


# Desarrollo de Plugins en python para software educativo TurtleArt

	<p><b>Autor:</b> Valentín Basel</p> <p><b>Bio:</b> Soy analistas en sistemas informáticos y me desempeño como encargado del área informática del CEA-CONICET. Comencé en el mundo de GNU/LINUX por el año 2000 cuando instale un SUSE 6.0 y decidí dedicarme al mundo del Software libre. Actualmente soy embajador Fedora por Argentina y mi aporte para la comunidad GNU es sobre robotica educativa libre con el proyecto ICARO, un entorno de software y hardware de bajo costo para el desarrollo de robots educativos en colegios primarios y secundarios.</p> <p>Mi paso por python es relativamente nuevo (un par de años) anteriormente siempre programe en C/C++, sin embargo la facilidad para crear código, la excelente documentación y demás comodidades hicieron que cada vez mas programe en python como lenguaje principal.</p> <p><b>Email:</b> <a href="mailto:valentinbasel@gmail.com">valentinbasel@gmail.com</a></p> <p><b>Web:</b></p> <ul style="list-style-type: none"><li>• <a href="http://www.sistema-icaro.blogspot.com/">http://www.sistema-icaro.blogspot.com/</a></li><li>• <a href="http://valentinbasel.fedorapeople.org/">http://valentinbasel.fedorapeople.org/</a></li></ul>
---	---

TurtleArt ([http://wiki.laptop.org/go/Turtle\\_Art](http://wiki.laptop.org/go/Turtle_Art)) es un entorno de programación gráfico basado en el lenguaje Logo ([http://en.wikipedia.org/wiki/Logo\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Logo_(programming_language))), en el que se pueden hacer pequeños programas y realizar diseños con una tortuga de forma gráfica.

Desde la versión 106, TurtleArt soporta el formato de plugins, permitiendo mejorar y agregar nuevas "primitivas" a nuestro software. En este artículo explicaremos la creación de un plugin personalizado para TurtleArt. Este artículo requiere cierto conocimiento previo en programación orientada a objetos, Python y la forma de trabajo de Sugar ([http://en.wikipedia.org/wiki/Sugar\\_\(desktop\\_environment\)](http://en.wikipedia.org/wiki/Sugar_(desktop_environment))).

## Preparando el entorno

Primero debemos tener instalado en nuestra computadora un entorno Sugar (no es excluyente pero siempre es conveniente); en la distribución GNU/Linux Fedora es tan simple como poner:

```
$ su -c 'yum install sugar*'
```

o también:

```
su -c 'yum groupinstall sugar'
```

Con esos comandos nuestro sistema Linux instalará todo el entorno Sugar con sus actividades y el emulador **sugar-emulator** (muy cómodo para las pruebas).

A continuación descargaremos la última versión de TurtleArt desde la página web:

<http://activities.sugarlabs.org/en-US/sugar/downloads/latest/4027/addon-4027-latest.xo?src=addondetail>

También pueden descargar del siguiente link la versión de TurtleArt con el plugin “**tortucaro**” (Icaro + TurtleArt) para estudiarlo tranquilos (versión 106).

<http://valentinbasel.fedorapeople.org/TurtleArt.tar>

**Atención: esta versión no es igual a la que analizamos en este artículo.**

Luego de realizar la descarga, desempaquetamos el archivo **.tar** en nuestro **home** y procedemos a ver la estructura de directorios.

Los archivos que nos importa “tocar” están dentro del directorio **plugins**. La idea de esta versión de TurtleArt es posibilitar el desarrollo de nuevas paletas por parte de los desarrolladores sin necesidad de modificar el código fuente original del software (como en las versiones anteriores). Así, antes, en la versión 106, para hacer nuestro plugin básicamente teníamos que crear un archivo con el siguiente formato:

```
<NOMBRE>_plugin.py
```

Donde “<NOMBRE>” es la denominación que le daremos a nuestro plugin, por ejemplo: “prueba\_plugin.py”.

A partir de la versión 107, se mejoró el sistema para acomodar todos los archivos del plugin directamente en una carpeta dentro del directorio plugins.

## Desarrollando un plugin

Vamos a crear una carpeta de nombre “**prueba**” y dentro crearemos un archivo “**prueba/prueba.py**”. En nuestro archivo implementaremos una clase de Python con el mismo nombre que el archivo, y solamente pondremos la primera letra en mayúscula: “Prueba” (**hay que respetar ese formato**).

Ahora iremos agregando el siguiente código al archivo

```
# es necesario importar todos estos módulos para poder trabajar con
# TurtleArt

import gst
import gtk
from fcntl import ioctl
import os
from gettext import gettext as _
from plugins.plugin import Plugin
from TurtleArt.tapalette import make_palette
from TurtleArt.talogo import media_blocks_dictionary, primitive_dictionary
from TurtleArt.tautils import get_path, debug_output
import logging

_logger = logging.getLogger('TurtleArt-activity prueba plugin')

class Prueba(Plugin):

    def __init__(self, parent):
        self._parent = parent
        self._status = False
```

La primera parte es fundamental para poder importar todos los módulos internos de TurtleArt; aquí, por razones de extensión, no explicaré en detalle de que se trata cada módulo.

Con nuestra clase creada e inicializada (`def __init__`), procedemos a colocar el método `setup`

```
# dentro de la clase Prueba
def setup(self):
    palette = make_palette('prueba',
                          colors=["#006060", "#A00000"],
                          help_string=_('esta es una prueba'))
```

Acá declaramos una nueva paleta de nombre `prueba`; el campo `colors` representa los dos colores con los que se hará el gradiente del botón que muestra TurtleArt (para diferenciarlos de las otras “primitivas” del sistema). El campo `help_string` es bastante entendible, básicamente se trata de información sobre la paleta (pero no sobre los botones de esa paleta, como ya veremos).

```
# dentro de la clase Prueba y dentro del metodo setup

#[1]
primitive_dictionary['boton'] = self._boton_prueba

#[2]
palette.add_block('boton',
                 style='basic-style-larg',
                 label=_('boton'),
                 prim_name='boton',
                 help_string=_('boton de prueba'))

#[3]
self._parent.lc.def_prim('boton', 1, lambda self, valor: primitive_dictionary['boton'](valor))
```

Este es nuestro primer botón, acá definimos su comportamiento, estilo, nombre y función a la que apunta. en **[1]** definimos una “primitiva” para el diccionario. Cuando la coloquemos, el sistema llamará a `_boton_prueba` (la función donde estará el código de acción).

Para **[2]** `Palette.add_block` crea nuestro botón, el cual estará dentro de nuestra paleta `prueba`; el primer campo es el nombre del botón. Con `style` definimos que tipo de botón sera, si tendrá o no argumentos, si en lugar de enviar información la recibirá o si es un estilo especial. Para saber que tipos de estilos se pueden usar hay que revisar el archivo “**tapalette.py**” dentro del sub directorio “**TurtleArt**”, en nuestro directorio principal en el que descomprimimos el `.tar` (Son bastantes, nosotros sólo usaremos el estilo básico con un argumento `basic-style-larg`). El campo `label` es el nombre que mostrara el botón en su cuerpo; en el campo `prim_name` escribo lo mismo que en el primer campo (`boton`); `help_string` es el texto de ayuda que mostrará TurtleArt cuando pasemos el mouse por arriba del botón. la línea **[3]** de código es donde el sistema enlaza el botón con la función que definamos. `Lambda` es una interesante sintaxis para definir funciones mínimas implementada por python, lo que hacemos es pasar a la función `_boton_prueba` mediante `primitive_dictionary['boton']` la variable `valor`. Como es un botón básico de un argumento, permite poner una caja de valor y almacenarlo en dicha variable.

Finalmente, lo único que nos falta es la función `_boton_prueba` donde haremos toda la lógica del botón propiamente dicho.

```
# dentro de la clase Prueba
def _boton_prueba(self, valor):
    print "el valor del boton ", valor
```

La función es muy sencilla, lo único que hace es mostrar un poco de código trivial para ilustrar el funcionamiento de un plugin.

Como verán, programar plugins para TurtleArt no es de lo más complejo. Si bien hay poca documentación y la mayoría está en inglés, con un poco de paciencia se pueden armar un montón de paletas personalizadas para las más variadas actividades.

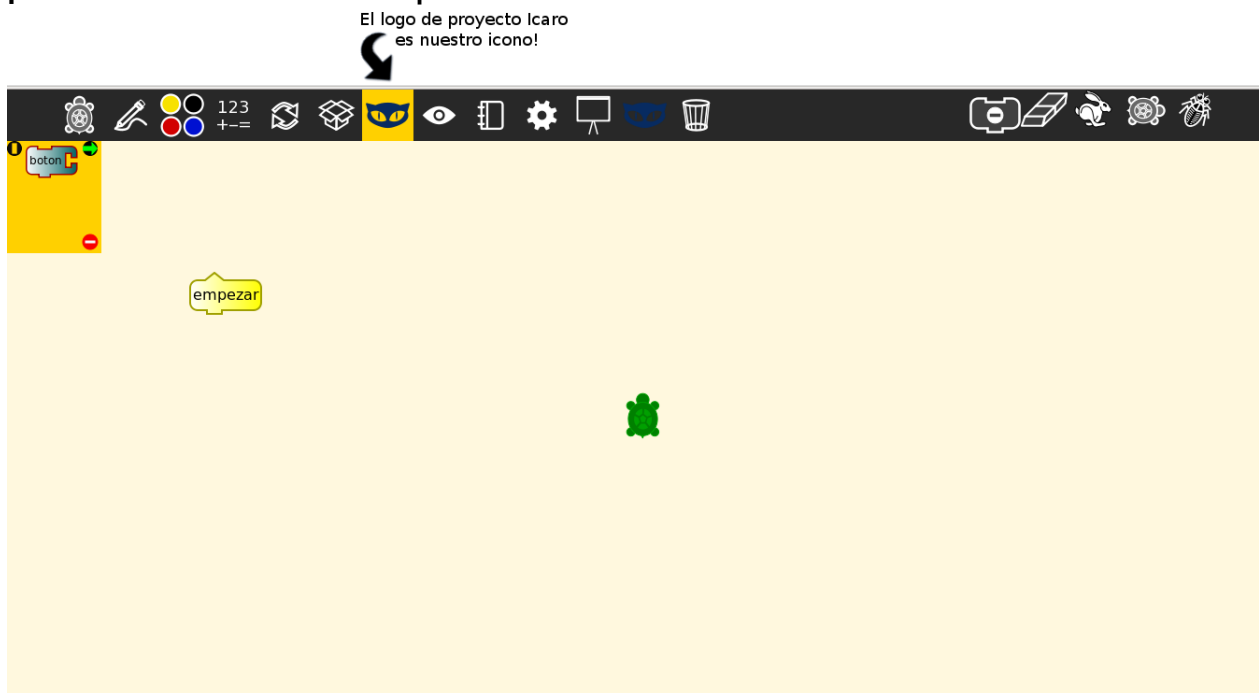
Con nuestro archivo “**prueba.py**” ya terminado lo único que nos falta es crear la carpeta “**prueba/icons**”, quedando el árbol de directorio de la siguiente manera:

```
~/TurtleArt/  
  plugins/  
    prueba/  
      icons/
```

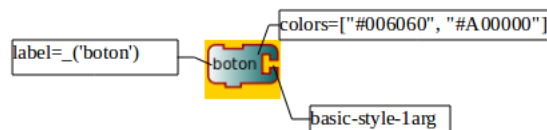
Dentro de **icons\*** hay que colocar dos archivos **.svg** de 55x55 píxeles con el nombre **pruebaon.svg** y **pruebaoff.svg** (el mismo nombre del plugin).

Con eso debería aparecer el icono **pruebaoff.svg** dentro de la barra de herramientas de TurtleArt.

### ¡TurtleArt con nuestro botón de prueba andando!



### Detalle del armado del botón dentro de la paleta.



Para terminar les presento el archivo **prueba.py** completo

```
import gst  
import gtk  
from fcntl import ioctl  
import os  
from gettext import gettext as _  
from plugins.plugin import Plugin  
from TurtleArt.tapalette import make_palette  
from TurtleArt.talogo import media_blocks_dictionary, primitive_dictionary  
from TurtleArt.tautils import get_path, debug_output
```

```
import logging
_logger = logging.getLogger('TurtleArt-activity prueba plugin')

class Prueba(Plugin):
    def __init__(self, parent):
        self._parent = parent
        self._status = False

    def setup(self):
        palette = make_palette('prueba',
                               colors=["#006060", "#A00000"],
                               help_string=_('esta es una prueba'))
        primitive_dictionary['boton'] = self._boton_prueba
        palette.add_block('boton',
                          style='basic-style-larg',
                          label=_('boton'),
                          prim_name='boton',
                          help_string=_('boton de prueba'))
        self._parent.lc.def_prim('boton', 1,
                                 lambda self, valor: primitive_dictionary['boton'](valor))

    def _boton_prueba(self, valor):
        print "el valor del boton ", valor
```