |
```

e, per estensione, il **ciclo** di recupero delle informazioni su tutte le particelle per 100 frames e di scrittura di tali dati in una stringa (da inserire nella list “dati”) può essere definito come segue:

```
>>> import math # Necessaria per round(n,2), che arrotonda n a due cifre decimali
>>> dati = []
>>> for i in range(100):
...     frameCorrente = i+1;
...     riga = "(";
...     bpy.ops.anim.change_frame(frame=frameCorrente)
...     for p in range(10):
...         x = round(float(SistemaParticellare.particles[p].location.x), 2)
...         y = round(float(SistemaParticellare.particles[p].location.y), 2)
...         z = round(float(SistemaParticellare.particles[p].location.z), 2)
...         riga = riga + str(x) + "," + str(y) + "," + str(z) + ")#("
...     riga = riga[:-2] # Serve a togliere "#(" a fine riga
...     riga = riga + "\n"
...     dati.append(riga)
...
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}
```

Segue la fase di **scrittura dei dati su file**: è sufficiente aprire un **descrittore di file** (un oggetto, associato ad un **file**) e scrivere, mediante ciclo *for*, le stringhe contenute in **dati** (stringhe che terminano in *\n* per implementare il ritorno a capo automatico), per poi **chiudere il descrittore** del file con *“close()”*.

```
>>> fileDati = open("fileDati.txt", "w")
>>> for i in range(100):
...     fileDati.write(dati[i])
...
180
180
180

>>> fileDati.close()
>>> |
```

Se non si specifica il **path assoluto**, il file verrà depositato nella **cartella principale di Blender** (ad esempio, "C:\Programmi\Blender Foundation\Blender\"); nello specificare un **path assoluto**, ricordarsi del **carattere di escape** \ prima di ogni \ nel path (ad es.: "**C:\\dati.txt**" se si desidera salvare il file "**dati.txt**" in "**C:**").

Nella pagina seguente è riportato il **codice sorgente completo** dell'esempio.

```
# Sistema Particellare su file; Francesco Milanese, 2011
# www.redbaron85.com; redbaron85@redbaron85.com
bpy.ops.object.add(type="EMPTY", location=(-10.0,0.0,0.0))
Empty1 = bpy.context.selected_objects[-1]
Empty1.name = "Empty1"
Empty1.select = False

bpy.ops.object.add(type="EMPTY", location=(10.0,0.0,0.0))
Empty2 = bpy.context.selected_objects[-1]
Empty2.name = "Empty2"
Empty2.select = False

dir(Empty1)
type(Empty1.field)

ForceField1 = Empty1.field
ForceField1.type = "FORCE"
ForceField1.strength = -10.0

ForceField2 = Empty2.field
ForceField2.type = "TURBULENCE"
ForceField2.strength = 20.0

bpy.ops.mesh.primitive_uv_sphere_add()
Emettitore = bpy.context.selected_objects[-1]
Emettitore.name = "Emettitore"

# In questo momento, l'unico oggetto selezionato e' proprio Emettitore
bpy.ops.object.particle_system_add()

SistemaParticellare = Emettitore.particle_systems[0]
SistemaParticellare.name = "Emissione"
SistemaParticellare.settings.frame_start = 1
SistemaParticellare.settings.frame_end = 10
SistemaParticellare.settings.lifetime = 250
SistemaParticellare.settings.count = 10
SistemaParticellare.settings.normal_factor = 0.0

bpy.context.scene.gravity = Vector((0.0, 0.0, 0.0))

SistemaParticellare.particles[0].location

import math # Necessaria per round(n,2), che arrotonda n a due cifre decimali
dati = []
for i in range(100):
    frameCorrente = i+1;
    riga = "(";
    bpy.ops.anim.change_frame(frame=frameCorrente)
    for p in range(10):
        x = round(float(SistemaParticellare.particles[p].location.x), 2)
        y = round(float(SistemaParticellare.particles[p].location.y), 2)
```

```
z = round(float(SistemaParticellare.particles[p].location.z), 2)
riga = riga + str(x) + "," + str(y) + "," + str(z) + ")#("
riga = riga[:-2] # Serve a togliere "(" a fine riga
riga = riga + "\n"
dati.append(riga)

fileDati = open("fileDati.txt", "w")
for i in range(100):
    fileDati.write(dati[i])
fileDati.close()
```

APPENDICE A

Case Study: il plugin “Add Point”

L’**Add-On** che esamineremo in questa sezione svolge un’operazione poco utile dal punto di vista pratico, limitandosi infatti ad aggiungere nella scena una mesh composta da un solo vertice e posizionata nell’origine dell’universo virtuale, ma dal punto di vista didattico ci consentirà di scoprire cosa dover fare per:

- creare ed installare un **Add-On** in Blender 3D 2.5;
- inserire una voce nel menù *Add → Mesh* della finestra **Info**;
- definire un **operatore** di trasformazione delle mesh personalizzato;
- **registrare** e richiamare tale operatore nell’ambiente di esecuzione di **Blender**.

Definiremo l’Add-On in un unico **file .py**, che possiamo creare mediante un qualsiasi editor di testo come, ad esempio, la finestra **Text Editor** di Blender 3D.

Dovremo creare delle classi Python per definire l’interfaccia grafica (le voci di menù) e l’operatore vero e proprio. Iniziamo la trattazione parlando proprio di questo argomento.

Per definire un nuovo operatore o modificatore, dobbiamo creare una classe che erediti da **bpy.types.Operator**; volendo chiamare la nostra classe “**ADD_POINT_OP**”, ad esempio, scriveremo:

```
class ADD_POINT_OP(bpy.types.Operator):  
    # Corpo della classe, campi e metodi, con la corretta indentazione  
bpy.utils.register_class(ADD_POINT_OP)
```

All'interno della classe possiamo definire a piacere campi e metodi ma due campi stringa sono obbligatori: ***bl_idname*** e ***bl_label***; il primo rappresenta il nome da associare all'operatore nelle **API Python** di **Blender** ed è con questo nome (*operator name*) che richiameremo il modificatore dalla voce di menù, mentre il secondo campo è il nome da visualizzare a video.

Su ***bl_idname*** c'è da dire inoltre che il nome di un **Operator** deve contenere almeno un punto; nel nostro esempio, utilizzeremo le seguenti istruzioni:

```
bl_idname = "wm.add_point_op"  
bl_label = "Aggiunge un punto in Origin"
```

Quando, a fine definizione classe (e fuori dal suo blocco), **registreremo** l'operatore ***ADD_POINT_OP*** in **Blender**, questo potrà essere richiamato via script con la chiamata:

```
bpy.ops.wm.add_point_op()
```

composta da ***"bpy.ops"*** (perché è un *Operator*) e dall'*operator name* definito in ***bl_idname***.

L'unico metodo presente in questa classe è quello che definisce cosa fare quando viene invocata la funzione ***bpy.ops.wm.add_point_op()***; si tratta del **metodo *execute(self, context)***.

All'interno di tale metodo inseriremo le istruzioni necessarie per:

- aggiungere un oggetto **Plane**;
- deselezionare tutti i vertici del Plane;
- passare in modalità selezione vertici, con ***mesh_select_mode***;
- mediante ciclo ***for***, selezionare di volta in volta tutti i vertici tranne il primo e cancellarli³;
- aggiornare ***l'Origin*** dell'oggetto, cosa che porrà lo stesso (composto a questo punto da un solo vertice) al centro dell'universo virtuale.

Le uniche annotazioni da fare per questa funzione riguardano ***ToolSettings***, il cambio di **contesto** e l'aggiornamento di ***Origin***.

E' noto che, nella **3D View** in **Edit Mode**, è possibile scegliere la modalità di selezione degli elementi tra vertici, spigoli e facce; tali opzioni corrispondono ai tre campi ***booleani*** di

³ Come esercizio, si può provare a modificare questa porzione di codice selezionando tutti i vertici tranne il primo e cancellare tutta la selezione mediante una sola invocazione di ***mesh.delete*** sull'intera selezione, anziché un vertice per volta.

mesh_select_mode di **bpy.context.tool_settings**, per cui impostando a (**True, False, False**) tali campi sceglieremo solo la prima modalità, ossia **vertex**.

All'interno del ciclo **for** dovremo cambiare il contesto, scegliendo **EDIT** o **OBJECT**, perché la selezione dei vertici in base all'indice avviene in modo **OBJECT**, mentre la cancellazione dei vertici selezionati deve avvenire nel contesto **EDIT**.

La funzione **origin_set** di **bpy.ops.object**, con parametro "**GEOMETRY_ORIGIN**", aggiorna l'origine dell'oggetto selezionato e opera nel contesto **OBJECT**.

Il codice della **classe dell'operatore**, inclusa la chiamata alla funzione di registrazione della stessa, è riportato nell'immagine seguente.

```
class ADD_POINT_OP(bpy.types.Operator):
    bl_idname = "wm.add_point_op"
    bl_label = "Aggiunge un punto in Origin"

    # Chiamare questo operatore con bpy.ops.wm.add_point_op()
    def execute(self, context):
        bpy.ops.mesh.primitive_plane_add(location=(0.0,0.0,0.0))
        bpy.context.active_object.name = "___temp___"
        nuovoOggetto = bpy.data.objects['___temp___']
        vertici = nuovoOggetto.data.vertices
        bpy.context.tool_settings.mesh_select_mode[0] = True
        bpy.context.tool_settings.mesh_select_mode[1] = False
        bpy.context.tool_settings.mesh_select_mode[2] = False
        for v in vertici:
            v.select = False
        numVertici = len(vertici)
        for i in range(numVertici-1, 0, -1):
            vertici[i].select = True
            bpy.ops.object.mode_set(mode='EDIT')
            bpy.ops.mesh.delete(type='VERT')
            bpy.ops.object.mode_set(mode='OBJECT')
        bpy.ops.object.mode_set(mode='OBJECT')
        bpy.ops.object.origin_set(type='GEOMETRY_ORIGIN')
        return {'FINISHED'}

# Registra l'operatore appena definito nell'ambiente di Blender
bpy.utils.register_class(ADD_POINT_OP)
```

NOTA: esistono due funzioni di **registrazione** di elementi in **bpy.utils**: **register_class** e **register_module**; il primo viene utilizzato per le classi che estendono da **Panel**, **Menu**, **Header**, **Operator**, **KeyingSetInfo** e **RenderEngine**, il secondo per tutte le altre.

Per la definizione della GUI (interfaccia grafica necessaria per richiamare l'operatore) è necessario creare una classe che estenda dal tipo **"Menu"**, in modo da generare un sottomenù da inserire in **Add → Mesh** (identificato con `bpy.types.INFO_MT_mesh_add`, nelle **API**); tale classe deve essere provvista del metodo **draw**, che contiene appunto le "istruzioni di disegno", ossia i metodi da inserire nel **layout**, oggetto che identifica la rappresentazione visiva della classe (che in questo caso è di tipo **Menu**).

Il "disegno" del menù avviene nella classe **ADD_POINT_MENU** dove, all'interno del metodo **draw**, si recupera il **layout** nativo dell'oggetto e vi si aggiunge un elemento **operator**.

La firma del metodo operator è composta dal **bl_idname** dell'operatore, dal testo del **Tooltip** e dall'**icona** da associare alla voce. Tramite il **bl_idname**, **Blender** invocherà direttamente il metodo **execute** della classe dell'operatore, quando si farà click sulla voce nel menù.

Il campo **ICON** serve a specificare l'icona da assegnare alla voce e il suo valore va scelto tra una lista di **costanti predefinite**; la lista completa delle costanti è reperibile nella **Documentazione** in **.pdf** a pagina 976 e occupa ben due pagine formato A4.

Il menù così definito viene aggiunto a **"Add → Mesh"** di **Info** mediante la **funzione menu_func**, invocata a sua volta da **register**: **register** passa infatti il suo contesto a **menu_func**, che provvede ad inserire nel **layout** di questo contesto (ossia in **INFO_MT_mesh_add**, cioè il menù **Add Mesh** nella finestra **Info**) il menù definito in **ADD_POINT_MENU**.

register si occupa anche della registrazione del modulo (la voce di menù) nell'ambiente.

```
class ADD_POINT_MENU(bpy.types.Menu):
    bl_idname = "ADD_POINT_MENU"
    bl_label = "Add Point"
    bl_description = "" # Opzionale

    def draw(self, context):
        layout = self.layout
        #layout.operator_context = "VOKE_REGION_WIN"
        layout.operator("wm.add_point_op", text="Aggiunge un vertice in 0,0,0", icon="MESH_DATA")

def menu_func(self, content):
    self.layout.menu("ADD_POINT_MENU", icon="PLUGIN")

def register():
    bpy.utils.register_module(__name__)
    # Aggiunge il menu' ADD_POINT_MENU al menu' INFO_MT_mesh_add di Blender
    bpy.types.INFO_MT_mesh_add.append(menu_func) # vd. menu_func(self, content), appena sopra

def unregister():
    bpy.utils.unregister_module(__name__)
    # Toglie il nuovo menu' ADD_POINT_MENU da ADD_MESH (quando si disattiva lo script)
    bpy.types.INFO_MT_mesh_add.remove(menu_func)

if __name__ == "__main__":
    register()
```

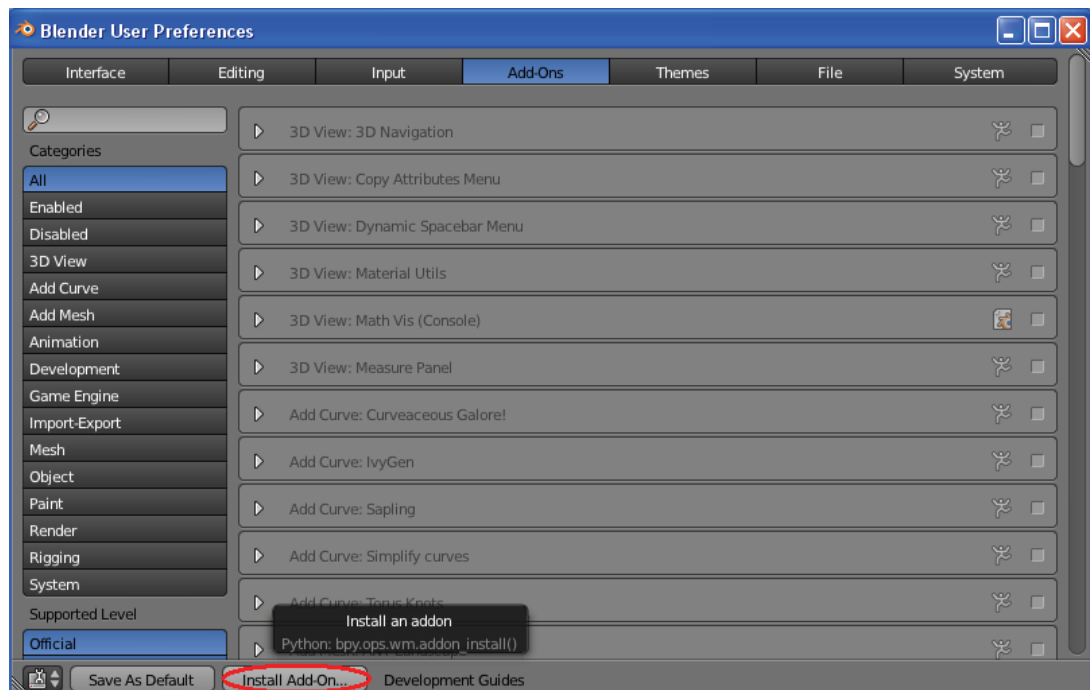
Completano la definizione del **file .py**, riportato per intero a fine sezione, l'importazione dei **moduli** da utilizzare e la definizione del campo **bl_info**, che avviene secondo le modalità illustrate nel capitolo 6.

```
import bpy
import mathutils

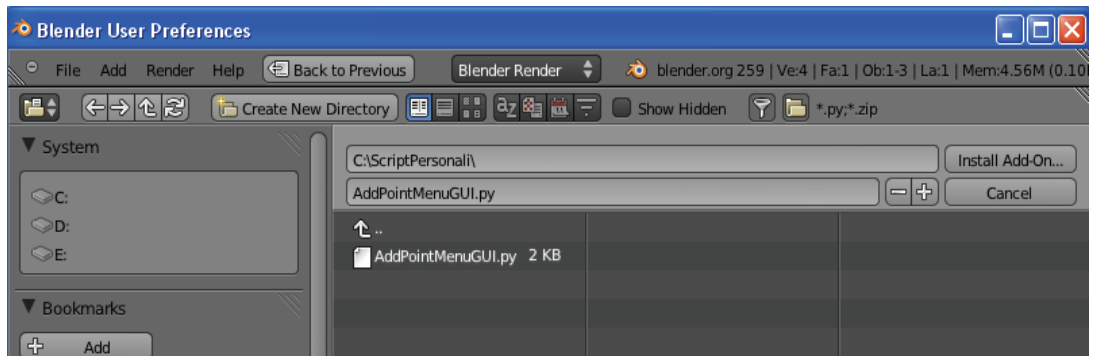
bl_info = {
    "name" : "Add Point",
    "author" : "Francesco Milanese",
    "version" : (1,0),
    "blender" : (2,5,9),
    "api" : 35853,
    "location" : "View 3D > Add > Mesh > Add Point",
    "description" : "Aggiunge una mesh di un solo vertice in 0,0,0",
    "warning" : "",
    "wiki_url" : "",
    "tracker_url" : "",
    "category" : "Add Mesh"
}
```

Salvato il file su disco, si procede alla sua **installazione in Blender** sotto forma di **Add-On**; per far ciò, è necessario:

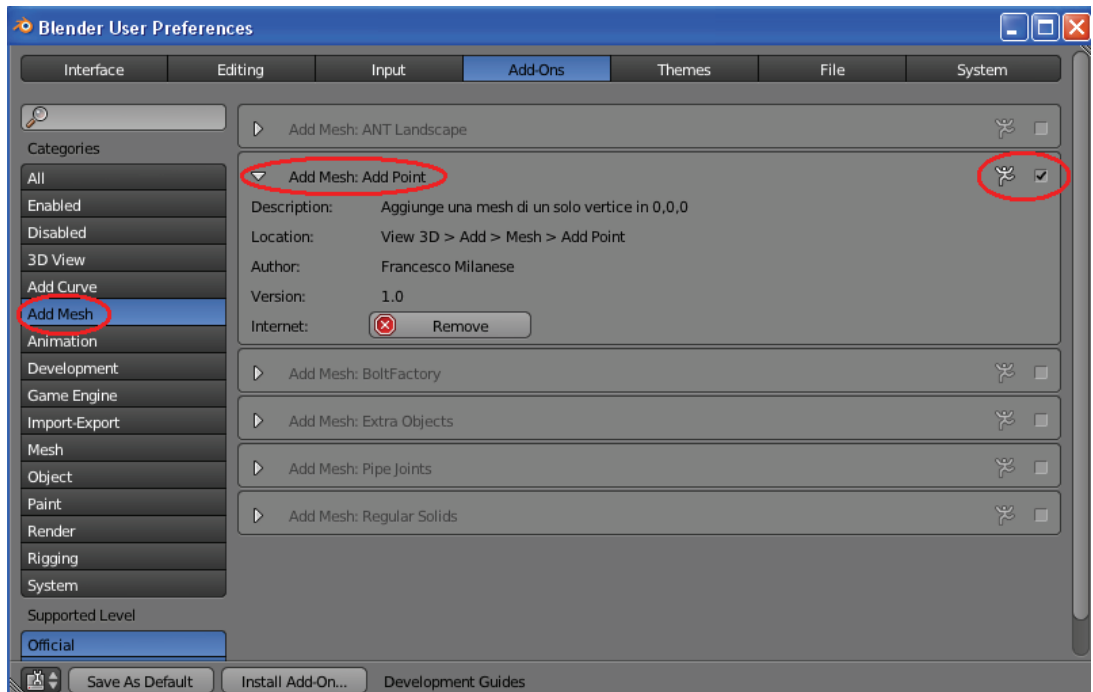
1. aprire la finestra **User Preferences** di **Blender**, sezione **Add-Ons**;



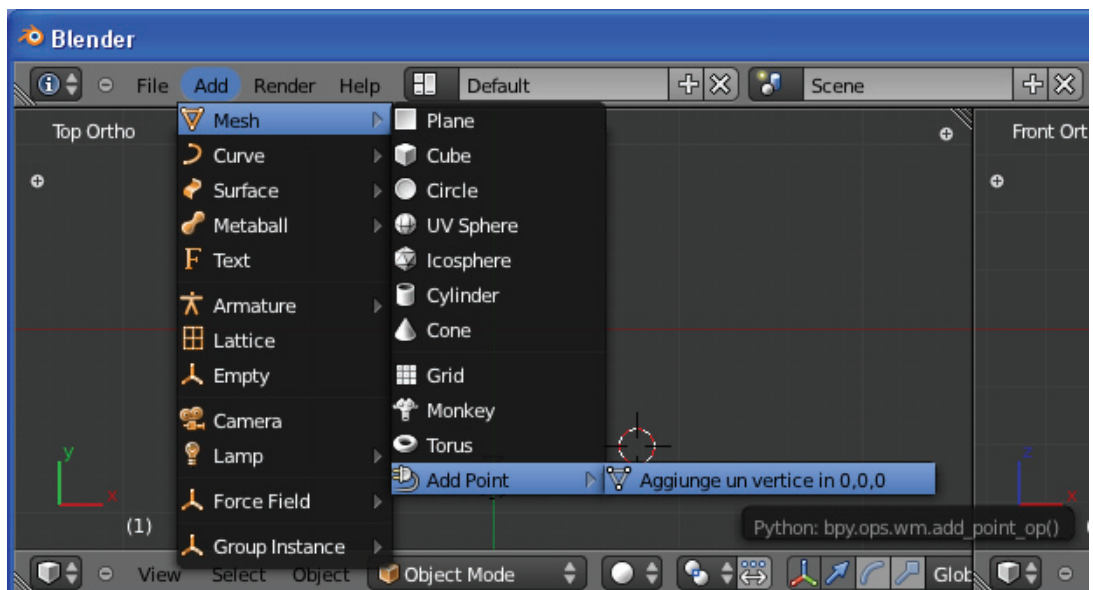
2. cliccare su **"Install Add-On..."**, in basso;
3. scegliere, dalla finestra stile esplora risorse, il **file .py** dello script che abbiamo salvato su disco;



4. selezionare la **categoria** del nostro script (in questo caso, *Add Mesh*) e, una volta individuato lo script nella parte destra della finestra, **attivarlo**, selezionando la checkbox.



A questo punto sarà sufficiente scegliere “*Add → Mesh → Add Point → Aggiungi un vertice...*” per eseguire lo script: nella scena, verrà inserito un oggetto composto da un solo vertice.



Verrà quindi inserita una nuova mesh, apparentemente vuota, nella scena; passando in **Edit Mode** e facendo click col tasto destro del mouse sul punto ove si trova l'**Origin** (che con la sua icona copre l'unico vertice che compone la mesh) si selezionerà il vertice, che potrà essere estruso per creare un nuovo vertice collegato al primo mediante uno spigolo.

Il codice del file **.py** è riportato per intero nella pagina seguente.

Suggerimenti per provare a implementare altri script:

- creare nuovi **operatori** che definiscano figure e solidi, aggiungendo vertici, spigoli e facce alla mesh in fase di editing;
- anziché creare un vertice da un Plane, definire proprio un nuovo **tipo** di Mesh composta da un solo vertice e aggiungerla alla scena;
 - utilizzare **bpy.data.meshes.new(nomeOggetto)** per aggiungere una nuova mesh (“vuota”: vertici, spigoli e facce andranno definiti in seguito e la mesh aggiornata con la funzione **.update()**);
 - utilizzare **from_pydata(list, list, list)** per aggiungere una lista di vertici, spigoli e facce alla mesh appena definita);
 - utilizzare **object_data_add** di **object_utils** (importato da **bpy_extras**) per aggiungere effettivamente la nuova mesh alla scena corrente;
 - vertici, spigoli e facce di una mesh sono oggetti **MeshVertex**, **MeshEdges**, **MeshFaces**, da aggiungere alle rispettive liste (**vertices**, **edges**, **faces**) della mesh; ad es., per creare ed aggiungere tre vertici connessi da 3 spigoli e costituenti una faccia in una mesh inizialmente vuota, scrivere:

```
nuovaMesh = bpy.data.meshes.new("Triangolo")
vertici = [(0.0, 0.0, 0.0), (3.0, 0.0, 0.0), (0.0, 5.0, 0.0)]
spigoli = [(0,1), (1,2), (2,0)]
facce = [(0,1,2)]
nuovaMesh.from_pydata(vertici, spigoli, facce)
nuovaMesh.update()
from bpy_extras import object_utils
object_utils.object_data_add(bpy.context, nuovaMesh, operator=None)
```

- per mostrare a video una **finestra pop-up** per immettere informazioni (es.: posizione iniziale della mesh), cfr. **“Dialog Box”**, pag. 634 **Documentazione pdf**.

* * *

```
# Add-On ADD POINT; Francesco Milanese, 2011
# www.redbaron85.com; redbaron85@redbaron85.com
import bpy
import mathutils

bl_info = {
    "name" : "Add Point",
    "author" : "Francesco Milanese",
    "version" : (1,0),
    "blender" : (2,5,9),
    "api" : 35853,
    "location" : "View 3D > Add > Mesh > Add Point",
    "description" : "Aggiunge una mesh di un solo vertice in 0,0,0",
    "warning" : "",
    "wiki_url" : "",
    "tracker_url" : "",
    "category" : "Add Mesh"
}

class ADD_POINT_OP(bpy.types.Operator):
    bl_idname = "wm.add_point_op"
    bl_label = "Aggiunge un punto in Origin"

    # Chiamare questo operatore con bpy.ops.wm.add_point_op()
    def execute(self, context):
        bpy.ops.mesh.primitive_plane_add(location=(0.0,0.0,0.0))
        bpy.context.active_object.name = "__temp__"
        nuovoOggetto = bpy.data.objects['__temp__']
        vertici = nuovoOggetto.data.vertices
        bpy.context.tool_settings.mesh_select_mode[0] = True
        bpy.context.tool_settings.mesh_select_mode[1] = False
        bpy.context.tool_settings.mesh_select_mode[2] = False
        for v in vertici:
            v.select = False
        numVertici = len(vertici)
        for i in range(numVertici-1, 0, -1):
            vertici[i].select = True
            bpy.ops.object.mode_set(mode='EDIT')
            bpy.ops.mesh.delete(type='VERT')
            bpy.ops.object.mode_set(mode='OBJECT')
        bpy.ops.object.mode_set(mode='OBJECT')
        bpy.ops.object.origin_set(type='GEOMETRY_ORIGIN')
        return {'FINISHED'}

# Registra l'operatore appena definito nell'ambiente di Blender
bpy.utils.register_class(ADD_POINT_OP)

class ADD_POINT_MENU(bpy.types.Menu):
    bl_idname = "ADD_POINT_MENU"
```

```
bl_label = "Add Point"
bl_description = "" # Opzionale

def draw(self, context):
    layout = self.layout
    #layout.operator_context = "INVOKE_REGION_WIN"
    layout.operator("wm.add_point_op", text="Aggiunge un vertice in 0,0,0",
icon="MESH_DATA")

def menu_func(self, content):
    self.layout.menu("ADD_POINT_MENU", icon="PLUGIN")

def register():
    bpy.utils.register_module(__name__)
    # Aggiunge il menu' ADD_POINT_MENU al menu' INFO_MT_mesh_add di Blender
    bpy.types.INFO_MT_mesh_add.append(menu_func) # vd. menu_func(self, content),
appena sopra

def unregister():
    bpy.utils.unregister_module(__name__)
    # Toglie il nuovo menu' ADD_POINT_MENU da ADD MESH (quando si disattiva lo script)
    bpy.types.INFO_MT_mesh_add.remove(menu_func)

if __name__ == "__main__":
    register()
```

APPENDICE B

Case Study: il plugin “Mixer Luci”

Nell’**Appendice A** abbiamo parlato della definizione della funzione *draw*, all’interno di una classe, per definire un’interfaccia grafica per un oggetto, facendo uso di *self.layout* per richiamare l’impostazione grafica del contesto in uso: *self* rappresenta infatti l’istanza dell’oggetto e “*layout*” è un campo predefinito che restituisce un oggetto di tipo *UILayout*⁴.

Questo tipo, definito in *bpy.types.UILayout*, mette a disposizione molti campi e metodi per suddividere l’area grafica in zone (*box* per un “quadrato”, *col* e *row* per definire colonne e righe) ed inserire al loro interno degli elementi, ossia gli *operators* (tramite la funzione *operator*, appunto), associati alle classi *Operator* definite mediante codice: cliccando sulle voci del menù o del *box*, viene richiamato l’*Operator* (il modificatore o la porzione di codice da noi definita) registrato nelle **API Blender Python**.

In questo secondo **Case Study** mostreremo come poter aggiungere un pannello alla **Tool Shelf** ed inserire al suo interno degli *Operators* particolari (le *props*) per effettuare alcune operazioni.

Per prima cosa, parliamo dei pannelli dal punto di vista del codice: si tratta di classi che estendono dal tipo *bpy.types.panel* e dove *bl_idname* non ha bisogno di essere specificato (o meglio, non in maniera esplicita: se non specificato, *bl_idname* prenderà il nome della classe all’interno del quale è specificato, per qualsiasi tipo di classe) quanto dei campi:

⁴ Per una descrizione completa di campi e metodi messi a disposizione da questo elemento si rimanda alla Documentazione delle API Blender Python.

- **bl_label** = l’etichetta da far comparire in cima al pannello, il suo “nome” visualizzato;
- **bl_space_type** = finestra principale dell’applicazione (*VIEW_3D*, *OUTLINER*, ...) dove dovrà apparire il pannello;
- **bl_region_type** = sotto-regione della finestra, specificata in **bl_space_type**, dove dovrà apparire il pannello (es.: *WINDOW*, *HEADER*, *TOOLS*, ...);
- **bl_context** = il contesto di esecuzione del pannello e degli operatori in esso contenuti; es.: *objectmode*.

Per un elenco completo delle costanti predefinite utilizzabili per **Space Type** e **Region Type** si rimanda alla **Documentazione** del tipo **bpy.types.Panel**; mostriamo invece un esempio pratico: scrivendo in una finestra **Text Editor** la seguente porzione di codice:

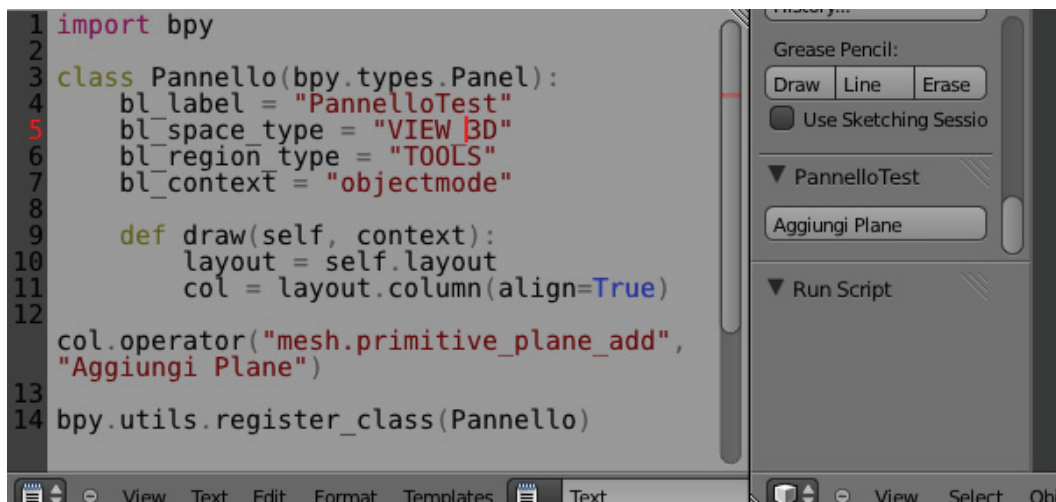
```
import bpy

class Pannello(bpy.types.Panel):
    bl_label = "PannelloTest"
    bl_space_type = "VIEW_3D"
    bl_region_type = "TOOLS"
    bl_context = "objectmode"

    def draw(self, context):
        layout = self.layout
        col = layout.column(align=True)
        col.operator("mesh.primitive_plane_add", "Aggiungi Plane")

bpy.utils.register_class(Pannello)
```

e cliccando su **Run Script** nell’header della finestra, nella **Tool Shelf** di una finestra **3D View** vedremo apparire una sezione con etichetta **“PannelloTest”** (lo stesso nome specificato nel campo **bl_label**) e un pulsante con etichetta **“Aggiungi Plane”** che, se cliccato, ci consentirà appunto di aggiungere un elemento di tipo **Mesh Plane** nella scena, nel punto ove si trova il **3D Cursor**, ricorrendo all’operatore primitivo di Blender **“mesh.primitive_plane_add”** di **bpy.ops**.



Ciò ci porta all'analisi del contenuto della funzione **draw**: dopo aver richiamato **self.layout** per ottenere l'oggetto **UILayout** “standard” del tipo **Panel**, aggiungiamo una colonna specificando che vogliamo attivare l'allineamento degli elementi al suo interno (il parametro **align**, che di default ha valore **False**, viene impostato a **True**), dopodiché inseriamo un **Operator** nella colonna.

L'**Operator** così definito avrà forma di pulsante, con etichetta specificata nel secondo parametro (**text**) e richiamerà la funzione primitiva di **Blender** che serve ad aggiungere un **Plane** di default nella scena.

Come sappiamo, il pulsante non è l'unico **layout** disponibile per effettuare delle operazioni o cambiare delle proprietà: abbiamo infatti *sliders*, *checkbox*, *color boxes*, *color ramps*, ecc....

La buona notizia è che per inserire i controlli delle proprietà è sufficiente utilizzare la funzione **prop** di **UILayout**, specificando almeno i due campi obbligatori (**data**, la variabile associata all'oggetto della scena da controllare, e **property**, il nome – sotto forma di stringa – della proprietà da modificare per l'oggetto indicato in **data**), per inserire il tipo di pulsante adatto in maniera automatica, ossia senza doverlo specificare esplicitamente.

Veniamo finalmente all'**Add-On** oggetto di questa sezione: un pannello, da inserire nella **Tool Shelf** in **Object Mode**, che consente di regolare mediante *sliders* l'intensità luminosa e il colore delle fonti di luce (oggetti **Lamps**) presenti nella scena.

La definizione dello strumento (costituito dal solo pannello) è molto semplice: è sufficiente creare e inserire un pannello che, per ogni fonte di luce **Lamp**, metta a disposizione i controlli associati ai vari parametri che vogliamo modificare; come anticipato, **Blender** recupererà automaticamente il tipo di controllo da utilizzare (pulsante, checkbox, ...) a seconda della **property** da modificare e lo inserirà nel pannello.

Esaminando il **Tooltip** restituito quando il cursore del mouse si trova sullo slider “**Energy**” della scheda **Object Data** di una **Lamp**, nella **Properties Window**, notiamo che lo slider influenza il campo “**energy**” di un oggetto **PointLamp**, che estende “**Lamp**” (dal quale ricava “**energy**” e altri campi e metodi); allo stesso modo, notiamo che il campo colore della **Lamp** si chiama “**color**”: limitiamoci a questi due campi.

Per suddividere il pannello in **box**, uno per ciascuna **Lamp**, utilizziamo appunto la funzione **box()** di **layout**, che restituisce un elemento di questo tipo, inserendo poi i controlli all'interno del

box (disposti, di default, in colonna, un elemento per riga) appena creato; per stampare il nome della **Lamp** in questione all'inizio di ogni **box**, utilizziamo la funzione *"label(text, icon)"*.

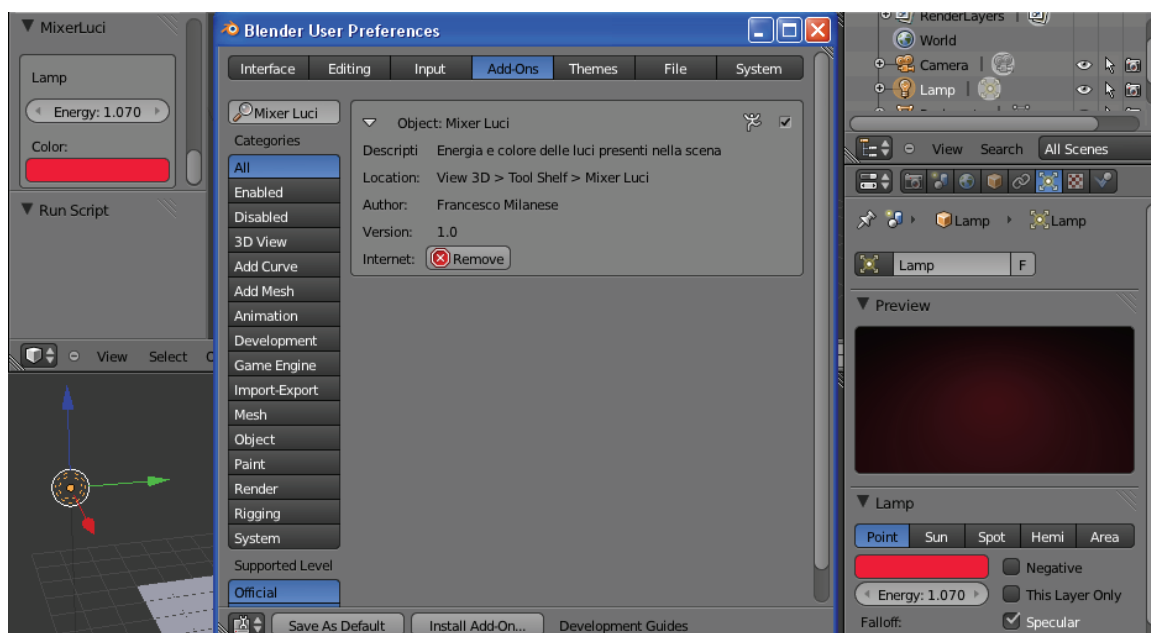
Attenzione: all'interno del blocco data vengono memorizzati anche gli elementi cancellati o non più utilizzati; per mostrare i comandi delle **Lamp** effettivamente presenti nella scena, all'interno del blocco *for* verrà posto un blocco *if* che controllerà che il numero di utilizzatori (*users*) del blocco dati in esame sia maggiore di 0; in caso contrario, non prenderà in considerazione quel blocco dati e non ne riporterà i parametri nel pannello.

Ecco quindi il corpo centrale dello script:

```
class MixerLuci(bpy.types.Panel):
    bl_label = "MixerLuci"
    bl_space_type = "VIEW_3D"
    bl_region_type = "TOOLS"
    bl_context = "objectmode"

    def draw(self, context):
        layout = self.layout
        col = layout.column(align=True)
        for l in bpy.data.lamps:
            if l.users > 0:
                box = layout.box()
                box.label(l.name)
                box.prop(l, "energy")
                box.prop(l, "color")
```

Completano la definizione dell'*Add-On* il dizionario *bl_info* e le funzioni per **registrare** e togliere classi e moduli dall'ambiente di **Blender**. Attivando lo script come **Add-On**, il suo pannello sarà sempre visibile nella **Tool Shelf** di ogni **3D View**.



Add-On Mixer Luci; Francesco Milanese, 2011

www.redbaron85.com; redbaron85@redbaron85.com

```
import bpy
```

```
bl_info = {
    "name" : "Mixer Luci",
    "author" : "Francesco Milanese",
    "version" : (1,0),
    "blender" : (2,5,9),
    "api" : 35853,
    "location" : "View 3D > Tool Shelf > Mixer Luci",
    "description" : "Energia e colore delle luci presenti nella scena",
    "warning" : "",
    "wiki_url" : "",
    "tracker_url" : "",
    "category" : "Object"
}
```

```
class MixerLuci(bpy.types.Panel):
    bl_label = "MixerLuci"
    bl_space_type = "VIEW_3D"
    bl_region_type = "TOOLS"
    bl_context = "objectmode"

    def draw(self, context):
        layout = self.layout
        col = layout.column(align=True)
        for l in bpy.data.lamps:
            if(l.users > 0):
                box = layout.box()
                box.label(l.name)
                box.prop(l, "energy")
                box.prop(l, "color")
```

```
def register():
    bpy.utils.register_module(__name__)
```

```
def unregister():
    bpy.utils.unregister_module(__name__)
```

```
if __name__ == "__main__":
    register()
```

NOTE