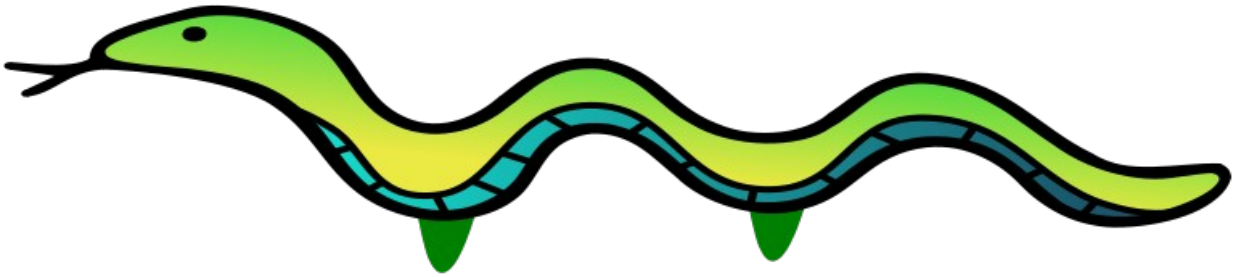
INTRODUCCIÓN A PYTHON



Los programas deben escribirse para que los lean las personas, y sólo de forma circunstancial para que los ejecuten las máquinas.

— Abelson y Sussman, Estructura e Interpretación de Programas de Computadora

Introducción a Python by Ángel Luis García García is licensed under a Creative Commons Reconocimiento 3.0 España License.

Se permite la reproducción, distribución y comunicación pública de la obra así como la generación de obras derivadas, incluso con fines comerciales.

Autor: Ángel Luis García García (angelluis78@gmail.com)
Versión del documento: 0.0.2
Versión Python: 2.6.6
Fecha: 25/02/2011

Antes de nada

Este manual pretende introducir a cualquier persona, que haya programado en cualquier otro lenguaje, en el desarrollo de software con **Python**. Solo es una iniciación, por lo que se dejan conceptos en el camino, ya que de lo contrario se titularía “TODO sobre Python”.

Inspirado en un nido de horas estudiando y programando en **Python**, este texto está redactado para ser leído (y probado) de manera secuencial, introduciendo progresivamente los conceptos fundamentales del lenguaje en los diversos temas. Puesto que no es lo mismo andar el camino que conocer el camino, se insta al lector a que pruebe todos los ejemplos que vienen en el documento. Y es que para aprender a desarrollar “en cualquier lenguaje de programación” hay que embarrarse las manos, no hay otra forma.

ÍNDICE

0. Introducción

Python es un lenguaje de programación diseñado por el holandés **Guido van Rossum**, a finales de los 80. Aunque Guido tiene la última palabra en cuestiones referentes a fijación de directrices y decisiones finales sobre **Python** (de hecho, a Guido se le conoce como ***el benevolente dictador vitalicio***), este último es administrado y desarrollado por la **Python Software Foundation** (PSF), teniendo licencia de código abierto (PSFL) compatible con GPL de GNU a partir de la versión 2.1.1. La PSF tiene además, como objetivo, fomentar el desarrollo de la comunidad **Python** (es una organización sin ánimo de lucro, creada en 2001).

Guido trabaja desde 2005 en Google (Mountain View). Más información en http://www.python.org/~guido/.

Cuando se introducen características nuevas en el lenguaje nos encontramos con las **PEP**, que son las siglas en inglés de **Python Enhancement Proposal**, esto es, en español **propuesta de mejora de Python**. Un PEP es un documento de diseño que suministra información a la comunidad Python ó bien describe una nueva característica en Python. Por tanto un PEP sirve para proporcionar una especificación técnica precisa de una característica y su justificación.

El sitio web de Python es www.python.org.

Python se usa en programación de sistemas, cálculo numérico, desarrollo web, software para dispositivos móviles (Symbian, Android), desarrollo de aplicaciones de escritorio, educación, simulación, prototipados, GIS y un largo etcétera.

De las empresas más destacadas que utilizan Python se pueden nombrar Google, NASA, Facebook, US National Weather Service, Corel, Lockheed Martin, Pixar, Industrial Light and Magic, etc.

Como software desarrollado en Python cabe resaltar Google App Engine (entorno Cloud Computing de Google), OpenERP, The Washington Post (Python + Django), Facebook (la parte de gestión de mensajes e información en tiempo real), Zope (servidor de aplicaciones), Plone (CMS [sistema de gestión de contenidos] utilizado por la NASA, Canonical, etc), BitTorrent, etc.
Más casos de éxito en http://python.org/about/success/.

1. Características principales de Python

Python es un lenguaje de programación de propósito general, de muy alto nivel (esto es, un alto nivel de abstracción, con el uso de listas, tuplas, diccionarios).

Python es un lenguaje interpretado (no es necesaria compilación), dinámico (no necesita identificar explícitamente los tipos de datos para inicializar variables, de modo que los tipos se validan durante la ejecución del programa) y fuertemente tipado (no pueden mezclarse tipos, es necesario hacer conversiones).

Python es un lenguaje multiplataforma (Windows, Mac, Linux, etc), multiparadigma (imperativo, orientado a objetos y en menos medida funcional) y con gestión automática de memoria.

Por último cabe destacar que Python es un lenguaje de programación con una sintaxis clara y sencilla, fácil de aprender, donde se pueden mezclar los diferentes paradigmas de programación de los que dispone, ampliamente documentado, extensible, que intenta obligar al desarrollador de software a programar de la manera correcta en el menor tiempo posible.

1.1. Implementaciones en Python

CPython ó (o Python) es la implementación de referencia. Interpreta, “compila”, y contiene módulos codificados en C estándar.

Jython es la implementación para la JVM (Máquina Virtual Java) accediendo a las bibliotecas de Java.

IronPython es la implementación para el CRL de Microsoft, con el propósito de acceder al framework de .NET.

PyPy es la implementación de Python escrita en Python (para rizar más el rizo).

1.2. Herramientas para desarrollar en Python

Para programar en Python lo único que se necesita es el intérprete de Python, (se puede obtener de www.python.org) y el/los fichero/s de código fuente Python a ejecutar. Por otra parte tenemos ciertas herramientas de desarrollo para utilizar con Python, a saber:

IDE: Son las siglas de entorno de desarrollo integrado. Es una aplicación (entorno de programación) para desarrollar software que está compuesto normalmente por un editor de código, un compilador ó un intérprete, herramientas auxiliares de desarrollo (creación de documentación, tests, empaquetado de software), un depurador y algunas veces un diseñador de interfaces gráficas incorporado. Se pueden nombrar *IDLE, Eclipse con el plugin pyDev, Aptana, NetBeans, Geany, pyScripter, Ninja IDE, Stani's Python Editor, Wingware Python IDE, Komodo, Pyragua, Eric.*

GUI: Son las siglas de interfaz gráfica de usuario. Son herramientas para crear interfaces gráficas, esto es, componentes gráficos con los cuales el usuario interacciona con la aplicación. Hay algunos IDE's que contienen GUI's integrados. Las GUI's dependen de la plataforma gráfica en la que trabajemos. Por ejemplo, para wxPython tenemos *wxDesigner, wxFormBuilder ó wxGlade. Para GTK, tenemos Glade, Qt dispone de Monkey Studio, etc.*

Editor: Es una aplicación para escribir código de programación en el lenguaje deseado. Contiene todas las herramientas de edición necesarias, y algunas veces características propias de IDE's. Hay veces que cuesta distinguir entre un IDE y un editor por la cantidad de opciones que tiene este último. Por ejemplo *Editra, emacs, Notepad++, SciTE*.

RAD: Son las siglas de desarrollo rápido de aplicaciones. Dependiendo del autor sirve para designar a las aplicaciones de desarrollo de interfaces gráficas ó a los IDE's con GUI's integrados, o a las dos cosas. Es un término dado a plataformas de desarrollo como PowerBuilder, Visual Studio ó Delphi, por poner ejemplos. De los más destacados para Python tenemos *BOA Constructor* y *SharpDevelop* (para IronPython).

Shell interactivo: Es una aplicación que mejora sustancialmente el intérprete Python original, esto es, es un intérprete con características especiales, que podrían incluir la completitud de código y el coloreado del mismo, navegación entre los namespaces, exportación de código, etc. Por ejemplo *PyCrust, Dreampie, PyShell*, etc.

Lo mínimo que se le debe de pedir a cualquier **IDE** ó **editor**, para desarrollar en **Python**, es la **indentación automática**, **coloreado** y **completitud de código**.

Más información sobre herramientas para desarrollar en Python en http://wiki.python.org/moin/PythonEditors.

1.3. Las versiones de Python

Actualmente en Python existen dos versiones activas, la 2.XX y la 3.XX. En el sitio web de Python existe un guión de ayuda para elegir entre la 2 y la 3:

http://wiki.python.org/moin/Python2orPython3

A diciembre de 2010 había estables las versiones de 2.6, 2.7 y en desarrollo la 3.2. El futuro es Python 3, el cual es incompatible con las versiones 2.XX.

Entonces, ¿cuál elegir?. Pues depende de nuestras exigencias, y de los módulos de extensión (aquellos que no están en la distribución de Python) que utilicemos (frameworks y demás). La mayoría siguen siendo compatibles con 2.XX pero no con 3 (aunque esto está cambiando). Este manual está hecho pensando en Python 2.6.6.

1.4. Composición de Python

A grandes rasgos podemos ver de qué está compuesta una aplicación Python:

PYTHON: **LENGUAJE + BIBLIOTECA ESTÁNDAR** + *MÓDULOS DE EXTENSIÓN*

El lenguaje y la biblioteca estándar conforman la instalación de base.

El lenguaje se compone de palabras clave (if, for, etc), funciones integradas (abs, print, etc), tipos básicos (números, secuencias, diccionarios, conjuntos, cadenas, ficheros, ...) y las reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones.

NOTA: Una función integrada es aquella no es necesaria llamarla desde ningún módulo, pues se puede acceder a ella por defecto, esto es, se carga automáticamente.

La biblioteca estándar es el conjunto de módulos, paquetes y frameworks que están en la instalación base de Python y que hay que declarar de manera explícita en el código. Por ejemplo, para acceder a las funciones del sistema operativo en el que se encuentre nuestra instalación Python invocaremos al módulo **os**.

Los módulos de extensión, que pueden ser frameworks, módulos ó paquetes, es software de terceros, que hay que instalar por separado. Más información sobre módulos de extensión en http://pypi.python.org/pypi. **PyPI** es el índice de paquetes de Python, esto es, un repositorio de software para el lenguaje de programación Python.

1.5. Filosofía Python

Python está planteado para desarrollar software a partir de unos principios de programación. Los usuarios de Python se refieren a menudo a la **Filosofía Python** que es bastante análoga a la filosofía de Unix. El código que sigue los principios de Python de legibilidad y transparencia se dice que es "**pythonico**". Contrariamente, el código opaco u ofuscado es bautizado como "no pythonico" ("unpythonic" en inglés). Estos principios fueron famosamente descritos por el desarrollador de Python **Tim Peters** en **El Zen de Python**:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Ralo es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Los errores nunca deberían dejarse pasar silenciosamente.
- A menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechazar la tentación de adivinar.
- Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea Holandés.
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ya.
- Si la implementación es dificil de explicar, es una mala idea.
- Si la implementacion es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!.

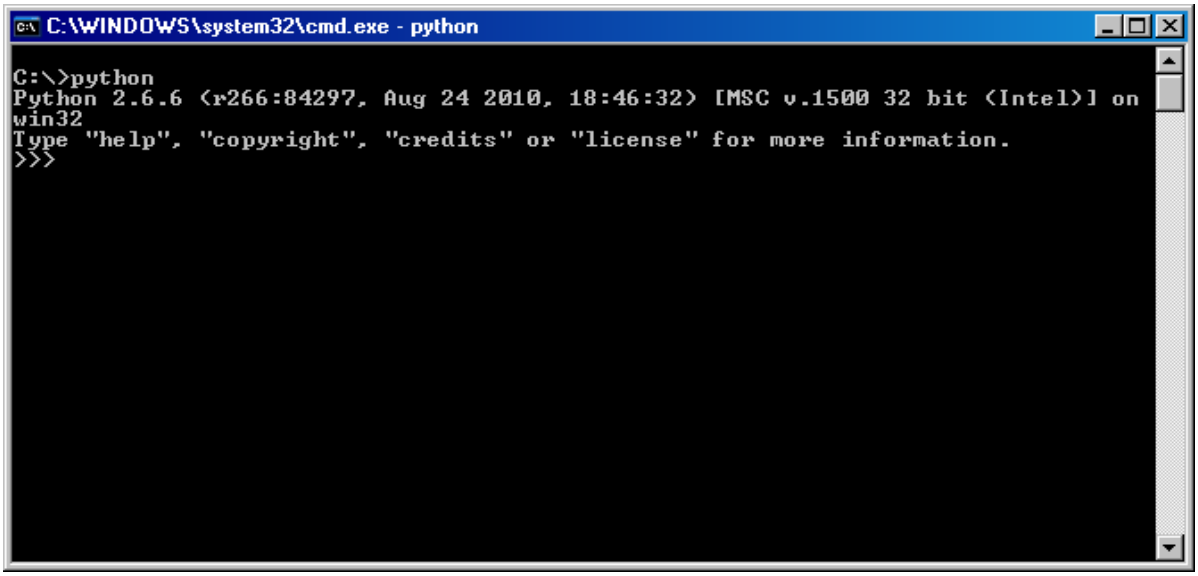
Estos principios son casi axiomáticos en Python, implementados además en el intérprete Python. Para visualizarlos ejecutar el intérprete Python y escribir:

```
import this
```

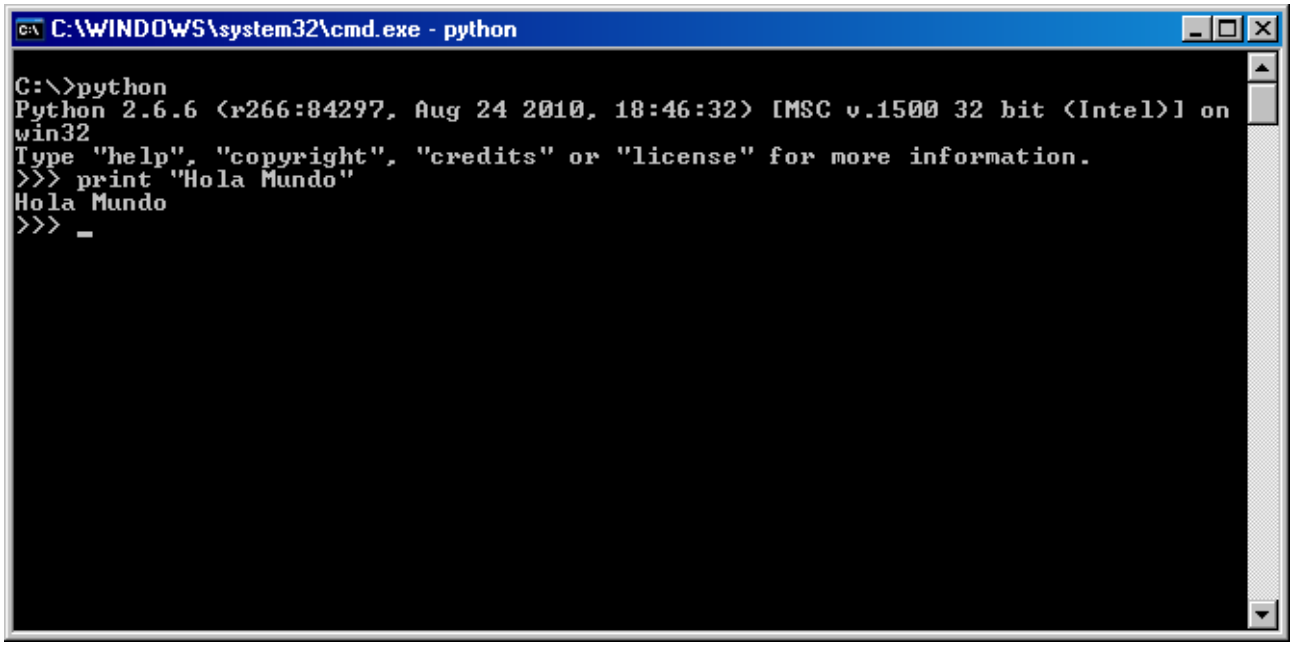
2. El intérprete de Python

Python, como se ha comentado, viene de serie en la mayoría de distribuciones Linux. Para Windows hay que descargarlo de su sitio web. Convendría para este último sistema incluir en la variable de entorno *path* la ruta donde se tenga instalado Python (por ejemplo c:\python26, si se tiene instalada la versión 2.6 en la unidad C).

Ejecutamos el intérprete mediante el comando **python**.



Escribimos nuestro primer “Hola mundo” con la función integrada **print**. Darse cuenta que estamos visualizando una cadena, mediante comillas dobles. También es válido las comillas simples.



Podemos realizar operaciones matemáticas in situ:

```
>>> 3 + 4
7
>>>
```

Asignar valores a una variable cualquiera (al ser tipado dinámico, no es necesario declarar el tipo de variable). Podemos visualizarlo con print.

```
>>> a = 57
>>> print a
57
>>>
```

Podemos volver a asignar un valor a dicha variable, con otro tipo diferente.

```
>>> a = 'Aprendiz de programador Python'
>>> print a
Aprendiz de programador Python
>>>
```

No podemos meclar tipos si no es haciendo conversiones. Intentaremos sumar un número a una cadena.

```
>>> 'Hola mundo' + 4
Traceback (most recent call last):
  File “(stdin”, line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

El mensaje nos dice que hay error de tipo (TypeError), no pudiendo concatenar un tipo str (string ó cadena) con otro tipo int (entero). Esto se debe a que Python es un lenguaje fuertemente tipado. Bien, podemos solucionarlo haciendo una conversión ó casting del número 4 para convertirlo a una cadena con la función integrada str.

```
>>> 'Hola mundo' + str('4')
'Hola mundo4'
>>>
```

Pedimos información al usuario mediante la función integrada **raw_input**.

```
>>> a = raw_input(“Introduce tu nombre: ”)
Introduce tu nombre: angel
>>> print a
angel
>>>
```

Ahora vamos a cargar un módulo (librería) de la biblioteca estándar para obtener información de la máquina en la que nos encontramos:

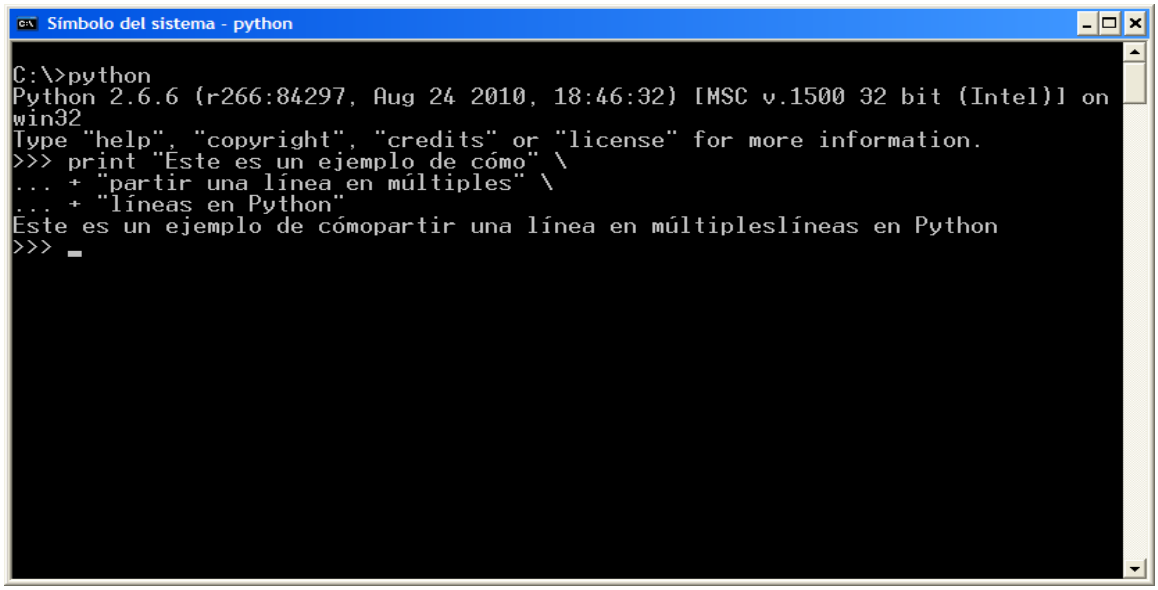
```
>>> import sys
>>> print sys.platform
```

2.1. Identificadores y declaraciones

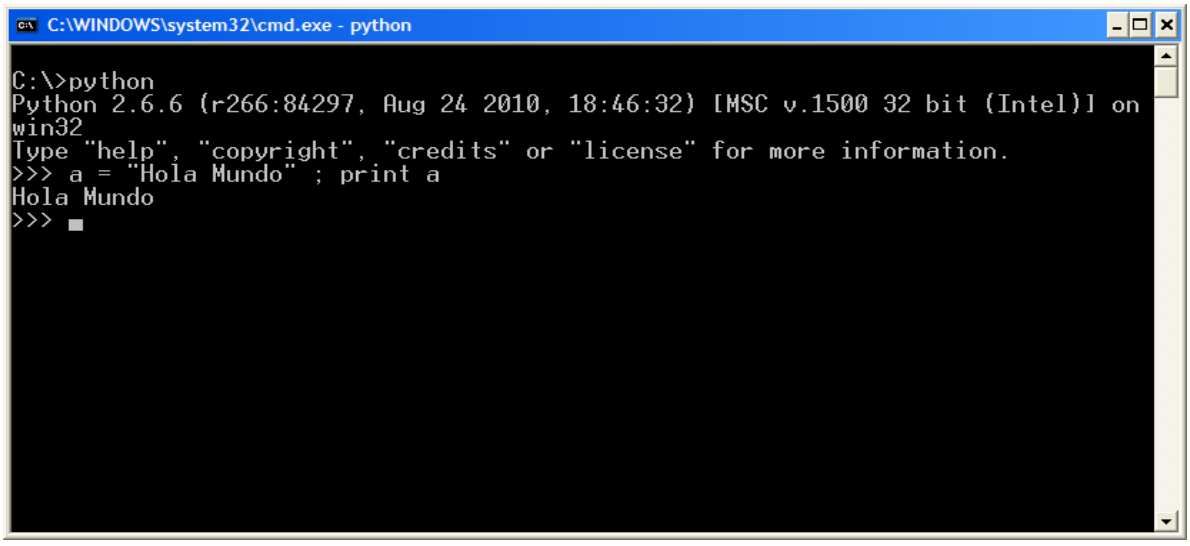
Los identificadores son nombres únicos que sirven para distinguir algo, utilizándose para designar variables, funciones, clases y objetos. Comienzan por una letra ó un guión bajo, pudiendo contener letras, guiones bajos ó dígitos, pero nunca signos de puntuación. Además no pueden existir identificadores que contengan los caracteres **$**, **?** ó **@** (solo es válido dentro de cadenas).

Además cabe descatar que los identificadores, palabras clave, atributos, etcétera, en Python son **case-sensitive**, esto es, dintinguen entre mayúsculas y minúsculas.

La declaración es la unidad básica de programación. En Python, una declaración debe de estar toda en una única línea. Por ejemplo, print “Hola Mundo”. Para “partir” una línea en múltiples líneas, se utiliza ****. Hay una excepción a esto último, y es que siempre se puede “partir” dentro de una tupla, lista ó diccionario, y en cadenas con triple comillas, sin necesidad de utilizar ****.

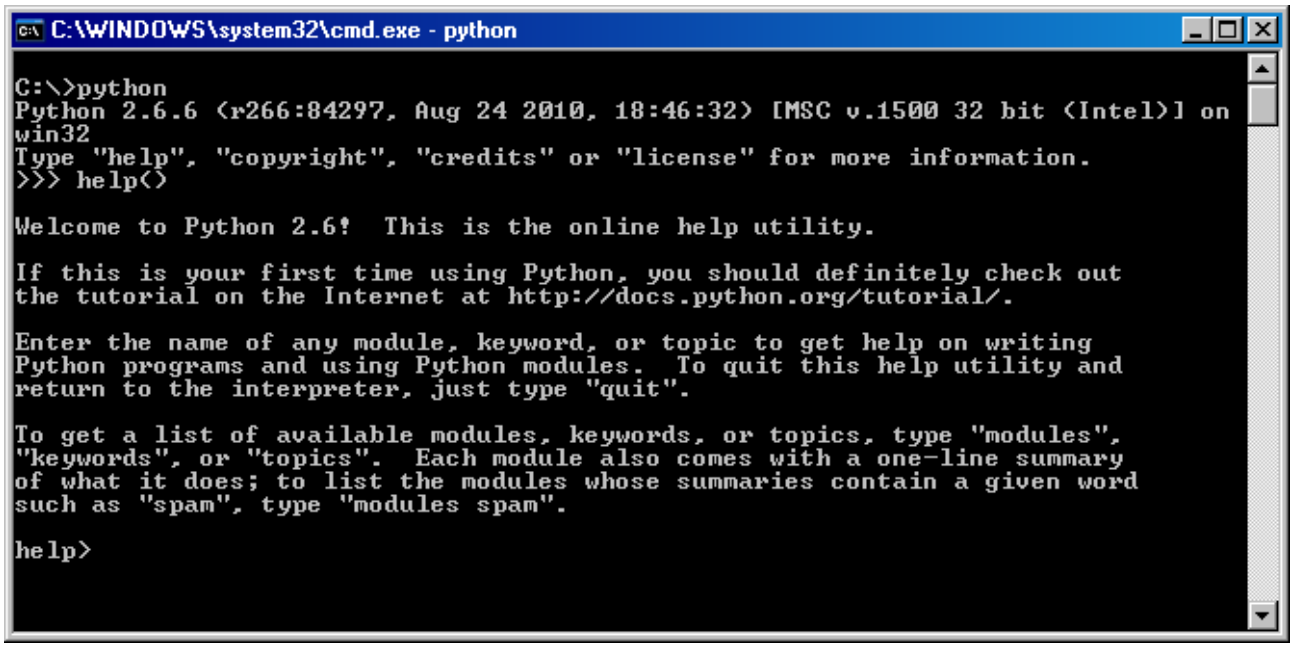


Por último comentar en este apartado que más de una declaración puede aparecer en una misma línea si están separadas por punto y coma **;**.

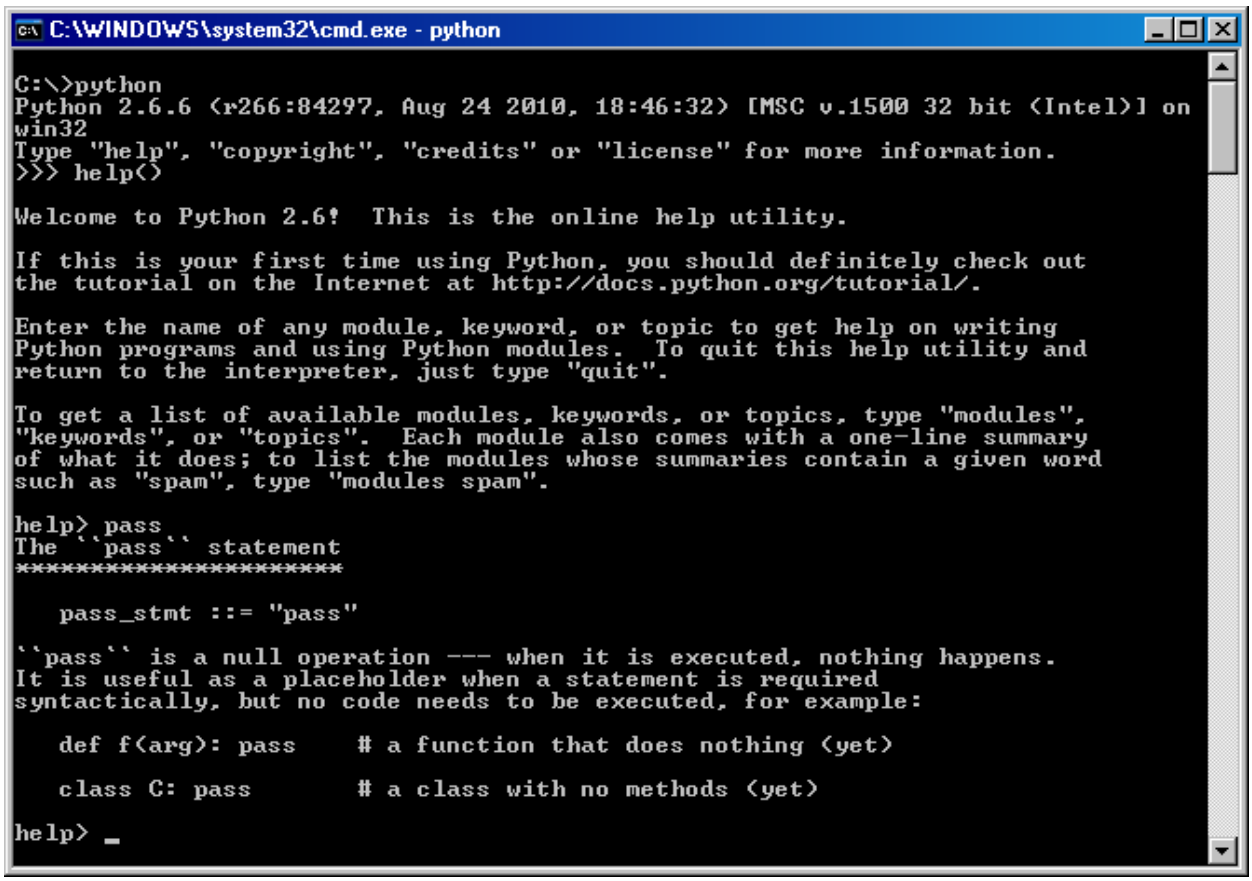


2.2 Ayuda

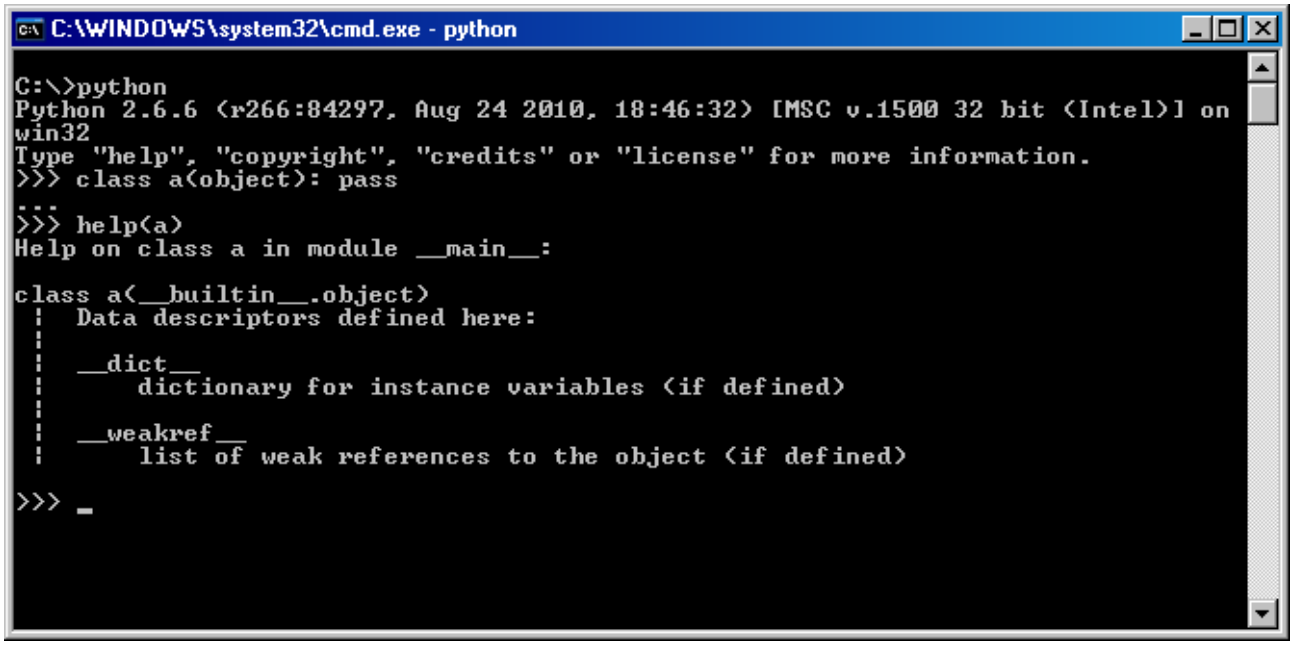
Si queremos ó necesitamos ayuda podemos obtenerla del intérprete simplemente tecleando **help()**:



A continuación podemos insertar la palabra de la cual queremos información. Por ejemplo, si quiero saber para qué sirve la palabra clave **pass**, únicamente escribo pass y pulsar *Enter*:



Si quiero salir de la ayuda, pulsar *Enter* sin introducir ningún texto. También se puede utilizar help(objeto) ó help("nombre"). En el ejemplo anterior podríamos haber utilizado *help("pass")*. Si creamos un clase llamada *a* y queremos obtener información sobre ella:



Para salir del intérprete utilizar **quit()**.

>>> quit()

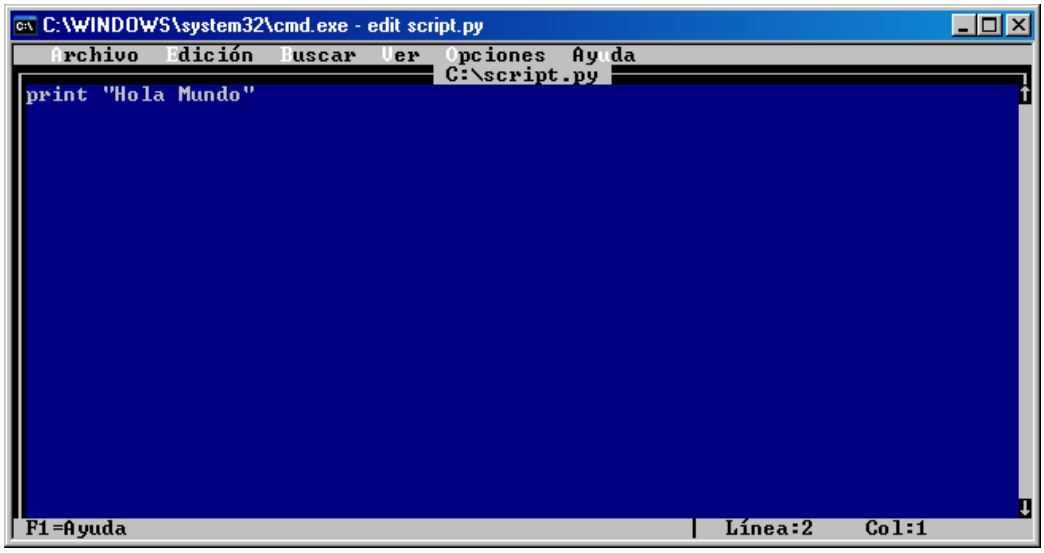
C:\>

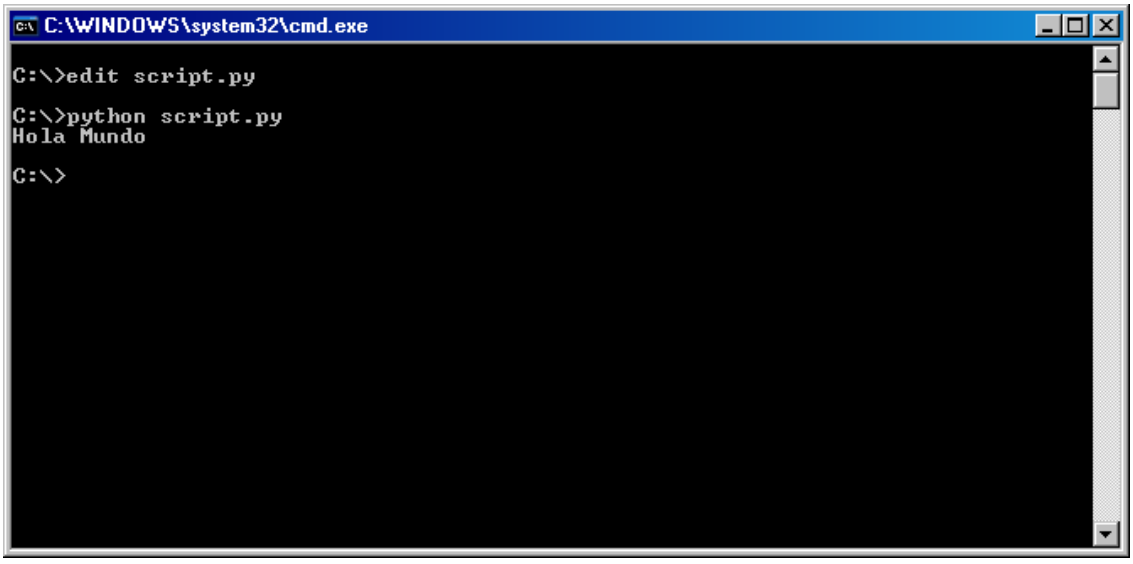
El ejecutar únicamente el intérprete de Python se utiliza para probar trozos de código, buscar ayuda, probar estructuras, expresiones regulares, comprobar que se han instalado correctamente módulos y algunas otras cosas. Evidentemente para escribir programas necesitamos otro tipo de operativa, que es lo que veremos a continuación.

3. Mi primer fichero con código fuente Python

El código a interpretar por Python puede ejecutarse dentro del propio intérprete ó en ficheros de texto plano con extensión **.py**. Evidentemente nuestras aplicaciones Python se guardarán en ficheros con dicha extensión. Podemos crear varios ficheros .py de manera que tengamos nuestro código ordenado según la jerarquía que más nos convenga. Crear librerías con funciones y clases es una buena idea para el buen mantenimiento del código en nuestras aplicaciones. Para trabajar con ficheros de código fuente es necesario un editor de texto plano. Se pueden utilizar editores, ide's ó rad's. Si se está empezando en Python lo mejor es utilizar un editor normal (Gedit, Bloc de notas, etc) ó como mucho el ide IDLE, que se puede encontrar en las distribuciones de Windows de manera predeterminada, en Mac y Linux. Una vez que se tenga experiencia ó se necesiten herramientas más potentes es cuando habría que dar el paso con otras alternativas de desarrollo anteriormente comentadas.

Crear un fichero con código Python es trivial. Creamos un fichero con extensión .py (*script.py*) que contenga el código para mostrar un “Hola Mundo”. Ejecutamos dicho script con el comando python script.py.





4. Unicode

Unicode es un sistema para representar caracteres de todos los diferentes idiomas del mundo.

Antes de Unicode había diferentes sistemas de codificación de caracteres para cada idioma, cada uno usando los mismos números (0-255) para representar los caracteres de ese lenguaje. El problema radicaba cuando se quería intercambiar documentos entre sistemas heterogéneos, restultando difícil, puesto que no había manera de que un ordenador supiera con certeza qué esquema de codificación de caracteres había usado el autor del documento, donde los números pueden significar muchas cosas (en japonés el carácter codificado 234 no es el mismo que el carácter codificado ruso 234, ni en español, que 234 es Û).

Para resolver este problema Unicode representa cada carácter como un número de 2 bytes (de 0 a 65535). Cada número de 2 bytes representa un único carácter utilizado en al menos un idioma del mundo (los caracteres que se usan en más de un idioma tienen el mismo código numérico). Hay exactamente un número por carácter, y exactamente un carácter por número. Los datos de Unicode nunca son ambiguos.

Evidentemente, los sistemas de codificación antiguos siguen existiendo. Los caracteres ingleses se codifican en ASCII de 7 bits (0 a 127). Idiomas europeos occidentales como el español ó el francés utilizan el ISO-8859-1 (ó Latin-1), que usa un ASCII extendido (0 a 255), para representar caracteres especiales, como la ñ, ó las tildes (á,é,í,ó,ú),por ejemplo. Unicode usa los mismos caracteres que el ASCII de 7 bits para los números de 0 a 127, y los mismos caracteres que el ISO-8859-1, del 128 a 255, y de ahí en adelante se extiende para otros lenguajes que usan el resto de los números del 256 al 65535.

La verdadera ventaja de Unicode reside en su capacidad de almacenar caracteres que no son ASCII, como la ñ española. Para crear una cadena Unicode en lugar de una ASCII normal solo hay que añadir la letra **u** antes de la cadena. Por ejemplo:

```
>>> x= u'angel'
>>> print str(x)
angel
>>>
```

Python normalmente convierte el Unicode a ASCII cuando necesita hacer una cadena normal partiendo de una Unicode. En este ejemplo, la función **str** intenta convertir una cadena Unicode en ASCII para poder imprimirla a continuación con print. ¿Y si hacemos x = u'ángel'?

```
>>> x = u'ángel'
>>> print str(x)
Traceback (most recent call last):
```

File ("stdin)", line 1, in (module)
UnicodeEncodeError: 'ascii' codec can´t encode character u'\xe1' in position 0: ordinal not in range(128)
>>>

Vemos que da error, puesto que la cadena Unicode *ángel* contiene carácteres (la á) que no son ASCII. Es por ello que Python se queja produciendo un error UnicodeDecodeError.

Esta conversión que la función str (lo hacen más funciones en Python, y en muchos más casos) realiza de Unicode a ASCII se puede modificar, ahorrándonos errores del tipo:

UnicodeDecodeError: 'ascii' codec can't decode byte 0x@@@ in position @@: ordinal not in range(128)

Lo que podemos hacer es decirle a Python que cuando se encuentre en el caso de que tenga que realizar una conversión, en vez de hacerlo a ASCII de 7 bits, lo haga al iso-8859-1, que es el que contempla el idioma español. ¿Cómo se hace esto?

Hay que crear un fichero, llamado **sitecustomize.py**. Este fichero puede estar en cualquier parte, siempre que el **import** pueda encontrarlo, pero normalmente se encuentra en el lib/site-packages de Python. Si por ejemplo se utiliza Python 2.5, este fichero debería de crearse en c:\Python25\Lib\site-packages.

El contenido sería el siguiente:

```
import sys
sys.setdefaultencoding('iso-8859-1')
```

Python intentará importar sitecustomize.py cada vez que arranque, de manera que ejecutará automáticamente cualquier código que incluya. La función setdefaultencoding establece la codificación por omisión. Éste es el esquema de codificación que Python intentará usar cada vez que necesite convertir automáticamente una cadena Unicode a una normal.

La codificación por omisión sólo se puede cambiar durante el inicio de Python; no se puede hacer más adelante. Es más, estando en el intérprete de Python no puede ejecutarse la sentencia sys.setdefaultencoding('iso-8859-1'), ya que nos dirá que no existe este atributo.

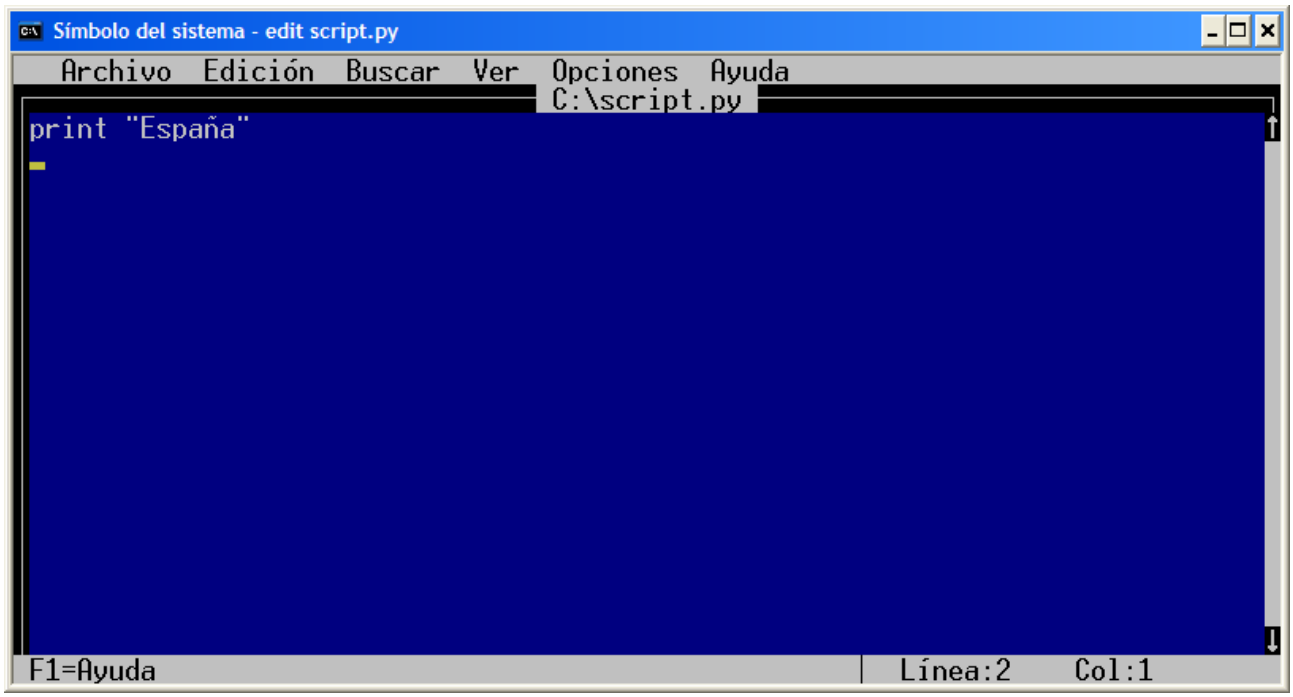
Ahora que el esquema de codificación por omisión incluye todos los caracteres que usa en la cadena, Python no tiene problemas en autoconvertir la cadena e imprimirla.

```
>>> x = u'ángel'
>>> print str(x)
ángel
```

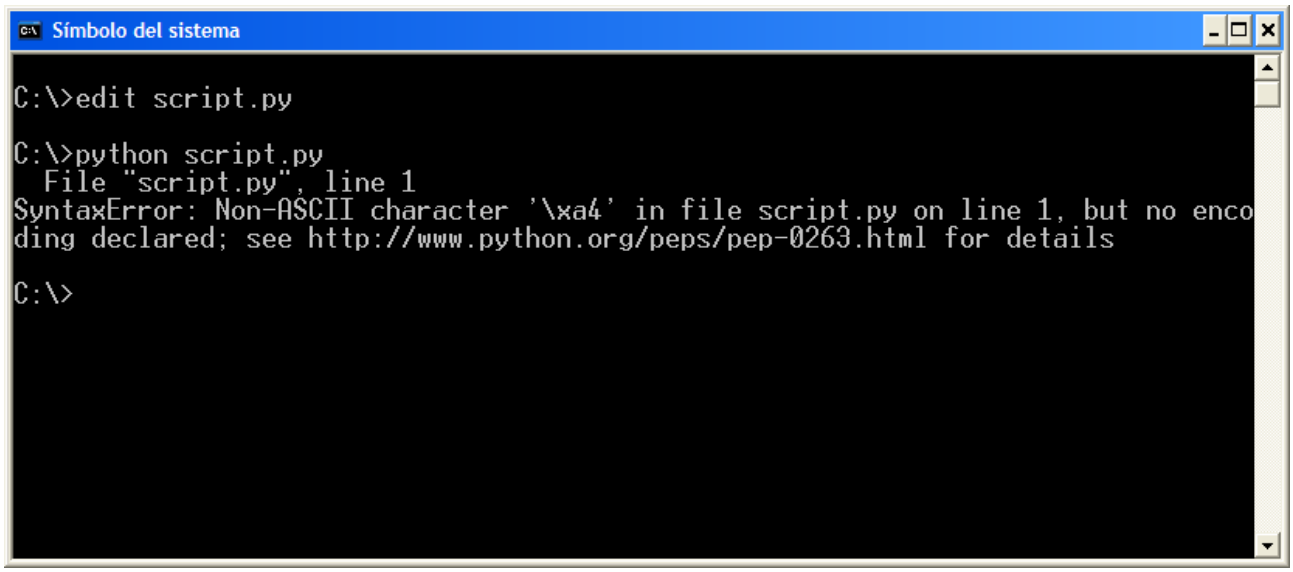
4.1 Scripts Python en Unicode

En Python el nombre de variables, funciones ó clases se deben de escribir en codificación ASCII, pero hay caracteres no ASCII que se pueden incluir en ficheros fuente, como comentarios ó cadenas de caracteres (entre comillas).

Imaginamos que creamos un script del siguiente tipo:



Al ejecutarlo nos aparece lo siguiente:



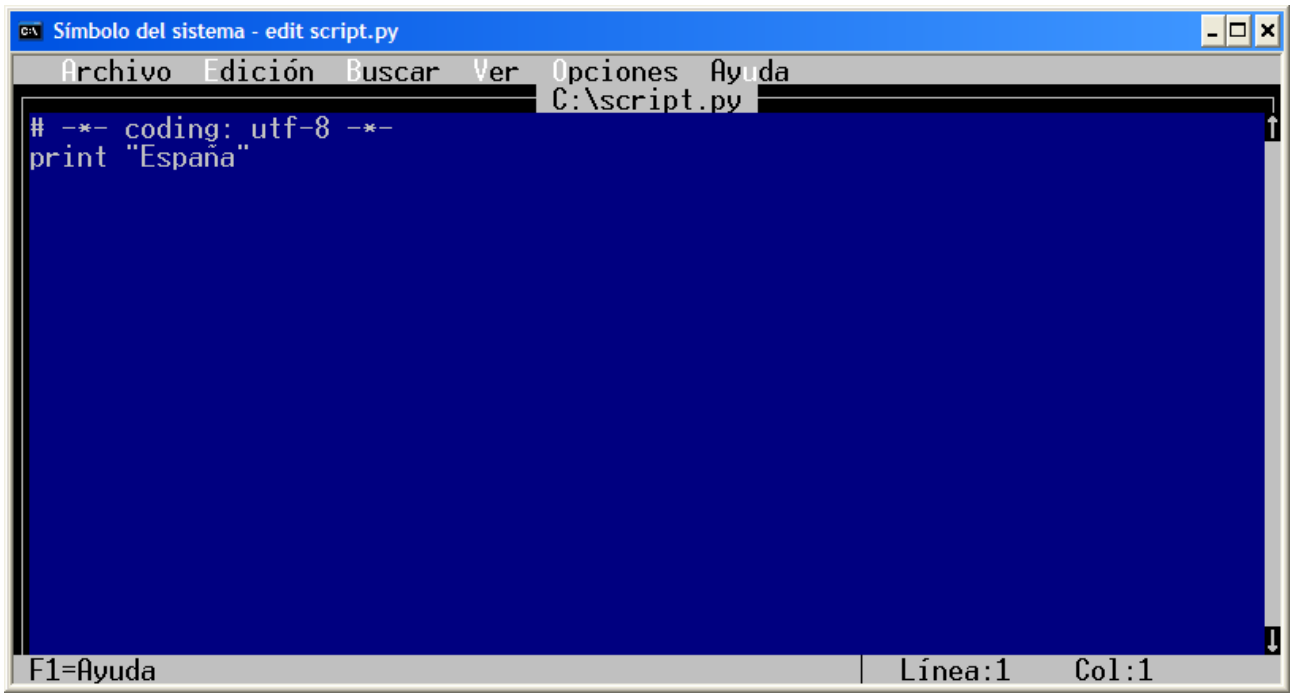
El carácter *\xa4* corresponde a la *ñ* española, el cual no pertenece a ASCII.

Para que funcione correctamente Python necesita saber que el script incluido

en el fichero no es ASCII. La manera más sencilla es colocar un comentario especial en la primera ó segunda línea del script, tal que así:

 # -*- coding: utf-8 -*-

Si volvemos a probar el código con la modificación, todo funciona como se esperaba, ya que le estamos diciendo a Python que interprete el script del fichero en Unicode.



Resultando:

C:\>python script.py
España

C:\>

Una cabecera de módulo típica es:

#!/usr/bin/env python
-*- coding: utf-8 -*-

5. Funciones en Python

En Python se pueden utilizar funciones, como en la mayoría de los lenguajes de programación. Su sintaxis es la siguiente:

def nombre_funcion**(**parámetros**):**
    <...código...>

Indentación de código

Para definir bloques de código en Python se utiliza la indentación (sangría) de código, en contra de otros lenguajes, como C, que utilizan las llaves {} para enmarcar un bloque. La identación tiene que ser uniforme desde el principio, así que si desde un primer momento se utilizan 4 espacios en blanco por ejemplo, han de usarse siempre, ya que de lo contrario el intérprete de Python nos lanzará un error de indentación de código. Algo importante a destacar sobre este tema es no mezclar sangrías de espacios en blanco con el carácter tabulador (TAB), ya que podría dar error de indentación.

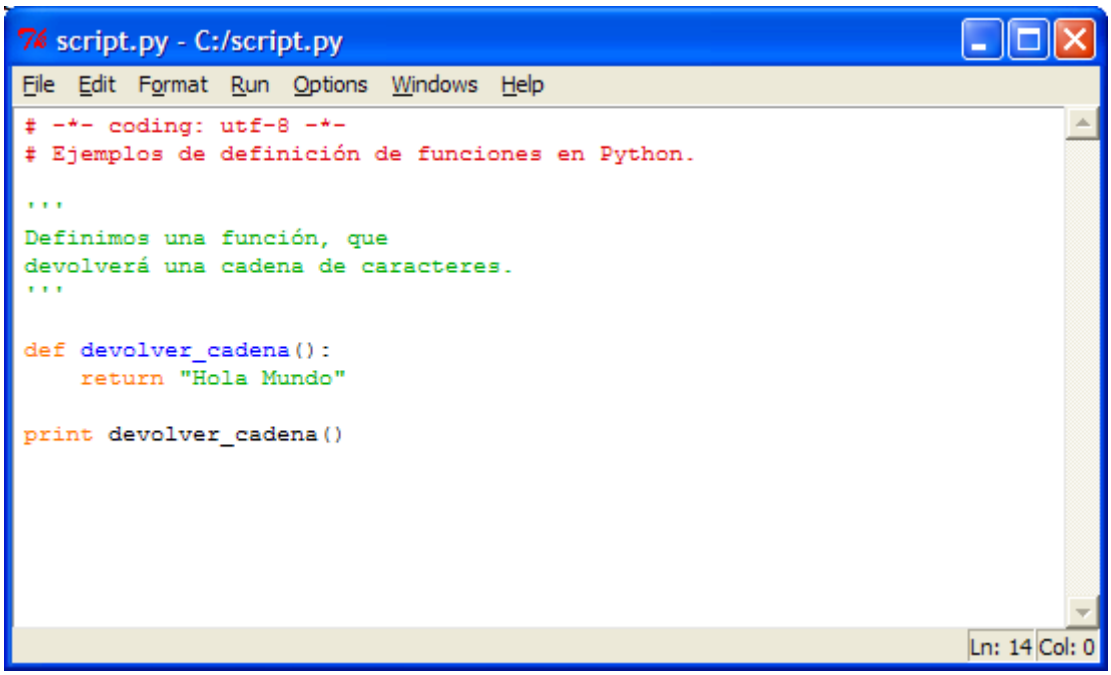
Una función puede devolver valores con la palabra clave **return** (si se omite, devuelve **None**). Los parámetros pasados por una función son siempre por referencia, excepto los numéricos y cadenas. Se permite dar valores por defecto a los parámetros (para cuando éstos se omiten en el cuerpo de la función), siempre y cuando aparezcan al final de la lista de parámetros. Las funciones pueden tener atributos.

Comentarios

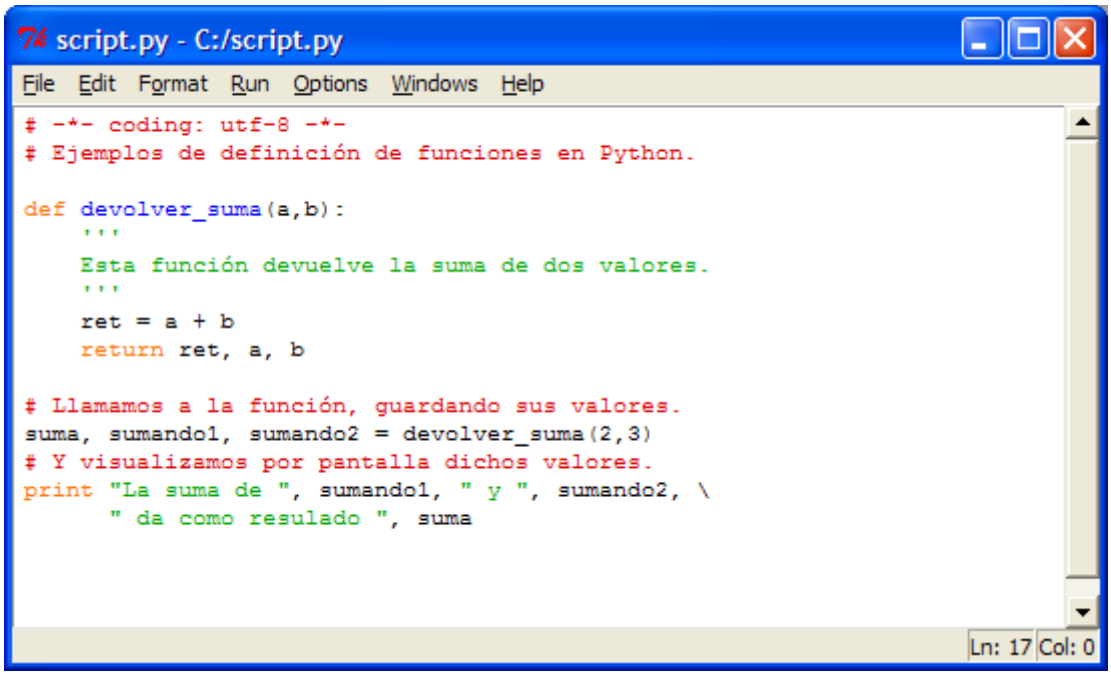
En Python se pueden insertar comentarios de una línea, precedidos por el carácter **#**, ó multilínea, con triple comillado simple ó doble.

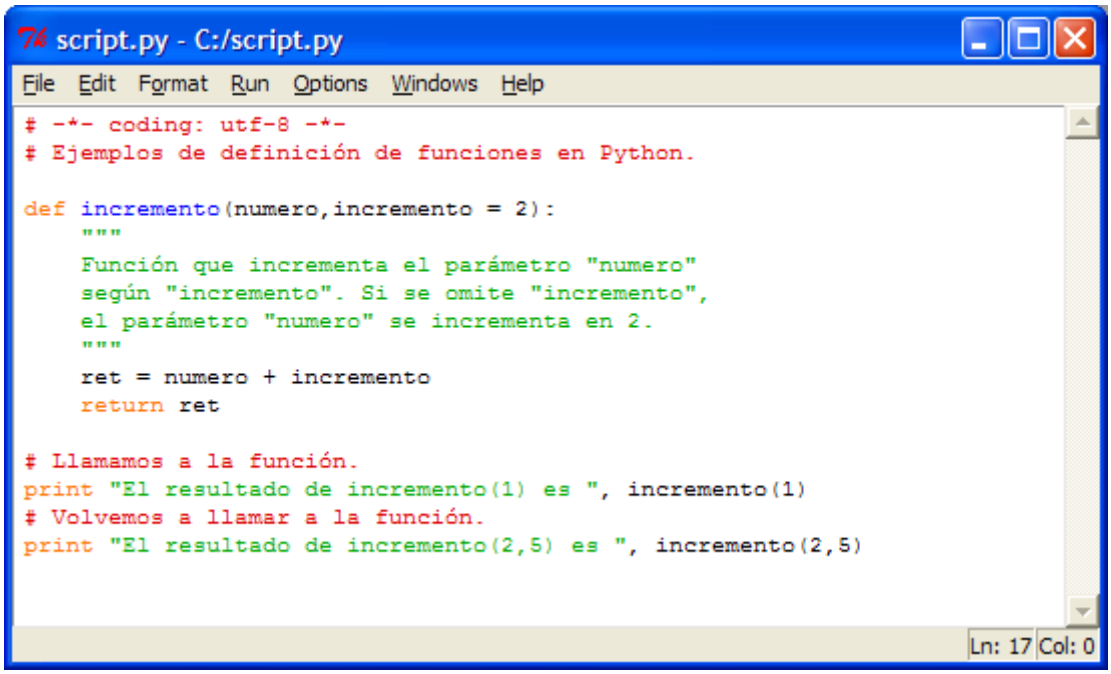
```
# Esto es un comentario.
'''
Esto es un comentario con
varias líneas.
'''
“””
Y esto
también
“””
```

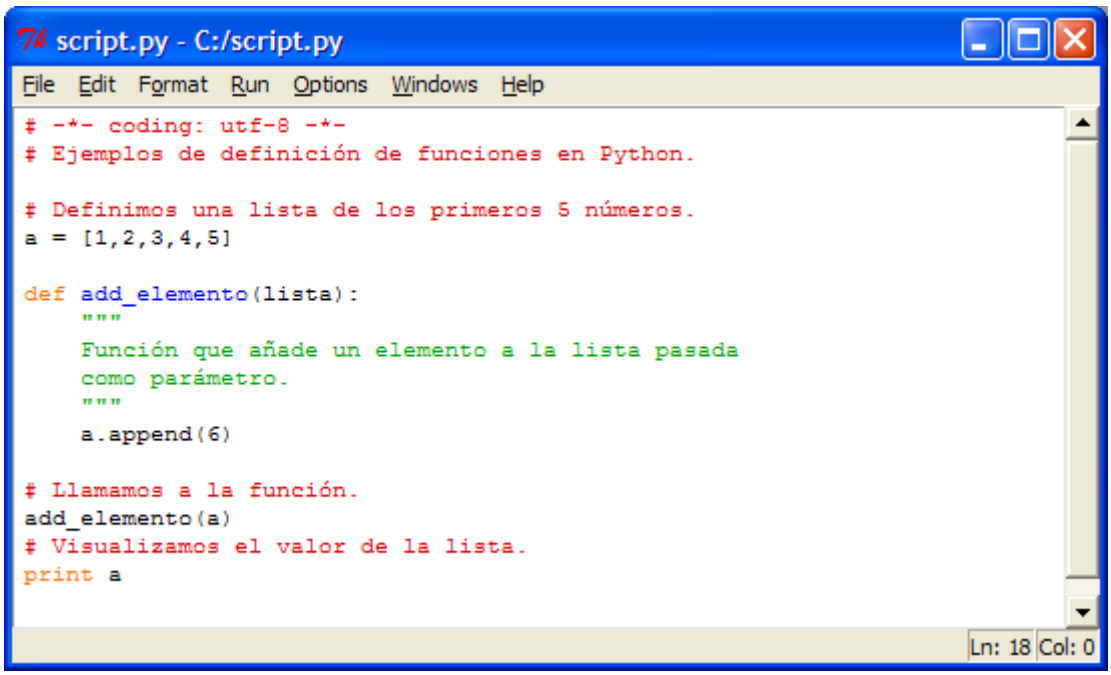
Volvemos a nuestro fichero *script.py*, y escribimos el código siguiente para ver todos los conceptos hasta el momento:



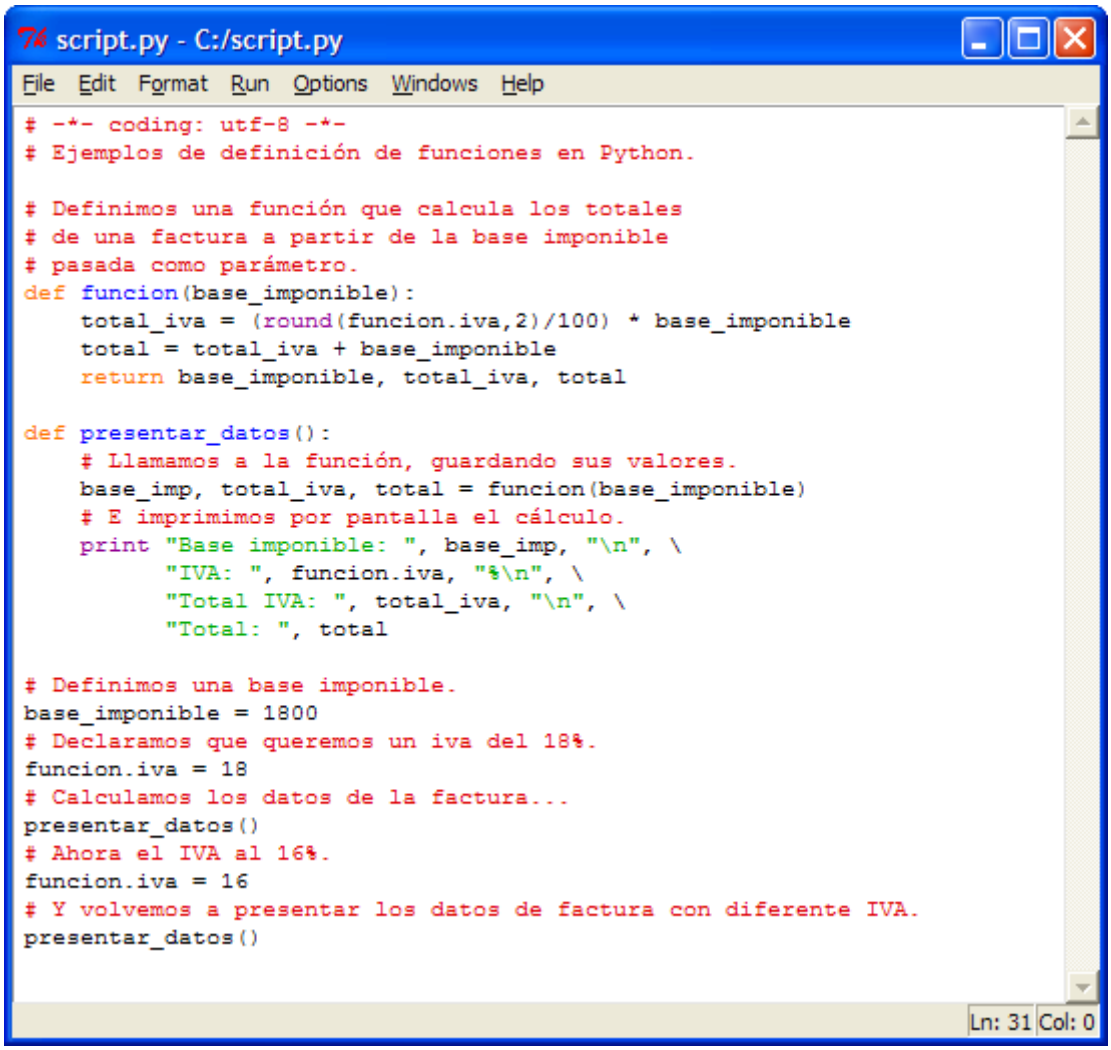
Darse cuenta de la sangría que utilizamos para enmcarcar el bloque de código correspondiente al cuerpo de la función.



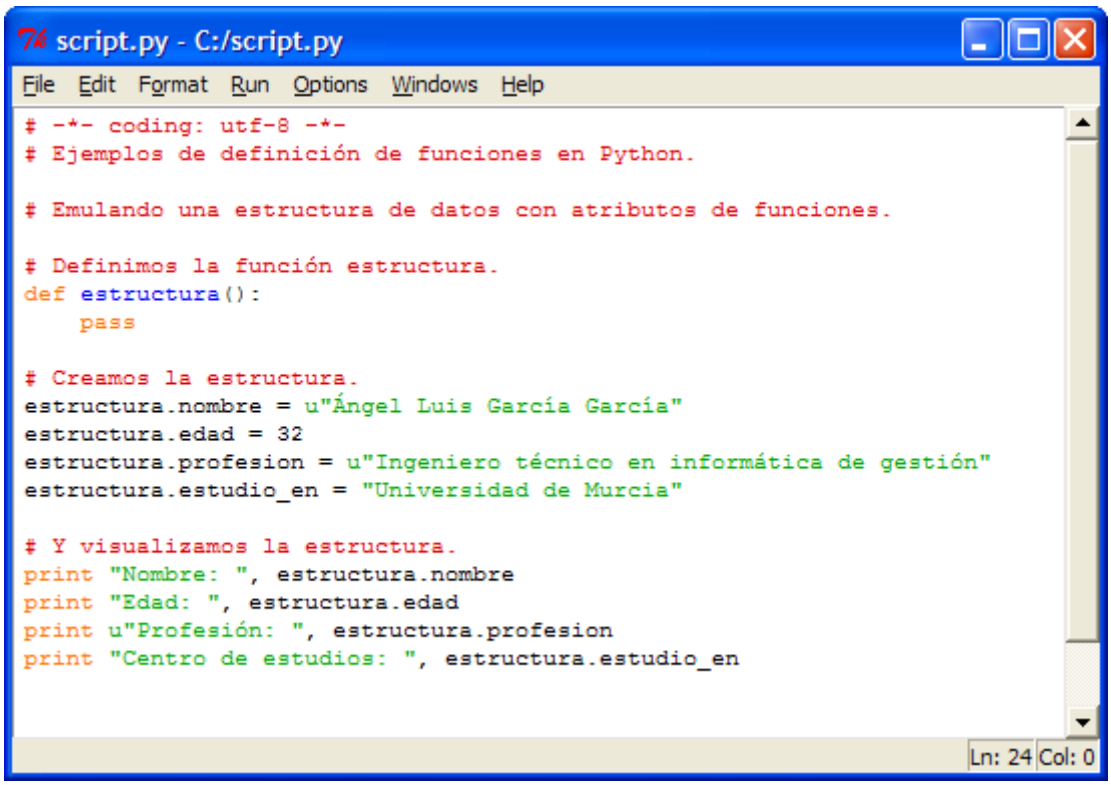




Una **lista** es un tipo de dato en Python, que se verá más adelante. Fijarse en que la función modifica la lista *a* por referencia, ya que dicha función añade (método append) el elemento “6” a dicha lista. Ahora la lista contiene los 6 primeros números.



Aquí podemos observar varias cosas. Se ha utilizado la función integrada **round** para redondear el valor del atributo *iva* de la funcion. Darse cuenta de la funcionalidad de los atributos de funciones. Si el lector se esfuerza intelectualemente un poco más podrá darse cuenta que podemos utilizar los atributos de funciones para emular el *tipo de dato estructura*, que se da en algunos lenguajes de programación, como C ó PowerBuilder, y del que Python carece.



La palabra reservada **pass** se utiliza para representar la declaración nula.

None

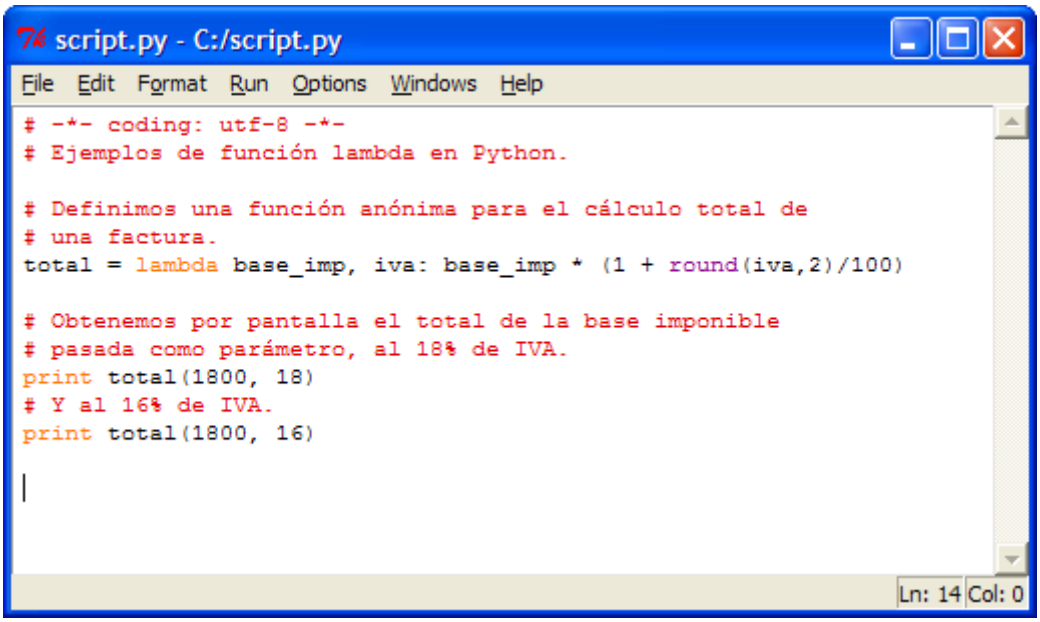
None se utiliza como valor de retorno por defecto en funciones. Es un objeto integrado de tipo *NoneType*. Es una constante que indica la ausencia de un valor.

5.1. Funciones anónimas: lambda

La sintaxis para crear una función anónima es:

lambda [lista_parámetros]**:** Expresión_devuelta

De esta forma se crea una función anónima, esto es, una función que no tiene identificador. *Expresión_devuelta* debe de ser una expresión, no una declaración (es decir, no “if xx:...”, “print xxx”, etcétera) y además no puede contener múltiples líneas. Esta función integrada forma parte del conjunto de programación funcional, junto con filter(), map() y reduce().

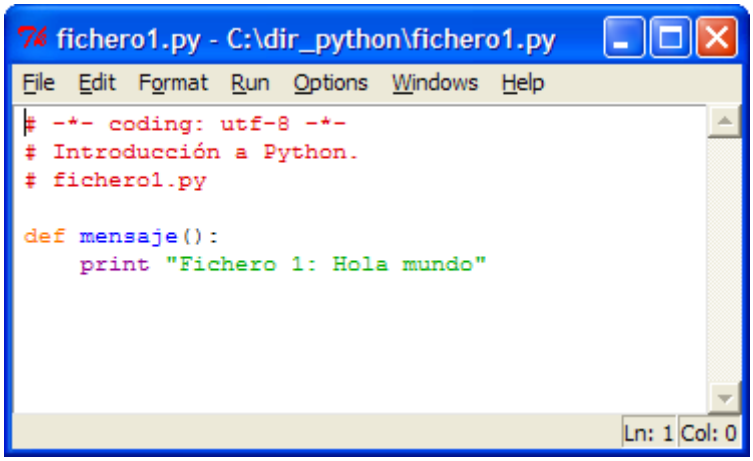


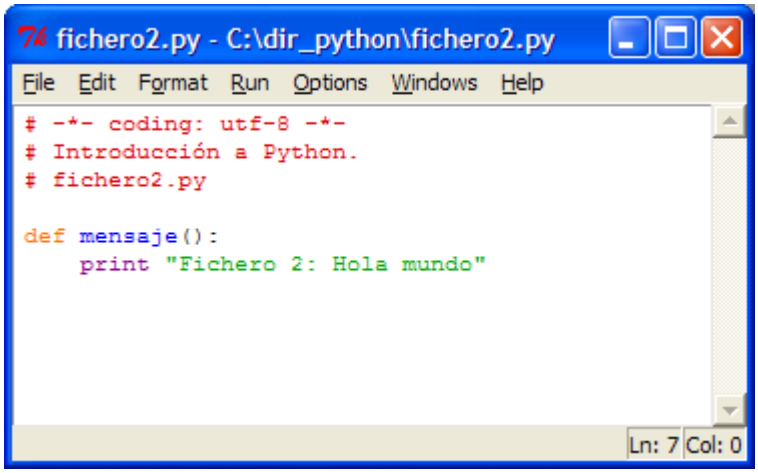
6. Módulos

Un módulo es una librería en Python, y normalmente corresponde a un archivo de programa. Un fichero con extensión **.py** es un módulo. El nombre del fichero **.py** es el nombre del módulo. Cada archivo es un módulo y los módulos importan a otros módulos para realizar distintas tareas de programación. Para importar un módulo se utiliza palabra clave **import**.

import puede ir en cualquier parte del código, aunque se recomienda incluirlo al principio del script, para mayor claridad y menor ofuscamiento del código.

Vamos a probar este concepto mediante un ejemplo. Se crea el directorio *dir_python*, y dentro de él, vamos a incluir 3 ficheros, con nombres *fichero1.py*, *fichero2.py* y *fichero3.py*, con el siguiente contenido:





fichero3.py - C:/dir_python/fichero3.py

File Edit Format Run Options Windows Help

```
# -*- coding: utf-8 -*-
# Introducción a Python.
# fichero3.py

def mensaje():
    print "Fichero 3: Hola mundo"
```

Ln: 7 Col: 0

Vamos a crear un cuarto fichero, denominado *principal.py*, que llamará a las distintas funciones de los módulos anteriormente creados.

principal.py - C:/dir_python/principal.py

File Edit Format Run Options Windows Help

```
# -*- coding: utf-8 -*-
# Ejemplo de utilización de import.

# Importamos los módulos fichero1, fichero2 y fichero3.
import fichero1, fichero2, fichero3

# Ejecutamos las funciones.
fichero1.mensaje()
fichero2.mensaje()
fichero3.mensaje()
```

Ln: 12 Col: 0

Se puede importar nombres específicos dentro de un módulo, de manera que no se importa todo el módulo, mediante:

from <modulo> **import** <simbolo>

principal.py - C:/dir_python/principal.py

File Edit Format Run Options Windows Help

```
# -*- coding: utf-8 -*-
# Ejemplo de utilización de import.

# Importamos los módulos fichero1, fichero2 y fichero3.
import fichero1
from fichero2 import mensaje
import fichero3

# Ejecutamos las funciones.
fichero1.mensaje()
mensaje()
fichero3.mensaje()
```

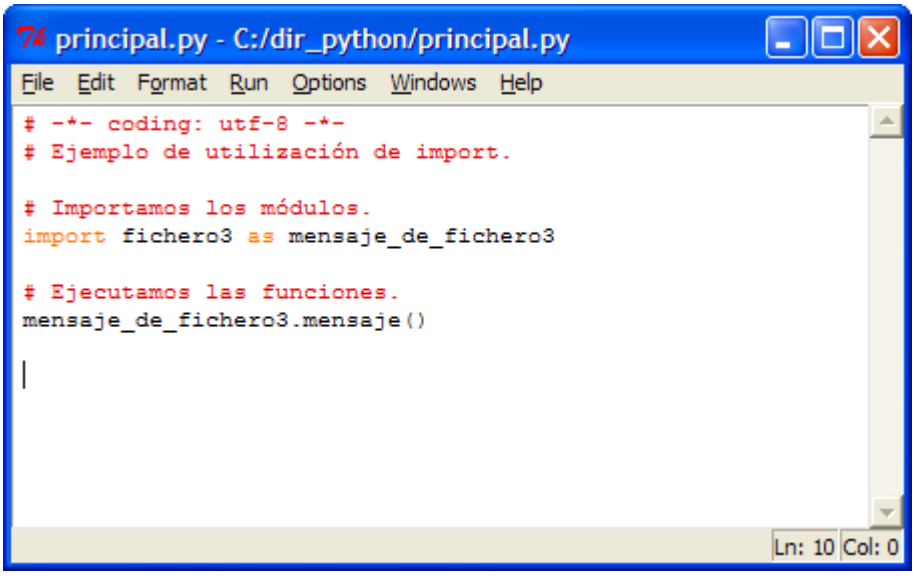
Ln: 14 Col: 0

También se puede crear un alias al nombre de un módulo, mediante:

import <modulo> **as** <alias_modulo>

Ó a la función ó clase incluida en el módulo, mediante:

from <modulo> **import** <simbolo> **as** <alias_simbolo>



Hay una última forma de importar módulos y es con la sintaxis:

from <modulo> **import** *

Esta manera es legal pero extremadamente desaconsejable, sobretodo si se crea gran cantidad de código, puesto que hace difícil su mantenimiento. Por ejemplo:

from a1 import *
from a2 import *
…
from a20 import *
….
….
….
f(x,y)
Error-----> ¿En qué módulo está implementada la función?
….
….
….

David Goodger en su obra *Code Like a Pythonista: Idiomatic Python*, expresa extraordinariamente bien la no utilización de esta expresión:

(Dagobah, jungla, pantanos, y niebla.)

LUKE: ¿Es mejor *from module import* * que los imports explícitos?

YODA: No, no mejor. Más rápido, más fácil, más seductor.

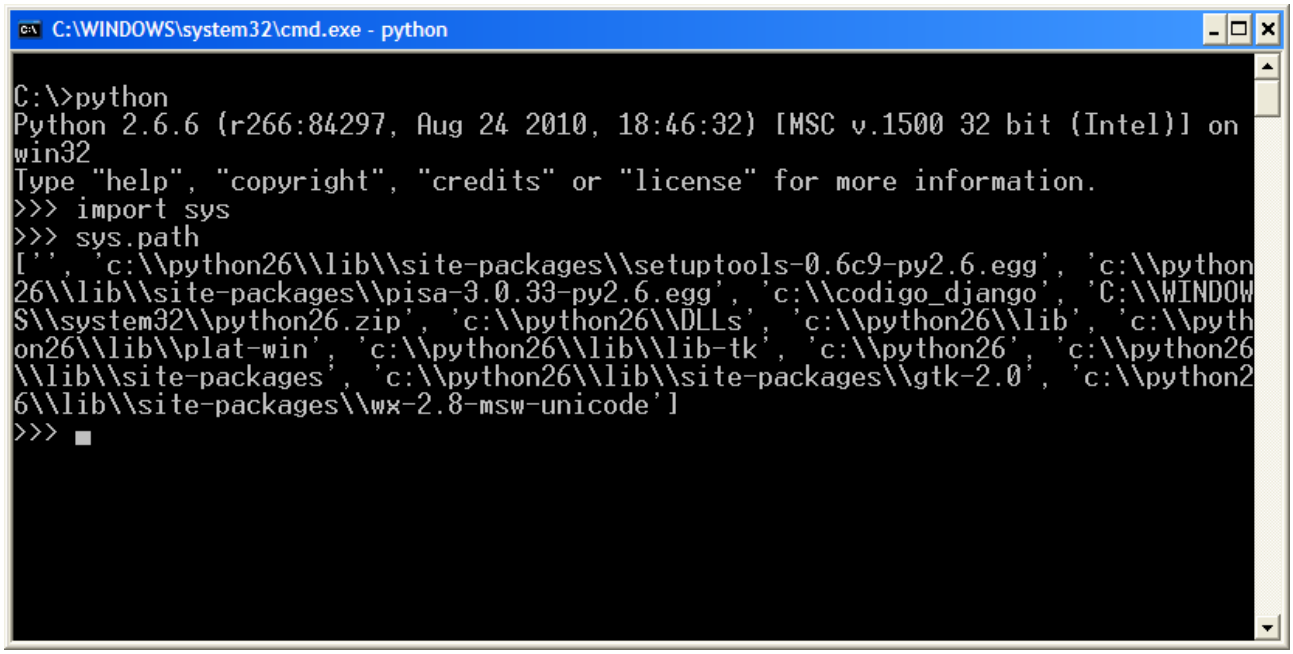
LUKE: Pero ¿cómo sabré por qué los imports explícitos son mejores que la forma con el carácter comodín?

YODA: Saberlo tu podrás cuando tu código intentes leer seis meses después.

6.1. Localización de módulos

Cuando incluimos la declaración **import** para la carga del módulo correspondiente no se le dice a Python donde buscar. Por defecto Python buscará primero en el directorio donde tengamos el programa principal ó donde resida el intérprete Python. Si no lo encuentra ahí, seguidamente buscará en la variable de entorno **PYTHONPATH**. Si tampoco tiene éxito, buscará por las bibliotecas estándar que vienen en la distribución base de Python.

Una manera de ver las rutas por las cuales Python intenta cargar los módulos mediante **import** es cargando el módulo de sistema **sys** (en el intérprete), y a continuación ejecutar **sys.path**. Esto nos devolverá una lista con todas las rutas a las que Python accede para buscar los módulos requeridos.



6.2. Listado de módulos en la librería estándar

El listado completo de la librería estándar para Python 2.6.6 se puede encontrar en:

http://docs.python.org/release/2.6.6/modindex.html

6.3. Packages

Un package (paquete) es un espacio de nombres (namespace) que se asigna a un directorio, el cual incluye módulos y un módulo especial de inicialización denominado **__init__.py**, que posiblemente esté vacío. Los packages/directorios se pueden anidar, por lo que se puede acceder a un módulo vía package.package.....modulo.simbolo. Esto que parece algo complicado en realidad no lo es. Y para entenderlo, que mejor que con un ejemplo.

Para ello vamos a crear un directorio *dir_python*. Dentro de él vamos a crear un package (es decir, un directorio), denominado *mi_package*, que se compondrá de otro package anidado, llamado *nivel1*, y dentro de él, un módulo llamado *script.py*, que contendrá una función que nos devolverá “Hola Mundo”. Esto es:

dir_python
    principal.py
    mi_package
        __init__.py
        nivel1
            __init__.py
            script.py

El módulo principal (*principal.py*) que llamará , mediante el uso de espacio de nombres, a la función que está en *script.py,* se guardará en *dir_python*, como se puede observar. El fichero *script.py* tiene el siguiente código:

```
def mensaje():
    print “Hola Mundo”
```

Para llamar a esta función habría que declarar en el módulo *principal.py* lo siguiente:

```
from mi_package.nivel1.script import mensaje

mensaje()
```

El lector debe de darse cuenta de las ventajas de utilizar espacios de nombres. Se pueden crear librerías de clases y funciones jerarquizadas, frameworks y demás estructuras de código Python (ó no Python). Todo completamente organizado. Crear namespaces es una buena idea.

7. Palabras clave y funciones integradas en Python

Si queremos saber las palabras clave que tiene nuestra versión actual de Python podemos acceder al módulo **keyword**, el cual recoge dicho conjunto de vocablos. Tal que así:

```
import keyword

print "Palabras clave", keyword.kwlist
```

Si nos ponemos algo más puntillosos, podemos mejorar la presentación, bien utilizando el módulo **pprint**, ó creando una presentación por columnas:

```
import keyword

tab = '\t'
enter = '\n'
contador = 0
cadena = ''

for i in keyword.kwlist:
    cadena += i + tab
    if contador == 5:
        contador = 0
        cadena += enter
    else:
        contador += 1
print cadena
```

Devolviendo las palabras clave (versión 2.6.6):

```
and     as      assert  break   class   continue
def     del     elif    else    except  exec
finally for     from    global  if      import
in      is      lambda  not     or      pass
print   raise   return  try     while   with
yield
>>>
```

Python trae consigo ciertas funciones que vienen de serie, también llamadas funciones integradas, esto es, no es necesario cargarlas desde ningún módulo, como **raw_input** ó **abs**. Dichas funciones dependen de la versión Python que tengamos instalada en nuestro sistema. Podemos encontrar las **built-in Functions** en:

http://docs.python.org/library/functions.html

8. Tipos de datos y sus operadores

Operadores de comparación definidos para cualquier tipo de dato

Son los habituales: **<**, **>**, **<=**, **>=**, **==** (igualdad), **!=** ó **<>** (no igual a), **is** (objeto identidad), **is not** (objeto identidad negada).

Operadores de asignación

a = b asigna el objeto b a la etiqueta a.
a += b equivale a a = a + b. Además **a -= b, a *= b, a /= b, a //= b, a %= b, a **= b**.
primero, segundo = l[0:2] equivale a primero=l[0] y segundo = l[1].
Se permite la asignación múltiple:

```
a = b = c = 0
lista1 = lista2 = [1,2,3]
```

c1,c2,c3 = 'abc' equivale a c1='a', c2='b', c3='c'.
x,y=y,x intercambia los valores de x e y.

Si se quiere saber el tipo de dato de cualquier objeto se puede utilizar la función integrada **type** en el intérprete de Python. Por ejemplo:

```
>>> type(3.4)
<type 'float'>
>>>
```

Los tipos de datos más importantes (que no los únicos) en Python son: None, booleano, número, secuencia (cadena, lista, tupla) y fichero.

8.1. None

None: Aunque en realidad es una constante, la podemos catalogar aquí, cuyo significado es “ausencia de valor”. None es lo que devuelve una función por defecto. El tipo de None es en realidad NoneType.

8.2. Booleano

Booleano: Tipo booleano, de las álgebras de Boole. Los valores de este tipo son **True** y **False**. Los operadores son **and**, **or** y **not**. La función integrada **bool(**expresión**)** convierte un valor a booleano. Así **bool()** devuelve False, **bool(**expresión**)** devuelve True si expresión es cierto y False en caso contrario. Se considera False la constante None, el número cero y las secuencias y diccionarios vacíos. Todo lo demás se considera True.

8.3. Números

Números: Los 4 tipos de números más importantes son: **int** (se implementa como long de C), **long** (limitado por los recursos de la máquina), **float** (se implementa como double de C) y **complex** (número complejo, representado como un par de números de punto flotante de doble precisión a nivel de máquina), esto es, con parte real e imaginaria).

El operador para el tipo numérico complejo es la función integrada complex(parte_real, parte_imaginaria), que genera un tipo de dato complex. Si z es un complex, z.real devuelve la parte real y z.imag devuelve la parte imaginaria.

Para el resto de números, los operadores de casting (conversión) son:
int(x**)** devuelve x convertido a tipo int.
long(x**)** devuelve x convertido a tipo long.
float(x**)** devuelve x convertido a tipo float.
coerce(x,y**)** devuelve la tupla (x,y), donde y es convertido al mismo tipo que x.

Los operadores de números son los que nos podemos encontrar en cualquier lenguaje de programación, a saber: -x, x + y, x – y, x * y, x / y (división entera), x % y (división modular), x**y (x elevado a y).

Además, existen otras funciones integradas para operaciones matemáticas, como:
abs(x**)** devuelve el valor absoluto de x.
pow(x,y**)** devuelve x**y.
divmod(x, y**)** devuelve la tupla (x/y, x%y).
cmp(x,y**)** devuelve -1 si x<y, 0 si x == y y 1 si x>y.
round(x,y**)** redondea x con y posiciones decimales.

Se pueden encontrar más funciones matemáticas en los módulos **math**, **cmath** y **random**. También se encuentran enteros binarios, hexadecimales y octales. Hay un tipo de número, llamado **Decimal**, útil para reseprentar fracciones, que está en el módulo **decimal**.

8.4. Secuencias

Una secuencia es un tipo de dato que contiene una serie de elementos, con un orden, en donde se pueden acceder a los mismos para obtener su valor mediante ciertos mecanismos.

Hay 2 tipos de secuencias, a saber, el tipo de secuencia mutable, que está compuesto por el tipo de dato lista, y las secuencias inmutables, que están compuestas por el tipo de dato tupla y el tipo de dato cadena.

8.4.1. Listas

Una lista es un tipo de dato secuencia de clase mutable (se puede modificar una vez instanciada), donde sus elementos pueden ser heterogénos ú homogéneos. Una lista en Python es de tipo **list**.

Lista = [] instanciamos una lista vacía.
Lista = [1,2,4,5,6]
Lista = [u'Ángel Luis', 32, 'Lorca', [u'Ingeniero Técnico Informática','Python']]

Como se puede observar una lista puede contener elementos del mismo tipo ó diferentes. Una lista puede contener listas y cualquier otro tipo de dato, objetos incluidos.

Mediante ciertos métodos de la lista se pueden añadir ó eliminar elementos de la lista. Es por ello que se dice que la lista es una secuencia mutable, ya que después de ser instanciada puede modificarse.

8.4.2. Tuplas

Una tupla es un tipo de dato secuencia de clase inmutable (no se puede modificar una vez instanciada), donde sus elementos pueden ser heterogéneos ú homogéneos. Una tupla en Python se define como de tipo **tuple**.

Tupla = () instanciamos una tupla vacía.
Tupla = (1,) una tupla con un elemento.
Tupla = (1,34)
Tupla = (“Hola Mundo”,2.3,[1,2,3],('a',3))

Al igual que las listas, los elementos de las tuplas pueden ser del mismo ó diferente tipo. Sin embargo las tuplas son inmutables, es decir, una vez instanciadas (creadas) no se pueden modificar, esto es, no se pueden eliminar ó añadir elementos a la tupla ya creada.

8.4.3. Cadenas

En Python los tipos **str** y **unicode** son tipos cadena. Una cadena es un tipo inmutable, ya que al igual que las tuplas, una vez creada no se puede modificar.

Cadena = 'Hola Mundo'
Cadena = “Hola Mundo”

Las cadenas contienen caracteres alfanuméricos.

8.4.4. Formas de acceder a elementos de secuencias

Hay varias maneras de poder acceder a elementos de una secuencia (válido para listas, cadenas y tuplas):

1) Acceso mediante índices.

   secuencia = ['Hola','Mundo','Python']

   secuencia[0] == 'Hola'
   secuencia[1] == 'Mundo'
   secuencia[2] == 'Python'

   La indización comienza en la posición 0, como se puede observar. Índices negativos significa contar hacia atrás desde el final de la secuencia, así tendremos que:

   secuencia[-1] == 'Python'
   secuencia[-2] == 'Mundo'
   secuencia[-3] == 'Hola'

   Si la secuencia contiene secuencias podemos acceder a sus elementos también mediante indización de la siguiente manera:

   secuencia[2][0] == 'P'
   secuencia[2][1] == 'y'
   secuencia[2][2] == 't'
   secuencia[2][3] == 'h'
   secuencia[2][4] == 'o'
   secuencia[2][5] == 'n'

2) Acceso mediante slicing.

   slicing significa en inglés “rebanar”, y en el diccionario de la RAE, en su segunda acepción, rebanar es “cortar o dividir algo de una parte a otra”. Bien, pues esta técnica es exactamente lo que hace. De lo que se trata es de obtener secciones de una secuencia. Esta forma de acceso es tremendamente poderosa en Python.

   El slicing funciona de la siguiente manera:

   [índice de comienzo:índice fin]

   donde “índice de comienzo” está incluido e “índice fin” está excluido. Si se omite el “índice de comienzo” por defecto toma valor 0. Si se excluye el “índice fin” por defecto toma el índice del último elemento de la secuencia (que coincide con len(secuencia)).

secuencia = (1,2,3,4,5)

secuencia[:] == (1,2,3,4,5) En este caso, puesto que se han omitido tanto los índices de comienzo, como de fín, el slicing da como resultado una copia de la secuencia.

secuencia[1:] == (2,3,4,5) Aquí estamos obteniendo una tupla, desde la posición 1 hasta el último elemento de la secuencia.

secuencia[2:3] == (3,) Darse cuenta que estamos obteniendo el elemento de la posición 2 hasta la posición 3 (el cual se excluye). Es por ello que se obtiene una tupla de un único elemento.

secuencia[:4] == (1,2,3,4) Se excluye la posición de "índice fin" y puesto que no hemos declararo el "índice de comienzo", se toma por defecto 0.

En el slicing hay un tercer elemento opcional, el "paso", y que sirve para indicar saltos en la secuencia. Si no se declara, por defecto es 1. La sintaxis es:

[índice de comienzo:índice fin:paso]

secuencia[::1] == secuencia[:] == (1,2,3,4,5)

secuencia[::**2**] == (1,3,5) Esta expresión lo que dice es a partir de la posición del "índice de comienzo" (incluido), selecciona los elementos que se encuentren a dos posiciones de longitud y así sucesivamente.

secuencia[:4:2] == (1,3) Ahora la expresión dice que a partir de la "posición de comienzo", y hasta la posición 4 (excluida), obtenga todos los elementos cada dos posiciones y así sucesivamente.

secuencia[1::2] == (2,4) Obtenemos todos los números pares.

Los índices negativos indican contar hacia atrás, desde el final de la secuencia. La característica de saltos en el slicing lo contempla:

secuencia[::-1] == (5,4,3,2,1) Orden inverso.

secuencia[::-2] == (5, 3, 1) Orden inverso de números impares.

NOTA: Si el paso es negativo, se intercambian los índices de comienzo y fín. Tener esto en cuenta, para no llevarse imprevistos en el resultado.

secuencia[4:0:-1] == (5, 4, 3, 2)

secuencia[4:1:-1] == (5, 4, 3)

secuencia[4:2:-1] == (5, 4)

De todas formas, no hay que complicar las cosas en exceso si se pueden

hacer explícitas.

3) Mediante **iteradores** y la estructura de control de flujo **for**, que veremos más adelante.

8.4.5. Operaciones sobre secuencias

x **in** secuencia: Devuelve True si x es un elemento de secuencia y False en caso contrario.

x **not in** secuencia: Devuelve True si x no es un elemento de secuencia y False en caso contrario.

secuencia1 **+** secuencia2: Concatena secuencias del mismo tipo.

n * secuencia : Contanena la secuencia n veces.

secuencia.**count(**x**)**: Devuelve el número de veces que está el elemento x en la secuencia.

secuencia.**index(**x**)**: Devuelve el índice del primer elemento x de la secuencia (el índice más pequeño).

len(secuencia**)**: Devuelve el número de elementos de la secuencia.

min(secuencia**)**: Devuelve el elemento más pequeño de la secuencia.

max(secuencia**)**: Devuelve el elemento más grande de la secuencia.

sorted(secuencia**)**: Devuelve la secuencia ordenada.

reversed(secuencia**)**: Devuelve un iterador de la secuencia en orden inverso.

Ejemplos:

```
>>> cadena = "Hola Mundo"
>>> lista = ["Hola", "Mundo"]
>>> tupla = ("Hola", "Mundo")
>>>
>>> "Hola" in cadena
True
>>> "mundo" in lista
False
>>> "Python" not in tupla
True
>>> cadena + cadena
'Hola MundoHola Mundo'
>>> tupla+tupla
('Hola', 'Mundo', 'Hola', 'Mundo')
>>> cadena * 4
'Hola MundoHola MundoHola MundoHola Mundo'
```

```
>>> lista.count("Mundo")
1
>>> tupla.count("Python")
0
>>> cadena.count("Hola")
1
```

```
>>> lista[0].count("a")
1
```

En este ejemplo lo que se hace es acceder al primer elemento de la lista y contar los elementos “a” de dicho elemento, es decir, cuantas a's hay en la palabra “Hola”.

```
>>> lista.index("Mundo")
1
>>> lista.index("Hola")
0
>>> len(lista)
2
>>> len(cadena)
10
>>> len(tupla)
2
>>> min(cadena)
' '
>>> min(lista)
'Hola'
>>> max(cadena)
'u'
>>> max(tupla)
'Mundo'
>>> sorted(cadena)
[' ', 'H', 'M', 'a', 'd', 'l', 'n', 'o', 'o', 'u']
>>> sorted(lista)
['Hola', 'Mundo']
```

8.4.5.1 Operaciones sobre listas

l[i] = x El elemento i de la lista l es reemplazado por x.

l[i:j] = k El slicing de i a j en la lista l se reemplaza por k.

del l[i:j] Es lo mismo que hacer l[i:j] = [].

l.append(x) Añade el elemento x al final de la lista l.

l.extend(l2) Añade la lista l2 al final de la lista l.

l.insert(i,x) Inserta el elemento x en la posición i de la lista l.

l.reverse() Invierte el orden de los elementos de la lista l.

l.sort() Ordena la lista l.

x = l.pop() Devuelve en x el último elemento de la lista l, y elimina de la lista l dicho último elemento (en otras palabras, saca el último elemento de la lista).

Ejemplos:

```
>>> lista = ['Curso','de','Python']
>>> lista2 = ['en','castellano']
>>> lista.extend(lista2)
>>> lista
['Curso', 'de', 'Python', 'en', 'castellano']
>>> lista.append('para Caldum')
>>> lista
['Curso', 'de', 'Python', 'en', 'castellano', 'para Caldum']
>>> lista[0] = 'Charla entre amigos'
>>> lista
['Charla entre amigos', 'de', 'Python', 'en', 'castellano', 'para Caldum']
>>> lista.pop()
'para Caldum'
>>> lista
['Charla entre amigos', 'de', 'Python', 'en', 'castellano']
>>> lista.insert(0,'Mega')
>>> lista
['Mega', 'Charla entre amigos', 'de', 'Python', 'en', 'castellano']
>>> lista.reverse()
>>> lista
['castellano', 'en', 'Python', 'de', 'Charla entre amigos', 'Mega']
>>> lista.reverse()
>>> lista
['Mega', 'Charla entre amigos', 'de', 'Python', 'en', 'castellano']
>>>
```


>>> lista = ['a','b','c','d','e']
>>> del lista[3:]
>>> lista
['a', 'b', 'c']
>>> lista = ['a','b','c','d','e']
>>> lista[3:] = []
>>> lista
['a', 'b', 'c']
>>> lista.reverse()
>>> lista
['c', 'b', 'a']
>>> lista.sort()
>>> lista
['a', 'b', 'c']
>>>

8.4.5.2. Operaciones sobre cadenas

Las cadenas en Python tienen una gran cantidad de métodos, de los cuales se presentan los más importantes en cuanto a su uso:

cadena.strip() Devuelve el contenido de cadena sin espacios en blanco al principio y al final de la misma.

cadena.split(sep) Devuelve una lista de las palabras que hay en cadena, usando como separador sep.

cadena.find(subcadena) Devuelve el primer índice (y menor) de la cadena en donde se encuentra subcadena.

Cadena.upper() Devuelve una copia de la cadena con todas las letras en mayúscula.

Cadena.lower() Devuelve una copia de la cadena con todas las letras en minúscula.

Cadena.capitalize() Devuelve una copia de la cadena con la primera letra en mayúscula y las restantes en minúscula.

Cadena.replace(antiguo, nuevo) Devuelve una copia de la cadena, reemplazando todos los elementos antiguo por el elemento nuevo.

sep.join(sec) Devuelve una concatenación de cadenas de la secuencia sec, separadas por el separador sep.

Más métodos son center, decode, encode, endswith, isalnum, isalpha, isdigit, istitle, islower, isupper, rfind, rindex, rpartition, rstrip, ljust, rjust, rsplit, splitlines, startswith, swapcase, title, translate, zfill.

Ejemplos:

```
>>> cadena = 'Python es el mejor lenguaje'
>>> cadena.split()
['Python', 'es', 'el', 'mejor', 'lenguaje']
>>> cadena = 'Python es el mejor lenguaje'
>>> cadena.find('es')
7
>>> cadena.upper()
'PYTHON ES EL MEJOR LENGUAJE'
>>> cadena.upper().lower()
'python es el mejor lenguaje'
>>> cadena.upper().capitalize()
'Python es el mejor lenguaje'
```

```
>>> cadena = '    Python mola!    '
>>> cadena.strip()
'Python mola!'
>>> cadena.replace('o','0')
'    Pyth0n m0la!    '
>>> lista = ['Python','es','el','mejor']
>>> " ".join(lista)
'Python es el mejor'
>>>
```

8.5. Tablas hashing (diccionarios)

Un diccionario ó tabla hashing es un tipo de dato donde cada elemento es un par <clave:valor>, siendo la clave unívoca para todo el diccionario, y valor cualquier tipo de dato, que puede repetirse. En Python los diccionarios son de tipo **dict**.

d = {} Diccionario vacío.
d = {1:'Python, 2:'C++',3:'R',4:'Fortran',5:'Java'}
d = {nombre:u'Ángel Luis,apellido1:'García,apellido2:'García',edad: 32}
d = {1:'UMU','Instituto':u'María Cegarra Salcedo'} Las claves pueden ser de tipos diferentes. El único requisito es que no puede haber duplicidades de claves.

8.5.1. Operaciones sobre diccionarios

d['x'] Devuelve el valor cuya clave es 'x'.
len(d) Devuelve el número de claves del diccionario.
del (d['x']) Elimina el elemento cuya clave es x y su valor d['x'] del diccionario.
d.copy() Devuelve una copia del diccionario d.
d.has_key(k) Devuelve True si existe la clave k y False en caso contrario. Esta expresión es equivalente a k in d.
d.get(k,x) Devuelve el valor de la clave k. Si no existe devuelve x.
d.setdefault(k,x) Devuelve el valor de la clave k si existe. Si no existe crea el elemento k:x (clave k y valor x) y devuelve x.
d.clear() Elimina todos los elementos del diccionario.
d.popitem() Devuelve y elimina un elemento.
d.items() Devuelve una lista de de todos los elementos, donde cada elemento es una tupla (clave, valor).
d.keys() Devuelve una lista con todas las claves.
d.values() Devuelve una lista con todos los valores.

En los diccionarios no existe ordenación. Para iterar sobre todos los elementos de un diccionario se pueden utilizar varias técnicas, entre ellas el uso de iteradores y de bucles for, que veremos más adelante.

Ejemplos:

```
>>> d = {'x':10,'y':20,'z':30}
>>> len(d)
3
>>> d1 = d.copy()
>>> d1
{'y': 20, 'x': 10, 'z': 30}
>>> del(d['x'])
>>> d
{'y': 20, 'z': 30}
>>> d.has_key('x')
False
>>> d.has_key('y')
True
>>> d['z']
30
>>>
```

```
>>> d = {'y':20,'z':30}
>>> d.get('x','No existe la clave "x"')
'No existe la clave "x"'
>>> d.setdefault("x",10)
10
>>> d
{'y': 20, 'x': 10, 'z': 30}
>>> d.popitem()
('y', 20)
>>> d
{'x': 10, 'z': 30}
>>> d.clear()
>>> d
{}
>>>
```

```
>>> d = {'x':10,'y':20,'z':30}
>>> d.items()
[('y', 20), ('x', 10), ('z', 30)]
>>> d.keys()
['y', 'x', 'z']
>>> d.values()
[20, 10, 30]
>>>
```

8.6. Ficheros

El tipo de dato fichero en Python es **file**. Se crean ficheros a partir de la función integrada **open()** ó su alias **file()**. También se pueden crear a partir de otros módulos, como codecs.

Crear un fichero en Python es tan simple como:

f = open(nombre_fichero, flag)

donde f es el manejador de fichero. nombre_fichero es una cadena que identifica el nombre del fichero y flag indica el modo de apertura del fichero, que puede ser r (lectura), w (escritura), a (añadir al final del fichero), r+ (random). También cabe la posibilidad de indicar que se trata de un fichero binario ó de un modo sin conversion EOL (fin de línea), mediante rb, wb, ab y r+b respectivamente.

Hay varias formas de leer un fichero en Python (una vez abierto):

f.read(t) Lee a lo sumo t bytes del fichero f y devuelve el contenido en un objeto cadena. Si t se omite devuelve el contenido del fichero hasta que encuentre EOF (fin de fichero)

f.readline() Lee una línea entera de un fichero, donde el final de línea es Enter (\n), excepto que se llegue a EOF (fin de fichero).

f.readlines() Lee hasta que encuentre EOF (fin de fichero) y devuelve una lista con las líneas leídas.

Y varias formas de escribir un fichero:

f.write(cadena) Escribe la cadena en el fichero f.
f.writelines(lista) Escribe una lista de cadenas en el fichero f. No se añade EOL (fin de línea).

Y finalmente, cerrar el fichero.

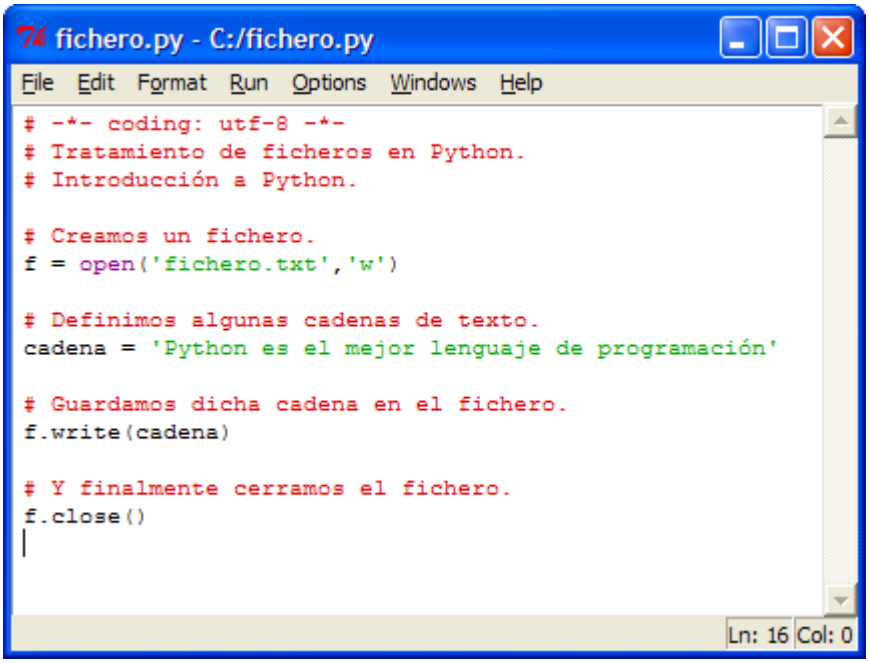
f.close()

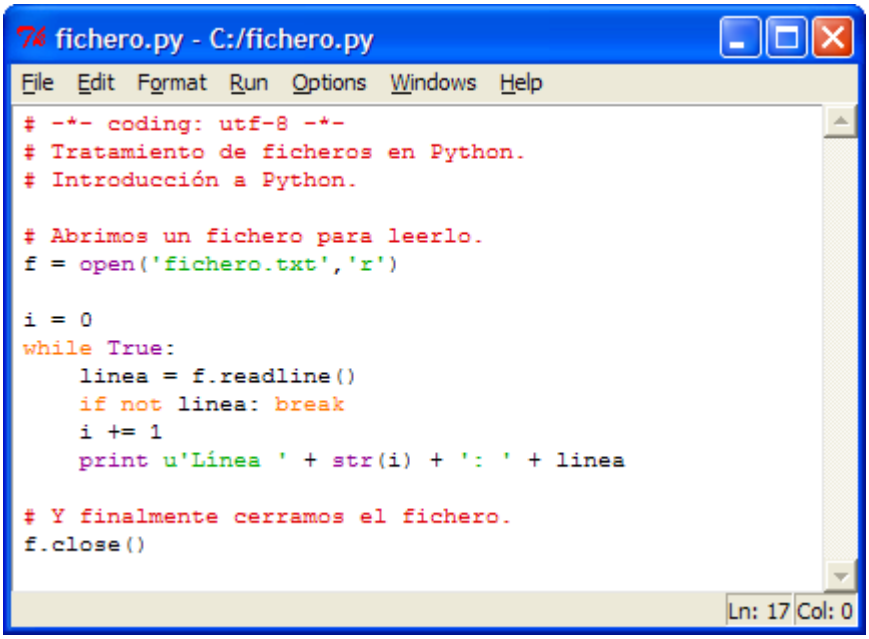
Hay otros métodos estándar para el tratamiento de ficheros (que encontramos en la mayoría de lenguajes) como seek, tell, truncate ó flush.

Pudiera ser que se necesitase abrir abrir un fichero con alguna codificación especial. Para ello está el módulo codecs. Por ejemplo, si se quiere abrir un fichero con codificación utf-8 se podría utilizar:

f = codecs.open('nombre_fichero',”rb”,”utf-8”)

Veamos ahora con ejemplos lo expuesto en el tratamiento de ficheros.


fichero.py - C:/fichero.py
File Edit Format Run Options Windows Help
# -*- coding: utf-8 -*-
# Tratamiento de ficheros en Python.
# Introducción a Python.
# Creamos un fichero.
f = open('fichero.txt','w')
# Definimos algunas cadenas de texto.
cadena = 'Python es el mejor lenguaje de programación'
# Guardamos dicha cadena en el fichero.
f.write(cadena)
# Y finalmente cerramos el fichero.
f.close()
Ln: 16 Col: 0


fichero.py - C:/fichero.py
File Edit Format Run Options Windows Help
# -*- coding: utf-8 -*-
# Tratamiento de ficheros en Python.
# Introducción a Python.
# Abrimos un fichero para leerlo.
f = open('fichero.txt','r')
i = 0
while True:
linea = f.readline()
if not linea: break
i += 1
print u'Línea ' + str(i) + ': ' + linea
# Y finalmente cerramos el fichero.
f.close()
Ln: 17 Col: 0

8.7. Más tipos en Python

En Python hay más tipos de datos, como conjuntos (set y frozenset), y otros que se pueden encontrar en módulos, como collections, time, datetime, array, UserDict, UserList, UserString, types, etc.

9. Conversiones entre listas, tuplas y tablas hashing

Se pueden hacer conversiones entre los tipos de datos más populares de Python, a saber, listas, tuplas y diccionarios. Para ello se utilizan las funciones integradas **dict**, **list** y **tuple**. Para ver este punto que mejor que hacerlo con un ejemplo:

```
l = ['Taller Caldum', 2011]
t = ('Universidad de Murcia','Python')
u = [(1,'Python'),(2,'wxPython')]
d = {1:'XML',2:'IronPython',3:'ReportLab'}
```

tuple(l**)** Devuelve ('Taller Caldum',2011)
list(t**)** Devuelve ['Universidad de Murcia','Python']
dict(u**)** Devuelve {1:'Python',2:'wxPython'}

d.**items()** Devuelve [(1,'XML'),(2,'IronPython'),(3,'ReportLab')]
d.**keys()** Devuelve [1,2,3]
d.**values()** Devuelve ['XML','IronPython','ReportLab']

Existe la función integrada **str()** para convertir cualquier parámetro en una cadena, si es posible. Por ejemplo:

str([1,2,3,4]**)** devuelve “[1,2,3,4]”

10. Formateo de cadenas

Normalmente cuando se definen cadenas que contienen variables, para formar otra cadena, puede utilizarse el operador **+** para concatenar todo y obtener el resultado deseado. Esta sería la primera forma de crear cadenas con datos variables. Esto es:

```
>>> lenguaje = 'Python'
>>> carac1 = 'multiparadigma'
>>> carac2 = 'multiplataforma'
>>> cadena = lenguaje + ' es un lenguaje ' + carac1 + ' y ' + carac2 + '.'
>>> cadena
'Python es un lenguaje multiparadigma y multiplataforma.'
>>>
```

La segunda forma, más elegante y eficiente es utilizar el operador **%**, de la siguiente manera:

```
cadena = '%s es un lenguaje %s y %s.' % (lenguaje,carac1,carac2)
```

% utiliza el flag s que corresponde a cadena (string). %s convierte cualquier tipo de argumento a cadena, ya que utiliza la función integrada **str()**. Darse cuenta de cómo se utiliza % y el paso de argumentos. Más elegante y sencillo de leer. El operador % también se utiliza para dar formato de salida a números.

Una tercera forma es utilizar el módulo **string**, el cual provee de un mecanismo para sustituir variables en cadenas *plantilla*. Su utilización es tal como sigue:

```
import string

p = string.Template('$lenguaje es un languaje $c1 y $c2.') Creamos la plantilla.

p.substitute({'lenguaje:'Python', 'c1':'multiparadigma','c2':'multiplataforma'})
Devuelve la cadena con las sustituciones hechas en la plantilla:

'Python es un lenguaje multiparadigma y multiplataforma.'
```

el método substitute lanza un error KeyError si no existe la clave correspondiente. Si se quiere ignorar esto se puede utilizar safe_substitute en su lugar. Por último cabe destacar que si se quiere utilizar en la plantilla el carácter $, tiene que ir precedido de $$.

Darse cuenta que las variables a sustituir comienzan con $. Los valores a sustituir son provistos mediante un diccionario vía substitute ó safe_substitute.

Una cuarta forma es utilizar el método de cadena **format()**.

11. Control de flujo

En Python hay 3 estructuras básicas de control de flujo de programa, que están en la mayoría de los lenguajes de programación que soportan el paradigma imperativo, a saber: la estructura condicional **if**, y las estructuras de bucles **for** y **while**.

11.1. Estructura de salto condicional if

La posibles sintaxis son:

if condición**:** <declaración>
elif condición**:** <declaración>

if condición**:** <declaración>
else: <declaración>

```
if condición:
        <declaraciones>
elif condición:
        <declaraciones>

if condición:
        <declaraciones>
else:
        <declaraciones>
```

Expresión condicional (no declaración)

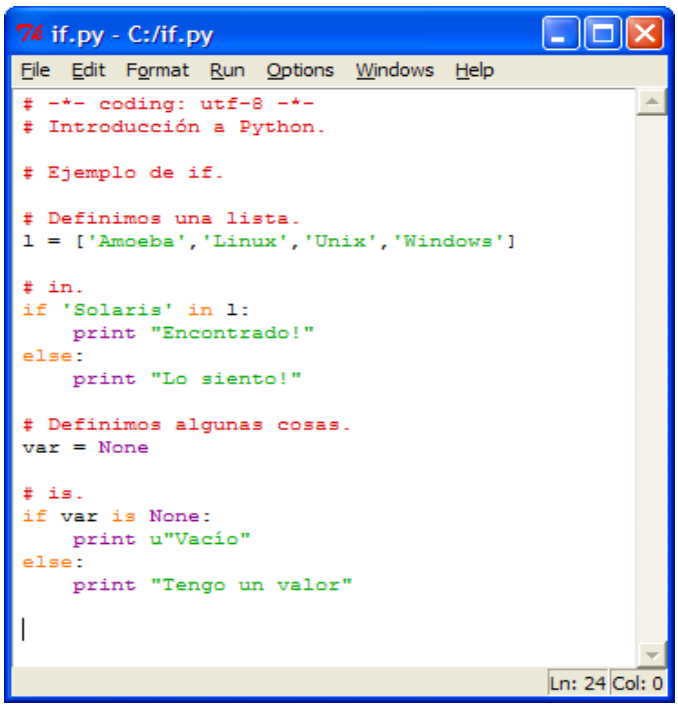
resultado = (ValorSiCierto **if** condición **else** ValorSiFalso)

es equivalente a:

```
if condición:
        resultado = ValorSiCierto
else:
        resultado = ValorSiFalta
```

NOTA: Los paréntesis no son obligatorios, aunque sí recomendables.

Veamos a continuación algunos ejemplos:



11.2. Estructura de bucle while

La sintaxis es la siguiente:

```
while <condición>:
        <declaraciones>
```

Para salir del bucle está la palabra clave **break**. Para forzar una nueva iteración se utiliza la palabra clave **continue**.

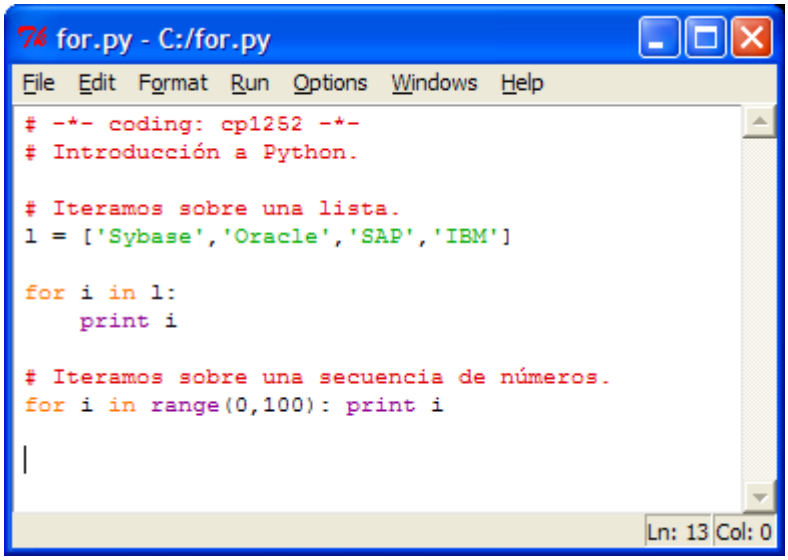
11.3. Estructura de bucle for

La sintaxis es:

for elemento in secuencia:
    <declaraciones>

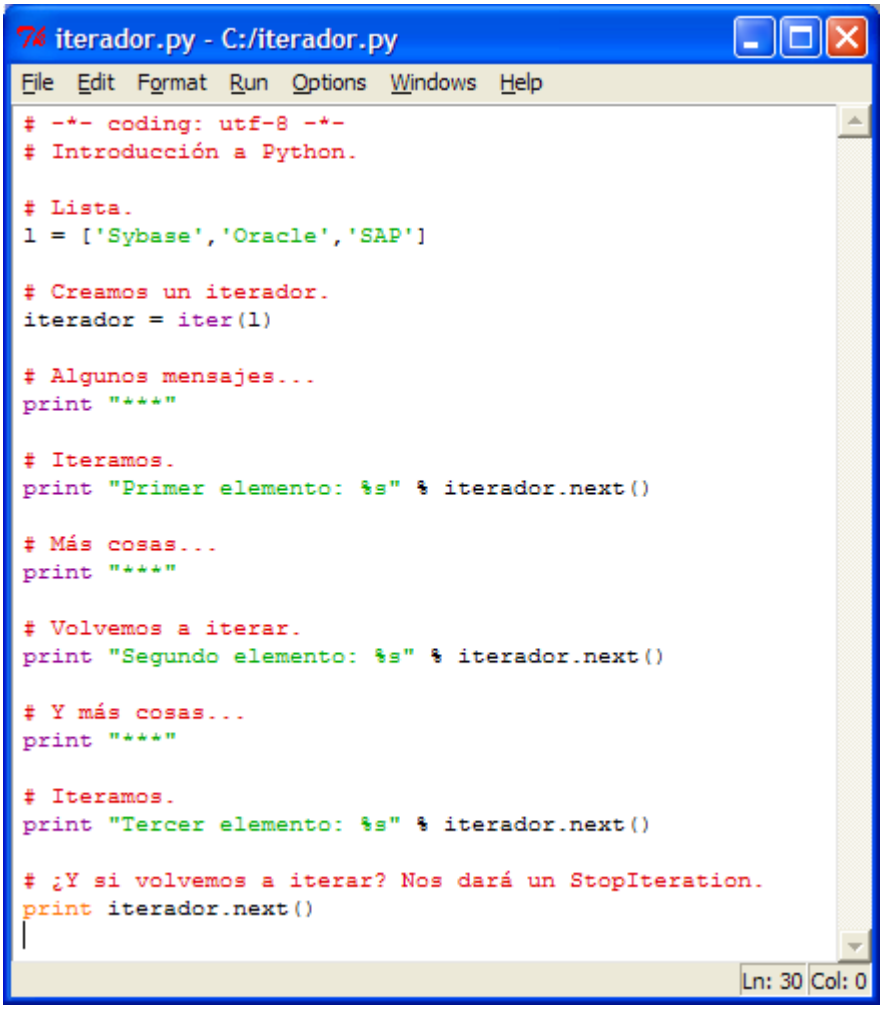
Lo que hace es iterar sobre la secuencia, asignando cada elemento de la misma a “elemento”. Se puede utilizar la función integrada **range** para iterar un número de veces. Al igual que con **while**, se utiliza **break** para salir incondicionalmente del bucle y **continue** para forzar una nueva iteración.

Veamos algunos ejemplos:



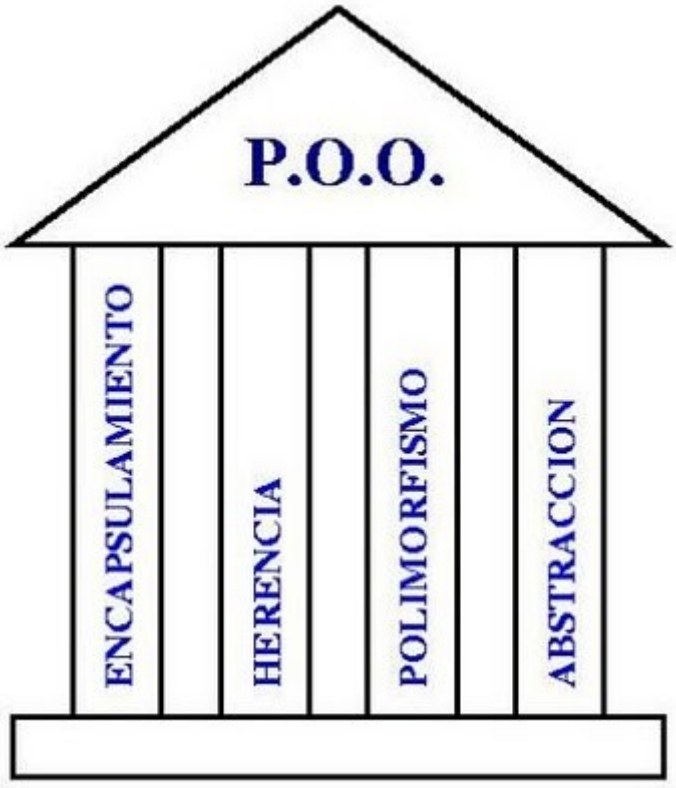
12. Iteradores

Un iterador enumera elementos de una colección. Es un objeto con un único método, denominado **next()**, que devuelve el siguiente elemento o lanza un error de *StopIteration*. Se puede crear un iterador mediante la función integrada **iter(**objeto**)**. Vamos a hacer algunos ejemplos de esto:



13. Programación orientada a objetos

La progamación orientada a objetos se basa en dos conceptos generales, el de **clase** y **objeto**, y 4 pilares fundamentales, a saber, **herencia**, **polimorfismo**, **encapsulación** y **abstracción**.



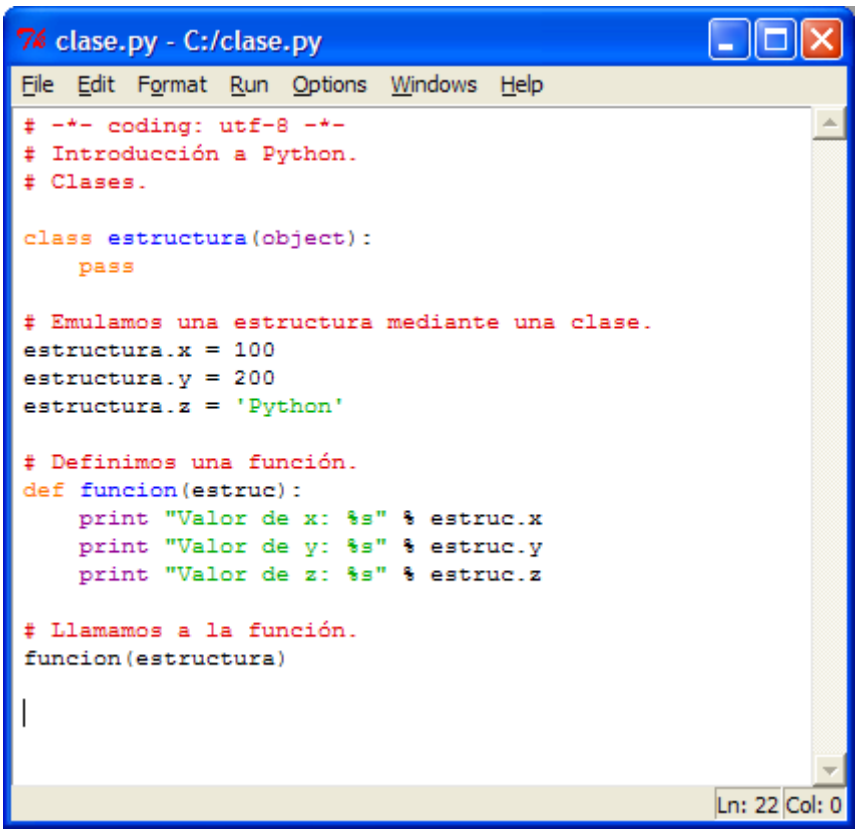
Python tiene herencia múltiple, no tiene polimorfismo entendido como sobrecarga de métodos (aunque haya formas de emularlo) y se puede implementar el encapsulamiento y la abstracción.

En Python todo es un objeto, y una clase siempre hereda del objeto **object**.

Una clase se define mediante:

```
class nombre_clase(object):
        <declaraciones>
```

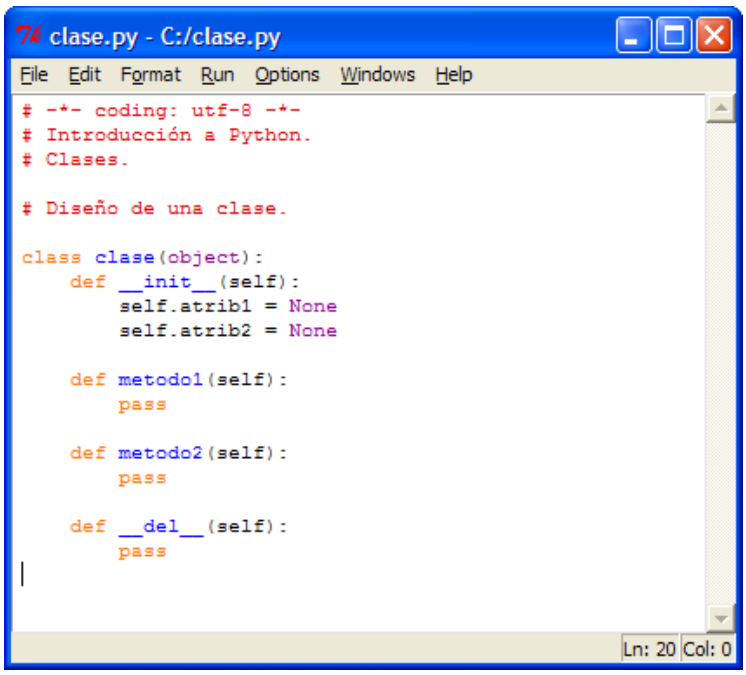
Las clases tienen atributos de clase, del mismo estilo que sucedía con las funciones. Vamos a ver esto con un ejemplo:



Normalmente una clase tiene atributos y métodos. En Python, un atributo se puede declarar en cualquier sitio, aunque lo propio sería en el método __init__, que aunque no es el constructor, sí que es el sitio donde habría que poner el código que ejecutaría el constructor.

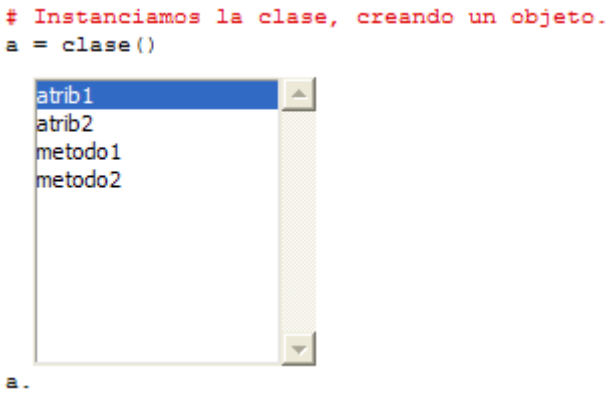
Los métodos especiales se inician y terminan con __. Los más destacados son:
__init__ Método donde incluiríamos todo que quisiéramos ejecutar en el constructor (no es el constructor).
__del__ Método destructor.

Veamos el siguiente ejemplo:

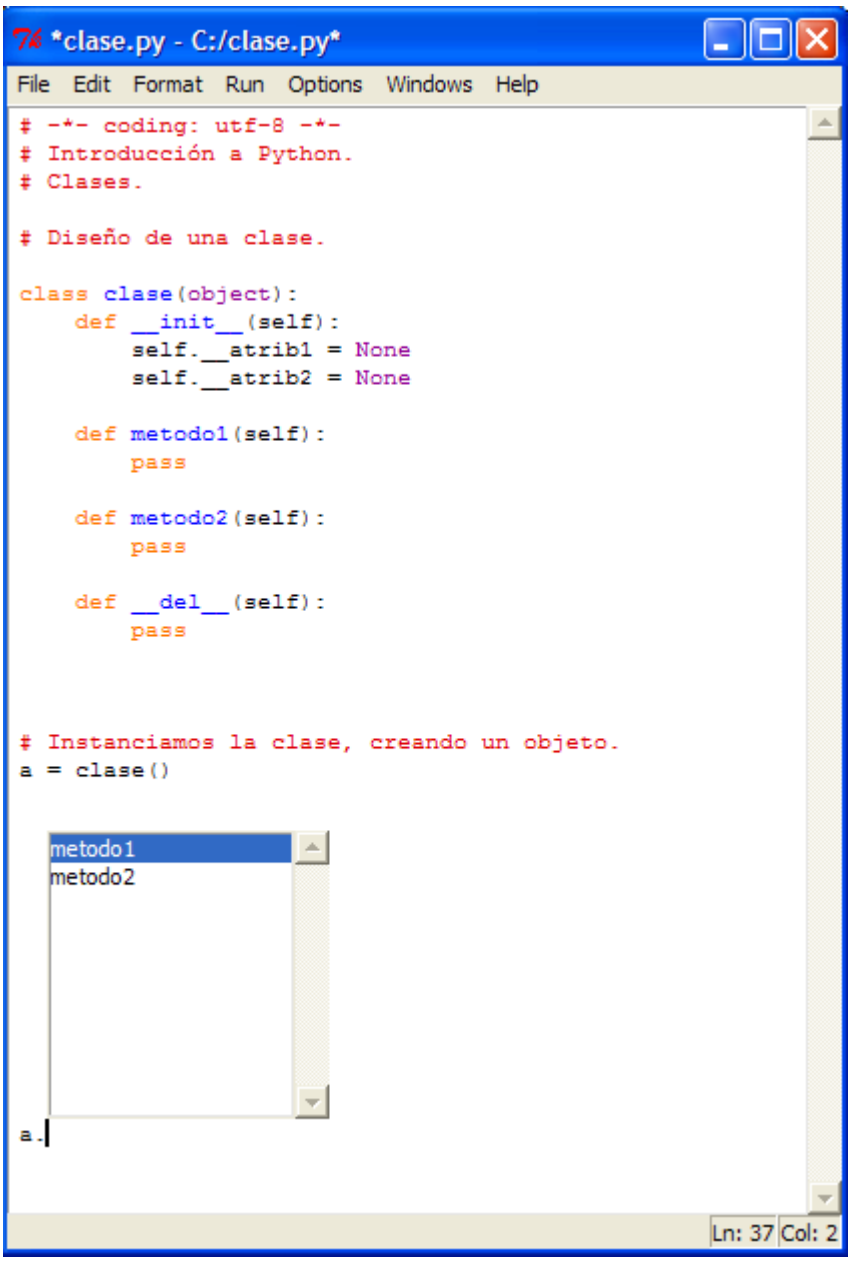


Los atributos siempre tienen que ir precedidos por **self**. Esto se hace para poder hacer referencia a ellos. Los métodos se definen con **def**. En los argumentos de los métodos siempre tiene que ir **self** como primer argumento. En el ejemplo la clase tiene 2 atributos inicializados a None y 2 métodos que no hacen nada (pass). Además se implementa el destructor.

Un objeto es la instanciación de una clase. Vamos a crear un objeto.

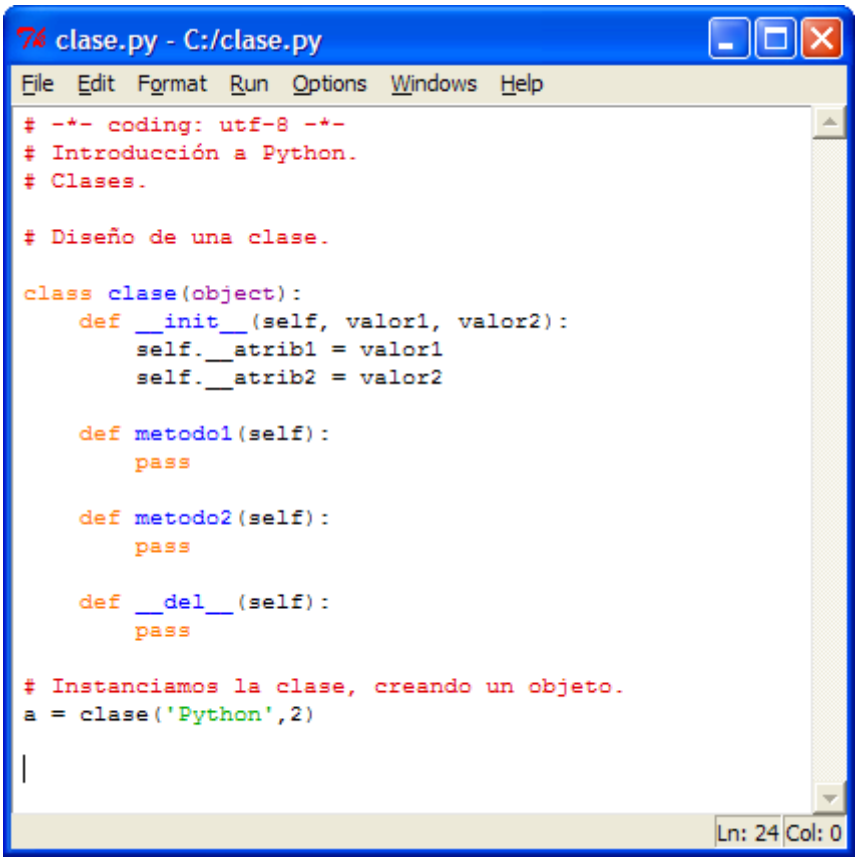


Como se puede observar son públicos tanto los atributos como los métodos. Podemos hacer atributos y métodos ocultos mediante el **name mangling**, que es incluir __ al principio del método ó atributo.



Ahora solo son públicos los métodos, ocultando los atributos. Darse cuenta que los métodos especiales también llevan __ al principio, por lo que son ocultos.

Los atributos de una clase se pueden inicializar en el momento de su instanciación, tal que así:



La ***encapsulación*** se considera una de las características definitorias de la orientación a objetos.

Cuando una clase existe (se define), se crean objetos a partir de ella, y se usan dichos objetos invocando los métodos necesarios. Es decir, creamos objetos para usar los servicios que nos proporciona la clase a través de sus métodos.

No necesitamos saber cómo trabaja el objeto, ni saber las variables que usa, ni el código que contiene.

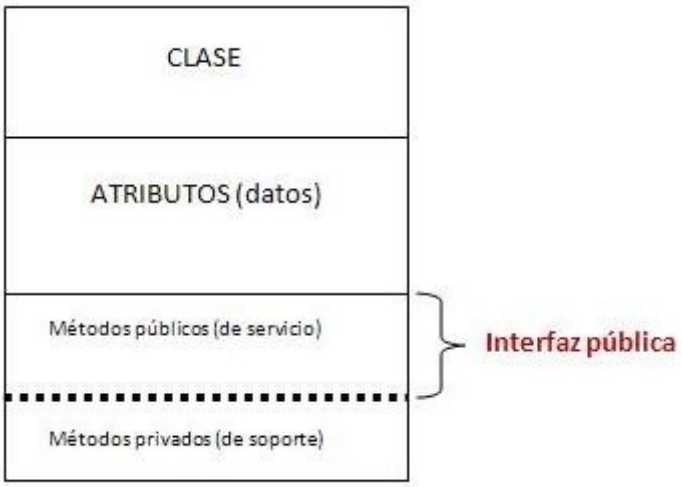
El objeto es una caja negra. **--> Modelo cliente - servidor**. Es decir, el objeto es un servidor que proporciona servicios a los clientes que lo solicitan.

La encapsulación describe el hecho de que los objetos se usan como cajas negras. Así, un objeto encapsula datos y métodos, que están dentro del objeto.

Interfaz pública de una clase: Es el conjunto de métodos (métodos de servicio) que sirve para que los objetos de una clase proporcionen sus servicios. Estos servicios son los que pueden ser invocados por un cliente.

Métodos de soporte: Métodos adicionales en un objeto que no definen un servicio utilizable por un cliente, pero que ayudan a otros métodos en sus tareas.

La ***encapsulación*** es un mecanismo de control. El estado (el conjunto de propiedades, atributos ó datos) de un objeto sólo debe ser modificado por medio de los métodos del propio objeto.



El concepto de encapsulamiento se apoya sobre el concepto de abstracción.

En POO solo necesitamos saber como interaccionar con los objetos, no necesitamos conocer los detalles de cómo está implementada la clase a partir de la cual se instancia el objeto. Sólo necesitamos conocer su interfaz pública.

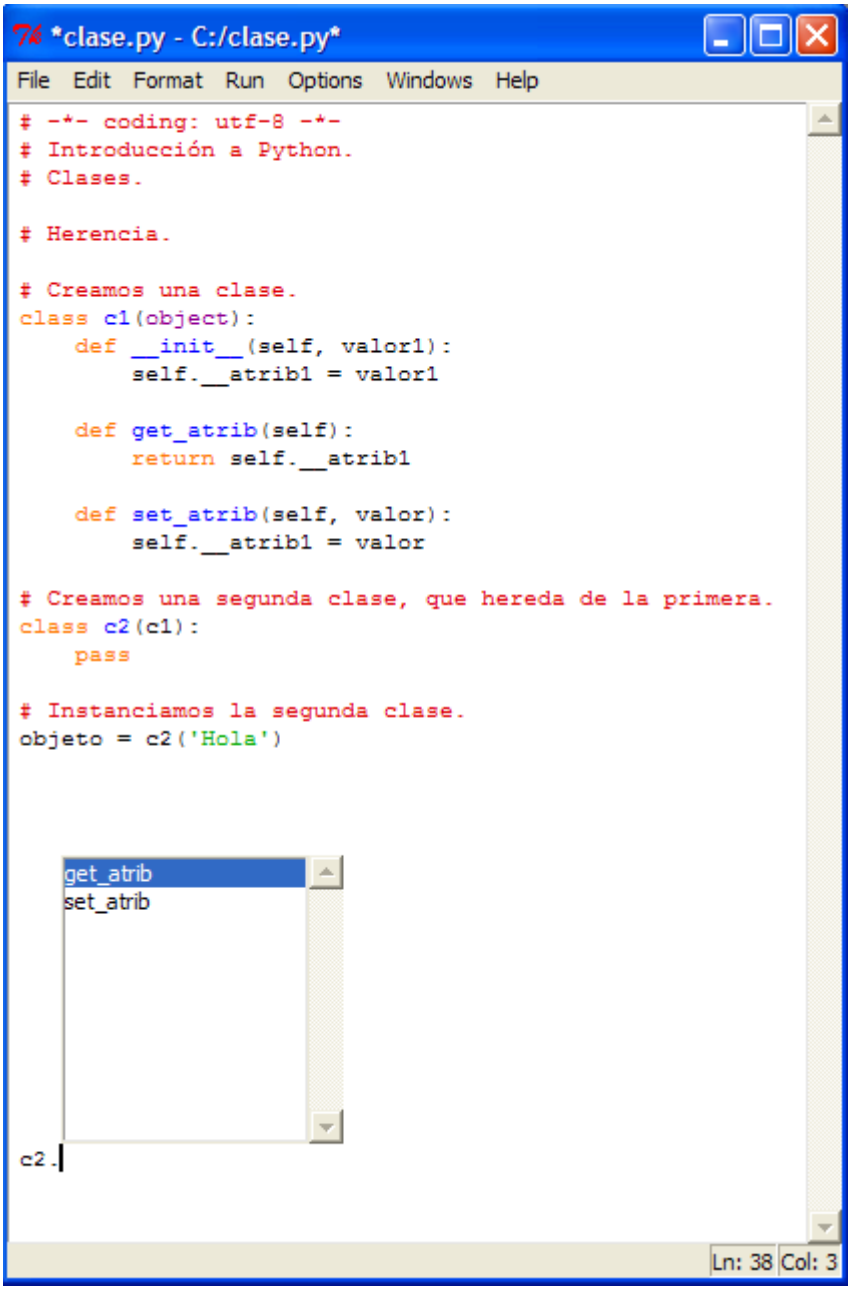
La encapsulación es una forma de abstracción.

La encapsulación es un mecanismo para llevar a la práctica la abstracción.

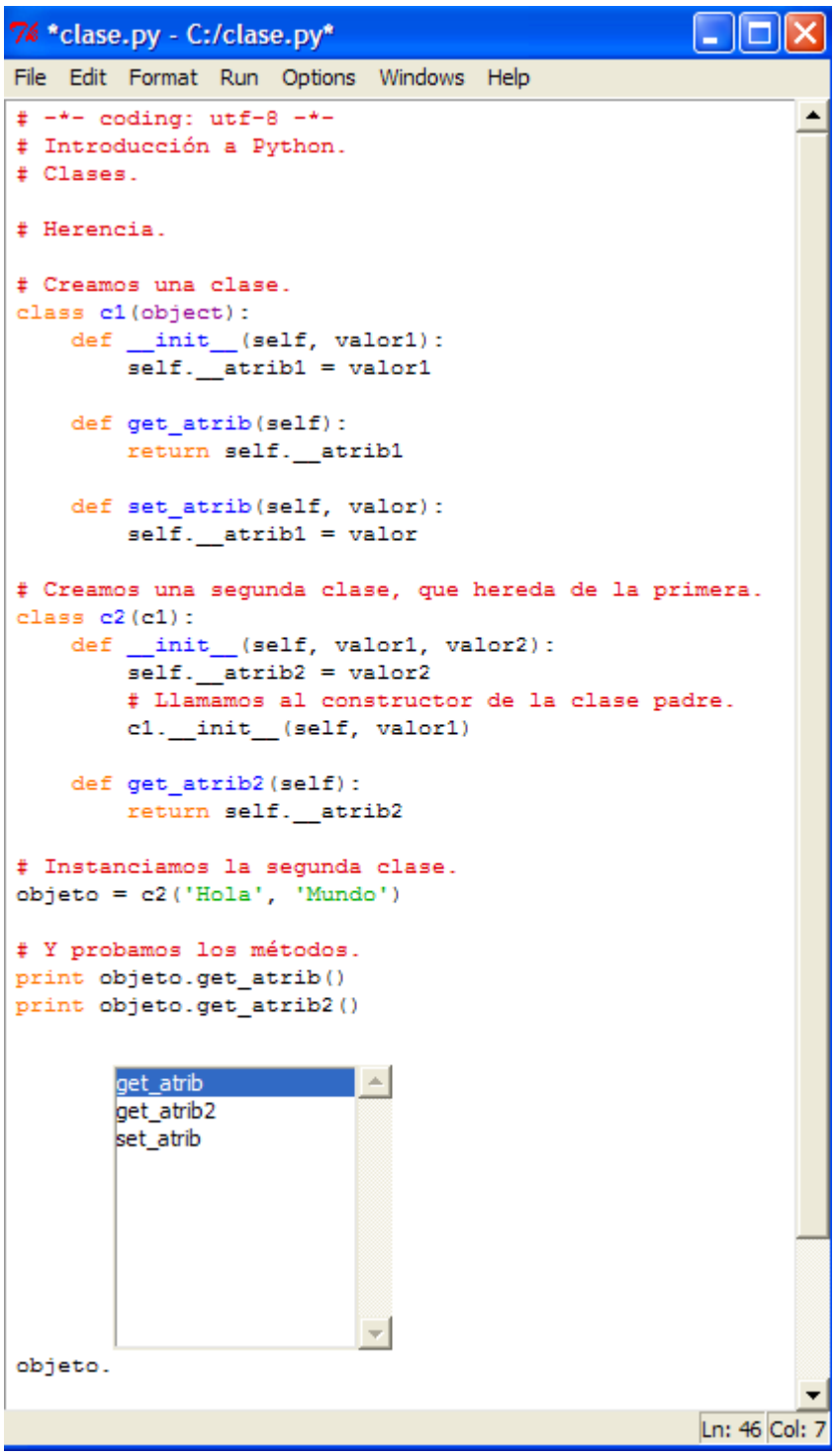
El nivel de abstracción puede ser bajo (en un objeto se manipulan datos y métodos individualmente), ó alto (en un objeto solo se usan sus métodos de servicio).

Por tanto para poner en práctica la encapsulación y abstracción se hace necesario diseñar una clase de la manera correcta, a saber: todos los atributos deben de ser ocultos, los métodos privados (o de soporte) también. únicamente los métodos son capaces de cambiar el estado de atributos, habiendo un conjunto de ellos que son públicos.

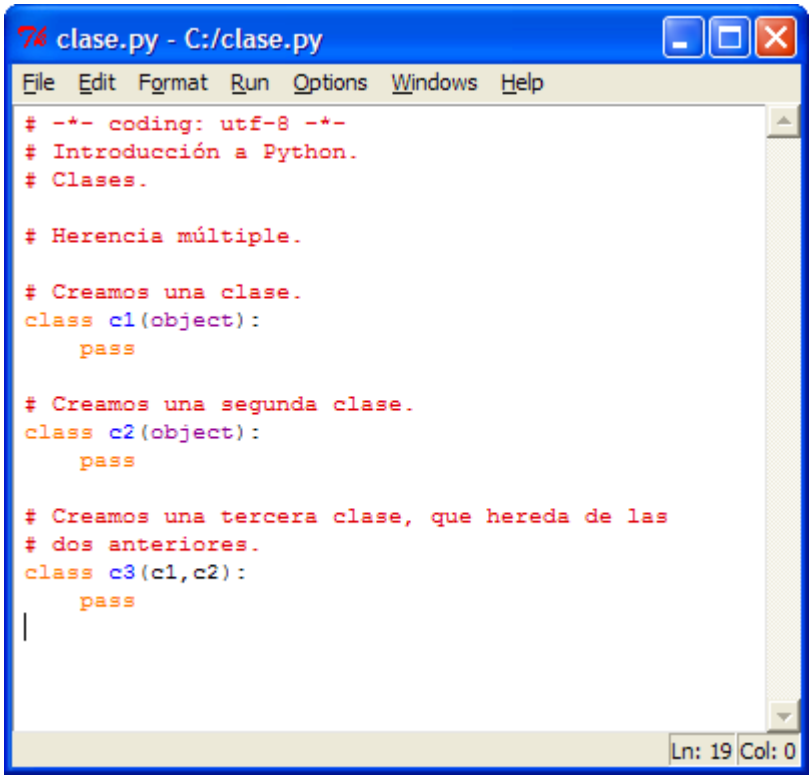
La herencia en Python es muy sencilla. Lo vemos:



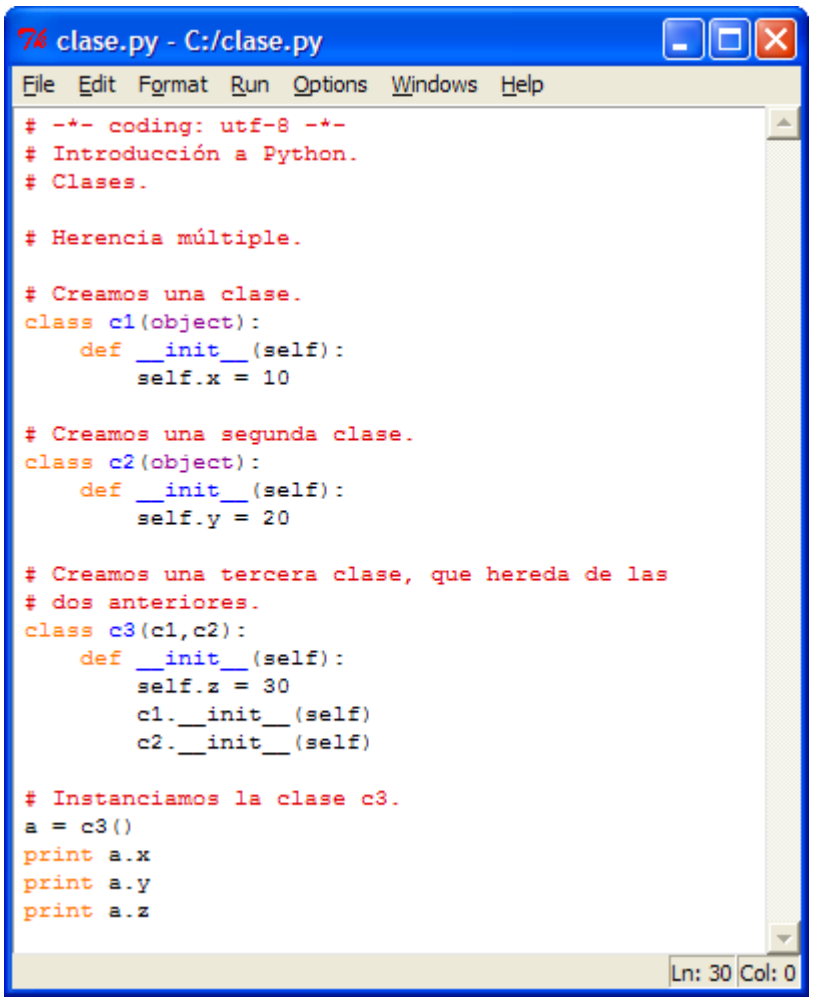
Si queremos incluir en la segunda clase, que hereda de la primera, algo en su __init__ tenemos que tener en cuenta que seguidamente hay que llamar al __init__ padre, ya que de lo contrario estamos haciendo override.


clase.py - C:/clase.py
File Edit Format Run Options Windows Help
# -*- coding: utf-8 -*-
# Introducción a Python.
# Clases.
# Herencia.
# Creamos una clase.
class c1(object):
def __init__(self, valor1):
self.__atrib1 = valor1
def get_atrib(self):
return self.__atrib1
def set_atrib(self, valor):
self.__atrib1 = valor
# Creamos una segunda clase, que hereda de la primera.
class c2(c1):
def __init__(self, valor1, valor2):
self.__atrib2 = valor2
# Llamamos al constructor de la clase padre.
c1.__init__(self, valor1)
def get_atrib2(self):
return self.__atrib2
# Instanciamos la segunda clase.
objeto = c2('Hola', 'Mundo')
# Y probamos los métodos.
print objeto.get_atrib()
print objeto.get_atrib2()
get_atrib
get_atrib2
set_atrib
objeto.
Ln: 46 Col: 7

Python soporta herencia múltiple, de la siguiente manera.



Si la clase que hereda de las dos anteriores modificamos su __init__ evidentemente tenemos que llamar ahí a los __init__ de las clases padres. La herencia múltiple es muy peligrosa si no se implementa con cuidado, ya que se puede hacer override de atributos ó métodos con el mismo nombre.


clase.py - C:/clase.py
File Edit Format Run Options Windows Help
# -*- coding: utf-8 -*-
# Introducción a Python.
# Clases.
# Herencia múltiple.
# Creamos una clase.
class c1(object):
def __init__(self):
self.x = 10
# Creamos una segunda clase.
class c2(object):
def __init__(self):
self.y = 20
# Creamos una tercera clase, que hereda de las
# dos anteriores.
class c3(c1,c2):
def __init__(self):
self.z = 30
c1.__init__(self)
c2.__init__(self)
# Instanciamos la clase c3.
a = c3()
print a.x
print a.y
print a.z
Ln: 30 Col: 0

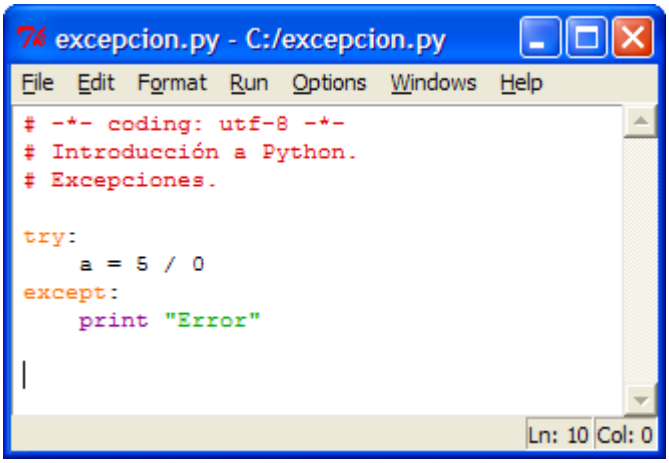
14. Manejo de excepciones

En Python el manejo de excepciones se trata con el bloque try/except. La sintaxis es la siguiente:

try:
    <declaraciones bloque1>
except:
    <declaraciones bloque2>

Si las declaraciones del bloque1 dan como resultado algún tipo de error, se salta incondicionalmente a ejecutar la declaración del bloque2.

Otro ejemplo del uso de este mecanismo.



Hay más opciones del manejo de excepciones, como **else**, **finally**, **as**, **with**, **tipos de excepciones**, que se deja al lector su estudio.

15. cPickle: persistencia de objetos

En Python se pueden serializar objetos, esto es, guardar un objeto que reside en memoria, a disco. Este proceso de serialización se puede hacer mediante un módulo, llamado **cPickle**.

La serialización de objetos se implementa en otros lenguajes con XML, Yaml, etc. Lo bueno que tiene cPickle es que es bastante rápido y muy fácil de usar.

La sintaxis para hacer un objeto persistente es:

import cPickle

cPickle.dump(x, fichero**)**

donde x es el objeto a serializar y fichero el manejador del fichero.

Si se quiere cargar el objeto del fichero a memoria:

x = **cPickle.load(**fichero**)**

Podríamos crear una clase genérica que lo que haga sea serializarse y cargarse a sí misma. De esta manera podríamos heredar de ella y daríamos una funcionalidad de serialización a todas las clases que heredaran de ella. Dicha clase podría ser esta:

```
import cPickle

class persistencia(object):
    def nombre_clase(self):
        return str(self).split(' ')[0].split('.')[1]

    def cargar(self, nombre_fichero = None):
        if nombre_fichero is None:
            nombre_fichero = self.nombre_clase()
        manejador_fichero = open(nombre_fichero,'r')
        objeto = cPickle.load(manejador_fichero)
        manejador_fichero.close()
        return objeto

    def salvar(self, objeto, nombre_fichero = None):
        if nombre_fichero is None:
            nombre_fichero = self.nombre_clase()
        # Fichero que contendrá el objeto.
        manejador_fichero = open(nombre_fichero,'w')
        # Volcamos el objeto de memoria al fichero.
        cPickle.dump(self, manejador_fichero)
        # Cerramos fichero.
        manejador_fichero.close()
```

16. Scripts en Python

En este apartado se presenta un ejemplo de script Python para contar caracteres en ficheros de texto pasados como parámetro. Se puede incluir flags, de manera que se puede configurar si se quiere contar caracteres especiales. Así, por ejemplo, si queremos contar los caracteres que un fichero, sin contar los espacios en blanco, simplemente haremos: python script.py fichero.txt -SPACE, donde es el script.py es el fichero que contiene el script y fichero.txt el archivo al cual queremos contar los caracteres.

Fichero **contar_caracteres.py**:

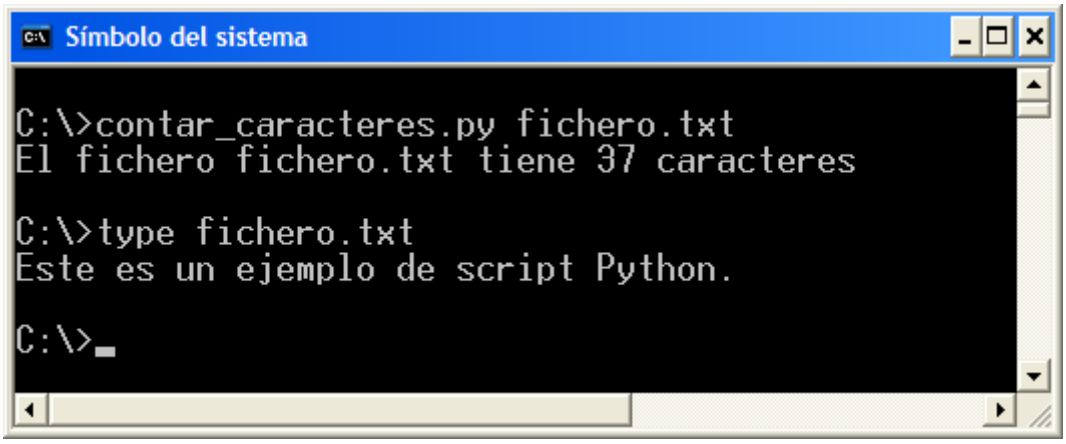
```
# -*- coding: utf-8 -*-

import sys

# Declaramos algunas constantes.
FF = '\f' # Form Feed
LF = '\n' # Line Feed
CR = '\r' # Carriage Return
TAB = '\t' # Tabulador
SPACE = ' ' # Blanco

try:
    excepcion = 0
    lista_excepcion = []
    fichero = sys.argv[1]
    opciones = sys.argv[2:]
    if '-FF' in opciones: lista_excepcion.append(FF)
    if '-LF' in opciones: lista_excepcion.append(LF)
    if '-CR' in opciones: lista_excepcion.append(CR)
    if '-TAB' in opciones: lista_excepcion.append(TAB)
    if '-SPACE' in opciones: lista_excepcion.append(SPACE)
    f = open(fichero,"r")
    caracteres = f.read()
    for i in caracteres:
        if i in lista_excepcion: excepcion += 1
    f.close()
    print "El fichero %s tiene %d caracteres" \
    %(fichero, len(caracteres) - excepcion)
except: print "Error al abrir el fichero."
```

Un ejemplo del uso de este script es el siguiente:



17. Programa de loterías, en Python

A continuación se presenta un programa muy sencillo, escrito en Python, para la generación aleatoria de combinaciones para juegos de azar. Se ha utilizado el módulo **random** para crear los números pseudoaleatorios.

```
# -*- coding: utf-8 -*-
import random, os

def combinacion(num_ele, total_num, repetir = False,ordenar_resultado = True):
    elementos = []
    if num_ele > total_num and not repetir:
        print "No puedes sacar más valores de los que ya tienes!"
        return

    aux = num_ele
    while aux > 0:
        numero = int(random.uniform(1,total_num))
        if repetir:
            elementos.append(numero)
            aux = aux - 1
        else:
            if elementos.count(numero) == 0:
                elementos.append(numero)
                aux = aux - 1

    if ordenar_resultado:
        elementos.sort()
    print elementos

def menu():
    print '''
================================
      MENU  PRINCIPAL
================================

Este programa genera combinaciones
de juegos de azar. Elige el juego
que más te guste.

1) Lotería primitiva
2) Euromillones
9) Salir
'''

    opcion = raw_input("")
    return opcion

def aplicacion():
    os.system('cls')
    opcion = ""
    while opcion != "9":
        opcion = menu()
        if opcion == "1":
            print ""
            print "Combinación para Lotería Primitiva: "
            combinacion(6,49)
            print ""

        if opcion == "2":
            print ""
            print "Euromillones"
            print "Combinación ganadora: "
            combinacion(5,50)
            print "Estrellas: "
            combinacion(2,9)
            print ""

# Ejecutamos el programa.
aplicacion()
```

18. Una agenda en Python

En este apartado vamos a ver como crear una agenda personal minimalista, en Python, que controla los nicks de personas repetidas, y que utiliza persistencia de objetos para guardar en disco.

```
# -*- coding: utf-8 -*-
# Creación de una agenda simple, en Python.
# Introducción a Python.

# Definimos un menú principal.
def menu():
    texto = u'''
AGENDA v0.0.1 - Taller CALDUM Febrero 2011

Agenda de contactos de personas.

Elige opción:

    1) Alta.
    2) Baja.
    3) Modificación.
    4) Listar.
    5) Guardar.
    0) Salir.
    '''
    print texto
    opcion = raw_input("Elige opción: ")

    #Devolvemos opción.
    return opcion

# Definimos una clase para implementar la persistencia.
import cPickle

class persistencia(object):
    def nombre_clase(self):
        return str(self).split(' ')[0].split('.')[1]

    def cargar(self, nombre_fichero = None):
        if nombre_fichero is None:
            nombre_fichero = self.nombre_clase()
        manejador_fichero = open(nombre_fichero,'r')
        objeto = cPickle.load(manejador_fichero)
        manejador_fichero.close()
        return objeto

    def salvar(self, objeto, nombre_fichero = None):
        if nombre_fichero is None:
            nombre_fichero = self.nombre_clase()
        # Fichero que contendrá el objeto.
        manejador_fichero = open(nombre_fichero,'w')
        # Volcamos el objeto de memoria al fichero.
        cPickle.dump(self, manejador_fichero)
        # Cerramos fichero.
        manejador_fichero.close()

# Creamos una clase agenda.
class mi_agenda(persistencia):
    def __init__(self):
        self.__lista = []

    def alta(self, persona):
        nick = persona[0].lower().strip()
        for i in self.__lista:
            if i[0].lower().strip() == nick:
                print "%s ya existe" % (nick)
                return
        self.__lista.append(persona)
        print "Alta correcta"
```

```
    def baja(self, nick):
        for i in self.__lista:
            if i[0].lower().strip() == nick.lower().strip():
                self.__lista.remove(i)
                print "%s se ha borrado" % (nick)
                return
        print "%s no existe" % (nick)

    def modificacion(self, persona):
        nick = persona[0]
        for i in self.__lista:
            if i[0].lower().strip() == nick.lower().strip():
                self.__lista.remove(i)
                self.__lista.append(persona)
                print "%s se ha modificado" % (nick)
                return
        print "%s no existe"

    def lista(self):
        return self.__lista

# Cargamos la agenda en memoria.
agenda = mi_agenda()
try: agenda = agenda.cargar()
except: pass

# Script principal de la aplicación.
while True:
    opcion = menu()
    # Salimos
    if opcion == '0':break
    # Alta.
    if opcion == '1':
        buffer_alta = []
        for i in ['Nick:','Nombre:','Apellidos:','Teléfono:']:
            dato = raw_input(i)
            buffer_alta.append(dato)
        # Intentamos dar de alta.
        agenda.alta(buffer_alta)
    # Baja.
    if opcion == '2':
        nick = raw_input('Nick:')
        agenda.baja(nick)
    # Modificación.
    if opcion == '3':
        buffer_modif = []
        for i in ['Nick:','Nombre:','Apellidos:','Teléfono:']:
            dato = raw_input(i)
            buffer_modif.append(dato)
        # Intentamos modificar.
        agenda.modificacion(buffer_modif)
    # Listar.
    if opcion == '4':
        lista = agenda.lista()
        for i in lista: print i
    # Guardar.
    if opcion == '5':
        agenda.salvar(agenda)
```

19. Tipos de ficheros Python

En Python existen varios tipos de ficheros. El primero y más importante, es el de extensión **.py** (fichero de código fuente).

Si en el directorio donde están los ficheros fuente .py de la aplicación se encuentran además ficheros con el mismo nombre y extensión **.pyc**, estos son las versiones “byte compiled” de nuestos ficheros .py. Si se modifican los ficheros fuente automáticamente se vuelven a generar los ficheros .pyc, esto es, no tenemos que hacer nada con los ficheros compilados .pyc.

Si se invoca Python con la opción **-o** se genera código optimizado en ficheros **.pyo** (byte code optimizado). En este caso se igoran los ficheros .pyc. Esta opción no hace mucho (eliminar los **assert**).

Un programa no se ejecuta más rápido si se lee desde ficheros .pyc ó .pyo que cuando se lee de ficheros .py. Lo único que es más rápido al usar .pyc ó .pyo es la velocidad con la que se cargan.

Los fichero **.pyd** son en realidad DLL's de Python.

Si se quiere ejecutar código Python en un entorno gráfico de modo que la consola de texto no aparezca, podemos renombrar el fichero principal .py a **.pyw**, y ejecutarlo. La consola de texto no aparecerá, solo la aplicación gráfica.

20. Módulos más importantes

No hay en Python un módulo más importante que otro, ya que dependiendo de nuestras exigencias habrá algunos imprescindibles y otros algo más livianos. Sin embargo, es cierto que hay módulos cuyo uso es muy común. Pasamos a enumerarlos:

os: Módulo interfaz con el sistema operativo (Windows, posix, Mac). En el espacio de nombres **os.path** encontramos métodos para operaciones de nombres de rutas. Ejemplos:

os.system(comando**)** Llamada al sistema para ejecutar comando.
os.startfile(f**)** Abre el fichero f con el programa asociado en el sistema.

re ó sre: Módulo de operaciones de expresiones regulares.

datetime y time: Módulos de tratamiento de tiempo y fechas.

shutil: Módulo de operaciones con ficheros de alto nivel, tales como copia, borrado (ficheros, directorios).

sys: Módulo de parámetros y funciones específicas del sistema. Ejemplos:

sys.platform Devuelve el sistema operativo.
sys.stdout, stdin, stderr Estándar entrada, salida, error.
sys.argv[1:] Devuelve lista de parámetros en la línea de comandos.

math y cmath: Módulo de operaciones matemáticas.

Se insta al novel en Python (si estás leyendo esto, es que es así), que investigue sobre estos módulos y haga pruebas sobre su funcionamiento. Hay muchos más, pero para empezar a indagar, están los más importantes.

En la web **Cats Who Code** hay una recopilación de los 50 módulos de extensión más importantes:

http://www.catswhocode.com/blog/python-50-modules-for-all-needs

21. Recursos documentales en Internet

Python para todos: Manual de Python en español de Raúl González Duque.

Packt Publishing: Editorial técnica de informática, que incluye una sección para libros sobre Python.

Proyecto de Documentación de Python en castellano en Sourceforge.

Comunidad de Python en Argentina. Enorme cantidad de recursos documentales de Python en castellano.

Sitio web oficial de Python. Todo tipo de información sobre Python, además de la última versión de este software.

Comunidad Plone. Comunidad Plone en España.

FreeTechBooks: Sitio web donde podemos encontrar libros sobre Python gratuitos y online.

QuickReferences: Guías de referencias rápidas de varios lenguajes de programación. Ideales para descargar e imprimirlas como ayuda rápida.

Richard Gruet's Python page: Guías rápidas para Python, en varias versiones.

Planeta Python: Aquí nos encontramos con toda la relación de lugares web, de primer nivel, en relación con Python.

Alltop: Noticias sobre Python.

IronPython URLs: Sitio web que agrega artículos, noticias y links sobre IronPython, la versión de Python escrita en C# para acceder al framework .NET de Microsoft.

Cheat-Sheets.org: Guías de referencia rápida. Multitud de lenguajes de programación.

Python portable: Sitio web en donde se nos presenta la forma en la que se puede configurar Python en un USB para ejecutarlo directamente.

Dive into Python 3: Guía de Python 3. Importante ya que Python 3 no es compatible con las versiones 2.

Blog de la historia de Python: Blog de Guido.

Mundo Python: Blog de Python, tutoriales y formación.

Python Links: Conjunto de links sobre programación en Python.

SQLAlchemy: Página oficial del ORM más potente de Python.

ActiveState Code Recipes: Recetas de problemas resueltos en Python.

Python entre todos nº 1: Revista sobre Python de PyAr.

Página personal de Stephen Ferg: Sitio personal del señor Stephen Ferg.

Introducción a Tkinter: Manual del framework gráfico Tkinter

Índice de paquetes para Python: Lugar donde se pueden encontrar el listado oficial de paquetes disponibles para Python (módulos de extensión).

Tutoriales de programación y ejemplos de código fuente: Gran variedad de lenguajes de programación, incluido Python. Ver: Ejemplos de código fuente Python organizados por temas.

Tutoriales de Python

Herramientas de desarrollo para Python: Lista de herramientas para desarrollar en Python

Recursos documentales para Python en español: Material, webs y blogs de recursos documentales sobre Python, en español.

Tutoriales para Python - Centro de aprendizaje: Sitio web, en inglés, en donde podemos encontrar recursos para aprender Python, inlcuyendo niveles básico y avanzado.

Tutorial Python: Tutorial, en español, de Python. Creado por PyAr.

El blog de Jesse Noller: Blog de programación Python.

El blog de Mike Driscoll: Blog de un gurú de Python.

Página de Spinecho: Código fuente de aplicaciones Python. Para aprender.

Linux Plátanos: Sitio web en donde encontrar temática GNU y Python.

El libro de Django: El libro online de Django. En inglés.

DABO: Sitio del software Dabo, para crear de manera rápida aplicaciones de escritorio en Python y wxPython.

CAMELOT: Sitio del software Camelot, para crear de manera rápida aplicaciones de escritorio en Python, Qt y SQLAlchemy.

Tutorial de sizers: Tutorial, en inglés, de uso de sizers (algoritmo de posicionamiento de widgets) en wxPython.

La web de Andrea Gavanna: Sitio de *Andrea Gavanna*, uno de los desarrolladores de widgets para wxPython.

Web de Andy Bulka: Software de patrones de diseño y UML**.**