

Introducción a Python, Pilas y Videojuegos (II)

En el capítulo anterior...



¡Mi cabezaaaa....!

función

variable

actor

objeto

... dejamos a nuestros intrépidos futuros programadores perdiendo la cabeza con tantos nuevos términos. Y es que lo son, claro. No en vano, así como en España utilizamos el término *ordenador*, en Latinoamérica se utiliza más el de *computadora*, ya que nuestros dispositivos hacen eso, computar, computar muchos datos distintos.

Y hablando de datos, ¿recuerdas que ha hemos usado? ¿Recuerdas nuestra brillante suma?

```
Type "help", "copyright"
>>> 3 + 5
8
>>>
```

Esos **3**, **5** y **8** son **datos numéricos**. Sí, sí, no te reprimas, abre una ventana del intérprete de Pilas y prueba allí también. ¡Anda, remolón...!

¡0 remolona!

Eso.



Bueno, a parte de números, puedes utilizar **textos**. Para escribirlos, debes encerrarlos entre dobles comillas “ ” o entre comillas simples ‘ ’, como tú prefieras. Y ya que estamos, hagamos algo divertido; ¿qué pasará si **sumamos** dos textos?...

```
>>> "Me gustan" + "las bicicletas"
'Me gustanlas bicicletas'
>>>
```



No pongas esa cara de sorpresa, que no es para tanto. De hecho, acaban de ocurrir varias cosas interesantes. Fíjate:

- *Sumar* dos textos lo que produce de resultado es otro texto que no es más que la unión de los dos, el segundo justo a continuación del primero. Eso se conoce como **concatenación**. Pero observa que, a los humanos, lo que nos llama la atención es que no hay un espacio en medio y sale como una sola palabra *gustanlas*. No te mosquees. **Python** ha hecho lo que le has pedido. Como simplemente une los dos pedazos de texto que le has dado, si quieres que separe *gustan* de *las*, debes dejarle un espacio en blanco *a posta*, en el texto. **¡Los espacios en blanco también importan!** Mira la diferencia:

¡Ah, amigo!

Aquí está el culpable

```
>> "Me gustan " + "las biciletas"  
'Me gustan las biciletas'  
>> |
```

Pero que conste que también
podríamos haberlo puesto aquí



Sí, sí, pero la
próxima vez pon "bicicletas" y no
"biciletas"

- También habrás visto que tú has escrito los textos con dobles comillas y **Python** te ha devuelto el texto con comillas simples. ¡No importa! Como hemos dicho, se pueden usar indistintamente. Hay más usos y más matices, pero por el momento, pueden esperar.
- El **intérprete de Pilas** colorea lo que escribes a medida que lo reconoce. ¿Te has fijado que los números y el signo más los pone en rojo? ¿Y que el texto entrecomillado lo pone en verde? Cool... ¿De qué color pondrá otros elementos, otros tipos de datos?

¡Un momento!

¿Hay más tipos de datos?

Tipos de Datos en Python



Ya conoces dos: los **números enteros** (a este tipo **Python** lo llama **int**) y las **cadenas de texto** (en este caso, **string**). Aquí tienes unos ejemplos de otros tipos típicos que incorpora **Python**:

Números decimales o de coma flotante (*float*)

```
>> 3.36  
>> 2.5e3
```

Notación científica

Significa $2.5 \times 10^3 = 2500$

Listas (*list*)

```
>> [3, 'hola', 22.5]
```

Tuplas (*tuple*)

```
>> ('hola', 'adios')
```

Varios elementos
separados por comas

Diccionarios (*dict*)

```
>> {'a':35, 'b':13}
```

Varios elementos
"etiquetados"

Booleanos (*bool*)

```
>> True  
>> False
```

Valores que significan
"verdadero" o "falso"



Pero... ¿Para qué necesito un valor que represente algo que sea 'verdadero' o que sea 'falso'? Humm....

Para empezar, como información. Prueba estas cuentas en el intérprete de Pilas y observa lo que obtienes:

```
» 3 > 5
False
» 5 + 2 == 7
True
```

¡Ojo! Son dos símbolos "igual". En Python se expresan así las comparaciones

```
» 14574 % 3 == 0
True
```

Python nos acaba de decir que 14574 es un múltiplo de 3...

... ya que ésta es la forma con la que Python calcula el resto de hacer la división.

Pero la existencia de los valores booleanos **True** y **False** nos permite hacer lo anterior más bonito. Prueba a hacer esto último de la siguiente manera:

Pulsa 'intro' cuando acabes de escribir esta línea

Python deja este espacio automáticamente

```
» if 14574 % 3 == 0:
..     print "14574 es divisible por 3"
..
14574 es divisible por 3
»
```

Deberás pulsar 'intro' dos veces para que ejecute el código

¿Qué ha ocurrido aquí? Lo que queríamos hacer es decirle al ordenador que

Si 14574 es divisible por 3...
...escribe que es divisible por 3

o, de forma más general, algo que siga el esquema

Si se cumple cierta condición...
...haz lo que te digo aquí

En lenguaje **Python**, ese **bloque** condicional se escribe con la instrucción **if** ('sí', en inglés, como es lógico, al provenir de un lenguaje del mundo anglosajón):

if se cumple cierta condición:
haz lo que te digo aquí

Veámoslo de nuevo en el ejemplo concreto anterior:

```
if 14574 % 3 == 0:  
    print "14574 es divisible por 3"
```



Fíjate en el
sangrado del código para
indicar qué es lo que está
relacionado con el **if**

¡Y también en los dos
puntos que indican que lo
deseado viene a continuación!

De hecho, los bloques **if** pueden ser más completos. Por ejemplo, podemos incluir una tarea para realizar en el caso de que no se cumpla la condición utilizando la instrucción **else**. Fíjate, de nuevo, en la importancia de los dos puntos y el sangrado del código:

```
>> if 65477 % 3 == 0:
..     print "65477 es divisible por 3"
.. else:
..     print "65477 no es divisible por 3"
..
65477 no es divisible por 3
>>
```

Habrás tenido que borrar varios espacios (en concreto **4 espacios**) para poder dejar el **else** con el sangrado adecuado, ¿verdad? Por cierto, muchas veces verás que se usa el término **indentado** en lugar de **sangrado**. En ambos casos nos referimos a lo mismo.



Humm... Vale. Pero resulta molesto escribir todo el código cada vez que quiero probar con un número distinto...
¿No?

... Y no hablemos de tener que escribir el mismo número varias veces...

Sí, claro. ¿Y si quiero saber ahora si 4503 es también divisible por 3? ¿Y 3400748? ¿Y 13028476? ¿Tengo que escribirlo todo de nuevo por cada número? ¿Cada vez?...

Nuestro amigo **Python** proporciona una forma de agilizar todo esto, un tipo de **bloque** llamado **función**. Escribe lo siguiente en el **intérprete de Pilas**:

```
» def divisibilidad_por_tres(numero):  
..     if numero % 3 == 0:  
..         print "Es divisible por 3"  
..     else:  
..         print "No es divisible por 3"  
..  
..  
»
```

Este sangrado no es automático. Acuérdate; debes poner 4 espacios para indicar el contenido del bloque

Con la instrucción **def** conseguimos **definir** un bloque de código (una **función**) que **Python** ejecutará cada vez que escribas su nombre, en este caso, *divisibilidad_por_tres*. Haz la prueba:

```
» divisibilidad_por_tres(4503)  
Es divisible por 3  
» divisibilidad_por_tres(3400748)  
No es divisible por 3  
» divisibilidad_por_tres(13028476)  
No es divisible por 3
```

Python ejecuta cada vez el código del bloque usando como *numero* el valor que pones entre paréntesis

Funciones (y otros bichos raros)

Sí, un poco bicho raro sí te parecerá. A fin de cuentas, el nombre

`divisibilidad_por_tres`

se las trae. Y es que podemos elegir el nombre que queramos para una **función** pero con ciertas condiciones. Y las mismas consideraciones sirven para las **variables** (algo de ello viste en el tutorial anterior) y para cualquier tipo de **objeto** que maneja **Python**.



Estos nombres no deben contener espacios (lo que es lógico pues **Python** no sabría distinguir si es un solo nombre o son dos) ni deben comenzar por un número ni usar caracteres no anglosajones (como la ñ) o especiales, con alguna excepción (como la `_`). Suele ser aconsejable utilizar nombres muy descriptivos que nos ayuden a recordar para qué sirven. Y como no podemos usar espacios, se suele acudir a la `_` o a mayúsculas para separar las palabras. Mira un par de ejemplos más:

`escribe_un_cuento`

`colorDeseado`

¡Ten cuidado! Python distingue entre mayúsculas y minúsculas, así que

`colorDeseado`

y

`colordeseado`

son dos nombres válidos diferentes.

Además, las **funciones** tienen un poco más de rareza, pues su nombre ha de incluir los paréntesis, lo que permite que les pasemos valores para que sean usados en el código que contienen. Y sí, decimos **valores**, en plural, por que podemos pasarle varios (en lenguaje técnico, se les llama **argumentos** o **parámetros**).

Un ejemplo; intenta entender cómo funciona y qué hace la siguiente función:

```
def divisibilidad(numero1, numero2):  
    if numero1 % numero2 == 0:  
        print str(numero1) + " es divisible por " + str(numero2)  
    else:  
        print str(numero1) + " no es divisible por " + str(numero2)
```

Prueba a ponerla en práctica utilizando diferentes números.

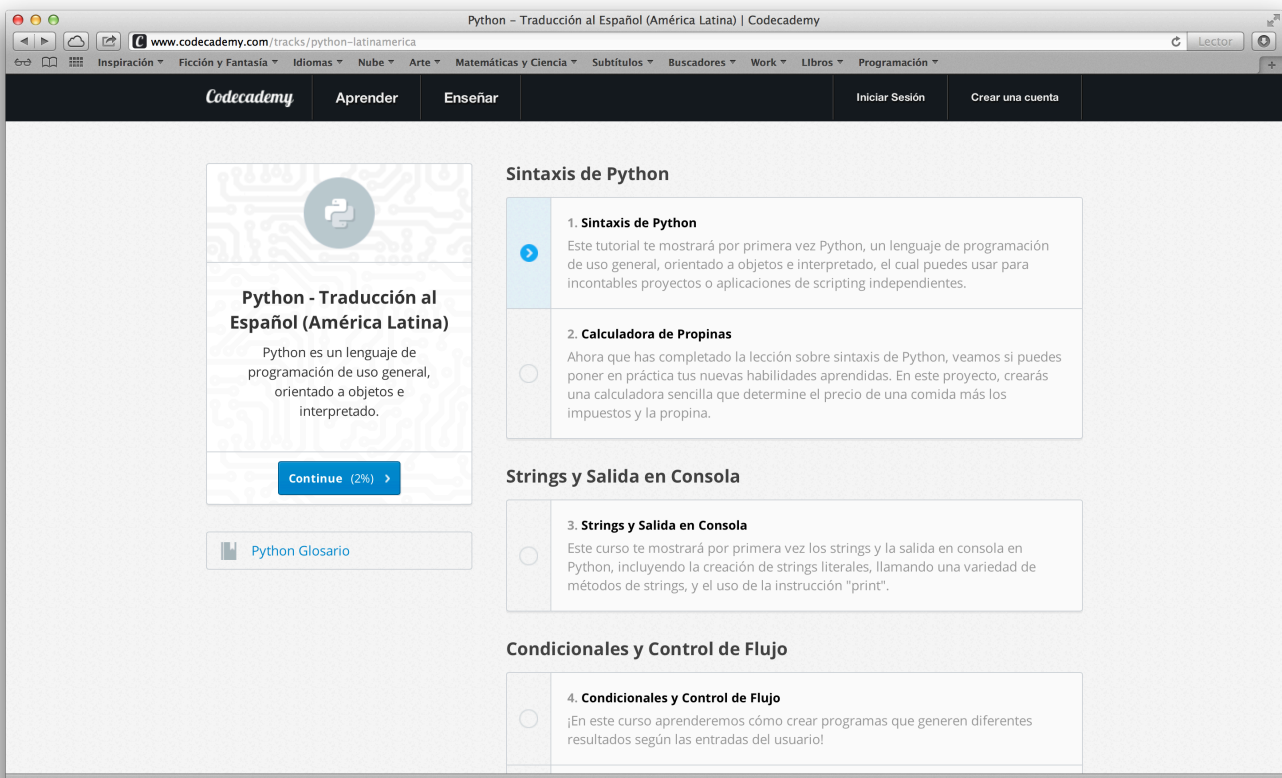
¿Se te escapa algo?



¡Para profundizar más!

Hay una web en la que puedes comprender éste y otros temas, de forma interactiva. Se trata de

[Python - Traducción al Español \(América Latina\) | Codecademy](https://www.codecademy.com/tracks/python-latnamerica)



¿A qué esperas?
¡Regístrate!

Si vas realizando el tutorial de **CodeAcademy** irás aprendiendo la **sintaxis de Python**, familiarizándote con las **funciones**, las **variables**, otros tipos de **bloques de código** y los diferentes **tipos de datos**, incluyendo lo que son las **clases** y su materialización como **objetos**. Sí, sí, ¿recuerdas que nombramos en el tutorial anterior a los **objetos** y cómo acceder a sus elementos con la **notación dot** (la de escribir detrás del nombre un punto y, a continuación, el nombre del elemento)?

Pues la noticia es que...

... esto... sí...

En Python **TODO** es un objeto

¿Todo?

TODO

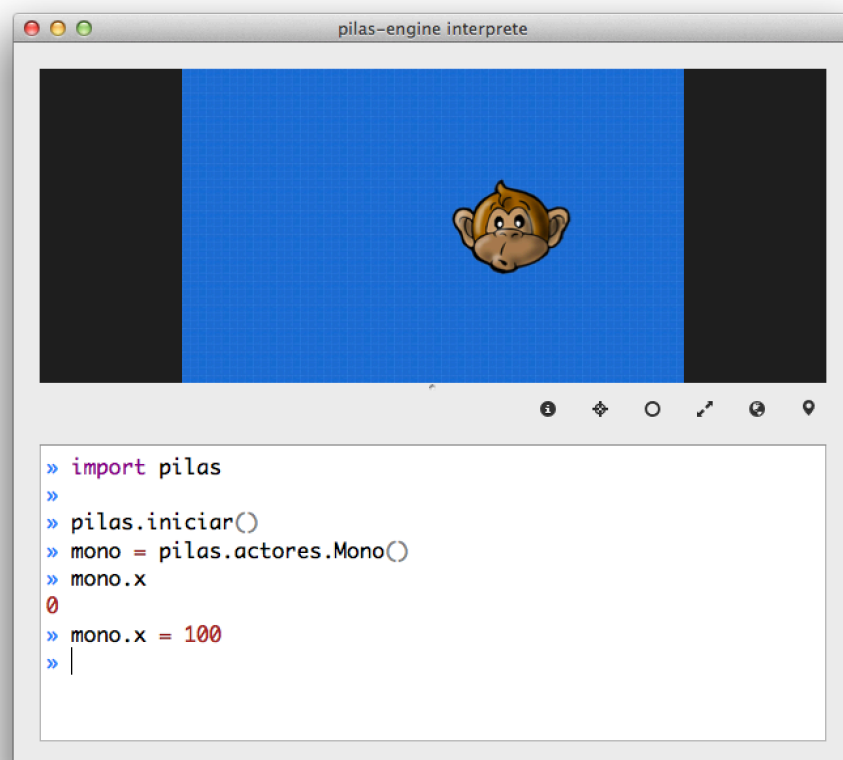
Que los **módulos**, como **Pilas**, son objetos lo puedes ver con facilidad. Fíjate que para iniciarlo lo que hacemos (lo pone el **intérprete de Pilas** de forma automática) es llamar a un elemento suyo, la función **iniciar()**, con la **notación dot**:

```
» import pilas
»
» pilas.iniciar()
» mono = pilas.actores.Mono()
»
```

Pero fíjate en la siguiente línea que es más divertida todavía. ¡Aparecen objetos en cadena! Lo que hacemos, aparentemente, es llamar a la **función Mono()** que es un elemento del **objeto actores** que, a su vez, es un elemento del **objeto pilas**. Y al **objeto resultante**, lo hemos llamado **mono**, para poder referirnos a él más adelante (es lo que se llama una **variable**). Y sí, es un objeto a su vez. Por ejemplo, tiene un elemento, una **propiedad**, llamada **x** que nos da la coordenada x de su posición en pantalla. Pruébalo:

```
>> mono.x
0
```

¡Incluso la puedes cambiar y verás que, automáticamente, nuestro objeto **mono** cambia de posición en la ventana gráfica!

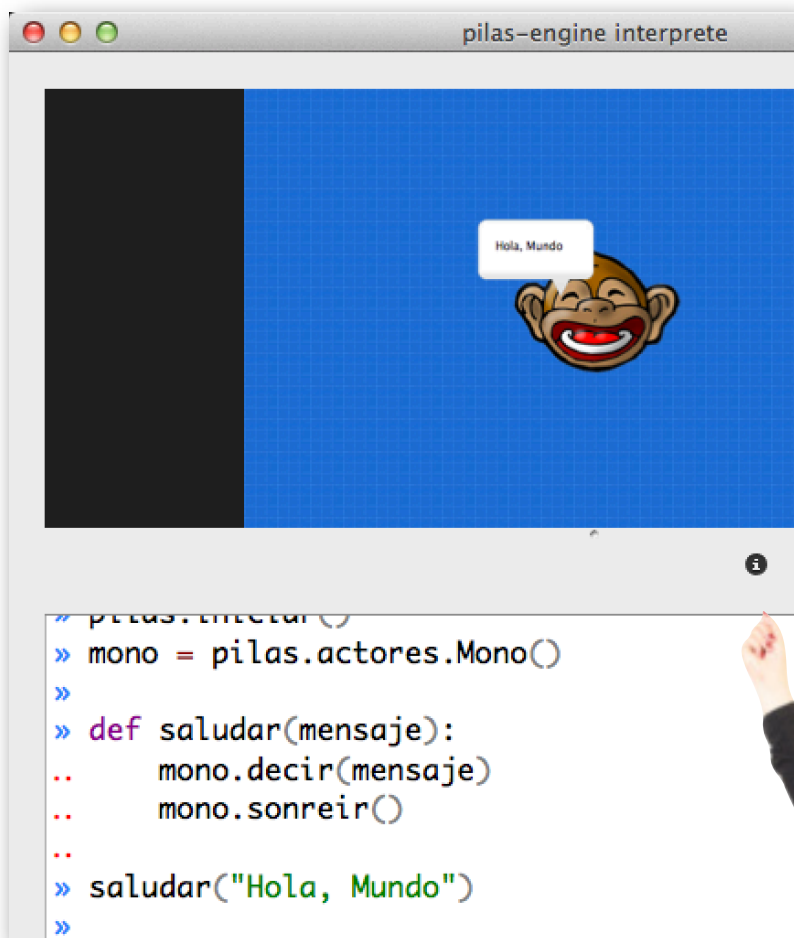


Y ¿puedo hacer cualquier cosa?

Bueno, cualquier cosa que sea viable hacer con los elementos definidos en los objetos. Nuestro **mono**, como ejemplo, posee también dos funciones que le hacen, respectivamente, hablar y reír. ¿Qué te parece si definimos nosotros mismos una función que las use para saludar, con el texto que deseemos? Sea:

```
» def saludar(mensaje):  
..     mono.decir(mensaje)  
..     mono.sonreir()  
..
```

Lo has adivinado. La función **decir()** hace que el mono hable y la función **sonreir()** hace que nos alegre la cara. ¿La probamos? Ejecuta la función con el texto “**Hola, Mundo**”:



Como ves, todos los elementos que maneja **Python** los puedes usar y mezclar para realizar tareas mucho más complicadas que realizar, simplemente, una suma.

¿Te convences?



Con un poco de suerte, ya habrás empezado a hacerte una buena idea de lo que son los objetos y asumirás que, de nuevo, en Python todo son objetos. ¿No lo crees? Las humildes **cadenas de texto**, por ejemplo, encierran mucho más de lo que parece. Tienen, entre otras, una **función miembro** que se encarga de convertir todas sus letras a mayúsculas (la función miembro **upper()**). Sí, sí, como lo oyes. La demostración:

```
>>> "Los textos son objetos".upper()
'LOS TEXTOS SON OBJETOS'
>>>
```

Si quieres hacer algo de magia y averiguar qué esconde un objeto, puedes probar con la función de Python **dir()**. No te asustes con el resultado.

```
>>> dir("Los textos son objetos")
['__add__', '__class__', '__contains__', '__delattr__',
['__doc__', '__eq__', '__format__', '__ge__',
['__getattr__', '__getitem__', '__getnewargs__',
['__getslice__', '__gt__', '__hash__', '__init__', '__le__',
['__len__', '__lt__', '__mod__', '__mul__', '__ne__',
['__new__', '__reduce__', '__reduce_ex__', '__repr__',
['__rmod__', '__rmul__', '__setattr__', '__sizeof__',
['__str__', '__subclasshook__',
'_formatter_field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>>
```

¿Te suena?

No se vayan
todavía...
¡Aún hay más!



Nos falta sólo la recta final para terminar este segundo tutorial y tenerlo todo listo para pasar a la acción y comenzar a hacer nuestros pinitos.

El **intérprete de Pilas** es una herramienta genial, pero necesitamos ayuda cuando se trata de escribir mucho código y cuando queremos interrumpir nuestra tarea y continuar más adelante. Afortunadamente, todo está pensado y, tanto **Python** como **Pilas** disponen de métodos para **ejecutar archivos de texto que contengan código válido**.

En el caso de **Pilas**, la forma es muy sencilla; **basta arrastrar y soltar el documento de texto que contiene el código sobre la ventana de Pilas**.

Hagamos la prueba. Coge un editor de texto cualquiera y escribe el código de nuestro mono sonriente. Presta atención a una cosa; como ahora no estás en el **Intérprete de Pilas**, éste no hace nada automático, así que tienes que añadir las líneas que aparecían por defecto. Además, tienes que añadir también otra línea indicando a Pilas que está todo listo y que debe ejecutar con su engine las instrucciones que le hemos dado.

```
import pilas  
pilas.iniciar()  
mono = pilas.actores.Mono()  
  
def saludar(mensaje):  
    mono.decir(mensaje)  
    mono.sonreir()  
  
saludar("Hola, Mundo")  
pilas.ejecutar()
```

Estas líneas son las que ponía antes
Pilas automáticamente

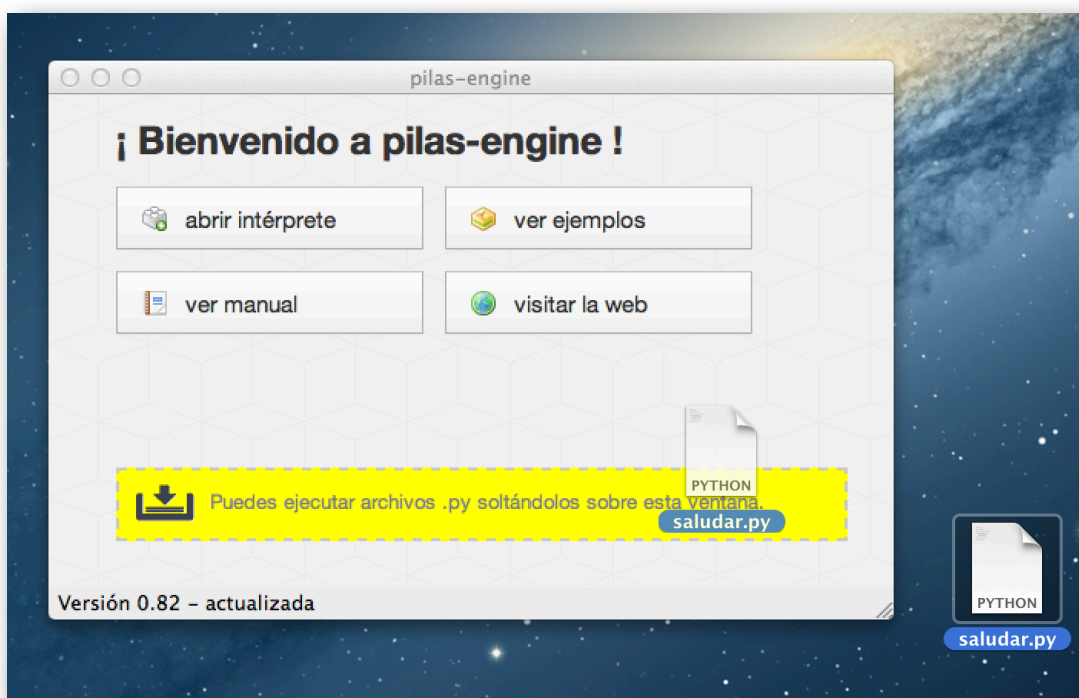
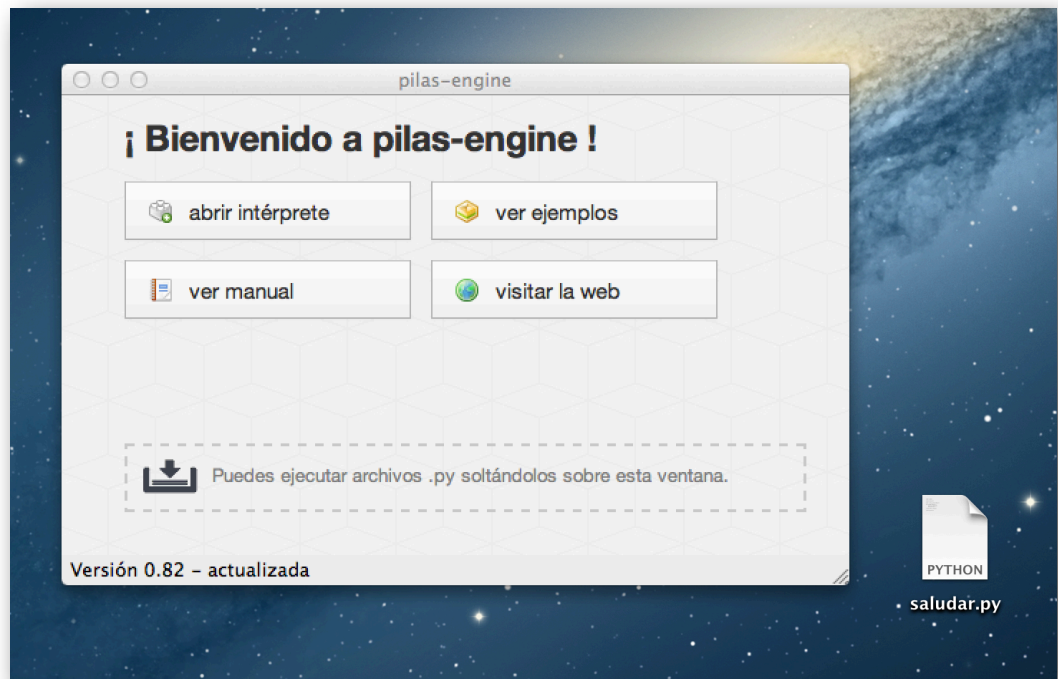
Ésta es la línea nueva que debes
añadir

Éste es el contenido del archivo de texto que
has de escribir. Guárdalo con un nombre
terminado en .py, por ejemplo, *saludar.py*

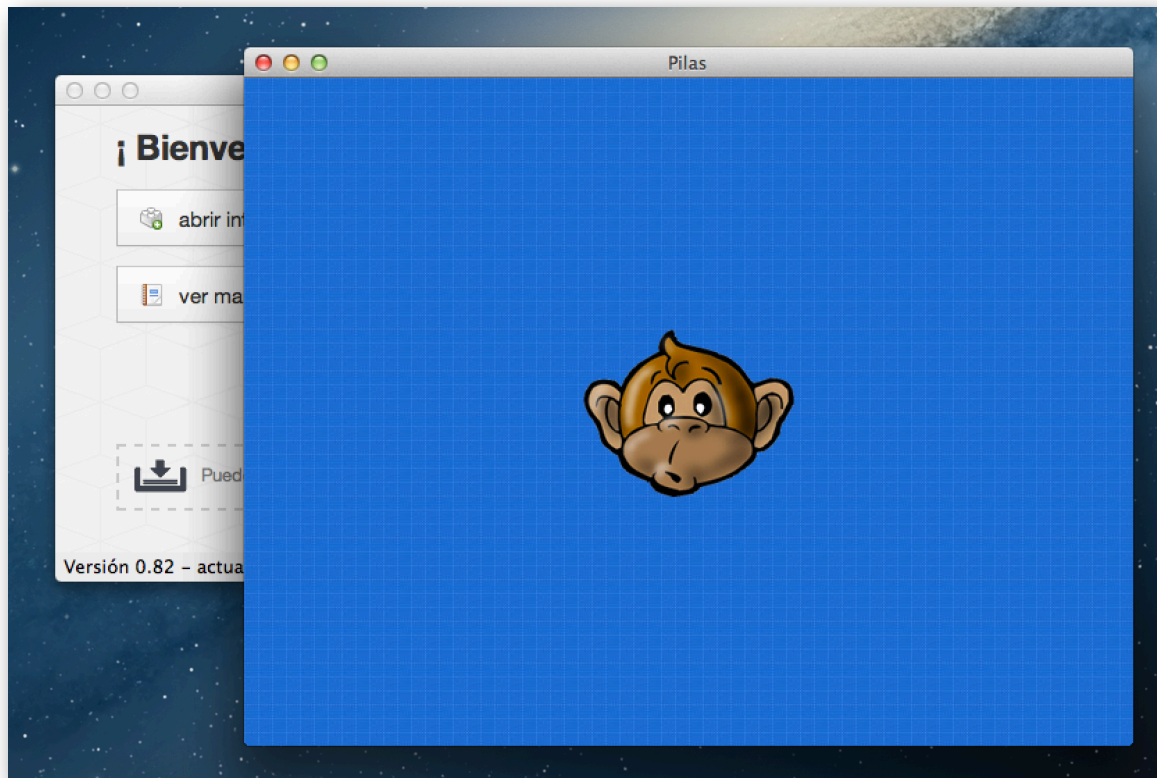
Guardar tu código **con un nombre terminado en .py** no es solo una buena idea, es una costumbre muy extendida entre los programadores **Python** (los *pythonistas*). Es un convenio muy útil que permite identificar rápidamente si un archivo contiene código **Python**.

¡Un aviso! Asegúrate que has guardado el archivo en formato de **texto plano**, no vaya a ser que tu editor de texto te lo guarde con información de tipos de letra o estilos de texto. Nosotros, de cara al código, no necesitamos nada de eso.

Bueno, ahora ya puedes lanzar Pilas y arrastrar tu archivo sobre su ventana. Verás que Pilas la detecta...

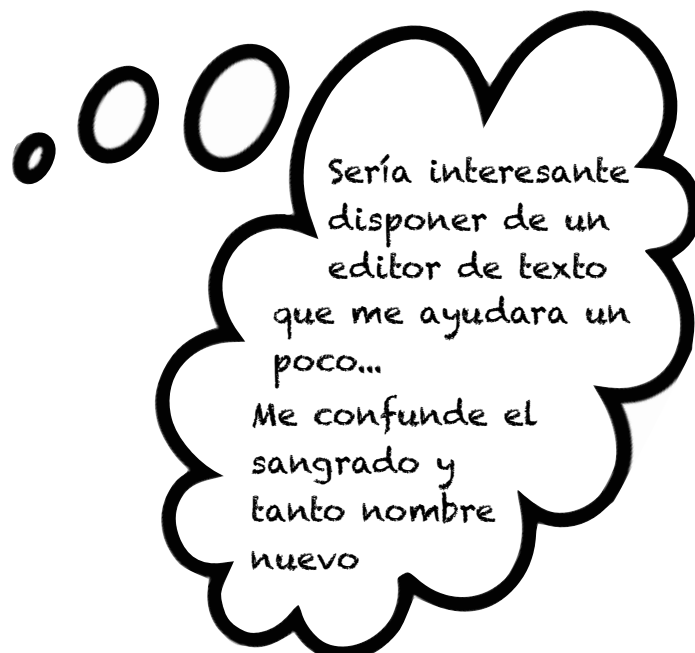


Suéltala y... ¡alehop!



Todo está en orden. Sabes cómo escribir un archivo de código y como ejecutarlo. Sólo te falta práctica y aprender las características de Pilas, poco a poco, para realizar tu videojuego.

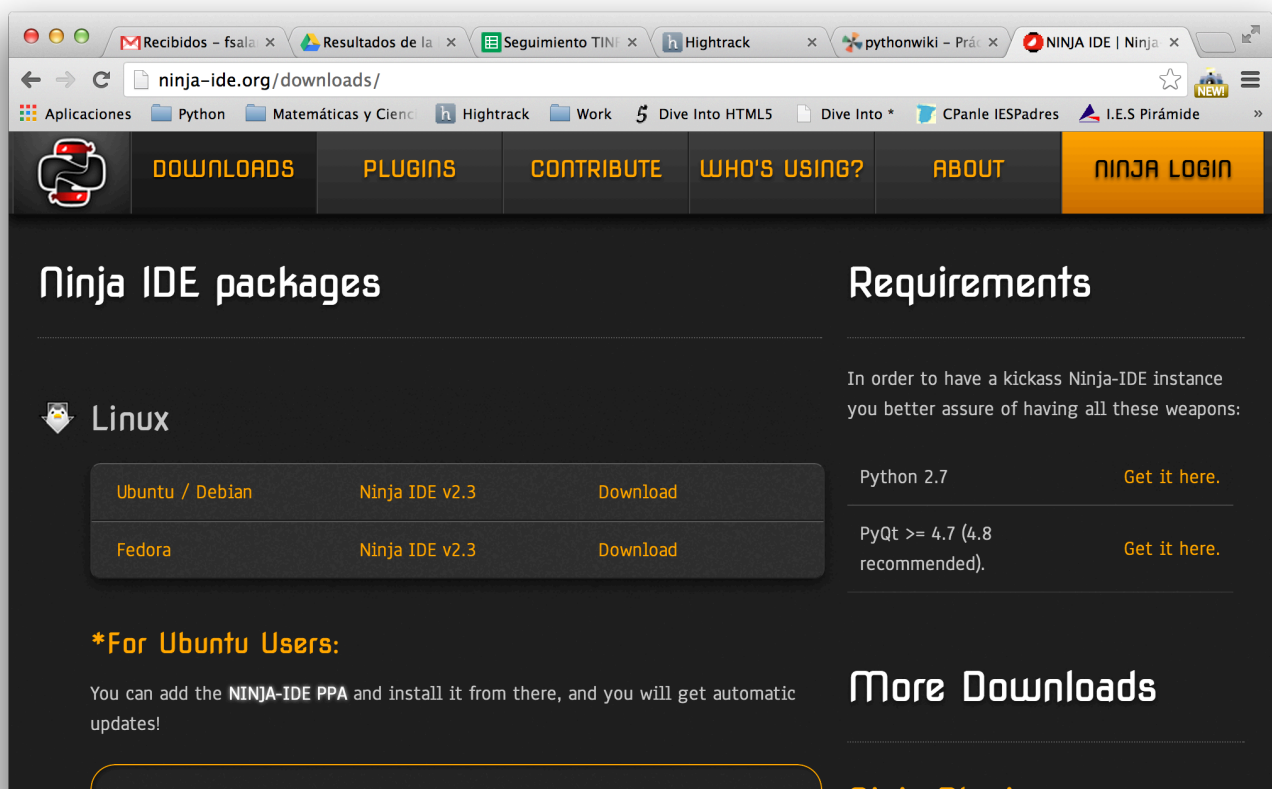
Pero, espera, ¿te inquieta algo?



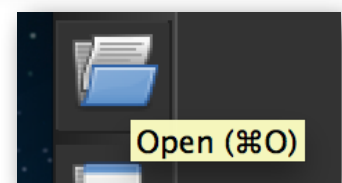


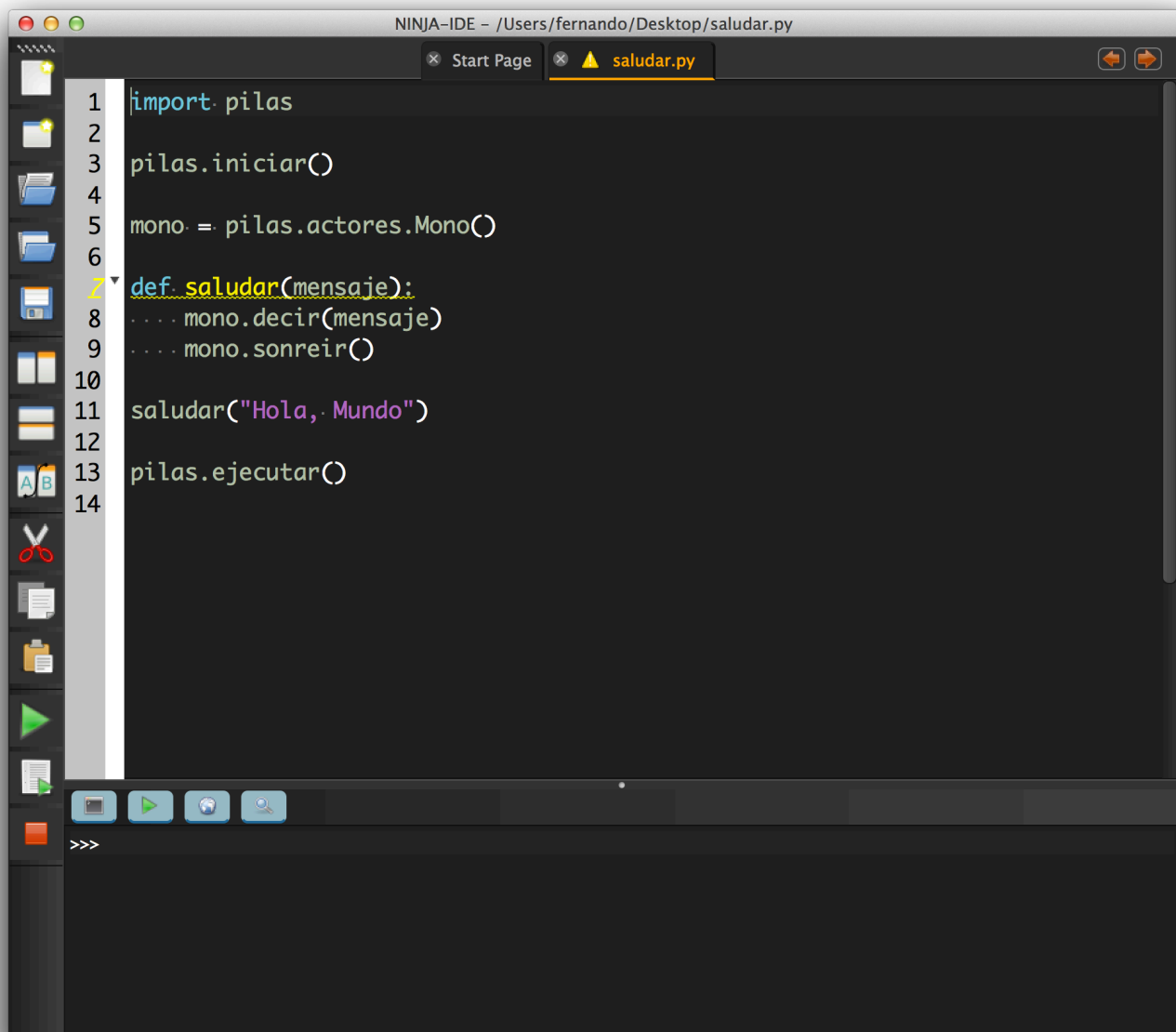
Bien, no hay que preocuparse puesto que existen muchos **editores de código**. Aunque puedes usar, por supuesto, el que prefieras, vamos a sugerirte uno: **Ninja-IDE**. Para obtenerlo, dirígete al apartado de descargas su web

ninja-ide.org

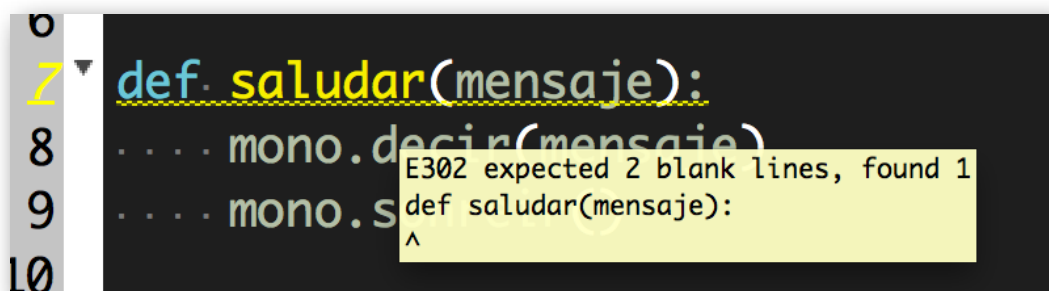


Una vez instalado, prueba a abrir nuestro código **saludar.py** desde el menú o desde el icono correspondiente de la barra lateral de **Ninja-IDE** (si pasas el ratón sobre ellos y esperas un momento, verás una ayuda emergente).





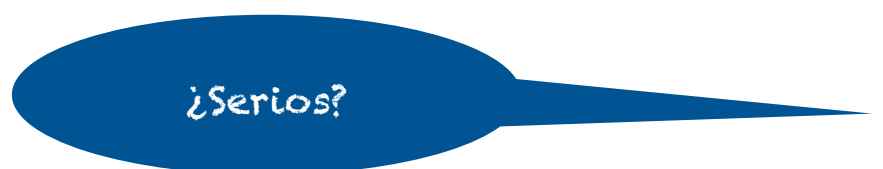
Tiene mejor pinta, ¿verdad? **Ninja-IDE** colorea la sintaxis de **Python**, coloca los 4 espacios en blanco cuando pulsas la tecla tabulador e incluso te hace sugerencias de estilo. Puedes ver una en la línea 7, marcada por un subrayado amarillo. Acerca, de nuevo, el cursor del ratón a esa línea para leer la ayuda emergente:



Ninja-IDE te sugiere, por claridad, que dejes dos líneas en blanco antes de la definición de la **función saludar()**. Si lo corriges verás que desaparece la línea amarilla...



Vaale. Pero a partir de ahora nos pondremos serios



... serios y manos a la obra. ¡Tu videojuego está en camino!

