Initiation to Python in Blender

Bbug - Belgian Blender User group
bbug.tuxfamily.org
by Collignon David
for Blender 2.69.0
2014 v. 0.3

Introduction

Python is a programming language created in the late 1980 (its implementation started around 1989) by **Guido van Rossum** (aka Benevolent Dictator for Life), it has the following characteristics :

- General-purpose
  (designed to be used for writing software in a wide variety of application domains)
- High-level
  (strong abstraction from the details of the computer such as memory allocation)
- Multiple programming paradigm
  - Object Oriented
  - Imperative programming (variable assignment, conditional branching, loop, ...)
  - Functional programming
- Interpreted (<> compiled)
- Dynamic typing (type checking at runtime, a value has a type but not a variable)
- Automatic memory management
- Free & Open source

Strength
Easy to use (simple syntax rules, high level of abstraction) so it's a good language to start programming (it is taught in both ULB and UCMons for example).
Code is compact and expressive and, as such, easy to read.
Python can be used in almost every 3D Package (Maya, C4D, 3ds Max, Modo, Nuke) or to write desktop softwares and even web applications (for example with the framework Django).
Cross platform (Win, Mac, Linux, Unix).
Free & Open source.

Weaknesses
Slow (compared to C++ for example)
Not as many libraries as C or Java.
No variable check, no constant or private variable, methods, ...

2.7.6 vs 3.3.3
Small syntax changes (ex.: print → print()).
Python 3 is the language future but not backward-compatible (conversion is mostly painless).
Many new functionalities were backported in 2.6 & 2.7.

Python in Blender
Blender use Python 3.3.0.
Python accesses Blender's data in the same way as the animation system and user interface; this implies that any setting that can be changed via a button can also be changed from Python.

Philosophy

Python philosophy is mostly about a clean, **easily readable** and **compact code**.

Python uses **whitespace indentation**, rather than curly braces or keywords, to delimit blocks. (as it help standardize code) and is case sensitive.

And finally, it was always intended to have fun while coding with Python !
(It's a reference to the Monty Python after all!)

“"There should be one—and preferably only one—obvious way to do it.”

“Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Readability counts.”

```
// Hello world in C
#include <stdio.h>
 int main(void) {
    printf("Hello, world!\n");
    return(0);
 }
```

```
// Hello World in Python
print ("Hello, world!")
```

Zen of python (for the fun)

```
// Copy paste and run this in Blender
import this

dir(this)
this.c
this.d
this.i
this.s
"".join([c in this.d and this.d[c] or c for c in this.s])
help(this)
```

Blender interface tips

Security setup in user preference
Python script won't run from an untrusted source.
Info Area : File > user Preferences > File : Check Auto run Python Script

Tooltip & right click to online API
Hover over buttons and the like in the UI to see the API equivalent.
Info Area : File > User Preferences > Interface : Show Python Tooltip
Look how it's done : Right button over UI > Online Python reference / Edit Source

Console
Show error messages and print() results.
Windows : Info area > Window > Toggle system console, Linux : start Blender from a console

Conflict internal/external file
Click on the **Life Buoy** icon that appear next to the script name (only in case of conflict) to set source (internal/external).

Note : There seems to be a bug with multiple internal files referencing each other which don't update correctly...

Workspace - Scripting

Change workspace via the drop-down in the Info area or Ctrl+Left / Right.
The new layout is divided in **6 area**, we won't look at the 3D View, Properties or the Outliner.

Python Console Area
Kind of a quick code sandbox.
Autocompletion (CTRL+Space) for all **methods** and **arguments** in bpy
Copy to clipboard (as script)
print() isn't required

```
1 + 1   # Simplest program ever : )
pi      # Work into any field in the Blender UI
```

Info Area
Show python code of what you just did by hand, simply copy-pasting (right-click toggle select/deselect, Ctrl+C, Ctrl+V) the code generated from 'add Cube', 'G' & 'R' can serve as a basis for a quick'n dirty script such as :

```
import bpy
import random

for i in range(0, 10):
        randomCoord = (random.randrange(-2, 3),
                       random.randrange(-2, 3),
                       random.randrange(-2, 3))
        rot = random.randrange(0, 360)

        bpy.ops.mesh.primitive_cube_add()
        bpy.ops.transform.translate(value = randomCoord)
        bpy.ops.transform.rotate(value = rot, axis = (0, 1, 0))
```

Please note that this script works only because translate and rotate are applied to the active object (i.e. the object just added by primitive_cube_add is already selected!). Thinking of the step if you would have done it by hand might help understanding this process.

Also, primitive_cube_add place the new object at the 3D cursor position as expected while using the UI.

Text Editor Area
Too few options but really good at handling very large files.
Minimal autocomplete : only works with words already present in the file.
Can convert text to 3D object.
Work with external files : Text Editor Area > Text > Open Text Block (Alt+O)
Register to launch script at start as module (useful for classes that need to be registered and unregistered).

Look at text Editor Area > Templates > Python > UI Panel Simple for example

Python Syntax

The **help()** function is part of the pydoc library, which has several options for accessing the documentation built into Python libraries.

One other useful function is **dir()**, which lists the objects in a particular namespace. Used with no parameters, it lists the current globals, but it can also list objects for a module or even a type:

```
print('Hello World')
help(print)      # Give info on print()
```

Variable assignation

```
my_var = 5     # Case-sensitive, must start with a letter, valid characters : numbers, letters, -, _
one, two, three, four = 1, 2, 3, 4       # Same as one = 1, two = 2, three = 3 and four = 4
x, y = y, x                 # Conveniently swap 2 variable's values
my_tuple = 1, 2, 3      # Packing and unpacking
```

Available data type

integers, floats, complex numbers, strings, lists, tuples, dictionaries, set, files objects & classes.
None # Empty value

String

Escape character /, difference between ' ', “ “ & ''' '''
String concatenation with both + & *

```
print('-' * 30)
```

Numbers

integer (1, -3), float (3.2), boolean (True, False), complex number (3+2j)

Operators : +, -, *, /, //, **, %

Mathematical order of operations :

1. power and roots
2. multiplication and division
3. addition and subtraction

Lists [ ]

Array, index, [start : finish], len()

```
my_list[0]
nested_list[0][1]
a_list = [x for x in range(1, 30, 3) if x % 2]    # List comprehension
```

```
# It's usually bad practice to update a list you're iterating through
# Faulty code /!\
a = ['a', 'b', 'c', 'd', 'e', 'f']
for x in a:
        print('Current value = ' + str(x))
        if x == 'a':
                index = a.index(x)
                print('-> index = ' + str(index))
                del a[index]
print(a)        # Where's my second loop ?
```

Tuples ( )
Tuples are like lists but **immutable**.

```
my_tuple = (1,)        # Python quirk : one element tuple
```

Set
A set in Python is an unordered collection of objects used in situations where **membership** and **uniqueness** in the set are main things you need to know about that object.

```
my_list = [0, 1, 1, 2, 2, 2, 3]
my_list_without_duplicate = set(my_list)
```

Dictionaries {}
Dictionaries access values by means of integers, strings, or other Python objects called **keys**.

```
my_agenda = {'Firstname': 'David', 'Family_name': 'Collignon', 'Answer': 42}
print(my_agenda['Firstname'])
```

Dynamically typed variable

```
a = '1 string'
a = 1           # Data type changed to int without any casting or raising any error
print(type(a))  # Get variable data type
```

Casting
Turn a variable into another data type with bool(), list(), set(), str()...

```
age = int(input('Please, give me your age '))
age += 1
print('Next year, you\'ll be ' + str(age))
```

Deep copy
Variables only reference a complex data type, therefore in this example both variables point toward the same data.

```
a = [1, 2, 3]
b = a
b[0] = 'one'
print(a, b)
```

To avoid that, make a deep copy

```
my_new_list = list(my_old_list)        # 1st method : new distinct but identical list
my_new_list = my_old_list[:]           # 2nd method : new list = old from start to finish
```

User input
Quickest (dirtiest ?) way to pass an used defined argument.

```
user_choice = input('Message string ')        # /!\ Only work with console open
```

Logic test

<, <=, >, >=, ==, != (<> is valid too), and, not, in

```
if 0 < x < 10:   # Shorthand for if 0 < x and x < 10:
```

Loop & Branching
while True :
for ... in ... :
range(StartIncluded, EndExcluded, Step) # Pythonic way for for($1 = 0; $i < 10; $1++) {...}
break, continue   # break loop, skip current iteration
pass   # needed in empty indented block (# ToDo)

```
if True:
    do_something()
elif True:
    do_something_else()
else:
    failsafe()
```

Exception
Try ... except ...

```
raise exception(args)
```

Exercise 1 & FAQ

Make a list of all prime numbers under 100 (Harder than it look).

```
import time

def is_prime(n):
	''' Check if integer n is a prime '''

	# 0, 1 and lower are not primes
	if n < 2:
		return False

	# 2 is the only even prime number
	if n == 2:
		return True

	# all other even numbers are not primes
	if n % 2 == 0:
		return False

	# range starts with 3 and only needs to go up the squareroot of n
	# for all odd numbers
	for x in range(3, int(n ** 0.5) + 1, 2):
		if n % x == 0:
			return False

	return True

prime = []
start = time.clock()

for x in range(-3, 101):
	if is_prime(x):
		prime.append(x)

timer = time.clock() - start

print(prime, timer)
```

Functions

Code block that can be called multiple times and even **reused** with **differents** parameters. It's considered best practice to **always return something** when using a function especially for instance when things got wrong (return False, -1, ...).

Function can have no argument, one or more arguments, an indefinite number of args and parameter names. Watch out for **local scope** of variables!

```
def capped_sum(*numbers, max = 999):
        ''' Return the sum of all numbers arguments
        capped by the max arg argument '''
        result = sum(numbers)
        return min(result, max)

print(capped_sum.__doc__)
print(capped_sum(1, 2, 3))
print(capped_sum(1, 2, 3, 4, 5, max = 5))
```

Exercise 2 & FAQ

```
def factoral(n):          # Cheapest factorial ever T__T
        if n < 0:
                return -1 # Fac only work for positive number
        elif 0 <= n <= 1:
                return 1
        else:
                result = 1
                tmpList = [x for x in range(2, n+1)]
                for i in tmpList:
                        result *= i
                return result

def factoral_recursive(n):      # Cheapest recursive factorial ever T__T
        if n < 0:
                return -1 #Fac only work for positive number
        elif 0 <= n <= 1:
                return 1
        else:
                return n * factoral_recursive(n - 1)

print(factoral_recursive(5), factoral(5))
```

Exercise 3

```
import bpy
import math

radius = 10
z = 0
size = [1] * 3

def angle_to_vector(ang):
	return (math.cos(ang), math.sin(ang))

for angle_in_degree in range(0, 360 * 10, 15):
	xy = angle_to_vector(math.radians(angle_in_degree))
	temp_loc = (xy[0] * radius, xy[1] * radius, z)
	z += 0.2
	size = [size[0] * 0.99] * 3

	bpy.ops.mesh.primitive_cube_add(location = temp_loc)
	bpy.ops.transform.resize(value = size)
```

Notation & Good habits
Always comment (# ...) your code with **meaningful** informations.
Use **explicit** variable and function names.

```
Variable and Function : lowercase_separated_by_underscore
Constant : UPPERCASE_SEPARATED_BY_UNDERSCORE # /!\ Convention ; still a variable
Class : CamelCase    # Class name must always start with a capital letter
Class method not part of the API : _lowercase_starting_with_underscore # /!\ Convention
Private variable : __variable # /!\ Still not a private variable as you would expect from Java
Class method not to be overridden by a subclass : __method()
Magick methods : __init__ # if you don't know what they're for, don't call them.

if is_playing:
	check_score()

def calculate_interest(amount, currency = 'USD', annual_interest_rate, time_period_in_year = 1):
	pass

Shape.get_area(self, width, height)
```

OOP

Object-oriented programming (OOP) is a programming paradigm that represents **concepts** as "objects" that have **attributes** that describe the object and associated function known as **methods**. Objects, which are usually **instances** of classes, are used to interact with one another to design applications and computer programs.

- **Class** and **Instance**
- **Encapsulation** (hide and protect some variables and implementations, access through getter/setter) and **abstraction** (we don't care how it's done). Well, that's the theory, in Python private and such don't exist !
- **Inheritance** and/vs **Compositing**

Inheritance
In object-oriented programming, inheritance is when an object or class is based on another object or class, using the same implementation; it is a mechanism for **code reuse**. The relationships of objects or classes through inheritance give rise to a **hierarchy**.

```
class Shape:
        def __init__(self, x, y):
                self.x = x
                self.y = y

        def move(self, delta_x, delta_y):
                self.x = self.x + delta_x
                self.y = self.y + delta_y

class Square(Shape):
        def __init__(self, side = 1, x = 0, y = 0):
                super().__init__(x, y) # Implicit call to parent constructor
                self.side = side

class Circle(Shape):
        def __init__(self, r = 1, x = 0, y = 0):
                super().__init__(x, y)
                self.radius = r

c = Circle(1, 0, 0)
c.move(2, 2)            # Circle inherited the move() method
print(c.x, c.y)
```

Example
Let's think about the old Asteroïd game from an OOP point of view...
What do we need ? Maybe 3 classes for Ship, Missile and Rock ?
Ship will need to handle rotate_left(), rotate_right(), shoot() and go_forward().
Missile will inherit ship rotation at instantiation through passing a parameter to the constructor.
Missile and Rock will test at each game tic collision versus each other, ship and screen borders.
All 3 need angular velocity and some sort of update_position() method (inheritance from Sprite).
Finally a MainGame class using composition will instantiate those three as often as needed and update score, UI, Game over and so on.

Static and Class methods
You can invoke **static methods** even though no instance of that class has been created so Python doesn't have to instantiate a bound-method for each object we instantiate (creating them has a cost).

Class methods are similar to static methods in that they can be invoked before an object of the class has been instantiated or by using an instance of the class. But class methods are implicitly passed the class they belong to as their first parameter. By using a class method instead of a static method, we don't have to hardcode the class name into get_pi(). That means any subclasses of Circle can still call it and refer to their own members, not those in Circle.

```
class Circle:
	pi = 3.14159   # Class variable

	def __init__(self, radius = 1):
		self.radius = radius    # Instance variable

	def area(self):
		return self.radius * self.radius * self.pi

	@staticmethod
	def get_pi():
		return Circle.pi         # Use class name

	@classmethod
	def get_PI(cls):
		return cls.pi            # cls = self.__class__

c = Circle(5)
print('Area of the instance c : ' + str(c.area()))
print(c.get_pi())
print(Circle.get_pi())
print(Circle.get_PI())
```

Import and Modules

```
import bge
from bpy import context as C
```

If you import somemodule the contained globals will be available via somemodule.someglobal. If you from somemodule import * ALL its globals (or those listed in __all__ if it exists) will be made globals, i.e. you can access them using someglobal without the module name in front of it. Using from module import * is discouraged as it clutters the global scope and if you import stuff from multiple modules you are likely to get conflicts and overwrite existing classes/functions.

Random Module

```
import random

r_n_g = random.Random()
#r_n_g.seed(1)          # Pseudo-random ; identical seed always return the same result

print('-' * 20)
for x in range (0, 10):
        print(r_n_g.randrange(0, 5))
```

Time Module

```
import time

time.sleep(3)  # Wait before executing the next line of code
time.clock()    # Use 2 calls before & after a block of code to check for bottleneck/performances
```

Sys Module

```
import sys

print(sys.version, sys.platform)
```

Python specific to Blender

A lot of change to be expected in the future for the bge.

http://code.blender.org/index.php/2013/06/blender-roadmap-2-7-2-8-and-beyond/

```
import bpy          # Blender Python
import bge          # Blender Game Engine (bpy won't work in the stand alone player)
import mathutils    # Vector, Matrix, ...
```

Application modules
The contents of the **bpy module** are divided into several classes, among which are:

- **bpy.context** - contains settings like the current 3D mode, which objects are selected, and so on.
- **bpy.data** - This is where you find the contents of the current document. Whether they exist on the scene or not.
- **bpy.ops** - Operators perform the actual functions of Blender; these can be attached to hotkeys, buttons or called from a script. When you write an addon script, it will typically define new operators. Every operator must have a unique name.
  Operators don't have return values as you might expect, instead they return a set() which is made up of: {'RUNNING_MODAL', 'CANCELLED', 'FINISHED', 'PASS_THROUGH'}.
  Calling an operator in the wrong context will raise a RuntimeError, there is a poll() method to avoid this problem.
- **bpy.types** - information about the types of the objects in bpy.data.
- **bpy.utils** - Utilities
- **bpy.path** - Path Utilities similar to os.path
- **bpy.app** - Application Data
- **bpy.props** - functions for defining properties. These allow your scripts to attach custom information to a scene, that for example the user can adjust through interface elements to control the behaviour of the scripts.

Standalone Modules

- **mathutils** - Math
- **bgl** - OpenGL Wrapper
- **blf** - Font Drawing
- **gpu** - GPU functions
- **aud** - Audio System
- **bpy_extras** - Extra Utilities
- **bmesh** - BMesh Module

The module **mathutils** defines several important classes that are used heavily in the rest of the Blender API :

- **Vector** - The representation of 2D or 3D coordinates.
- **Matrix** - The most general way of representing any kind of linear transformation.
- **Euler** - A straightforward way of representing rotations as a set of Euler angles, which are simply rotation angles about the X, Y and Z axes. Prone to well-known pitfalls such as gimbal lock.
- **Quaternion** - On the face of it, a more mathematically abstruse way of representing rotations. But in fact this has many nice properties, like absence of gimbal lock, and smoother interpolation between two arbitrary rotations. The latter is particularly important in character animation.
- **Color** - A representation of RGB colours and conversion to/from HSV space (no alpha channel).

How to Find Something ?
bpy.data.objects can be used as a list/dictionary of all object on the scene (bpy.data.objects[0] or bpy.data.objects['Cube'])

You can check the index with **bpy.data.objects.find("Cube")**. The return value will be index number of object named "Cube".

```
list(bpy.context.selected_objects)   # Get only selected objects, depend on context obviously
bpy.context.active_object            # Get the active object

list(bpy.data.objects)               # Get only Objects
list(bpy.data.meshes)                # Get only Meshes
list(bpy.data.materials)             # Get only Materials
```

Every object have (amongst a lot of other things) :

1. a type defined by a constant : **bpy.data.objects[0].type** (Read-only)
   ARMATURE, CAMERA, CURVE, EMPTY (included Force Fields), FONT (Text objects), LAMP, LATTICE, MESH, META (Metaball), SPEAKER, SURFACE
2. a name : **bpy.data.objects[0].name** (Read/Write)
3. a location represented by a vector : **bpy.data.objects[0].location** (Read/Write)
4. a scale represented by a vector : **bpy.data.objects[0].scale** (Read/Write)
5. you can select an object with **bpy.data.objects[0].select** (Read/Write : select = True)

```
cube = bpy.data.objects['Cube']              # Shorten data path as variable
bpy.context.scene.objects.active = cube      # Operation are applied to the active object
cube.select = True                           #
```

Operation - Manipulate Selection

```
bpy.ops.transform.rotate(value = 45, axis = (1, 0, 0))        # Ops apply only to the selection
bpy.ops.transform.translate(value = (1, 0, 0))
bpy.ops.transform.resize(value = (1, 0, 0))

bpy.ops.object.material_slot_add()
bpy.ops.object.rotation_clear()

bpy.data.objects['Cube'].location += mathutils.Vector((1, 1, 1))
```

Manipulate Vertex Coordinates
Local coordinates

```
import bpy

for item in bpy.data.objects:
        print(item.name)
        if item.type == 'MESH':
                for vertex in item.data.vertices:
                        print(vertex.co)
```

World coordinates.

```
import bpy

current_obj = bpy.context.active_object

for face in current_obj.data.polygons:
        verts_in_face = face.vertices[:]

        print("face index", face.index)
        print("normal", face.normal)

        for vert in verts_in_face:
                local_point = current_obj.data.vertices[vert].co
                world_point = current_obj.matrix_world * local_point
                print("vert", vert, " vert co", world_point)
```

bpy.context.mode return a Read-only value of the current context :
EDIT_MESH, EDIT_CURVE, EDIT_SURFACE, EDIT_TEXT, EDIT_ARMATURE, EDIT_METABALL, EDIT_LATTICE, POSE, SCULPT, PAINT_WEIGHT, PAINT_VERTEX, PAINT_TEXTURE, PARTICLE, OBJECT

Final exercise

1. Dynamically create 125 cubes
Position them as a 5x5x5 'cube' in XYZ
Iterate through all objects in the scene (avoid camera and the like) and randomly displace 2 vertices in each mesh also rotate and scale each object.
Apply a second material on each odd numbered.

2. Care to read, try to understand and port to Python 3 and Blender 2.67 this old script ?

```
# random_face_color.py (c) 2011 Phil Cote (cotejrp1)
#
# ***** BEGIN GPL LICENSE BLOCK *****
#
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version 2
# of the License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software Foundation,
# Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
#
# ***** END GPL LICENCE BLOCK *****
bl_info = {
        'name': 'Random Face Color',
        'author': 'Phil Cote, cotejrp1, (http://www.blenderaddons.com)',
        'version': (0,2),
        "blender": (2, 5, 9),
        "api": 39307,
        'location': '',
        'description': 'Generate random diffuse faces on each face of a mesh',
        'warning': 'Don\'t use on meshes that have a large number of faces.',
        'category': 'Add Material'}
"""
How to use:
- Select a material with no more than one material on it.
```

```
- Hit "t" to open the toolbar.
- Under "Random Mat Panel", hit the "Random Face Materials" button.
Note: as of this revision, it works best on just one object.
It works slightly less well when colorizing multiple scene objects.
"""
import bpy
import time
from random import random, seed

# start simple be just generating random colors
def getRandomColor():
	seed( time.time() )
	red = random()
	green = random()
	blue = random()
	return red, green, blue

def makeMaterials( ob ):
	for face in ob.data.faces:
		randcolor = getRandomColor()
		mat = bpy.data.materials.new( "randmat" )
		mat.diffuse_color = randcolor

def assignMats2Ob( ob ):
	mats = bpy.data.materials

	# load up the materials into the material slots
	for mat in mats:
		bpy.ops.object.material_slot_add()
		ob.active_material = mat

	# tie the loaded up materials o each of the faces
	i=0
	faces = ob.data.faces
	while i < len( faces ):
		faces[i].material_index = i
	i+=1

getUnusedRandoms = lambda : [ x for x in bpy.data.materials
	if x.name.startswith( "randmat" ) and x.users == 0 ]

def clearMaterialSlots( ob ):
	while len( ob.material_slots ) > 0:
```

```
                bpy.ops.object.material_slot_remove()

def removeUnusedRandoms():
        unusedRandoms = getUnusedRandoms()

        for mat in unusedRandoms:
                bpy.data.materials.remove( mat )

class RemoveUnusedRandomOp( bpy.types.Operator ):
        bl_label = "Remove Unused Randoms"
        bl_options = { 'REGISTER'}
        bl_idname = "material.remove_unusedmats"

        def execute( self, context ):
        removeUnusedRandoms()
        return {'FINISHED'}

class RandomMatOp( bpy.types.Operator ):

        bl_label = "Random Face Materials"
        bl_idname = "material.randommat"
        bl_options = { 'REGISTER', 'UNDO' }

        def execute( self, context ):
                ob = context.active_object
                clearMaterialSlots( ob )
                removeUnusedRandoms()
                makeMaterials( ob )
                assignMats2Ob( ob )
                return {'FINISHED'}

        @classmethod
        def poll( self, context ):
                ob = context.active_object
                return ob != None and ob.select

class RandomMatPanel( bpy.types.Panel ):
        bl_label = "Random Mat Panel"
        bl_region_type = "TOOLS"
        bl_space_type = "VIEW_3D"

        def draw( self, context ):
                self.layout.row().operator( "material.randommat" )
```

```
                row = self.layout.row()
                self.layout.row().operator( "material.remove_unusedmats" )

                matCount = len( getUnusedRandoms() )
                countLabel = "Unused Random Materials: %d" % matCount
                self.layout.row().label( countLabel )

def register():
        bpy.utils.register_class( RemoveUnusedRandomOp )
        bpy.utils.register_class( RandomMatOp )
        bpy.utils.register_class( RandomMatPanel )

def unregister():
        bpy.utils.unregister_class( RandomMatPanel )
        bpy.utils.unregister_class( RandomMatOp )
        bpy.utils.unregister_class( RemoveUnusedRandomOp )

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

Notes

This is only a draft of a training material for the free courses given each month in Brussels, Belgium by the Bbug (Belgian blender user group).

If you've learn anything useful here, please feel free to contribute and improve this first modest essay. Now and then, I've copy pasted explanations from BlenderSushi, The Quick Python Book Second Edition and some other online resources mentioned in the links section.

Give credit where credit is due :)

Found an error or a typo ? Please tell me and help me improve myself and this document.
And feel free to contact me at bbug.tuxfamily.org.

Free online courses about Python
Coursera - https://www.coursera.org/course/interactivepython
Udacity - https://www.udacity.com/course/cs101

Blender & Python
Python Book of magic - http://wiki.blender.org/index.php/User:Kilon/Python_book_of_magic

Blender API Quickstart
http://www.blender.org/documentation/blender_python_api_2_69_release/info_quickstart.html

Blender Noob to pro
http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Advanced_Tutorials/Python_Scripting/Introduction

Bbug - http://bbug.tuxfamily.org/index.php?p=/discussion/48/useful-links-liens-utiles-nuttige-links

Books
Free [FR]
Gérard Swinnen 2012 Apprendre à programmer avec Python 3 - http://inforef.be/swi/python.htm

Paid [EN]
Vernon C. 2010, 'The Quick Python Book Second Edition', Manning (ISBN 1-935182-20-X)

A few links & some references
Python, Python Software Foundation & Guido van Rossum
http://www.python.org
http://en.wikipedia.org/wiki/Python_%28programming_language%29
http://en.wikipedia.org/wiki/Guido_van_Rossum
http://en.wikipedia.org/wiki/High-level_programming_language

Zen of Python - http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html
Official Python Tutorial - http://docs.python.org/3/tutorial/
Coding style guide - http://www.python.org/dev/peps/pep-0008/

Methods with _, __ & __x__ & Magick methods
http://igorsobreira.com/2010/09/16/difference-between-one-underline-and-two-underlines-in-python.html
http://www.rafekettler.com/magicmethods.html

Class & static methods
http://stackoverflow.com/questions/12179271/python-classmethod-and-staticmethod-for-beginner