Tipos de datos

Python

Literales

Literales enteros: 3, 12, -23
Literales reales: 12.3, 45.6
Literales cadenas:

```
'hola que tal!'
"Muy bien"
'''Podemos \n
ir al cine'''
```

- Los literales nos permiten representar valores.
- Estos valores pueden ser de diferentes tipos → diferentes tipos de literales:

  - **Literales numéricos**
    - Para representar números enteros utilizamos cifras enteras (Ejemplos: 3, 12, -23).
    - Para los números reales utilizamos un punto para separar la parte entera de la decimal (12.3, 45.6). Podemos indicar que la parte decimal es 0, por ejemplo 10., o la parte entera es 0, por ejemplo .001.

  - **Literales cadenas**

Nos permiten representar cadenas de caracteres.
Para delimitar las cadenas podemos usar el carácter ‘ o el carácter “.
También podemos utilizar la combinación ‘’’ cuando la cadena ocupa más de una línea.
Ejemplos:

'hola que tal!’
"Muy bien“
'''Podemos \n
ir al cine'''

El caracter \n es el retorno de carro (los siguientes caracteres se escriben en una nueva línea).

Variables

- Una variables es un identificador que referencia a un valor. Para que una variable referencia a un valor se utiliza el operador de asignación =.

- El nombre de una variable, ha de empezar por una letra o por el carácter guión bajo, seguido de letras, números o guiones bajos.

```
>>> var = 5
>>> var
5
```

- python distingue entre mayúsculas y minúsculas en el nombre de una variable, pero se recomienda usar sólo minúsculas.

Expresiones

- Una expresión es una combinación de variables, literales, operadores, funciones y expresiones, que tras su evaluación o cálculo nos devuelven un valor de un determinado tipo.
- Ejemplos:

```
a + 7
(a ** 2) + b
```

Operadores

- Los operadores que podemos utilizar se clasifican según el tipo de datos con los que trabajen.
- Ejemplos:
  - Operadores aritméticos: +, -, *, /, //, %, **.
  - Operadores de cadenas: +, *
  - Operadores de asignación: =
  - Operadores de comparación: ==, !=, >=, >, <=, <
  - Operadores lógicos: and, or, not
  - Operadores de pertenencia: in, not in

Operadores- precedencia

1. Los paréntesis rompen la precedencia.
2. La potencia (**)
3. Operadores unarios (+ -)
4. Multiplicar, dividir, módulo y división entera (* % //)
5. Suma y resta (+ -)
6. Operador binario AND (&)
7. Operadores binario OR y XOR (^ |)
8. Operadores de comparación (<= < > >=)
9. Operadores de igualdad (<> == !=)
10. Operadores de asignación (=)
11. Operadores de pertenencia (in, in not)
12. Operadores lógicos (not, or, and)

Tipos de datos

- Numéricos
  - Tipo entero (int)
  - Tipo real (float)
- Booleanos (bool)
- Secuencia
  - Tipo lista (list)
  - Tipo tuplas (tuple)
- Cadenas de caracteres
  - Tipo cadena (str)
- Mapas o diccionario (dict)

Función *type()*

- La función *type()* nos devuelve el tipo de dato de un objeto dado.
- Ejemplos:

```
>>> type(5)
<class 'int'>
>>> type(5.5)
<class 'float'>
>>> type("hola")
<class 'str'>
>>> type([1,2])
<class 'list'>
```

Datos numéricos- TIPOS

- Enteros (int): Representan todos los números enteros (positivos, negativos y 0), sin parte decimal. En python3 este tipo no tiene limitación de espacio.
- Reales (float): Sirve para representar los números reales, tienen una parte decimal y otra decimal. Normalmente se utiliza para su implementación un tipo double de C.

```
>>> entero = 7
>>> type(entero)
<class 'int'>
>>> real = 7.2
>>> type (real)
<class 'float'
```

Operadores aritméticos

- +: Suma dos números
- -: Resta dos números
- *: Multiplica dos números
- /: Divide dos números, el resultado es float.
- //: División entera
- %: Módulo o resto de la división
- **: Potencia
- +, -: Operadores unarios positivo y negativo

Funciones predefinidas que trabajan con números

- abs(x): Devuelve al valor absoluto de un número.
- divmod(x,y): Toma como parámetro dos números, y devuelve una tubla con dos valores, la división entera, y el módulo o resto de la división.
- hex(x): Devuelve una cadena con la representación hexadecimal del número que recibe como parámetro.
- bin(x): Devuelve una cadena con la representación binaria del número que recibe como parámetro.
- pow(x,y): Devuelve la potencia de la base x elevado al exponente y. Es similar al operador**`.
- round(x,[y]): Devuelve un número real (float) que es el redondeo del número recibido como parámetro, podemos indicar un parámetro opcional que indica el número de decimales en el redondeo.

```
>>> abs(-7)
7
>>> divmod(7,2)
(3, 1)
>>> hex(255)
'0xff'
>>> pow(2,3)
8
>>> round(7.567,1)
7.6
```

Conversión de tipos

- int(x): Convierte el valor a entero.
- float(x): Convierte el valor a float.

```
>>> a=int(7.2)
>>> a
7
>>> type(a)
<class 'int'>
>>> a=int("345")
>>> a
345
>>> type(a)
<class 'int'>
>>> b=float(1)
>>> b
1.0
>>> type(b)
<class 'float'>
>>> b=float("1.234")
>>> b
1.234
>>> type(b)
<class 'float'>
```

- Si queremos convertir una cadena a entero, la cadena debe estar formada por caracteres numéricos, sino es así, obtenemos un error:

```
a=int("123.3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123.3'
```

Otras operaciones

- En el módulo math encontramos muchas de estas operaciones. Para utilizarlas vamos a importar el módulo, por ejemplo para realizar una raíz cuadrada:

```
>>> import math
>>> math.sqrt(9)
3.0
```

Datos booleanos- TIPOS

- El tipo booleano o lógico se considera como un subtipo del tipo entero.
- Se puede representar dos valores: verdadero o falso (True, False).
- Cuando se evalúa una expresión, hay determinados valores que se interpretan como False:
  - False
  - Cualquier número 0. (0, 0.0)
  - Cualquier secuencia vacía ([], (), ‘’)
  - Cualquier diccionario vacío ({})

Operaciones de comparación

- Las expresiones lógicas utilizan operadores de comparación
- Permiten comparar dos valores y devuelven un valor booleano, dependiendo de lo que este comparando.
  - ==: Igual que
  - !=: Distinto que
  - >: Mayor que
  - <: Menor que
  - <=: Menor o igual
  - >=: Mayor o igual

Operadores booleanos o lógicos

- Los operadores booleanos se utilizan para operar sobre expresiones booleanas
- Se suelen utilizar en las estructuras de control alternativas (if, while):
  - x or y: Si x es falso entonces y, sino x. Este operados sólo evalúa el segundo argumento si el primero es False.
  - x and y: Si x es falso entonces x, sino y. Este operados sólo evalúa el segundo argumento si el primero es True.
  - not x: Si x es falso entonces True, sino False.

VARIABLE

- Una variables es un identificador que referencia a un valor.
- No hay que declarar la variable antes de usarla, el tipo de la variable será el mismo que el del valor al que hace referencia.
- El tipo de una variable puede cambiar en cualquier momento:

```
>>> var = 5
>>> type(var)
<class 'int'>
>>> var = "hola"
>>> type(var)
<class 'str'>
```

Creación, borrado y ámbito de variables

- Para crear una variable simplemente tenemos que utilizar un operador de asignación, el más utilizado = para que referencia un valor.
- Si queremos borrar la variable utilizamos la instrucción del.
- El ámbito de una variable se refiere a la zona del programa donde se ha definido y existe esa variable.
- Las variables creadas dentro de funciones o clases tienen un ámbito local, es decir no existen fuera de la función o clase.

```
>>> a = 5
>>> a
5
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Modificación del valor de una variable

- En cualquier momento podemos cambiar el valor de una variable, asignándole un nuevo valor:

```
>>> a = 5
>>> a
5
>>> a = 8
>>> a
8
```

- También podemos modificar el valor de una variable, por ejemplo si queremos incrementarla en uno, podríamos usar:

```
>>> a = a + 1
```

- Aunque también podemos utilizar otro operador de asignación:

```
>>> a+=1
```

- Otros operadores de asignación: +=, -=, *=, /=, %=, **=, //=.

FUNCION *INPUT*

- No permite leer por teclado información.
- Devuelve una cadena de caracteres y puede tener como argumento una cadena que se muestra en pantalla.
- Ejemplos:

```
>>> nombre=input("Nombre:")
Nombre:jose
>>> nombre
'jose'
>>> edad=int(input("Edad:"))
Edad:23
>>> edad
23
```

FUNCION *PRINT*

- No permite escribir en la salida estándar.
- Podemos indicar varios datos a imprimir, que por defecto serán separado por un espacio.
- Podemos también imprimir varias cadenas de texto utilizando la concatenación.

```
>>> print(1,2,3)
1 2 3

>>> print("Hola son las",6,"de la tarde")
Hola son las 6 de la tarde

>>> print("Hola son las "+str(6)+" de la tarde")
Hola son las 6 de la tarde
```

Formateo cadenas de caracteres

- Con la función *print* podemos indicar el formato con el que se va a mostrar los datos
- Ejemplo:

```
>>> print("%d %f %s" % (2.5,2.5,2.5))
2 2.500000 2.5

>>> print("El producto %s cantidad=%d precio=%.2f"%
("cesta",23,13.456))
El producto cesta cantidad=23 precio=13.46
```

CADENAS de Caracteres- Definición

- Las cadenas de caracteres (str): Me permiten guardar secuencias de caracteres.
- Podemos definir una cadena de caracteres de distintas formas:

```
>>> cad1 = "Hola"
>>> cad2 = '¿Qué tal?'
>>> cad3 = '''Hola,
    que tal?'''
```

Operaciones básicas

- **Concatenación**: +: El operador + me permite unir datos de tipos secuenciales, en este caso dos cadenas de caracteres.

```
>>> "hola " + "que tal"
'hola que tal'
```

- **Repetición**: *: El operador * me permite repetir un dato de un tipo secuencial, en este caso de cadenas de caracteres.

```
>>> "abc" * 3
'abcabcabc'
```

- **Indexación**: Puedo obtener el dato de una secuencia indicando la posición en la secuencia. En este caso puedo obtener el carácter de la cadena indicando la posición (empezando por la posición

```
>>> cadena = "josé"
>>> cadena[0]
'j'
>>> cadena[3]
'é'
```

- Para obtener la longitud de un caracteres que tiene), utilizamos la función len:

```
>>> cadena = "josé"
>>> len(cadena)
4
```

Comparación de cadenas

- Las cadenas se comparan carácter a carácter, en el momento en que dos caracteres no son iguales se compara alfabéticamente (es decir, se convierte a código unicode y se comparan).
- Ejemplos

```
>>> "a">"A"
True

>>> "informatica">"informacion"
True

>>> "abcde">"abcdef"
False
```

Estructuras de control

Python

Estructuras de control - Alternativa simple

- Al ejecutarse la instrucción if se evalúa la condición lógica.
  - Si la condición lógica es True se ejecutan de manera secuencial el bloque de instrucciones .
  - Si la condición es False no se ejecuta el bloque de instrucciones.
  - Una vez ejecutado el if (opción verdadera o falsa) se continúa la ejecución de forma secuencial por la siguiente instrucción (bloque de instrucción no identado).

```
edad = int(input("Dime tu edad:"))
if edad>=18:
    print("Eres mayor de edad")
print("Programa terminado")
```

Estructuras de control - Alternativa doble

- Al ejecutarse la instrucción if se evalúa la condición lógica.
  - Si la condición lógica es True se ejecutan de manera secuencial el primer bloque de instrucciones.
  - Si la condición es False se ejecuta el segundo bloque de instrucción.
  - Una vez ejecutado el if (opción verdadera o falsa) se continúa la ejecución de forma secuencial por la siguiente instrucción (bloque de instrucción no identado).

```
edad = int(input("Dime tu edad:"))
if edad>=18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
print("Programa terminado")
```

Estructuras de control - Alternativa múltiple

- En este caso tenemos varias opciones. Vamos preguntando por cada una de las opciones y según el valor de la expresión ejecutamos un bloque o otro.

```
nota = int(input("Dime tu nota:"))
if nota >=1 and nota <= 4:
    print("Suspenso")
elif nota == 5:
    print("Suficiente")
elif nota == 6 or nota == 7:
    print("Bien")
elif nota == 8:
    print("Notable")
elif nota ==9 or nota == 10:
    print("Sobresaliente")
else:
    print("Nota incorrecta")
print("Programa terminado")
```

Estructura repetitiva - While

- La instrucción while ejecuta una secuencia de instrucciones mientras una condición sea verdadera.

```
while <condición>:
    <instrucciones>
```

  - Al ejecutarse esta instrucción, la condición es evaluada. Si la condición resulta verdadera, se ejecuta una vez la secuencia de instrucciones que forman el cuerpo del ciclo. Al finalizar la ejecución del cuerpo del ciclo se vuelve a evaluar la condición y, si es verdadera, la ejecución se repite. Estos pasos se repiten mientras la condición sea verdadera.
  - Se puede dar la circunstancia que las instrucciones del bucle no se ejecuten nunca, si al evaluar por primera vez la condición resulta ser falsa.
  - Si la condición siempre es verdadera, al ejecutar esta instrucción se produce un ciclo infinito. A fin de evitarlo, las instrucciones del cuerpo del ciclo deben contener alguna instrucción que modifique la o las variables involucradas en la condición, de modo que ésta sea falsificada en algún momento y así finalice la ejecución del ciclo.

ejemplo

- Crea un programa que pida al usuario una contraseña, de forma repetitiva mientras que no introduzca “asdasd”. Cuando finalmente escriba la contraseña correcta, se le dirá “Bienvenido” y terminará el programa.

```
secreto = "asdasd"
clave = input("Dime la clave:")
while clave != secreto:
    print("Clave incorrecta!!!")
    clave = input("Dime la clave:")
print("Bienvenido!!!")
print("Programa terminado")
```

Instrucción *break*

- Termina la ejecución del bucle, además no ejecuta el bloque de instrucciones.
- Ejemplo

```
secreto = "asdasd"
clave = input("Dime la clave:")
while clave != secreto:
    print("Clave incorrecta!!!")
    otra = input("¿Quieres introducir otra clave (S/N)?:")
    if otra.upper()=="N":
        break;
    clave = input("Dime la clave:")
if clave == secreto:
    print("Bienvenido!!!")
print("Programa terminado")
```

Instrucción *continue*

- Deja de ejecutar las restantes instrucciones del bucle y vuelve a iterar.
- Ejemplo: usamos la instrucción *continue* para mostrar los números pares del 1 al 10:

```
cont = 0
while cont<10:
    cont = cont + 1
    if cont % 2 != 0:
        continue
    print(cont)
```

Instrucción *repetir – hasta que*

- En Python podemos simular el comportamiento de la estructura *repetir-hasta que* utilizando un bucle *while* con la instrucción *break*:

```
secreto = "asdasd"
while True:
    clave = input("Dime la clave:")
    if clave != secreto:
        print("Clave incorrecta!!!")
    if clave == secreto:
        break;
print("Bienvenido!!!")
print("Programa terminado")
```

Estructura repetitiva - for

- La estructura *for* nos permite recorrer los elementos de una secuencia (lista, tupla, cadena de caracteres,…).
- Vamos a usar *for* de forma similar a la instrucción *Para* de pseudocódigo, es decir, ejecutar una secuencia de instrucciones un número determinado de veces, desde un valor inicial, hasta un valor final y con un posible incremento.
  - Para ello vamos a usar el tipo de datos *range* que nos permite generar listas de números. Vamos a usar *for* para crear bucles de instrucciones donde sabemos a priori el número de iteraciones que hay que realizar.

ejemplos

- Escribir en pantalla del 1 al 10

```
for var in range(1,11):
    print(var," ",end="")
```

- Escribir en pantalla del 10 al 1

```
for var in range(10,0,-1):
    print(var," ",end="")
```

- Escribir los número pares desde el 2 al 10

```
for var in range(2,11,2):
    print(var," ",end="")
```

Variables- Contador

- Un **contador** es una variable entera que la utilizamos para contar cuando ocurre un suceso.

- Se inicializa a un valor inicial.

  cont = 0;

- Se incrementa, cuando ocurre el suceso que estamos contado se le suma 1.

  cont = cont + 1;

- Otra forma de incrementar el contador:

  cont += 1

- Ejemplo. Introducir 5 números y contar los pares

```
cont = 0;
for var in range(1,6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        cont = cont + 1
print("Has introducido ",cont," números pares.")
```

Variables- Acumulador

- Un **acumulador** es una variable numérica que permite ir acumulando operaciones. Permite ir haciendo operaciones parciales.
- Se inicializa a un valor inicial según la operación que se va a acumular:
  - a 0 si es una suma
  - a 1 si es un producto.
- Se acumula un valor intermedio. acum = acum + num;

- Ejemplo. Introducir 5 números y sumar los pares

```
suma = 0;
for var in range(1,6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        suma = suma + num
print("La suma de los números pares es ",suma)
```

Variables- Indicador

- Un **indicador** es una variable lógica, que usamos para recordar o indicar algún suceso.

- Se inicializa a un valor lógico que indica que el suceso no ha ocurrido.

  indicador = False

- Cuando ocurre el suceso que queremos recordar cambiamos su valor.

  indicador = True

- Ejemplo. Introducir 5 números e indicar si se ha introducido algún número par

```
indicador  =  False;
for var in range(1,6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        indicador  = True
if indicador:
    print("Has introducido algún número par")
else:
    print("No has introducido algún número par")
```

Secuencias

Tipo de datos

Cadenas de caracteres

- Las cadenas de caracteres (str) permiten guardar secuencias de caracteres.

- Además de las operaciones:
  - Concatenación: +: El operador + me permite unir datos de tipos secuenciales, en este caso dos cadenas de caracteres.
  - Repetición: *: El operador * me permite repetir un dato de un tipo secuencial, en este caso de cadenas de caracteres.
  - Indexación: Puedo obtener el dato de una secuencia indicando la posición en la secuencia. En este caso puedo obtener el carácter de la cadena indicando la posición (**empezando por la posición 0**).
- Para obtener la longitud de una cadena (número de caracteres que tiene), utilizamos la función len.

Operaciones

```
>>> cadena = "informática"
>>> for caracter in cadena:
...     print(caracter,end="")
...
informática
```

- Las cadenas de caracteres se pueden recorrer:
- Operadores de pertenencia: Se puede comprobar si un elemento (subcadena) pertenece o no a una cadena de caracteres con los operadores in y not in.

```
>>> "a" in cadena
True
>>> "b" in cadena
False
>>> "a" not in cadena
False
```

- Slice (rebanada): Puedo obtener una subcadena de la cadena de caracteres. Se indica el carácter inicial, y el carácter final, además podemos indicar opcionalmente un salto. Si no se indica el carácter inicial se supone que es desde el primero, sino se indica el carácter final se supone que es hasta el final. Por último podemos usar salto negativo para empezar a contar desde el final.

Mas operaciones

- cadena[start:end] # Elementos desde la posición start hasta end-1
- cadena[start:] # Elementos desde la posición start hasta el final
- cadena[:end] # Elementos desde el principio hasta la posición end-1
- cadena[:] # Todos Los elementos
- cadena[start:end:step] # Igual que el anterior pero dando step saltos.

- Ejemplos:

```
>>> cadena[2:5]
'for'
>>> cadena[2:7:2]
'frá'
>>> cadena[:5]
'infor'
>>> cadena[5:]
'mática'
>>> cadena[-1:-3]
''
>>> cadena[::-1]
'acitámrofni'
```

Conversión de tipos

- Podemos convertir cualquier número en una cadena de caracteres utilizando la función str:

```
>>> cad = str(7.8)
>>> type(cad)
<class 'str'>
>>> print(cad)
7.8
```

Propiedad – inmutabilidad (I)

- Cuando creamos una variable de tipo cadena de caracteres, estamos creando un objeto de la clase str.
- Cada vez que creamos una variable de una determinada clase, creamos un objeto, que además de guardar información (en nuestro caso los caracteres de la cadena) puede realizar distintas operaciones que llamamos métodos.
- Ejemplo: el método upper() nos permite convertir la cadena a mayúsculas.

Propiedad – inmutabilidad (II)

- No podemos cambiar los caracteres de una cadena de la siguiente forma:

```
>>> cadena = "informática"
>>> cadena[2]="g"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- Esto implica que al usar un método la cadena original no cambia, el método devuelve otra cadena modificada. Ejemplo:

```
>>> cadena = "informática"
>>> cadena.upper()
'INFORMÁTICA'
>>> cadena
'informática'
```

- Si queremos cambiar la cadena debemos modificar su valor con el operador de asignación:

```
>>> cadena = cadena.upper()
>>> cadena
'INFORMÁTICA'
```

Métodos de formato

- capitalize() nos permite devolver la cadena con el primer carácter en mayúsculas.
- lower() y upper() convierte la cadena de caracteres en minúsculas y mayúsculas respectivamente.
- swapcase(): devuelve una cadena nueva con las minúsculas convertidas a mayúsculas y viceversa.
- title(): Devuelve una cadena con los primeros caracteres en mayúsculas de cada palabra.

- Ejemplos

```
>>> cad = "hola, como estás?"
>>> print(cad.capitalize())
Hola, como estás?
```

```
>>> cad = "Hola Mundo"
>>> print(cad.lower())
hola mundo

>>> cad = "hola mundo"
>>> print(cad.upper())
HOLA MUNDO
```

```
>>> cad = "Hola Mundo"
>>> print(cad.swapcase())
hOLA mUNDO
```

```
>>> cad = "hola mundo"
>>> print(cad.title())
Hola Mundo
```

Métodos de búsqueda

- count(): Es un método al que indicamos como parámetro una subcadena y cuenta cuantas apariciones hay de esa subcadena en la cadena.
- Además podemos indicar otro parámetro para indicar la posición desde la que queremos iniciar la búsqueda. Y otro parámetro optativo para indicar la posición final de búsqueda.
- find() nos devuelve la posición de la subcadena que hemos indicado como parámetro. Sino se encuentra se devuelve -1.

- Ejemplos

```
>>> cad = "bienvenido a mi aplicación"
>>> cad.count("a")
3
```

```
>>> cad.count("a",16)
2
>>> cad.count("a",10,16)
1
```

```
>>> cad.find("mi")
13
>>> cad.find("hola")
-1
```

Métodos de validación

- startswith() nos indica con un valor lógico si la cadena empieza por la subcadena que hemos indicado como parámetro. Podemos indicar también con otro parámetro la posición donde tiene que buscar.
- endswith() igual que la anterior pero indica si la cadena termina con la subcadena indicada. En este caso, se puede indicar la posición de inicio y final de búsqueda.
- Otras funciones de validación: isdigit(), islower(), isupper(), isspace(), istitle(),…

- Ejemplos

```
>>> cad.startswith("b")
True
>>> cad.startswith("m")
False
>>> cad.startswith("m",13)
True
```

```
>>> cad.endswith("ción")
True
>>> cad.endswith("ción",0,10)
False
>>> cad.endswith("nido",0,10)
True
```

Métodos de sustitución

- replace(): Devuelve una cadena donde se ha sustituido las apariciones de la primera subcadena indicada por la segunda subcadena indicada como parámetro.

```
>>> buscar = "nombre apellido"
>>> reemplazar_por = "Juan Pérez"
>>> print ("Estimado Sr. nombre apellido:".replace(buscar, reemplazar_por))
Estimado Sr. Juan Pérez:
```

- strip(): Devuelve una cadena donde se han quitado los espacios del principio y del final. Si indicamos una subcadena como parámetro quitará dicha subcadena del principio y del final.

```
>>> cadena = "   www.eugeniabahit.com
>>> print(cadena.strip())
www.eugeniabahit.com
>>> cadena="00000000123000000000"
>>> print(cadena.strip("0"))
123
```

Métodos de unión y división

- el método split() nos permite convertir una cadena en una lista.

```
>>> hora = "12:23:12"
>>> print(hora.split(":"))
['12', '23', '12']
```

- splitlines(): Nos permite separar las líneas que hay en una cadena (indicada con el carácter \n) en una lista.

```
>>> texto = "Linea 1\nLinea 2\nLinea 3"
>>> print(texto.splitlines())
['Linea 1', 'Linea 2', 'Linea 3']
```