Tutorial de Python 3 en Windows

Quico Saval Vicente
Última modificación: jueves, 29 de noviembre de 2012

1. Introducción

Este tutorial ha sido creado siguiendo estos videotutoriales en inglés sobre este lenguaje de programación.

Mi única labor ha sido traducirlos al castellano y juntarlos de forma que sean accesibles a personas ciegas.

2. Instalación

Lo primero que haremos será instalar el lenguaje y un editor para programar cómodamente, ya que mi experiencia me dice que el que viene con el lenguaje no es accesible. A continuación están los enlaces a los instaldores:

- El lenguaje Python 3
- El editor EdSharp

No hablaré de la instalación de estos programas porque es común a la de cualquier otra aplicación en Windows.

Lo único que reseñaré es una pequeña configuración que hay que hacer en el editor EdSharp. Lo que hay que hacer es ir a la barra de menú, concretamente a la opción "misc". Ahí se escogerá la opción "manual options". En el diálogo que se abra hay que pulsar el botón "main". Entre todas las líneas que aparecen hay que buscar la que dice:

```
YieldEncoding=""
```

Y cambiarla por esta otra:

```
YieldEncoding="UTF-8"
```

Hecho esto ya se puede guardar el documento con CTRL+s.

Con esto ya está preparado para seguir el tutorial. Le recomiendo que se asegure de tenerlo todo correctamente configurado, porque las cosas se entienden mucho mejor con la práctica.

3. Nuestro primer programa

A continuación escribiremos nuestro primer programa. Será muys encillo. Se abirrá, mostrará un mensaje en la pantalla y se cerrará. Sin embargo, nos será muy útil, ya que nos permitirá hablar de las dos formas que utilizaremos para ejecutar cualquier cosa en Python.

3.1. La consola de Python

Es un modo muy eficiente de probar lo que hace alguna instrucción. Se le recomienda que lo utilice tanto en los próximos capítulos como cada vez que esté programando y tenga alguna duda.

Para abrir la consola, basta con buscar el submenú de Python entre todos los programas en el menú inicio y ejecutar la opción "Python (command line)".

Cuando la abra se le mostrará un mensaje que, principalmente, le informará sobre la versión que tiene instalada. Escriba lo siguiente y pulse intro:

```
print("Esto es una prueba.")
```

Si todo ha ido bien, el mensaje aparecerá en la pantalla. Fácil, ¿Verdad?

Lo que ha ocurrido es que ha usado la función "print", que se encarga de imprimir un mensaje en la pantalla.

Aprovechando esto vamos a realizar un pequeño cambio en la configuración para que no tenga problemas con los acentos y las eñes. Entre en el menú del sistema con alt+espacio y busque la opción propiedades. Allí, en la ventana correspondiente asegúrese de que la fuente es "lucida console". Acepte los cambios y, cuando Windows le pregunte, diga que quiere modificar el acceso directo. Esto hará que cualquier ventana de consola de Python que abra respete esta configuración.

Ahora vamos a ejecutar la función que nos permitirá salir de la consola. Escriba esto y pulse intro:

```
exit()
```

No se olvide de los paréntesis. Ya hablaremos de ello en su momento. Puede probar a escribirlo sin los paréntesis y verá cómo Python le da un mensaje de error en inglés.

3.2. Escribir, guardar y ejecutar un programa

Sin duda, la consola está muy bien. Usted escribe algo, pulsa intro e inmediatamente obtiene el resultado. Pero tiene un gran inconveniente. Todo ese trabajo no lo puede guardar en ningún sitio. Por eso, a continuación aprenderá cómo guardar el mismo programa que ha escrito antes en la consola, para ejecutarlo cuantas veces quiera.

En su disco duro cree una carpeta en la que irá guardando todos sus programas. Por ejemplo, programas.

Abra el editor EdSharp y escriba lo siguiente:

```
print("Esto es una prueba.")
input()
```

Guárdelo en la carpeta programas con el nombre "prueba.py". Esta extensión indicará que es un programa escrito en Python.

Existen dos formas de ejecutar el programa:

1. Desde EdSharp pulsando f5.
2. Haciendo doble click sobre el archivo (como cualquier otro programa).

La función "input" hace que no se salga del programa hasta que se pulse intro. Si no, el mensaje desaparecería tan rápido que no daría tiempo a verlo.

Los usuarios de JAWS deben hacer alguna de estas cosas para poder leer el mensaje:

- Conmutar el eco de pantalla a todo con insert+s.
- Llevar el cursor de JAWS a la posición del cursor del PC.
- Leer todo el diálogo con insert+b.
- Enviar la ventana al visualizador virtual con alt+insert+w. Una vez leída se sale con escape.

4. Números y operaciones matemáticas

Python maneja tres tipos de números. El primero son los números enteros (como 3 o 100). El segundo son los números decimales, a los que Python llama flotantes, (como 33.4). El último tipo son los números complejos.

Los números complejos se basan en J, que es la raíz cuadrada de -1. Esto se escribiría como 1j. El doble sería 2J y así sucesivamente.

Los operadores matemáticos que maneja Python son los siguientes:

Operación	Signo
Suma	+
Resta	-
Multiplicación	*
División	/
Potencia	**
Parte entera del cociente	//
Módulo (resto de la división)	%

5. Cadenas de caracteres

Las cadenas de caracteres son secuencias de caracteres (palabras o frases) que siempre van encerradas entre comillas. Existen dos tipos de comillas. Las simples (') y las dobles ("). Por lo tanto:

```
'Esto es una cadena de caracteres válida'
"Y esta es otra cadena de caracteres válida"
```

Lo importante es que se acostumbre a un tipo de comillas y que las utilice siempre.

5.1. La concatenación

La operación principal que se realiza con este tipo de datos es la concatenación, que no es más que la unión de varias cadenas de caracteres. Esto se realiza con el signo de suma. Así, esta operación:

```
"Hola"+"Pepito"
```

Daría como resultado la cadena "HolaPepito". Por ello hay que poner un espacio en blanco al final de la primera cadena o bien al principio de la segunda.

6. Variables

De una forma muy informal, podemos decir que una variable es una especie de "caja" a la que usted asigna un nombre que contendrá cualquier tipo de dato que soporte Python (en particular números y cadenas de caracteres).

Asignar un valor a una variable es muy sencillo. Se escribe el nombre de la variable, después un signo igual y a continuación lo que contendrá. Tenga en cuenta que Python distingue entre las mayúsculas y minúsculas.

A la hora de asignar un nombre a una variable tenga en cuenta las siguientes reglas:

- El nombre debe empezar por una letra.
- No se permiten espacios, eñes ni caracteres acentuados.
- No se pueden usar palabras clave, es decir, palabras que signifiquen algo para Python.

Una vez que se ha asignado algo a una variable ésta puede ser usada en cualquier operación que se pueda llevar a cabo con ese dato. Veamos un ejemplo:

```
base=6
altura=5
area=base*altura/2
print("El área del triángulo es",area)
```

Se imprimirá: "El área del triángulo es 15.0".

Supongamos que usted crea una variable llamada **a** y que le asigna el valor 5. Si usted hace cualquier operación (por ejemplo a+2) la variable pasará a tener el nuevo valor y el antiguo se habrá perdido. Una manera de solucionar esto es haciendo una copia. Por ejemplo:

```
a=5
b=a
```

Puede probar a modificar el valor de a realizando cualquier operación y verá cómo se modifica el valor de **a** y cómo se mantiene el valor de **b**.

7. Imprimiendo

Como hemos visto en nuestro primer programa, la función print imprime cualquier cosa que se escriba entre los paréntesis. Ahora puede comprender mejor aquel programa:

```
print("Esto es una prueba.")
```

Lo único que se hace es imprimir una cadena de caracteres.

Con la función print se puede imprimir cualquier cosa:

```
print(15*2)
```

Y se imprime el número 30.

Para que se vaya familiarizando con el vocabulario de los lenguajes de programación, le diré que print es una función y lo que le mandamos a imprimir (lo que va entre paréntesis) se llama argumentos. En los dos ejemplos anteriores sólo le hemos pasado un argumento. En el primer caso era una cadena de caracteres y en el segundo era el resultado de una operación matemática. Cuando queremos pasar más de un argumento a una función debemos separar cada uno de ellos con una coma. Lo que hará print es imprimir cada uno de los argumentos que reciba insertando entre ellos un espacio en blanco. Veamos un ejemplo:

```
print("Hola","Pepito")
```

Y se imprime "Hola Pepito". Veamos otro ejemplo:

```
print("Pepito tiene",25,"años.")
```

Lo que se imprime es: "Pepito tiene 25 años".

Esto nos indica que se pueden imprimir juntos números y cadenas de caracteres. ¿Qué pasaría si lo hiciéramos así?

```
print("Pepito tiene "+25+" años.")
```

Lo que ocurre es que Python da un error porque no se puede sumar un número a una cadena de caracteres. Así que, si sólo se van a imprimir cadenas de caracteres se pueden separar como argumentos o se pueden concatenar. Pero si se van a imprimir cadenas de caracteres y números es preciso separarlos como argumentos.

Existe una función muy interesante llamada type. Esta función devuelve el tipo al que pertenece lo que se le pase como argumento. Esta información siempre se devolverá en inglés, por lo que se recomienda que se familiarice con este idioma.

8. Funciones

8.1. Introducción

Las funciones son bloques de código que se dedican a realizar determinadas tareas. Esto tiene la ventaja de que las tareas se programan una única vez. Existen dos tipos de funciones. Las internas, que son aquellas que ya tiene Python y las funciones definidas por el usuario. En apartados anteriores hemos visto tres funciones internas que son print, type y exit.

Por otra parte, también podemos agrupar las funciones según si reciben parámetros o no. Recuerde que si una función recibe parámetros éstos se escriben entre los paréntesis y que cada uno de ellos se separa del siguiente mediante una coma. Un buen ejemplo de ello es la función print.

```
print("David tiene",30,"años.")
```

En este caso a la función print le hemos dado tres parámetros:

- Una cadena. "David tiene"
- Un número entero. 30
- Otra cadena. "años"

Si lo piensa, print tiene que recibir parámetros obligatoriamente. Si no fuese así, ¿cómo sabría lo que debe aparecer en la pantalla? Del mismo modo type necesita un parámetro que, si recuerda, es el dato del cual queremos averiguar su tipo.

En cambio, la función exit no necesita parámetros, porque no se requiere de ninguna información para salir del programa.

En los próximos apartados se explicará cómo puede crear sus propias funciones, por lo que se recomienda que estos conceptos estén claros antes de seguir avanzando.

8.2. Nuestra primera función

Crear una función es muy simple. Esto es lo que debe hacer:

- Escribir la palabra clave "def". Esto indica que empieza la definición de una función. Si piensa en def como abreviatura de definición lo recordará fácilmente.
- Deje un espacio en blanco.
- Escriba el nombre que le va a dar a la función. Aquí se aplican las mismas reglas que a los nombres de las variables.
- Escriba el signo de abrir paréntesis, los parámetros que recibirá la función separados por comas y el signo de cerrar paréntesis. Aunque no haya parámetros debe incluir los paréntesis.
- Escriba el signo de dos puntos.
- En las siguientes líneas escribirá el cuerpo de la función.
- Deje una línea en blanco para terminar.

El cuerpo de la función son las instrucciones que se ejecutarán cada vez que se llame a esa función, es decir, la tarea que la función llevará a cabo.

Para que Python sepa lo que pertenece al cuerpo de la función y lo que no debe indentar las instrucciones del cuerpo, es decir pulsar el tabulador al menos una vez. EdSharp dirá "level 1" la primera vez, "level 2" la segunda y así sucesivamente. Pulsando alt+i se le dirá en qué nivel de indentación está. Si dice "level 0" querrá decir que está al principio de la línea, es decir, que aún no ha pulsado ninguna vez el tabulador. Cuando se corrigen problemas de indentación, si pulsa el tabulador EdSharp indentará toda la línea independientemente de si el cursor está al principio de la misma o no.

Para ejemplificar todo esto, vamos a crear una función que se va a llamar "mensajes". Esta función no llevará parámetros e imprimirá dos mensajes. El primero dirá "hola" y el segundo dirá "Hasta siempre".

```
def mensajes():
      print("Hola")
      print("Hasta siempre")
```

Ahora debería quedarle más claro todo lo explicado anteriormente. Revise cómo se le ha dado nombre a la función, cómo se ha indicado que no lleva parámetros y las dos instrucciones que forman parte del cuerpo. Para ejecutarla simplemente escriba:

```
mensajes()
```

Es muy similar a cuando en la consola ejecutaba la función "exit()", que tampoco lleva parámetros.

8.3. Funciones con parámetros

Observe el código de esta función.

```
def suma(a,b):
      return a+b
```

Repasemos el código según lo visto en el apartado anterior.

- Aparece la palabra "def" y, por lo tanto, se define una función.
- La función se llama suma.
- El cuerpo de la función contiene una única instrucción.

Ahora explicaremos la parte nueva. Esta función recibe dos parámetros que se llaman "a" y "b". La sintaxis es la misma que cuando se llama a esta función. Recuerde cómo se imprimían varios parámetros con la función "print".

En el cuerpo de la función se utiliza la instrucción return. Return en inglés podría traducirse como devolver. La función devuelve, en este caso, la suma de los dos números que le indiquemos. Por ejemplo:

```
suma(5,7)
```

Devolverá 12.

Observe cómo Python da un error si no se le pasan los parámetros que requiere la función.

8.4. Funciones con valores por defecto

Vamos a hacer una modificación a la función suma para que sólo tengamos que escribir un valor si lo deseamos:

```
def suma(a,b=1):
      return a+b
```

La modificación está en la definición de los parámetros. El usuario puede especificar los dos números que se van a sumar. Si sólo escribiese uno, entonces ese sería asignado a la variable "a", mientras que "b" valdría 1. Por ejemplo:

```
suma(5,2)
```

- a vale 5
- b vale 2
- se imprimirá 7

Veamos otro ejemplo:

```
suma(19)
```

- a vale 19
- b vale 1 (el valor por defecto)
- se imprimirá 20

Es importante tener en cuenta el orden. Primero debe escribir los parámetros que el usuario debe especificar y después los parámetros que lleven valores por defecto. Si no Python dará un error.

8.5. Variables locales

Un último aspecto que hay que tener en cuenta es la coincidencia de nombres de variables dentro y fuera de una función.

Las variables que hayan sido creadas fuera de cualquier función se llaman globales. Sin embargo, las que se crean dentro de una función se llaman locales, porque sólo pueden ser modificadas por esa función.

Supongamos que definimos una variable llamada "x" con un valor de 42 y que, posteriormente, dentro de una función hacemos que "x" valga 36. Observemos lo que pasa cuando imprimimos x fuera de la función.

```
x=42
def funcion():
      x=36
      print(x)

print(x)
funcion()
print(x)
```

Fuera del cuerpo de la función, cuando imprimimos "x" da 42, pero cuando llamamos a la función "x" vale 36. Esto es porque dentro de la función Python ha creado una variable local llamada "x". Como esta variable sólo se puede ver y modificar dentro de la función no entra en conflicto con la variable global x (la que vale 42). Si hubiéramos querido modificar la variable global "x" debíamos haber escrito esto:

```
x=42
def funcion():
      global x
      x=36
      print(x)

print(x)
funcion()
print(x)
```

Por tanto, tenga cuidado con los nombres que asigna a las variables y use la palabra "global" a su conveniencia para evitar problemas.

9. Módulos

Los módulos son conjuntos de variables y funciones que podemos agregar a Python en aquellos programas en los que necesitemos. Veamos un ejemplo.

Supongamos que en un programa necesitamos calcular la raíz cuadrada de un número. Esto se lleva a cabo con la función llamada "sqrt". Probemos:

```
sqrt(16)
```

Python, en lugar de calcular la raíz cuadrada de 16 y devolver 4 dice que "sqrt" no está definido. Esto se debe a que esta función no está disponible por defecto como sucedía con las que hemos estado viendo hasta ahora, sino que está dentro de un módulo que se llama "math" (matemáticas en inglés). Por tanto lo primero que tenemos que hacer es importar este módulo. Para ello escribiremos la palabra clave "import", dejaremos un espacio en blanco y escribiremos el nombre del módulo.

```
import math
```

Y ahora probemos a ejecutar la función:

```
sqrt(16)
```

Y nos da el mismo error que antes porque no le hemos dicho que busque esa función en el módulo correspondiente. Para ello se escribe el nombre del módulo, después un punto y después la función.

```
math.sqrt(16)
```

Finalmente, ya no se producen errores y la función devuelve el resultado que esperábamos.

Hay una forma de abreviarnos la sintaxis. Supongamos que para nosotros es mucho más comprensible (y por supuesto corta) la palabra "raiz2", que significa raíz cuadrada. Podemos crear una variable que contenga el nombre de la función de la raíz cuadrada y luego ejecutarla como si fuera una función predefinida de Python. La secuencia completa de instrucciones sería la siguiente:

```
import math
raiz2=math.sqrt
raiz2(16)
```

Además, existen otras dos posibilidades con la instrucción "import".

```
from math import sqrt, pi
```

Esta instrucción importaría únicamente la función "sqrt" y la constante correspondiente al número "pi".

```
from math import *
```

Con esta instrucción se importarían todas las funciones y constantes que contiene el módulo "math".

En cualquiera de estos dos casos, para calcular la raíz cuadrada tan sólo tendríamos que ejecutar la función normalmente:

```
sqrt(16)
```

La forma más segura de trabajar con los módulos es importándolos (como "import math"). De esta forma nos obligaremos a escribir el módulo en el que está la función y evitaremos una confusión por culpa de una duplicidad de nombres.

Para averiguar los módulos que tenemos disponibles cuando se instala Python, basta con ir a la carpeta en la que esté instalado (que por defecto se llama "Python32" y está en la raíz del disco duro) y, dentro de ella, entrar en la carpeta "lib".

Por último, cabe reseñar que la función "help" nos da información sobre cualquier nombre de función o módulo que se le pase entre paréntesis. Esta información está en inglés.

La información es tan extensa que no cabe en una única pantalla. Por lo tanto, se recomienda a los usuarios de JAWS que envíen la ventana al visualizador virtual con alt+insert+w. Una vez que se haya terminado de leer la ventana se sale del visualizador virtual con escape. Para cambiar de pantalla se usa la barra espaciadora y para salir de la documentación sin terminar de leerla se usa la "q".

10. Preguntando al usuario

La función input recibe como parámetro una pregunta que se quiera hacer al usuario y devuelve una cadena de caracteres conteniendo lo que el usuario haya escrito. Veamos un ejemplo:

```
nombre=input("Escribe tu nombre: ")
print("hola "+nombre+".")
```

Este programa esperará a que el usuario teclee su nombre y pulse intro. Entonces se imprimirá el mensaje.

Lo que teclea el usuario se vuarda como una cadena de caracteres. Si lo que esperamos es un número, habrá que convertirla usando una de estas dos funciones:

Función		Acción que realiza
int(cadena)	Convierte el contenido de la variable "cadena" a un número entero.	
float(cadena)	Convierte el contenido de la variable "cadena" a un	

	número decimal.	

Por lo tanto, podemos unir estos dos pasos en una sola línea. Se trata de escribir la función "int" o "float" según corresponda y, entre paréntesis, la función "input".

```
entero=int(input("Escriba un número entero: "))
decimal=float(input("Escriba un número decimal: "))
```

Ya podríamos hacer cualquier tipo de operación matemática con "entero" o con "decimal".

11. Instrucciones condicionales

11.1. Introducción

Hasta ahora, todos los programas que hemos escrito ejecutaban el código de principio a fin. Sin embargo, muchos de los problemas a los que nos enfrentamos requieren diferentes soluciones dependiendo del cumplimiento de alguna condición.

Por ejemplo, supongamos que nos dan la edad de una persona y queremos averiguar si es mayor o menor de edad. Suponiendo que la mayoría de edad se alcanza a los 18 años la solución al problema sería la siguiente:

- Se debe comparar la edad del usuario con 18.
- Si la edad es mayor o igual se debe decir que el usuario es mayor de edad.
- En caso contrario, se debe decir que el usuario es menor de edad.

Para esto, la sintaxis que usa Python es la siguiente:

```
if(condicion):
      lo que pasa si se cumple
else:
      lo que pasa si no se cumple
```

A continuación se muestran los principales operadores de comparación.

Operador		Descripción
==	Es igual a	
!=	Es distinto de	
>	Es mayor que	
>=	Es mayor o igual que	
<	Es menor que	

<=	Es menor o igual que

Como habrá notado al leer la tabla anterior, el signo que compara la igualdad de valores está formado por dos signos igual juntos. Recuerde que un único signo igual corresponde al operador de asignación, que ya fue explicado en el apartado correspondiente a las variables.

Ahora ya tenemos todo lo necesario para construir nuestro programa:

```
edad=int(input("Escriba su edad: "))
if(edad<18):
      print("Es menor de edad")
else:
      print("Es mayor de edad")
```

Si prueba este programa con edades como 16, 18 y 20 verá que la condición funciona perfectamente y que siempre se imprimirá el mensaje adecuado.

11.2. Evaluando múltiples condiciones

Es posible que necesitemos evaluar al mismo tiempo si se cumple una de entre muchas condiciones. Para ello contamos con la instrucción "elif" que nos permite ir evaluando condiciones dentro de una misma estructura. La sintaxis es la siguiente:

```
if(unacondicion):
      lo que pasa si se cumple
elif(otracondicion):
      si se cumple otra condicion
elif(yotradistinta):
      lo que pasa si se cumple la distinta
else:
      lo que pasa si ninguna es cierta
```

Supongamos que en un país existe educación obligatoria hasta los 16 años, edad a partir de la cual se puede trabajar. Supongamos también que una persona se debe jubilar a los 67 años. Supongamos que tenemos una variable "edad" que ya ha almacenado el valor introducido por el usuario. Veamos cómo sería la estructura condicional:

```
if(edad<16):
      print("Tiene que seguir estudiando.")
elif(edad<67):
      print("Puede trabajar")
else:
      print("Debe estar jubilado.")
```

Si una persona tiene 15 años se cumple la primera condición y se imprime el mensaje correspondiente. Si, por ejemplo tiene 45 saltaría a la segunda condición (porque la primera es falsa) e imprimiría el segundo mensaje. Finalmente, si una persona tiene 70 años ambas condiciones se evaluarían como falsas y se imprimiría el último mensaje.

12. Repitiendo código

En este apartado hablaremos de la estructura "while" que se encarga de repetir un cierto bloque de código mientras se cumpla una determinada condición. Su estructura es muy parecida a la primera parte de la estructura condicional.

```
while(condicion):
      código que se repite
```

Por ejemplo, supongamos que queremos contar los números de 1 a 5. El código sería:

```
x=0
while(x<5):
      x=x+1
      print(x)
```

En alguna ocasión puede ocurrir que no sepamos qué condición se está cumpliendo mientras se ejecuta este bloque repetitivo (llamado bucle) pero sí que sepamos con certeza el momento en el que debe finalizar. Para esto existen dos herramientas. La primera de ellas es la condición:

```
while(True):
```

La palabra "True" significa verdadero en inglés. Lo que estamos diciendo es que el código se repita siempre. Note que "True" está escrito en mayúscula. La otra palabra clave es "break", que en inglés significa romper. Esta instrucción hará que se salga del bucle cuando Python la encuentre.

A continuación reescribiremos el programa anterior usando estas dos características:

```
x=0
while(True):
      x=x+1
      print(x)
      if(x==5):
              break
```

Como puede observarse, ambos programas producen el mismo resultado.

13. Secuencias

Una secuencia es un tipo de datos muy importante en Python. Se trata de colecciones ordenadas de datos. Los principales tipos son listas, tuplas y cadenas de caracteres, que ya fueron vistas en el capítulo de variables.

13.1. Definición

Para definir una lista se escribe el nombre que recibirá, el operador de asignación y los valores que contendrá encerrados entre corchetes y separados por comas. La tupla se define exactamente igual sólo que encerrando los valores entre paréntesis. Ambas pueden contener tanto cadenas de caracteres como valores numéricos. Veamos un ejemplo de cada una de ellas.

```
lista=["manzanas","naranjas","peras","plátanos"]
tupla=(1,2,3,4,5)
```

La principal diferencia entre las listas y las tuplas es que las listas se pueden modificar, es decir, podemos añadir y eliminar elementos de la lista, mientras que las tuplas permanecerán invariables.

13.2. Obteniendo valores

Podemos imprimir un valor concreto de una tupla o de una lista. La sintaxis que usaremos es la siguiente:

```
listaotupla[indice]
```

El valor que hay que escribir entre corchetes es el número de la posición que ocupa el elemento que queremos imprimir. Nótese que siempre usaremos corchetes para este propósito independientemente de si trabajamos con listas o tuplas.

Es muy importante recordar que el primer elemento siempre será la posición 0. Así, si trabajamos con la lista y la tupla creadas en el apartado anterior pdemos realizar las siguientes operaciones:

```
tupla[0]
```

Imprimirá el valor "1" que es el primer elemento de la tupla.

También podemos imprimir los últimos valores.

```
lista[-1]
```

Imprimirá "plátanos" que es el último valor de la lista.

13.3. Porciones de una lista

Hasta ahora hemos visto cómo imprimir un único elemento de la lista. A continuación, veremos cómo imprimir un determinado número de elementos de la lista. Esta es la secuencia con la que trabajaremos:

```
numeros=[1,2,3,4,5,6,7,8]
```

Si para imprimir un único elemento pasábamos como referencia el índice de su posición, para imprimir una porción pasaremos dos índices separados por dos puntos.

```
numeros[1:7]
```

En este caso se imprimirá "2, 3, 4, 5, 6" porque en esta porción no se incluye el segundo índice. Es decir, que cuando ponemos "[1:4]" en realidad estamos pidiendo las posiciones 1, 2 y 3.

Si se omite el primer valor, se tomará como origen 0, que es el principio de la lista. De igual modo, si se omite el segundo valor se entenderá que la porción debe llegar hasta el final de la lista.

Opcionalmente, podemos incluir un segundo signo de dos puntos y un número que incluye el salto que debe dar Python al imprimir los elementos. Por defecto es 1, es decir, que imprimirá elementos de forma consecutiva. Por ejemplo:

```
numeros[::2]
```

Imprimirá los números impares. Si quisiéramos los pares haríamos:

```
numeros[1::2]
```

14. Operando con listas

En los apartados anteriores hemos visto lo que son las secuencias y algunas operaciones que son comunes a todas ellas. A continuación, nos centraremos específicamente en las listas. Para ello, vamos a crear una lista con frutas:

```
frutas=["manzana","fresa","plátano","naranja"]
```

A partir de ahora trabajaremos con dos tipos de funciones. Unas serán funciones genéricas que reciben como argumento la lista. Por ejemplo:

```
print(frutas)
```

Las otras son específicas del tipo lista y por ello habrá que escribir el nombre de la lista, un punto y la función que queramos utilizar con sus argumentos correspondientes. Esta sintaxis es idéntica a la que se utiliza cuando se trabaja con módulos. Revise ese apartado si tiene alguna dificultad.

Podemos cambiar uno de los valores de la lista asignándolo como si fuera una variable poniendo el índice correspondiente. Por ejemplo:

```
frutas[2]="pera"
```

Si ahora imprimimos la lista veremos que tenemos manzana, fresa, pera, naranja.

Con la función "len" podemos averiguar cuántos elementos tiene la lista.

```
len(frutas)
```

En este caso tiene 4 elementos. Por lo tanto, como el primer elemento de la lista es el 0, el último no es el 4 sino el 3.

Supongamos que ahora queremos eliminar la fresa de la lista. Como ocupa el lugar 1 haremos lo siguiente:

```
del(frutas[1])
len(frutas)
print(frutas)
```

La primera función específica del objeto lista que vamos a ver es "append", que sirve para añadir un objeto al final de la lista.

```
frutas.append("ciruela")
```

Si imprimimos la lista, veremos que ahora el último elemento es "ciruela".

La función index sirve para conocer el índice de la primera vez que aparezca un elemento en la lista.

```
frutas.index("naranja")
```

En este caso devuelve 2.

Existen los operadores condicionales "in" y "not in" para preguntar si cierto elemento pertenece o no a la lista. Por ejemplo:

```
"pera" in frutas
"fresa" not in frutas
```

Ambas expresiones son ciertas, ya que la pera está en la lista pero la fresa no.

Añadamos la piña a la lista.

```
frutas.append("piña")
```

Podríamos eleminar la piña conociendo su índice como ya hicimos con el plátano. Sin embargo, podemos usar otra función:

```
frutas.remove("piña")
```

Esto eliminará únicamente la primera aparición de la piña en la lista.

Por último veremos la función "sort" que ordenará alfabéticamente nuestras frutas.

```
frutas.sort()
```

Si la lista está compuesta por cadenas de caracteres, éstas se ordenarán alfabéticamente. Si está compuesta por números, éstos serán ordenados de menor a mayor. Sin embargo, si están mezclados los números y las cadenas de caracteres no se puede aplicar esta función porque no existe un criterio para ordenar los elementos de esa lista.

15. Diccionarios

Los diccionarios son otro tipo de datos en Python. Se dedican a almacenar parejas de datos en los que uno es la clave y otro es el valor asociado a esa clave. Por ejemplo, un diccionario corriente también se comporta así, ya que para cada palabra (clave) existe una definición (valor).

Al igual que en un diccionario corriente la clave debe ser única, ya que la necesitamos para acceder al valor que tiene asociada. Además, la clave debe ser un dato mutable (una lista o una cadena de caracteres), mientras que el valor puede ser cualquier tipo de dato.

La sintaxis de los diccionarios es la siguiente. Van encerrados entre llaves ({ y }) y se escribe la clave, el signo de dos puntos y luego el valor. Cada pareja clave-valor es separada de la siguiente por una coma. Por ejemplo:

```
figuras={"cubo":"Figura geométrica que tiene seis
caras.","esfera":"Figura geométrica que no tiene caras, aristas ni
vértices."}
```

Aquí hemos creado un diccionario con dos datos. Las claves serían cubo y esfera y cada una de ellas lleva un valor asociado que, en este caso, es una posible definición.

Para acceder a cualquiera de los elementos se hace de una forma muy similar a las secuencias. Aquí, en lugar de escribir el número de posición que ocupa el dato escribiremos la clave:

```
figuras["cubo"]
```

Esta misma sintaxis se puede utilizar para agregar un nuevo valor. Se debe escribir el nombre del diccionario, entre corchetes la nueva clave, el operador de asignación y el valor que llevará asociado.

```
figuras["pirámide"]="Figura geométrica que posee una cúspide."
```

Además, con esta sintaxis se puede cambiar el valor para una clave.

```
  figuras["pirámide"]="Los egipcios te dirán mejor que nadie lo que
es."
```

Al igual que con las listas podemos usar los operadores "in" y "not in" para comprobar si cierta clave está en el diccionario. Además, podemos usar la función "del" para borrar un elemento del diccionario. En el argumento se escribiría entre corchetes la clave a eliminar.

```
  del(figuras["pirámide"])
```

Las funciones principales que podemos usar con los diccionarios son:

```
  figuras.keys()
```

Esta función devuelve una lista en la que cada elemento corresponde a una clave del diccionario.

```
  figuras.values()
```

Esta función también devuelve una lista que, en este caso, contiene los valores.

```
  figuras.items()
```

Esta función devuelve una lista de tuplas. Cada tupla corresponde a un elemento del diccionario. El primer elemento de la tupla es la clave y el segundo es el valor que tiene asociado.

16. Bucles para secuencias

En este tutorial ya se ha hablado de un tipo de bucles. Son esas estructuras que se inician con la palabra clave "while" y una determinada condición. Estas estructuras se encargaban de repetir un cierto código mientras una condición fuese cierta.

En este apartado veremos un bucle que recorrerá una secuencia elemento por elemento y repetirá el código mientras queden elementos en la secuencia. Estos bucles se inician con la palabra clave "for" (para en inglés), después irá una variable y luego la secuencia en cuestión. Esta línea finalizará con dos puntos y luego se escribirá el cuerpo, es decir, las instrucciones que se repetirán.

Para este primer ejemplo usaremos la función "range" que toma dos números como argumento, el primero de los cuales debe ser menor que el segundo. Esta función creará una lista que empezará en el primer número y no incluirá el último. Así, la función:

```
  range(1,6)
```

Creará una lista con los números del 1 al 5. Esto se puede probar en un bucle como este:

```
for i in range(1,6):
      print(i)
```

Lo que ha ocurrido es lo siguiente:

- La función range ha creado la lista con los números del 1 al 5.
- La variable i toma el primer valor (1) y lo imprime.
- Como hay más valores en la lista toma el siguiente (en este caso 2) y ejecuta el código del cuerpo, es decir, lo imprime.
- Y así sucesivamente hasta que acaba con todos los valores de la lista.

En el apartado correspondiente tiene dos ejemplos sobre cómo obtener este mismo resultado utilizando un bucle "while". Compare los tres programas.

Se puede interactuar con cualquier lista que haya creado, por ejemplo:

```
lista=["verde","rojo","amarillo"]
for i in lista:
      print("El",i,"es un color muy bonito.")
```

Supongamos que queremos hacer un bucle sobre un diccionario. Vamos a crear uno con los tres colores de la lista como claves y, como valores, insertaremos elementos que tienen esos colores:

```
colores={"Verde":"Césped.","Rojo":"Fuego","Amarillo":"Sol."}
```

Probemos con este bucle:

```
for i in colores:
      print(i)
```

Lo único que nos imprime son las claves del diccionario. Si queremos obtener los valores debemos utilizar la sintaxis propia para acceder a cada clave. Por ejemplo, para ver qué valor tiene la clave "rojo" escribiríamos:

```
colores["rojo"]
```

Por lo tanto, en el bucle debemos hacer lo mismo pero haciendo que la clave que nos interese sea la que en ese momento tenga la variable. Un bucle en el que combinemos claves y valores podría ser el siguiente:

```
for i in colores:
      print(i+":",colores[i])
```

17. Funciones específicas para cadenas de caracteres

A continuación vamos a hablar de funciones que se encargan de modificar cadenas de caracteres. Lo primero será crear la cadena con la que trabajaremos:

```
  principio="En un lugar de la Mancha, de cuyo nombre no quiero
acordarme."
```

Ahora ya estamos listos para aplicar estas funciones.

```
  principio.lower()
```

Esta función convierte toda la cadena a minúsculas.

Sin embargo, si usted manda a imprimir el contenido de la variable principio verá que la cadena sigue siendo la original. Si quiere conservar la modificación que realiza tanto la función "lower" como las demás funciones que veremos en este apartado deberá almacenar estos resultados en una variable.

```
  principio.upper()
```

Convierte toda la cadena a mayúsculas.

```
  principio.replace("a","1")
```

En este ejemplo se sustituiría la letra "a" por el número "1".

```
  principio.split()
```

Divide la cadena por el punto en el que encuentra un espacio en blanco y forma una lista a partir de dicha división.

Esta función acepta dos argumentos. el primero es el caracter por el que se debe separar. el segundo es la cantidad de separaciones que se deben realizar. Por ejemplo:

```
  principio.split(" ",1)
```

Esta llamada a la función crearía una lista con dos elemenntos. El primero sería la primera palabra y el segundo sería el resto de la cadena.

Para la última función necesitamos crear una lista.

```
  lista=["Una","serie","de","palabras."]
```

Vamos a unir todos esos elementos en una única cadena de caracteres. Supongamos que, como son palabras, queremos separar cada una de la siguiente por un espacio en blanco. En este caso, haríamos lo siguiente:

```
" ".join(lista)
```

Como se puede apreciar, estas funciones se pueden aplicar tanto sobre una cadena concreta de caracteres como sobre la variable que la contenga.

En el caso de esta última función la cadena sobre la que actuará "join" es el grupo de caracteres que separará cada elemento de la lista del siguiente.

Dejo como ejercicio que el lector pruebe a aplicar esta función sobre la cadena con la que veníamos trabajando.

18. Formateando cadenas

En este apartado veremos cómo se formatean las cadenas, es decir, cómo presentarlas de una forma bonita y amigable al usuario. Veamos un ejemplo:

```
nombre="Pedro"
edad=35
"{0} tiene {1} años.".format(nombre,edad)
```

Repasemos el código que ya conoce. Lo primero que se hace es asignar un par de variables. En la última línea hay algo extraño encerrado entre comillas, después un punto y un nombre con las variables encerradas entre paréntesis. Las comillas deben indicarle que, por muy extraño que parezca su contenido, no es más que una cadena de caracteres. Los paréntesis deben indicarle sin ningún tipo de dudas que se está llamando a una función. Como hay un punto en medio ya debería saber que la función es propia de las cadenas de caracteres. Es tiempo de saber lo que hace la función.

Esta función, que se llama "format", crea una lista con lo que recibe como parámetro y va sustituyendo los elementos en el lugar de la cadena en el que encuentre un número encerrado entre llaves. En nuestro ejemplo crea la lista con el contenido de las dos variables. Después recorre la cadena y encuentra un cero entre llaves. En ese punto imprime lo que hay en la posición cero de la lista (el contenido de la variable "nombre"). Después sigue recorriendo la cadena y cuando llega al uno entre llaves busca la posición uno en su lista, que es el contenido de la variable "edad" y realiza la sustitución pertinente.

Este es el formateo más básico que se puede realizar a una cadena de caracteres.

Cabe decir que la función "format" admite prácticamente cualquier cosa como parámetro (variables, números, operaciones matemáticas, cadenas, listas, tuplas o diccionarios). En el resto del apartado nos centraremos en lo que se puede poner entre las llaves.

Lo primero que cabe reseñar es que se puede acceder a un valor concreto de una lista, tupla o diccionario:

```
"{0[2]}".format(lista)
```

Esto mostraría únicamente el elemento que ocupa la posición tres de la lista.

Después del número del elemento a sustituir se puede escribir el signo de dos puntos y una serie de caracteres que afectan a la forma en que se imprimirá ese elemento.

El primero de estos caracteres afecta a la alineación:

- Un signo de menor que (<) hará que los caracteres se alineen a la izquierda.
- Un signo de mayor que (>) hará que los caracteres queden alineados a la derecha.
- Un signo de acento circunflejo (^) provocará que los caracteres sean centrados.

A continuación de la alineación se puede escribir un número que especificará el ancho del que se dispone para la alineación.

La última opción está disponible únicamente para los números decimales. A continuación del ancho se puede escribir un punto y un número que provocará un redondeo a una cierta cantidad de decimales. El número indica la cantidad de dígitos que se imprimirán entre la parte entera y la parte decimal. Veamos un ejemplo:

```
from math import pi
"{0:.5} {1:.5}".format(50.23456789,pi)
```

Se imprimiría:

"50.235 3.1416"

Para finalizar el apartado, veamos un ejemplo relativo a la alineación y el ancho. Supongamos que queremos hacer un programa que, para cada número del 1 al 10 imprima ese número y, a continuación, su cuadrado. Con lo que sabíamos nos quedaría así:

```
for i in range(1,11):
      print(i,i**2)
```

Este programa da un resultado correcto, pero no hace una gran presentación del mismo. Para hacerlo más bonito formatearemos la cadena de la siguiente forma:

- Ambos números de cada fila estarán alineados a la derecha.
- El número de la izquierda (i) tendrá una anchura de dos, puesto que el mayor es 10 que tiene dos cifras.
- El segundo número (i al cuadrado) tendrá una anchura de 3 porque elmayor es 100 que tiene tres dígitos.

Por lo tanto, el programa quedaría así:

```
for i in range(1,11):
      print("{0:>2} {1:>3}".format(i,i**2))
```

Dejo que el lector compare la salida que producen ambos programas.

19. Clases

Este apartado es el primero de un bloque muy importante en Python. Se trata de la programación orientada a objetos.

En este tipo de programación se trata de describir cómo es cada cosa, qué características tiene y que es capaz de hacer. Esta descripción genérica es la definición de una clase. Una vez definida la clase se pueden crear objetos y, en el momento del programa en el que nos interese, podemos hacer que uno de esos objetos ejecute alguna de los métodos (funciones) que tiene definidos.

Así, por ejemplo, podríamos tener una clase "estudiante". Esta clase podría tener unos atributos (características) como pueden ser el nombre, la edad, los días que ha asistido a clase o las calificaciones obtenidas. Además, podría tener métodos (funciones) como "iraclase()" que añadiría un día a los días que ha asistido a clase o "media()" que calcularía la media de las calificaciones.

Aunque todo esto le parezca nuevo en realidad no lo es. Python tiene definida la clase "str" que contiene todos los métodos sobre cadenas de caracteres (join, split, replace, format, lower, upper...). Si recuerda, estos métodos ya han sido vistos en otro apartado de este documento. Así, en este código:

```
cadena="Un texto entre comillas."
cadena.upper()
```

Lo que sucede es lo siguiente. Creamos un objeto de la clase "str" y después ejecutamos un método propio de esa clase.

Por las características de la llamada a la función debería comprender que los módulos también son objetos en Python.

19.1. Creación de una clase

La creación de una clase empieza con la palabra clave "class". Después viene el nombre que se le quiere dar a la clase. La línea finaliza con dos puntos y, a partir de ahí, se crea el cuerpo de la clase, en el que se especificarán sus métodos y atributos. Ni que decir tiene que todas las líneas del cuerpo deben llevar la indentación que les corresponde.

Existen dos clases de métodos. Los métodos especiales y los métodos del usuario. Por ahora la única diferencia estará en el nombre de los métodos especiales que van encerrados entre parejas de guiones bajos.

La definición del método empieza con la palabra clave "def" (al fin y al cabo no es más que una función), después el nombre del método, un par de paréntesis encerrando los parámetros y el signo de dos puntos. En las siguientes líneas se escriben las instrucciones que se ejecutarán cuando se llame al método.

Es obligatorio definir como primer método el método constructor. Es un método especial de Python que se ejecuta automáticamente cuando se crea un objeto. Este método se llama "init" (inicializar en inglés).

Por último, debe saber que todos los métodos deben llevar como parámetro la palabra clave "self". Con esto Python sabrá que los parámetros que se definen a continuación pertenecen al propio objeto. Además, si dentro de un método queremos modificar un atributo del propio objeto la instrucción sería así:

```
self.atributo=nuevovalor
```

Retomemos el ejemplo de la clase "estudiante". Con todas estas explicaciones debería entender cómo se programa. Le recomiendo que revise el código releyendo los párrafos anteriores para aclarar cualquier duda.

```
class estudiante:
      def __init__(self,nombre,nota=[],diasquefue=0):
            self.nombre=nombre
            self.nota=nota
            self.diasquefue=diasquefue
            print("Hola. Me llamo {0}.".format(self.nombre))

      def iraclase(self):
            self.diasquefue=self.diasquefue+1

      def nuevasnotas(self,nota):
            self.nota.append(nota)

      def notamedia(self):
            return sum(self.nota)/len(self.nota)
```

En la función "notamedia()" se ejecuta una nueva función ("sum" que calcula la suma de una lista formada por valores numéricos).

Vamos a crear un objeto de la clase estudiante:

```
alumno=estudiante("Miguel")
```

El mensaje que aparece le recuerda que el método constructor se ejecuta automáticamente al crear el objeto.

Veamos ahora cuántos días ha ido a clase:

```
alumno.diasquefue
```

Vamos a hacer que vaya durante 100 días a clase. Para ello, usaremos un bucle:

```
for i in range(1,101):
      alumno.iraclase()
```

Ahora vamos a añadir notas que ha obtenido el alumno:

```
alumno.nuevasnotas(10)
alumno.nuevasnotas(8)
alumno.nuevasnotas(5)
alumno.nuevasnotas(7)
alumno.nuevasnotas(9)
```

Como última comprobación vamos a conocer el nombre, los días que asistió a clase y sus notas. Además calcularemos la nota media.

```
alumno.nombre
alumno.diasquefue
alumno.nota
alumno.notamedia()
```

19.2. El método especial "str"

Este método sirve para controlar el mensaje que recibirá el usuario cuando mandemos imprimir el objeto. Por defecto se imprimiría información relativa a la clase a la que pertenece y la posición de memoria que ocupa.

Únicamente debe recibir "self" como parámetro y sólo debe contener la instrucción "return" con la cadena de caracteres que se imprimirá.

19.3. El método especial "del"

Este método recibe un sólo parámetro ("self") y se encarga de borrar el objeto. Podemos aprovechar para imprimir un mensaje apropiado, ya que es una buena práctica que el usuario conozca lo que ocurre, para que el programa le sea más amigable.

19.4. Métodos y variables de la clase

Una clase puede tener variables y métodos propios. Esto quiere decir que serán suyas y no de los objetos.

Se deben declarar fuera de los métodos de los objetos y siempre después de sus métodos (sobre todo del método constructor).

Las variables se declaran y asignan como ya se vio en su momento. Los métodos se escriben igual que los de los objetos, pero sin recibir la palabra "self" como parámetro.

19.5. La clase persona

Después de la exposición teórica de estos nuevos métodos veremos un ejemplo práctico. Para ello crearemos la clase persona. El código es el siguiente:

```
  class persona:
        def __init__(self,nombre,edad):
                self.nombre=nombre
                self.edad=edad
                persona.poblacion=persona.poblacion+1
                print("¡Demos la bienvenida a
{0}!".format(self.nombre))
                persona.censo()

        def __str__(self):
                return "¡Mucho gusto! Me llamo {0} y tengo {1}
años.".format(self.nombre,self.edad)

        def __del__(self):
                persona.poblacion=persona.poblacion-1
                print("Con lo buena persona que era {0}. Le echaremos
de menos. Descanse en paz.".format(self.nombre))
                persona.censo()

        poblacion=0

        def censo():
                if(persona.poblacion==0):
                        print("Aquí no hay nadie.")
                elif(persona.poblacion==1):
                        print("Aquí hay una persona.")
                else:
                        print("Aquí hay {0}
personas.".format(persona.poblacion))
```

Para crear una persona haríamos lo siguiente:

```
  p1=persona("Antonio",30)
```

Podemos imprimir este objeto.

```
  print(p1)
```

Y también podemos eliminarlo.

```
  del(p1)
```

20. Trabajando con archivos

A veces es conveniente guardar en un archivo lo que hace nuestro programa para que esto quede almacenado y no se pierda una vez finalizada la ejecución.

En este apartado aprenderá todo lo relacionado con la escritura y la lectura de archivos en Python.

Antes de empezar es importante que tenga claro que en un programa la operación con un archivo es la misma que la que lleva a cabo en su vida diaria con cualquier procesador de textos. En primer lugar abre el archivo, después lee lo que tenga que leer y modifica lo que tenga que modificar o, si lo crea desde cero, escribe el contenido. Por último, cuando ha acabado simplemente lo cierra.

20.1. Abriendo un archivo

Para abrir un archivo primero debe crear un objeto por el que se referirá al archivo. A este objeto le asignará el archivo que se abre.

Para abrir un archivo se utiliza la función open (abrir en inglés) que recibe como parámetros dos cadenas. La primera es la ruta y el nombre del archivo. La segunda contiene dos caracteres. El primero informa sobre la finalidad para la que se abre el archivo y es una de las siguientes letras:

- w si quiere crear el archivo para escribir.
- r si quiere abrir un archivo existente para leer el contenido.
- a si va a abrir un archivo existente para añadir contenido.

La segunda letra de la cadena será una t si quiere abrir el archivo en modo texto o una b si quiere abrirlo en modo binario.

Vista la teoría vamos a abrir un archivo para escritura. El nombre será prueba.txt, será de tipo texto y nos referiremos a él con la palabra escritura. La sintaxis debe ser la siguiente:

```
escritura=open("prueba.txt","wt")
```

A partir de este momento, tanto para escribir como para leer (si hubiera sido de lectura) como para cualquier operación que deseemos hacer con el archivo tendremos que usar la sintaxis propia de los objetos, es decir, el identificador (en este caso escritura), un punto y la función que queramos utilizar.

20.2. Escribiendo en el archivo

Para escribir en el archivo usaremos el método write (escribir en inglés) que recibe como parámetro la cadena de caracteres que se debe escribir. Tenga en cuenta que cuando quiera que en el archivo se inserte un salto de línea se debe escribir una barra invertida y la letra ene.

En el siguiente ejemplo escribiremos tres líneas en el archivo que teníamos abierto. Como es lo único que haremos aprovecharemos para cerrar el archivo con el método close (cerrar en inglés) que no requiere parámetros.

```
escritura.write("Primera línea.\n")
escritura.write("Segunda línea.\n")
escritura.write("La última línea es más larga.\n")
```

```
escritura.close()
```

Si escribe estas líneas en la consola de Python el archivo se habrá creado en la carpeta en la que se haya instalado Python. Ábralo con un editor de textos y vea cómo ha quedado.

20.3. Leyendo el archivo

Vamos a leer el contenido del archivo que hemos creado antes. Como lo hemos cerrado, lo tendremos que abrir:

```
lectura=open("prueba.txt","rt")
```

Observe que ahora nos referiremos al archivo como lectura. Da igual qué nombre utilice, simplemente tenga claro cuál corresponde a qué archivo. Por otra parte, observe que en la función open el segundo parámetro hace referencia a lectura en modo texto.

Para leer el archivo y posicionar el cursor de forma imaginaria al final del mismo se usa el método read (leer en inglés) que no recibe ningún parámetro. Probémoslo:

```
lectura.read()
```

Lo que le ha devuelto es una cadena en la que se incluyen los símbolos de salto de línea. Para la siguiente prueba vamos a cerrar el archivo y a volverlo a abrir.

```
lectura.close()
lectura=open("prueba.txt","rt")
```

Lo que debe hacer es usar la función print para imprimir la lectura del fichero. Así, se mostrará el contenido tal y como está escrito. Para que le resulte más comprensible asignaremos el contenido de la lectura a una variable y luego imprimiremos esta variable. Como no tenemos nada más que hacer, cerraremos el archivo.

```
texto=lectura.read()
lectura.close()
print(texto)
```

20.4. Otros métodos para archivos

A continuación se comentan algunos métodos para trabajar con archivos.

- Seek. Este método posiciona el cursor los bytes que se le pasen como parámetro a partir del principio del fichero. El caso más común de uso de esta función es pasándole 0 como parámetro, lo que hará que el cursor vaya al principio del archivo.
-
- ```
  fichero.seek(0)
  ```
  ```

- El método tell devuelve en qué byte se encuentra el cursor. No acepta parámetros.
-
- `fichero.tell()`
- El método readline lee una única línea del fichero a partir de la posición del cursor y la devuelve en formato cadena.
-
- `linea=fichero.readline()`
- El método readlines genera una lista con lo que encuentre a partir de la posición del cursor. Cada elemento es una cadena de caracteres que contiene una línea del fichero.
-
- `lista=fichero.readlines()`

20.5. Gestión de ficheros de forma más sencilla

Existe una forma más sencilla de gestionar los ficheros. Se trata de crear un bloque en el que se agrupen las instrucciones a realizar.

Supongamos que queremos abrir un fichero, que nos queremos referir a él como lectura y que queremos guardar su contenido en una variable. Con este bloque haríamos lo siguiente:

```
with open("prueba.txt","rt") as lectura:
      texto=lectura.read()
```

Python abrirá el archivo y almacenará el contenido en la variable. Cuando se termina el bucle Python cierra automáticamente el archivo y nos ahorramos cerrarlo a mano.

Además, existe otra forma de leer el contenido de un fichero:

```
with open("prueba.txt","rt") as leer:
      for linea in leer:
              print(linea)
```

20.6. Escribiendo objetos en un archivo

Existe un modo de escribir una lista en un archivo. Existen dos diferencias principales con respecto a la forma que se ha explicado para trabajar con archivos:

1. Trabajaremos con un módulo que se llama pickles.
2. El archivo se debe abrir en modo binario.

Para insertar una lista en un archivo haremos lo siguiente:

1. Crear la lista.
2. Abrir el archivo en modo binario para escritura.
3. Usar la función "dump" que recibe los siguientes parámetros:
    - El primer parámetro es la lista.

- El segundo parámetro es el identificador por el que nos referimos al fichero.

Veamos un ejemplo:

```
oficina=["papel","grapas","clips","lápices","bolígrafos"]
import pickles
with open("material.txt","wb") as material:
      pickles.dump(oficina,material)
```

Para recuperar la lista de un archivo hay que hacer lo siguiente:

1. Abrir el archivo en modo binario de lectura.
2. Asignar a una variable el resultado de la función "load" que recibe como parámetro el identificador con el que nos referimos al archivo.

Trabajemos con el archivo anterior:

```
import pickles
with open("material.txt","rb") as material:
      comprar=pickles.load(material)
```