	Python	Color fondo#272820

Varios

Añadir temas personalizados a IDLE (IDE de python). En la dirección pegar el archivo config-highlight.cfg.
Para salir de un programa en modo intérprete: exit().
Fuera de IDLE, hay que guardar el archivo con la extensión .py.
Al contar los elementos de una cadena se comienza por el 0. Los espacios cuentan como un puesto más.
Usando el intérprete, para copiar un mensaje ya escrito, ir usando Alt+P.

Introducción

Comentarios

para comentar una línea.
Abrir y cerrar con tres comillas, simples o dobles, para comentar varias líneas.

Palabras reservadas

No se pueden usar para identificar variables, objetos, etc. Las palabras son:

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

Sin embargo, si se pueden usar si se pone alguna mayúscula, ya que Python es sensible.

Mostrar en pantalla

Función print(). Comillas simples o dobles solo para texto (números puedes usarse como texto).
Un nuevo print saldrá en una línea nueva.

```
print('Hello world')          # Hello world
print(14)                     # 14
```

Algunos casos especiales:

- Poner comillas simples (si el texto va entre comillas simples): \'
- Poner comillas dobles (si el texto va entre comillas dobles): \"
- Salto de línea: \n
- Tabular: \t
- Añadir saltos de líneas y tabulaciones al escribir directamente: """ ## """

Para poder imprimir en un mismo print varias cosas, sean cadenas o números, basta con separar entre comas. La coma generará un espacio entre los dos elementos. De esta manera, no hay que pasar los números a texto.

```
print('Tengo',23,'años o',22.6)
Tengo 23 años o 22.6
```

Operaciones

- Suma: +
- Resta: -
- Multiplicación: *
- División: /
- División sin resto: //
- Resto de división: %
- Exponenciación (También con floats para raíces): **

```
print(2**3**2)          # da 512, 2^(3^2)
print(8**(1/3))         # da 2.0, raíz cúbica de 8
print(20 // 6)          # da 3, el cociente de 20 entre 6
print(20 % 6)           # da 2, el resto de dividir 20 entre 6
print(1.25 % 0.5)       # da 0.25, el resto de dividir 1.25 y 0.5
```

Si la variable donde se guarda la operación es parte de la misma, se puede acortar. Sirve para todos los operadores de arriba.

```
x=2
x += 3                  # x = x + 3 -> x=5
```

El de suma también se puede usar para cadenas.

Cadena

Se ponen entre comillas simples o dobles. También se pueden poner entre triples comillas simples o dobles, y escribir la cadena en varias líneas, ya que se guardarán los saltos de línea usando \n.

Números reales

Se representan con un punto y se produce con:

- Una división.
- Una operación con al menos un float.

Por defecto vienen con un único decimal a no ser que venga de una operación con más decimales significativos. Para elegir cuantos decimales usar round()

```
x=4.7
print(round(x,2))       # queremos mostrar x con 2 decimales
4.7                     # solo muestra 1 porque los demás son insignificantes
x=4.732
print(round(y,2))       # queremos mostrar y con 2 decimales
4.73
```

Números complejos

Se expresan mediante el tipo complex.

```
x=5+6j          # da 3 <class 'int'>
```

type()

Podemos saber de que tipo es una variable con type(), poniendo dentro la variable. Para verlo en pantalla, se usa un print.

```
x=3
print(x,type(x))          # da 3 <class 'int'>
y=4.25
print(y,type(y))          # da 4.25 <class 'float'>
```

Concatenación

```
print("8" + "B")          # 8B
print("hola" + "hola")    # holahola
```

También se pueden multiplicar las cadenas por números enteros (ni floats ni otras cadenas).

```
print("ab"*2)             # abab
```

Variables

Solo se puede usar letras, números y guiones bajos para el nombre de la variable. No puede comenzar por números. Diferencia minúsculas y mayúsculas.

```
usuario = "user1"
cant = 8
º
```

A una variable se le puede ir asignando cadena y número, pero no es recomendable.
Se pueden asignar varias variables en la misma línea aún siendo de distinto tipo.

```
x,y=3,"Hola"
```

Obtener información del usuario

Con la función input(), se pide al usuario introducir un valor que se guarda como cadena, aunque se introduzca un número. Solo guarda la primera línea.

```
x=input()          # pide escribir al usuario, que se guarda en la variable x
print(x)           # muestra lo que se ha escrito
```

Si se pone dos input() seguidos, espera recibir al menos dos líneas, que son las que guardará.
Podemos introducir dentro un texto que se mostrará al usuario.

```
name=input("Tu nombre: ")                    # 11
```

Convertir tipo de variable

Para convertir una variable a número entero se usa int(). Si no recibe un número entero da error. Lo mismo para pasar a float, usando float(), que si admite un entero que pasa a formato de número real. Y para pasar de cadena a número complejo se usa complex().

```
edad=int(input())          # guarda como número lo introducido
nota=float(input())        # si recibiese 5, se guarda como 5.0
```

Para convertir una variable a cadena se usa str(), que es útil para imprimir una variable de número.

```
edad=20
print("La edad es " + str(edad)) # asi se puede usar una variable de número en print
```

Buscar dentro de una cadena

Con find() podemos buscar una subcadena en concreto dentro de una cadena. Devuelve la posición de la cadena donde comienza la subcadena. Si la subcadena está más de una vez, se quedará solo con la primera vez.

```
men="hola caracola"
men_parte=men.find("caracola")          # buscamos caracola dentro de la cadena men
print("men_parte)          # da 5, posición de men donde comienza la subcadena buscada
```

Si no hay esa subcadena, devuelve un -1.

Extraer parte de una cadena

Sacar parte de una cadena para tratarla a parte. Cuando indicamos el carácter inicial y final, cuidado por que el final no lo cogerá, si no que el último será el anterior.

```
men="hola caracola"
men_extraer=men[5:13]       # extraemos parte de la cadena men
print(men_extraer)          # da caracola, parte de la cadena men extraída
```

Booleanos

Otro tipo de dato, que admite True y False. Se usan los comparadores relacionales, que devuelven True si es verdad y False si es falso.

- Iguales: ==
- Diferentes: !=
- Mayor: >
- Menor: <
- Mayor o igual: >=
- Menor o igual: <=

Python distingue entre mayúsculas y minúsculas. Se pueden comparar enteros con reales.

```
bool1=True
print(bool1)                # True
print(1==2)                 # False
print("abb"!="abc")         # True
print(3<2)                  # False
```

Se puede usar el comparador de mayor o menor para ordenar las cadenas por orden alfabético.

```
print("A">"B")              # False
print("Antonio">"Ana")      # True
```

Estructuras de control

If

Ejecuta las instrucciones si la condición es True. Para definir que líneas después del if son las instrucciones, hay que añadirles a todas espacio o tabulación respecto a la línea del if (indentación).

```
if a>b:
        print("a es mayor")      # Al estar indentado forma parte del if
print("fin")                     # Al no estar indentado no forma parte del if, que se acaba
```

```
num=15
if num>10:
        print("Mayor a 10")
        if num <= 20:                          # Un if dentro de otro if
                print("Entre 5 y 20")
```

else

Poner else, para ejecutar un código si la condición es False en un if. Solo un else por if.

```
x=4
if x==5:
        print("Si es 5")         # Esta parte no se ejecuta al ser la condición False
else:
        print("No es 5")         # Esta parte si se ejecuta al ser la condición False
```

elif

Else con condición. Se pueden encadenar varios.

```
num=3
if num==1:
        print("Uno")
elif num==2:
        print("Dos")
elif num==3:
        print("Tres")
else:
        print("Distinto")                # Solo se cumple si ningún if o elif es verdadero.
```

Lógica booleana

Para poner más condiciones se usan los operadores:

- Devuelve True si se cumplen todos: and
- Devuelve True si se cumple alguno: or
- Invierte un argumento pasándolo de True a False y al revés: not

```
print(1==1 and 1+1==2 and 2+2==2**2)        # True
print(5==4 or 5==5 or 5==6)                 # True
```

```
print(not 1==1)                          # False
print(not 1==1 or not 3!=3)              # True
```

while

```
x=2
while x<5:
        print(“x es menor a 5”)
        x+=1
print(“x igual a 5”)
```

Para interrumpir un bucle while se usa break.

```
x=1
while x<10:
        print(x,’’,end=’’)
        x+=1
        if x==5
                break          # en el momento que se hace el break, sale del bucle
1 2 3 4
```

Para interrumpir una iteración se usa continue que hará que se vaya directamente a ejecutar la nueva iteración, siempre que no se acabe el bucle en ese momento.

```
x=0
while x<10:
        x+=1
        if x==5:
                continue       # en la iteración que x=5, no ejecuta lo que está bajo, el print
        print(x,'',end='')
1 2 3 4 6 7 8 9 10
```

for

Se usa para iterar sobre una secuencia determinada, como una lista. La palabra elem (se puede usar la palabra que queramos) se usa para referirse a el elemento actual de la lista en cada iteración.

```
list=[“a”,”b”,”c”,”d”]
for elem in list:
        print(elem*2,end=””)
aa bb cc dd
```

Lo mismo funciona para cadenas.

```
list='abababababbaba'
b=0
for elemento in list:
        if elemento=="b":
        b+=1
```

```
print(“Hay”,b,”b”)
Hay 6 b
```

Se puede usar las declaraciones continue y break igual que en un bucle while. for se suele usar para un número de iteraciones conocido y un while para cuando no lo sabemos.

Extras

end (dentro de print)

Se usa para añadir una cadena al final de la salida de una impresión por print(). Evita el salto de línea automático que causa el print.

```
print(“Ejemplo ”,end=””)        # En este caso, la coma no hace un espacio, hay que ponerlo
print(“primero”)                        # True
Ejemplo primero
```

sep (dentro de print)

Al usar comas en el print se genera un espacio automático. Con sep podemos controlar la separación.

```
print("1","2","3","4","5")
1 2 3 4 5
print("1","2","3","4","5",sep='')
12345
print("1","2","3","4","5",sep='_')
1_2_3_4_5
```

len

Sirve para obtener la longitud de una cadena o los elemento que componen un objeto. El espacio cuenta como carácter.

```
print(len(‘Hola’))
4
x=len(Caracola)
print(x)
8
print("1","2","3","4","5",sep='_')
1_2_3_4_5
```

Range()

Devuelve una secuencia de números. Por defecto, comienza desde 0 y se incrementa en 1 hasta antes del número especificado. El rango se convierte en una lista usando list(). La sintaxis completa es range(start,stop,step), donde solo es obligatorio stop. Al poner dos argumentos, definimos los límites del rango(se acaba justo antes del segundo) y el tercer argumento es el incremento, que puede ser negativo también.

```
print(list(range(2,16,3)))
```

```
[2, 5, 8, 11, 14]
```

Se suele combinar con un bucle for para realizar una iteración un número fijo de veces. Aquí no hace falta llamar a list ya que no se está indexando.

```
for i in range(5):                    # la iteración se hace 5 veces
        print("A ",end='')
A A A A A
```

None

Usado para representar que no hay valor. Al igual que valores vacíos como 0 o [], da False al pasarlo a booleano. Cuando una función no está pensada para devolver valores, da un None.

```
def some_func():
    print("Hi!")

var = some_func()
print(var)

Hi!
None
```

In, not in

Sirve para comprobar si algo es dentro de algo, devolviendo True o False. Puede ser para listas, diccionarios, arrays…

```
frase='Que está pasando'
print('Que' in frase)

True
```

```
elem=[2,3,5,7,8]
print(4 not in elem)

True
```

Al usarlo en diccionarios, se fijará en las claves, no en los valores que tengan estas.

```
diccio={1:323,2:343,3:332}
print(1 in diccio)
print(323 in diccio)

True
False
```

Listas

Introducción

Poner varios elementos en una variable, que pueden ser de varios. Cada elemento puede incluir una lista dentro, haciendo una matriz. Se pueden modificar los valores. Usando len, devuelve los elementos que hay en la lista, y si usamos len en un elemento de la lista, devuelve el tamaño de ese elemento. Puede haber una coma después del último elemento, pero no es obligatorio, aunque si es recomendable para listas o otros elementos a los que dentro del código se les puede añadir a modificar el contenido.
Se puede acceder a el úlimo elemento de una lista con el índice -1, y asi succesivamente.

```
cos=[]                                  # se puede dejar en blanco para llenar después
cosa=[“abc”,”def”,”ghi”]
print(cosas[1])
def
print(cosas[0:2])
[‘abc’,’def’]
cosa=[“abc”,[“d”,”e”,”f”],”ghi”]
print(cosa[1])
[‘d’,’e’,’f’]
print(cosa[1][0])
d
print(cosa[-1])
ghi
```

A una cadena simple, también se le puede llamar algún elemento. Listas y cadenas son muy similares, una cadena se puede considerar una lista de caracteres que no se pueden cambiar. Las listas se pueden sumar o multiplicar.

```
nums=[1,2,3]
print(nums+[4,5,6])
[1,2,3,4,5,6]
print(nums*3)
[1,2,3,1,2,3,1,2,3]
```

Comprobar que un elemento está en la lista

Devuelve True o False dependiendo de si está.

```
nums=[1,2,3,’uno’,’dos’,’tres’]
print(2 in nums)
True
print(“dos” in nums)
True
```

También se usa para saber si una subcadena está en una cadena.

```
nums=”UnoDosTres”
print(“Dos” in nums)
True
```

I para comprobar, tanto en una cadena o lista si algo no está, la diferencia es poner not delante

```
nums=[1,2,3,'uno','dos','tres']
print(not 4 in nums)
True
print("cuatro" not in nums)
True
```

Añadir un elemento

```
list=[1,2,3,4,5]
list.append(6)                    # append solo para añadir un elemento
print(list)
[1, 2, 3, 4, 5, 6]
list.extend([7,8,9])
[1, 2, 3, 4, 5, 6, 7, 8, 9]       # extend puede añadir uno varios
```

Añadir un elemento en mitad de la lista

Elegir la posición donde poner el nuevo elemento y su valor. Si elegimos la posición 1, el elemento que estaba en esa posición se desplaza a la derecha, así como todos los siguientes.

```
list=[1,2,3,4,5]
list.insert(1,7)
print(list)
[1, 7, 2, 3, 4, 5]
```

Eliminar un elemento de la lista

Elimina un elemento, haciendo que el siguiente elemento de la lista ocupe su puesto, y así sucesivamente.

```
list=[1,2,3,4,5]
list.remove(2)
print(list)
[1, 3, 4, 5]
```

Para eliminar el último elemento aún sin saber en qué posición está:

```
list=[1,2,3,4,5]
list.pop(2)
print(list)
[1, 3, 4, 5]
```

Ordenar una lista

Para ordenar una lista alfabéticamente o de menor a mayor. Daré error si en la lista hay números y cadenas mezclados.

```
list=[5,3,1,4,2]
```

```
list.sort()
print(list)
[1, 2, 3, 4, 5]
```

Invertir una lista

Para invertir una lista poniendo el primer elemento al final, el último al principio y así.

```
list=[1, 2, 3, 4, 5]
list.reverse()
print(list)
[5, 4, 3, 2, 1]
```

Encontrar posición de un elemento

Devuelve en que posición se encuentra un elemento. Si está varias veces, solo importa donde está el primero.

```
list=[5, 3, 1, 4, 2]
print(list.index(4))
3
```

Cuantas veces está un elemento en una lista

```
list=[1, 2, 3, 2, 2, 1, 3, 2]
print(list.count(2))
4
```

Vaciar una lista

```
list=[1, 2, 3, 2, 2, 1, 3, 2]
list.clear()
print(list)
[]
```

Elemento mayor o menor de una lista

En caso de ser una lista de cadenas, max devuelve el último en orden alfabético y min el primero.

```
list=[7, 4, 3, 8, 4, 5, 4]
print(max(list),min(list))
8 3
```

Diccionarios

Introducción

Estructura de datos con un conjunto de elementos, como una lista, pero donde a cada uno se le puede asignar una clave. Son indexados igual que una lista.

En cada elemento a la derecha está el valor y a la izquiera su clave, separados por dos puntos. Se usan las llaves.

```
ages = {"Dave": 24, "Mary": 42, "John": 58}
print(ages["Dave"])
print(ages["Mary"])

24
42
```

```
bad_dict = {
    "a":[4, 5, 2],
    "b":[7, 1, 8],
}
print(bad_dict["a"][1])

5
```

Errores

Al intentar indexar una clave que no forma parte del diccionario aparece una excepción del tipo KeyError.
Además, como claves no se pueden usar datos mutables, como pueden ser listas o diccionarios.
En caso de usar una variable como clave, se guardará como clave el valor que tiene en ese momento la variable. Luego, no se podrá indexar el elemento usando la variable o el valor que tiene está, en caso de que haya cambiado.

```
x=3

bad_dict = {
    1:[1, 2, 3],
    x:[4, 5, 6],

}

x=12
print(bad_dict[3])
#No se puede usar print(bad_dict[x]) pues x!=3

[4, 5, 6]
```

Añadir claves

Se pueden añadir nuevas claves, a las que hay que asignarle un valor también.

```
orden = {1:'f'}
orden[2]='c'
orden[1] = 'g'
print(orden)

{1: 'g', 2: 'c'}
```

Get

Hace lo mismo que indexar, pero en caso de no existir la clave devuelve un valor específico, None por defecto, aunque se puede cambiar.

```
pairs = {1: "apple",
    "orange": [2, 3, 4],
    True: False,
    None: "True",
}

print(pairs.get("No está en el diccionario"))
print(pairs.get(7))
print(pairs.get(12345, "No está en el diccionario"))


[2, 3, 4]
None
No está en el diccionario
```

```
fib = {1: 1, 2: 1, 3: 2, 4: 3}
print(fib.get(4, 0) + fib.get(7, 5))

8
```

Funciones y módulos

Funciones

Se usa la sentencia def y dentro van loa argumentos que se van a usar en caso de que sí.

```
def func_hola()
        print("H_",end)
        print("O")
        print("L")
        print("A")
func_hola()
H_o_l_a_
def exclam(word,veces)
        print(word*veces,"!"*veces)
exclam("hola",3)
holaholahola !!!
```

Todas las variables que se usan creando una función, desaparecen fuera de la definición de la función. Para usarlas fuera hay que volver a definirla.
Podemos hacer que la función devuelva un valor mediante return. Al utilizarse, la función deja de ejecutarse. Si se pone return si nada, lo único que hará es salir de la función.

```
def sum(x,y):
```

```
        tot=x+y
        return tot
print(sum(4,5))
9
```

Podemos cambiar en cualquier momento los nombres usados para llamar a una función.

```
def multiply(x,y):
        return x*y
a,b=4,7
operation=multiply
print(operation(a,b),end=' ')
print(multiply(a,b))
28 28
```

Las funciones también puedes ser usadas como argumentos de otras funciones.

```
def square(x):
        return x*x
def test(func,x):          # al llamar una función, se debe añadir los argumentos que tiene
        print(func(x))
test(square,7)
49
```

Módulo

Trozos de código de otras personas con todo tipo de funciones que se llaman con import. Un ejemplo es el módulo random y dentro de este la función randomint

```
import random
print(random.randint(1,6))
4                                   # podría dar cualquiera entre 1 y 6
```

También se puede importar solo parte de las partes de un módulo. Por ejemplo, importar del módulo math solo la constante pi y la función sqrt.

```
from math import pi, sqrt
print(pi)
3.141592653589793
```

Importar un módulo no disponible dará un ImportError.
A las funciones importadas también se le puede cambiar los nombres usados para referirse a ellas. También se puede hacer de primeras.

```
from math import sqrt as squa re_root
print(square_root)
10.0
```

Hay tres tipos de módulos: Los que creas tú, los que instalas de fuera y los preinstalados en Python, que son la biblioteca estándar.

Módulos más comunes

random

rand.int(x,y)

math
- math.sqrt(x)
- pi
- cos

string
re
datetime
os
mutiprocessing
subprocess
socket
email
json
doctest
unittest
pdb
argparse
sys

Excepciones

Las excepciones ocurren cuando se ejecuta un código. Al no tener que compilar el código antes del error se ejecuta y al llegar a ese punto aparece un mensaje de error, que especifica el fallo: Entre ellos:

- ImportError: Fallo al importar.
- IndexError: Uso de un elemento de la lista que no existe.
- NameError: Usada variable que no existe.
- SyntaxError: No se puede analizar bien el código.
- TypeError: Se llama una variable de un tipo no válido.
- ValueError: El tipo de variable es correcto pero el valor no.
- AssertionError : Se crea manualmente con la función assert.
- KeyError: Indexado de un diccionario un elemento que no existe.

Para evitar excepciones, usando código si ocurre se usa el try y except. En el try va el código que puede dar error, y en el except código por si ocurre. En except se define la excepción a manejar. Útil para la entrada de datos. Aunque en el try haya un error. la parte anterior al error si se ejecuta.

Podemos poner varios except y para cada uno actuar en caso de más de un tipo de error. Si ponemos un except de un error en concreto y luego después un except general, si el error es del tipo en concreto solo se ejecutara el primer except.

Si no se especifica en el except que error, se usará para todos. Esto se debe evitar, ya que puede ocultar errores de programación que no esperamos.

```
try:
    variable = 10
    print(variable + "hello")
    print(variable / 2)
except ZeroDivisionError:
```

```
    print("Divided by zero")
except (ValueError, TypeError):
    print("Error occurred")

Error occurred
```

Para asegurar que corra un código en cualquier caso usamos finally. Se coloca debajo del except, y se ejecuta haya error o no.

```
try:
    print("Hello")
    print(1 / 4)
except ZeroDivisionError:
    print("Divided by zero")
finally:
    print("This code will run no matter what")

Hello
Divided by zero
This code will run no matter what
```

Puedes generar una excepción, aunque no haya un error de código, con raise. Se puede especificar los detalles del error. Se puede usar en cualquier parte del código.

```
name = "123"
print("1")
raise NameError("Invalid name!")
print("2")

1

Traceback (most recent call last):
File "file0.py", line 2, in <module>
raise NameError("Invalid name!")
NameError: Invalid name!
```

Una aserción sirve para comprobar una expresión, y si el resultado es falso, se crea una excepción. Se usa la sentencia assert. Se suele poner antes y después de una función para comprobar una entrada y salida válidas. Se le puede poner como segundo argumento un mensaje de error en la aserción.

```
print(1)
assert 2 + 2 == 4
print(2)
assert (1 + 1 == 3),"Los dos lados no valen igual"
print(3)

1
2

Traceback (most recent call last):
File "file0.py", line 4, in <module>
assert (1 + 1 == 3),"Los dos lados no valen igual"
AssertionError: Los dos lados no valen igual
```

Usar archivos

Abrir archivo

Primero se debe abrir el archivo con la función open cuyo argumento es la localización. Se le asigna una variable al archivo. Un segundo argumento, define que haremos con el archivo:

- Modo lectura, por fefecto: “r”
- Modo escritura: “w”
- Modo anexo, para añadir cosas al final del archivo: “a”

Añadiendo una b después de la letra de cada tipo abre el archivo en binario, útil para imágenes por ejemplo.

```
myfile = open("filename.txt", "r")
```

Cerrar archivo

Después de usar un archivo, hay que cerrarlo con close.

```
myfile.close()
```

Es muy recomendable asegurarse que se cierra el archivo aunque ocurra una excepción. Para ello poner el código donde se abre un archivo y se manipula dentro de un try y en un finally el comando de close.

```
try:
    f = open("filename.txt")
    print(f.read())
finally:
    f.close()
```

El código anterior ha dado error, pero el archivo se ha cerrado igualmente.
Otra manera de hacerlo es usar with y as. De esta manera, el archivo de cierra automáticamente al final del bloque with. En este modo, se crea una variable temporal que solo sirve en el bloque with.

```
with open('texto.txt','w') as fi:
    fi.write('Hola que pasa')

with open('AAA.txt') as fi:
    print(fi.read())
```

Leer archivos

En modo lectura usamos read para leer todo el archivo. (El salto de línea ocupa un espacio)

```
file = open("filename.txt")
cont = file.read()
print(cont)
file.close()
```

Podemos leer solo parte del código, poniendo un número en el argumento de la función read que determina los bytes a leer. El siguiente read() lo hará a partir de donde acabó el primero. Al llegar al final del archivo, todo intento de leer devolverá una cadena vacía.

```
file = open("filename.txt", "r")
print(file.read(16))
print(file.read(4))
print(file.read())
file.close()
```

Para leer línea a línea se usa readLines() que guarda el contenido de cada línea en un elemento de una lista, incluido el salto de línea (\n).

```
file = open("filename.txt", "r")
print(file.readlines())
file.close()
```

También se podría hacer con un bucle for. Entre cada línea aparece una en blanco ya que print() añade una línea automáticamente y al leer la línea entera recoge también el salto de línea que hay en el archivo de texto.

```
file = open("filename.txt", "r")
for line in file:
    print(line)
file.close()
```

En vez de asignar una variable al archivo, podemos leer el archivo a la vez que lo abrimos para ahorrar código.

```
open("test.txt").readlines()
```

Escribir en archivos

Se debe haber abierto el archivo en modo escribir o anexo, y si no existe el archivo se crea uno nuevo.

```
file = open("newfile.txt", "w")
file.write("This has been written to a file")
file.close()
file = open("newfile.txt", "r")
print(file.read())
file.close()

This has been written to a file
```

Al abrir un archivo en modo escribir, el contenido de este se borra. El comando write devuelve el número de bytes que se han escrito en el archivo.

```
var="Hola mundo"
file=open('file.txt','w')
print(file.write(var)==len(var))
```

Solo acepta string, para escribir otra tipo de dato hay que convertirlo.