Máster Telefónica en Big Data y Business Analytics

Módulo 4. Datascience Tradicional: Análisis de Datos y Aprendizaje Automático con Python.

Unidad 1. Introducción a Python.

2016

Contenidos

1. Objetivos de este módulo

El objetivo principal de este módulo es que el alumno comprenda los conceptos básicos de programación en Python. Se ha elegido este lenguaje ya que, gracias a la gran cantidad de librerías orientadas al análisis de datos, que se han creado en los últimos tiempos y a la simplicidad de su sintaxis hace que sea un lenguaje ideal para los principiantes.

En este documento se tratarán los aspectos básicos del lenguaje empezando por los tipos de datos y las estructuras de control para después ir añadiendo conceptos un poco más complejos como es la inclusión de clases o el uso de diferentes paquetes.

Al finalizar este documento, se espera que el alumno sea capaz de crear programas simples en Python y que esté preparado para comprender y utilizar las librerías de análisis de datos que se mostraran en el siguiente módulo.

2. Primeros pasos

La pregunta que suele hacerse todo el mundo cuando empieza a estudiar un lenguaje de programación es, por qué utilizar este lenguaje y no otro. La respuesta en este caso se debe a la simplicidad de su sintaxis que permite aprender a programar con facilidad y al gran número de librerías que están disponibles para realizar gran parte de las tareas que necesitamos.

2.1. El zen Python

En el propio intérprete podemos ver cuál es la filosofía de Python.

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
```

```
Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!
```

2.2. Instalación del interprete

Instalar el intérprete de Python en Windows es muy sencillo, una vez descargado[1] el instalador de la página de Python, solo tenemos que realizar los siguientes pasos.

Seleccionamos para que usuario queremos instalar el intérprete.

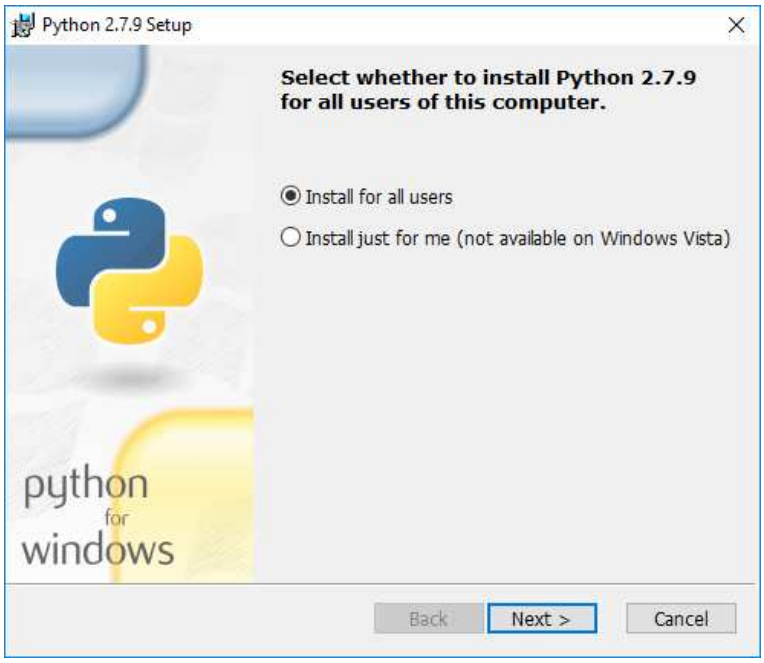


Ilustración 1 – Instalación Python paso 1

[1] Página de descarga del intérprete de python https://www.python.org/downloads/

Después solo tenemos seleccionar el directorio de instalación.

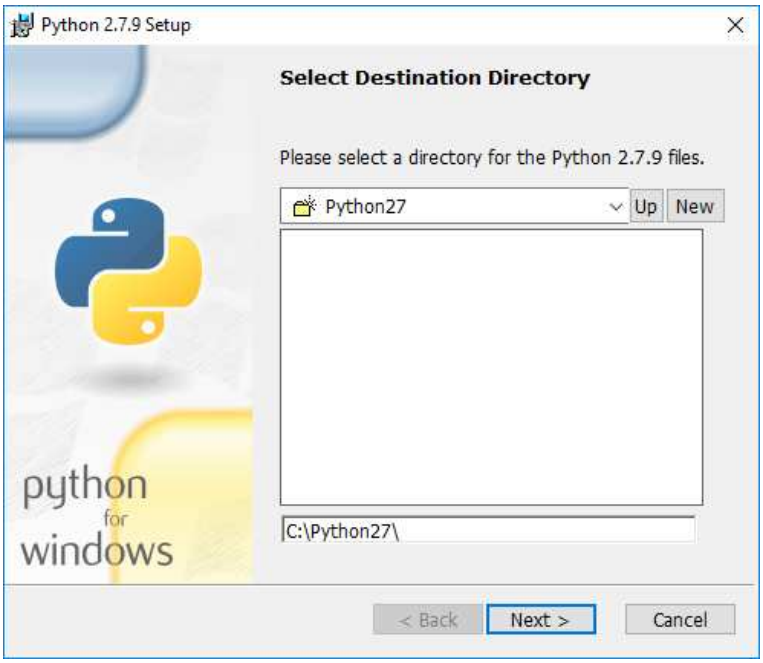


Ilustración 2 – Instalación Python paso 2

Añadimos las librerías por defecto.

Ilustración 3 – Instalación Python paso 3

Y le damos a instalar.

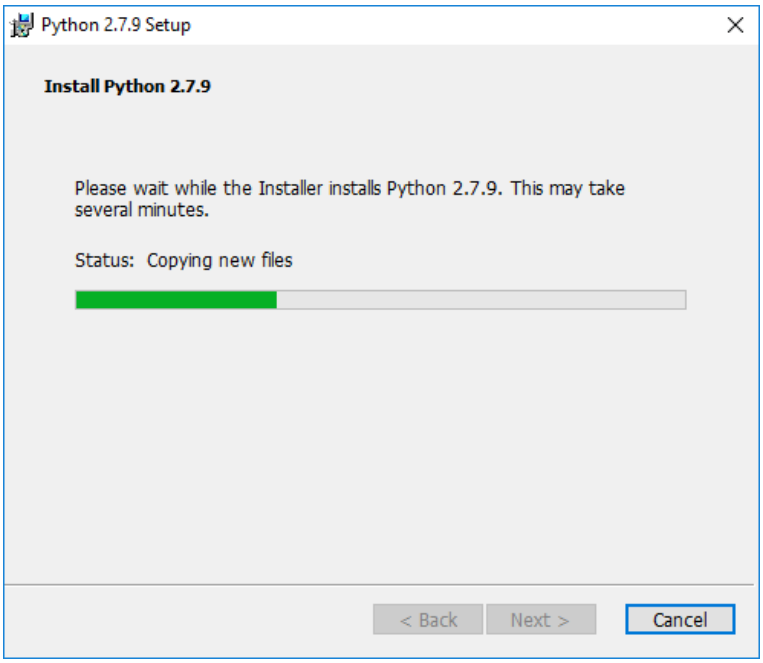


Ilustración 4 – Instalación Python paso 4

Después de unos minutos podemos ver como la instalación se ha completado.

Ilustración 5 – Instalación Python paso 5

Una vez completada la instalación es recomendable añadir la carpeta que contiene el ejecutable de *python* al path para poder ejecutarlo sin tener que escribir la ruta absoluta.

2.2. Instalación de un editor: PyCharm

Además de utilizar la línea de comandos para programar en Python existen múltiples entornos de desarrollo (IDE) que además de desarrollar el código permiten realizar otras muchas tareas. Uno de estos entornos de desarrollo es PyCharm que dispone de una versión community, que nos podemos descargar[2] de forma gratuita. Aquí podemos ver los pasos que hay que seguir para instalarlo.

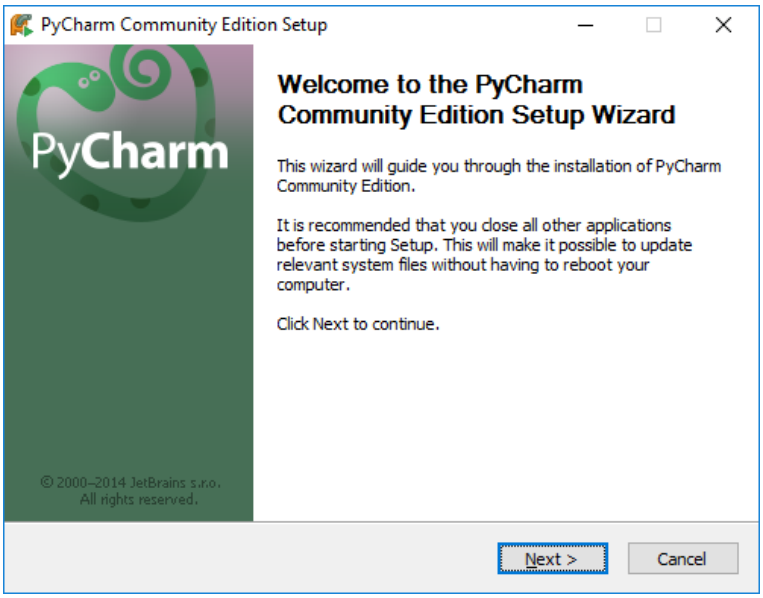


Ilustración 6 – Instalación PyCharm paso 1

[2] Página de descarga de pycharm https://www.jetbrains.com/pycharm/download/

Seleccionamos la ruta donde queremos instalarlo.

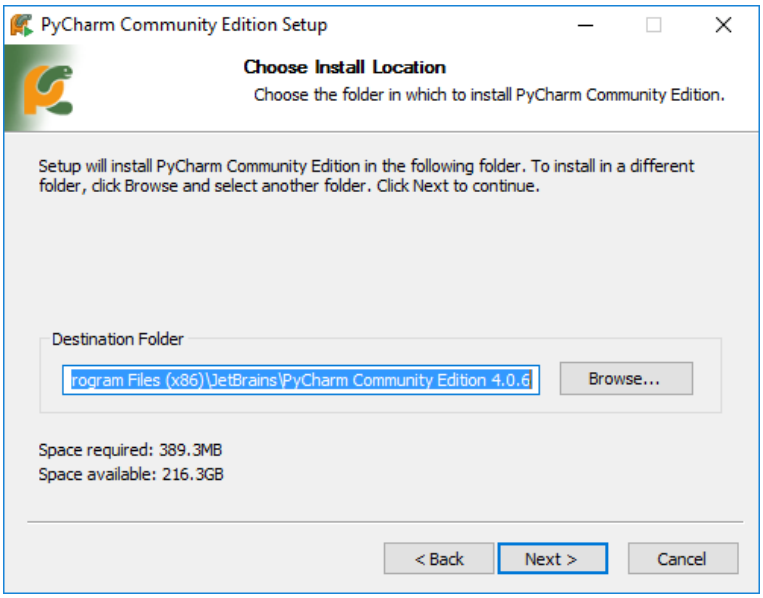


Ilustración 7 – Instalación PyCharm paso 2

Nos pregunta si queremos crear un acceso directo en el escritorio y si queremos que el S.O. abra los ficheros con extensión .py con pycharm.

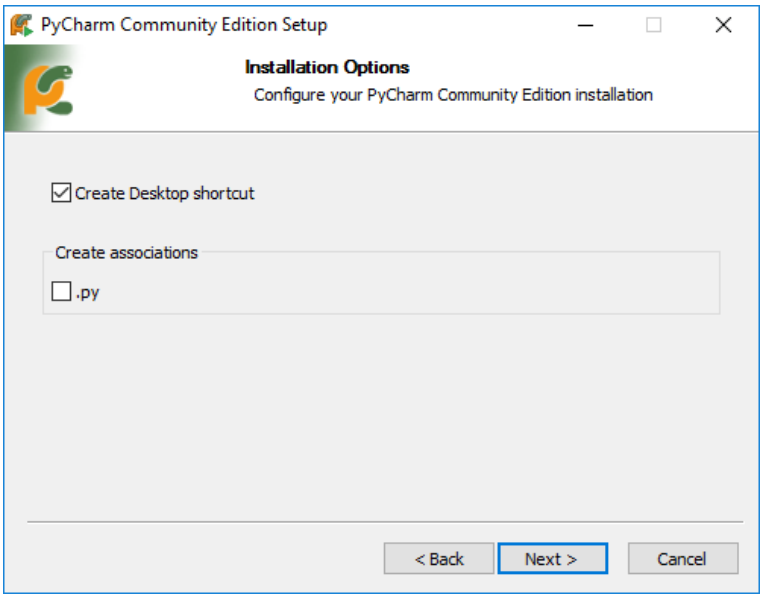


Ilustración 8 – Instalación PyCharm paso 3

Si queremos añadir una carpeta al menú inicio.

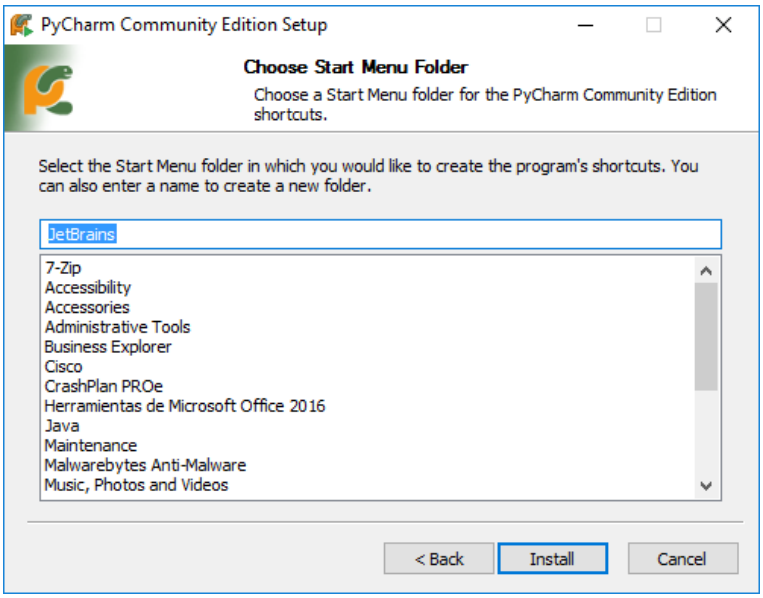


Ilustración 9 – Instalación PyCharm paso 4

Y con eso comenzara la instalación.

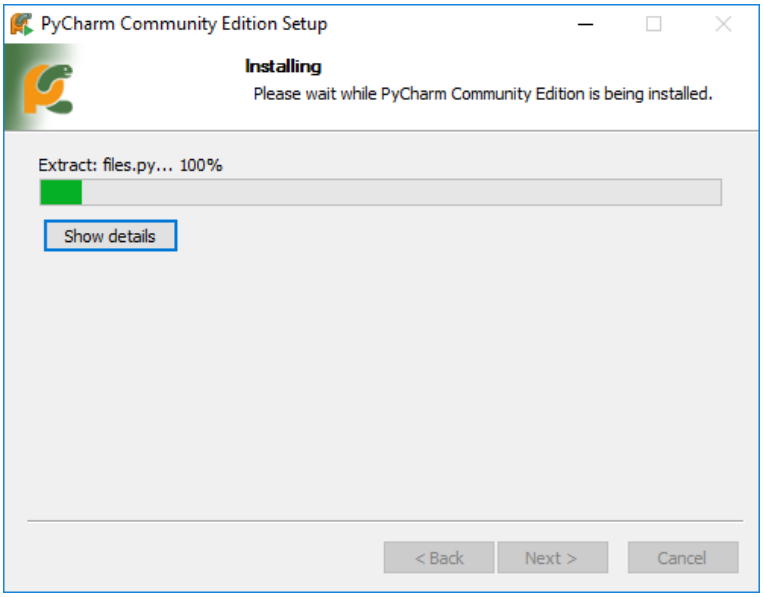


Ilustración 10 – Instalación PyCharm paso 5

Después de unos minutos nos indicara que la instalación se ha completado correctamente.

Ilustración 11 – Instalación PyCharm paso 6

2.3. Instalando un notebook: Anaconda

Una forma rápida de ejecutar código Python es utilizar un notebook de Python, para esto podemos descargarnos[3] el paquete anaconda que incluye muchas utilidades entre ellas el notebook jupyter. Este notebook nos permite escribir código de una manera sencilla. Aquí vemos como instalarlo:

[3] Dirección para descargar anaconda https://www.continuum.io/downloads

Ilustración 12 – Instalación anaconda paso 1

Aceptamos la licencia.

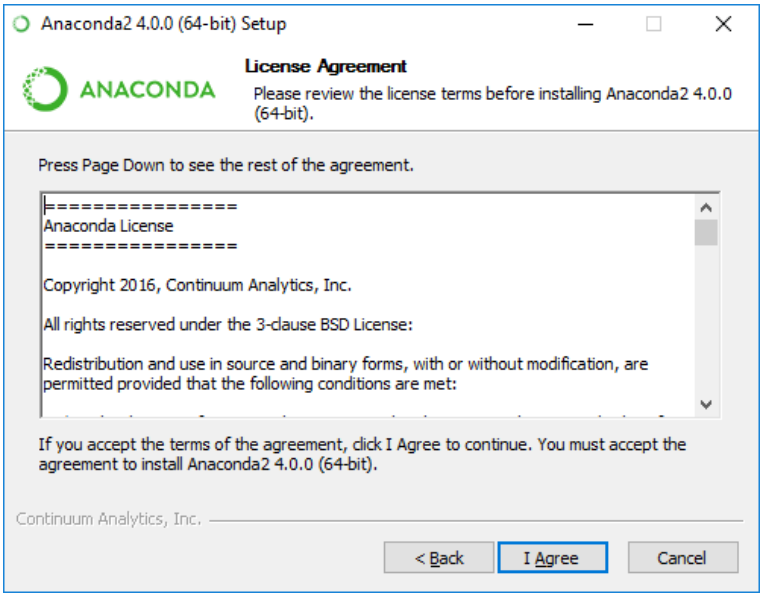


Ilustración 13 – Instalación anaconda paso 2

Elegimos para que usuario queremos que se instale.

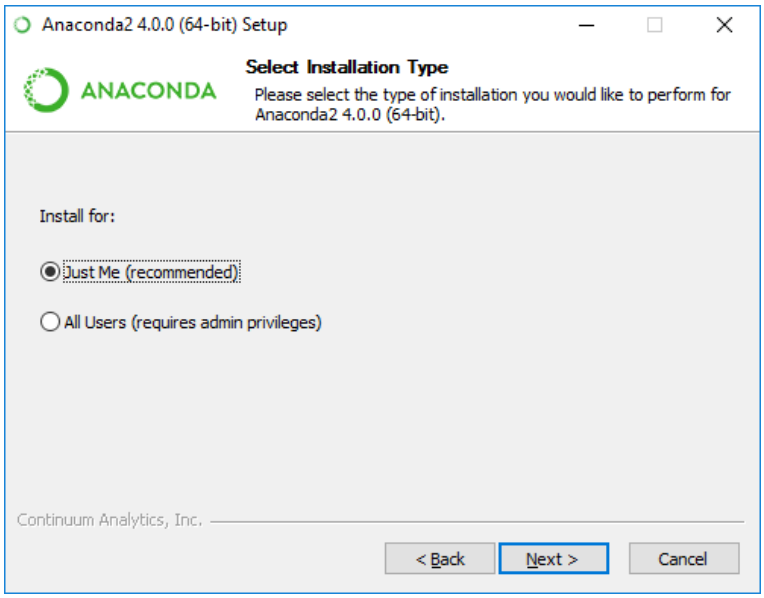


Ilustración 14 – Instalación anaconda paso 3

Seleccionamos el directorio de instalación.

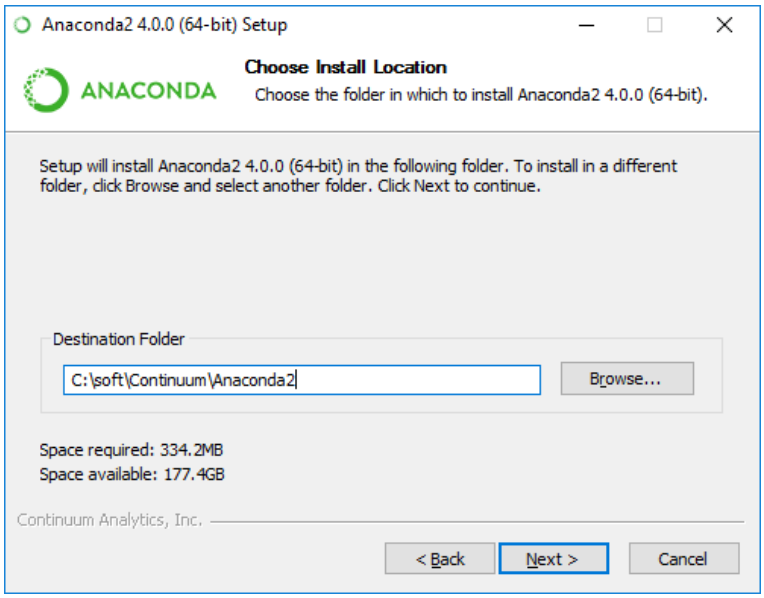


Ilustración 15 – Instalación anaconda paso 4

Y por último tenemos que decidir si añadimos las variables de entorno necesarias al PATH y si queremos que la instalación de python que viene con anaconda sea la instalación por defecto en nuestro entorno.

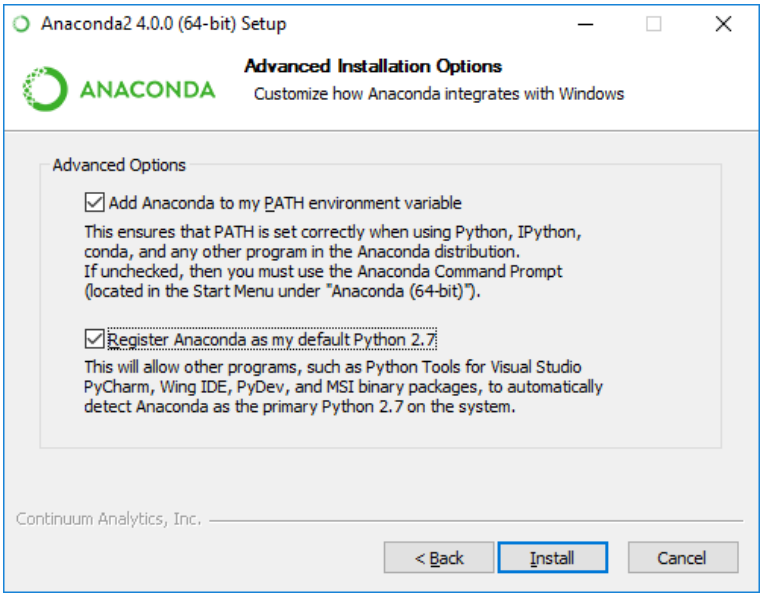


Ilustración 16 – Instalación anaconda paso 5

Una vez hecho esto comienza la instalación.

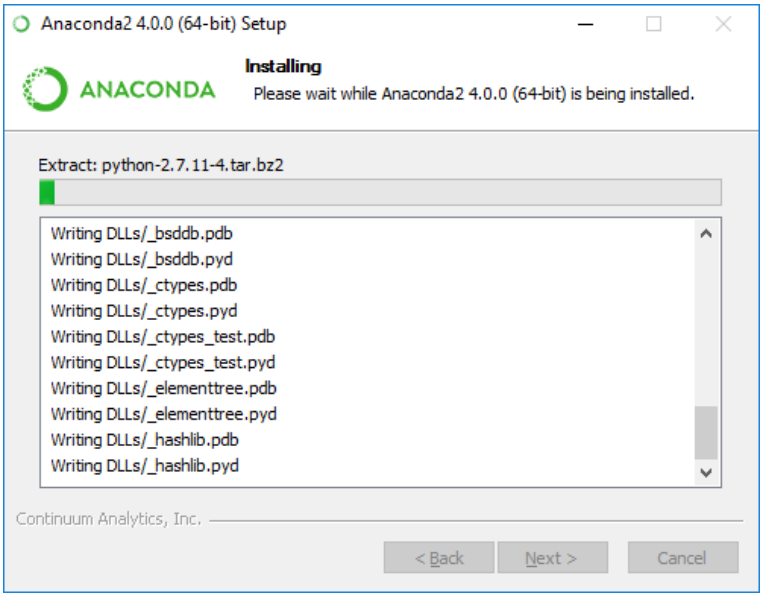


Ilustración 17 – Instalación anaconda paso 6

Después de unos minutos habremos terminado de instalar anaconda en nuestra máquina.

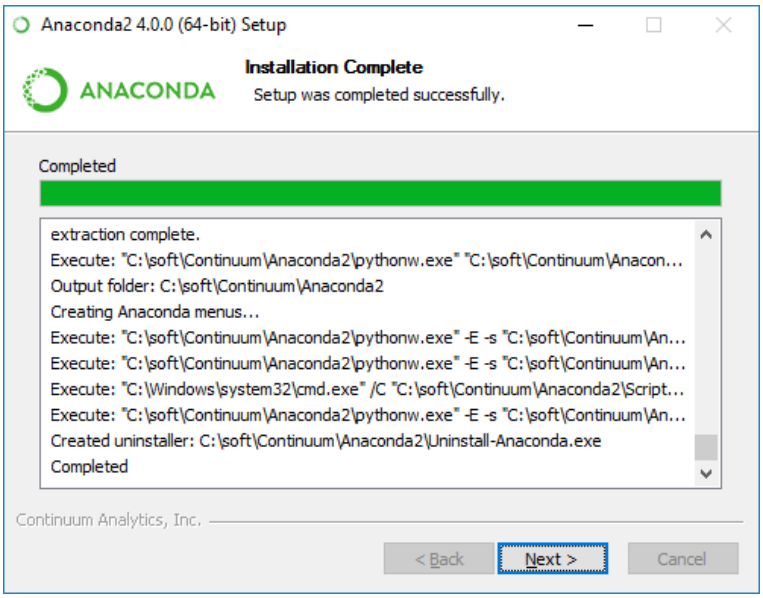


Ilustración 18 – Instalación anaconda paso 7

Solo queda dar a finalizar y listo.

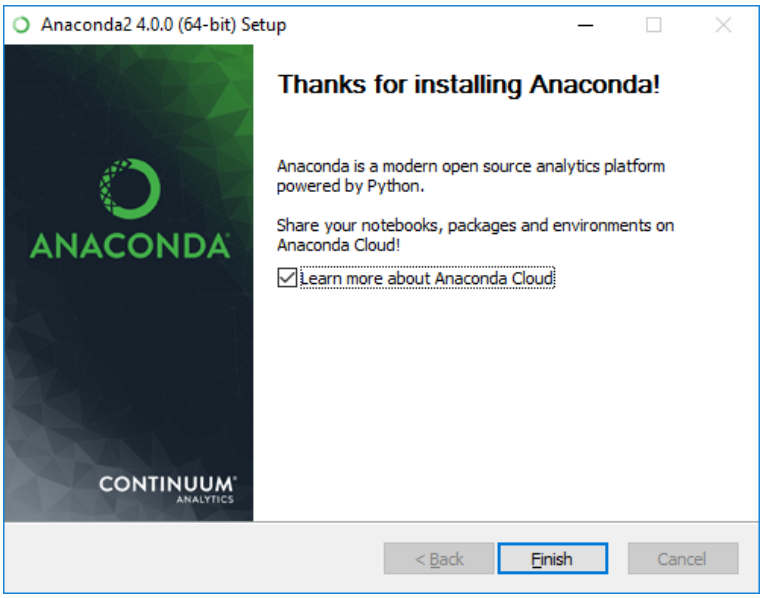


Ilustración 19 – Instalación anaconda paso 8

2.4. Nociones básicas

Una vez instalado el intérprete podemos ejecutarlo simplemente ejecutando el comando python desde la línea de comandos[4].

```
  $> python
  Python 2.7.9 (default, Dec 10 2014, 12:24:55) [MSC v.1500 32 bit (Intel)] on
win32
  Type "help", "copyright", "credits" or "license" for more information.
  >>>
```

Una vez hecho esto se puede empezar a escribir código, este sería el típico programa inicial.

```
print 'hola mundo'
```

Código 1 – Primer código

Este programa permite escribir por pantalla el texto entrecomillado, para ello utilizamos la sentencia `print` que nos permite escribir un texto por la salida estándar, habitualmente la pantalla.

Muchas veces cuando escribimos código necesitamos añadir comentarios para dejar constancia de lo que estamos haciendo. Para escribir un comentario usaremos el carácter `'#'`. Desde la posición de este carácter hasta el final de la línea se considera un comentario y no se tendrá en cuenta al interpretar el código.

```
print 'hola mundo' #esto es un comentario
```

Código 2 – Comentario

Otra característica interesante es como definir variables, en Python las variables se consideran definidas en el momento en que se hace una asignación y el tipo de una variable cambiará si se le asigna un valor de un tipo de dato distinto.

[4] Solo si hemos añadido la ruta al path, manualmente o bien con la opción de anaconda.

```
A = 'Hola'
type(A)
A = 5
type(A)
```

Código 3 – Cambio de tipo dinámico

Una última consideración importante antes de empezar a ver los tipos de datos es que python utiliza el sangrado (indentation en inglés) para separar los distintos bloques de código por lo tanto cada vez que tengamos un bloque de código todas las líneas de ese bloque deberán utilizar el mismo sangrado, típicamente un tabulador o varios espacios. En el modo interactivo del intérprete las líneas en blanco también tienen que mantener el sangrado puesto que no hay forma de saber cuándo acaba un bloque.

3. Tipos de datos predefinidos [5]

Los principales tipos predefinidos en el intérprete de Python son numéricos, secuencias, mapas, ficheros, clases, instancias y excepciones.

Muchas operaciones están definidas para varios tipos distintos; en particular, casi todos los objetos pueden ser comparados, evaluados para ver si son cierto o convertidos a una cadena de texto.

3.1. Booleanos

Un tipo de dato lógico o booleano (`bool`) es en computación aquel que puede tomar dos valores, normalmente *verdadero o falso*. Cualquier objeto puede ser evaluado para ver si es verdadero o falso, para utilizarse como operando de una operación lógica.

En Python se definen los siguientes valores como falso:

- La constante `None`.
- La constante `False`.
- El cero de cualquier tipo numérico (por ejemplo: `0, 0L, 0.0, 0j`)
- Cualquier secuencia vacía (por ejemplo `'', (), []`)
- Cualquier diccionario vacío (por ejemplo `{}`)
- Las instancias de una clase definida por el usuario, si la clase define los métodos `__nonzero__()` o `__len__()` y devuelven un numérico cero o un valor booleano *False*.

El resto de valores se consideran verdaderos, por eso los objetos de muchos tipos se consideran que devuelven el valor booleano verdadero. Además de estos valores también está definida la constante `True` como valor verdadero.

[5] Fuente: https://docs.python.org/2/library/stdtypes.html

3.1.1. Operaciones con booleanos

Las operaciones que están definidas para los tipos de datos booleanos son las siguientes.

3.1.1.1. Operaciones lógicas: And, or y not

Las operaciones lógicas que están definidas son las siguientes, ordenadas por orden de prioridad descendente:

Operación	Resultado	Notas
`x or y`	Si el valor de x es falso devuelve el valor de y, sino devuelve el valor lógico de x.	Solo se evalúa el segundo argumento si el primero es `False`.
`x and y`	Si el valor de x es falso devuelve el valor de x, sino devuelve el valor de y.	Solo se evalúa el segundo argumento si el primero es `True`.
`not y`	Si el valor de x es falso, devuelve `True`, sino devuelve `False`	

Tabla 1 – Operaciones lógicas

```
A = True
B = False

A or B
A and B
not A
not B
```

Código 4 – Operaciones lógicas

3.1.1.2. Comparaciones

Las comparaciones están definidas para todos los objetos. Todas tienen la misma prioridad (y esta es superior a las operaciones booleanas). Estas comparaciones pueden encadenarse de forma arbitraria y en todas ellas solo evaluara una comparación si todas las anteriores devuelven el valor lógico True.

En esta tabla se puede ver el resumen de todas operaciones de comparación:

Operación	Resultado	Notas
`<`	Estrictamente menor que	
`<=`	Menor o igual que	
`>`	Estrictamente mayor que	
`>=`	Mayor o igual que	
`==`	Igual	
`!=`	Distinto	Esta operación también puede escribirse como <> pero se recomienda no utilizar esta forma.
`Is`	Comparación de identidad	
`is not`	Comparación de identidad negada	

Tabla 2 – Comparaciones

Hay que tener en cuenta que objetos de distintos tipos, sin tener en cuenta los objetos numéricos y los diferentes tipos de cadenas de caracteres, nunca devuelven un resultado de iguales en una comparación. Estos objetos se ordenarán de una forma arbitraria (aunque consistente) y en base a ese orden se obtendrán los resultados de las comparaciones.

Las instancias de una clase que no sean la misma referencia siempre devuelven un resultados de distinto, a menos que la clase defina los métodos `__eq__()` o `__cmp__()` y estos métodos devuelvan el valor de igualdad.

Si queremos ordenar las instancias de una misma clase deberemos implementar los métodos de comparación `__lt__()`, `__le__()`, `__gt__()` y `__ge__()` o implementar el método `__cmp__()`.

3.2. Numéricos

En Python existen cuatro tipos de datos numéricos: enteros (`int`), enteros largos (`long`), números decimales de punto flotante (`float`) y números complejos (`complex`). Además, se puede considerar que los booleanos son un subconjunto de los números enteros.

Los numéricos se pueden crear como literales numéricos o como el resultado de una función u operación. Estos numéricos siempre se considerarán como enteros a menos que su valor sea demasiado largo para para ser almacenado de esa forma en cuyo caso se almacenarán como enteros largos. Los literales enteros con una `'L'` o `'l'` como sufijo se almacenarán como enteros largos. Se recomienda utilizar `'L'` puesto que la `'l'` es demasiado parecida a un 1. Un literal numérico que contenga un punto decimal o un signo de exponente se almacena como un número decimal de punto flotante. Si añadimos al literal una `'j'` o `'J'` se almacenará como un número real con parte imaginaria igual a cero.

En Python es posible mezclar diversos tipos numéricos en una misma operación en este caso el operando con el tipo más reducido se ampliará para igualarse al más amplio.

Se pueden utilizar los constructores `int()`, `long()`, `float()` y `complex()` para obtener un numérico de un tipo en concreto.

3.2.1. Operaciones numéricas

Las operaciones que están definidas para todos los tipos de datos numéricos son las siguientes:

Operación	Resultado	Notas
x + y	Suma de x e y	
x - y	Diferencia entre x e y	
x * y	Producto de x e y	
x / y	Cociente de x e y	Para enteros el resultado es un entero.

x // y	Parte entera del cociente de x e *y*	Deprecado para números complejos.
x % y	Resto de x / y	Deprecado para números complejos.
-x	x cambiado de signo	
+x	x sin cambios	
abs(x)	Valor absoluto de x	
int(x)	x convertido en entero	Los decimales se truncan a cero.
long(x)	x convertido en entero largo	Los decimales se truncan a cero.
float(x)	x convertido a punto flotante	
complex(re,im)	Un numero complejo con parte real *re*, y parte imaginaria *im*. Si no existe *im* por defecto es cero.	
c.conjugate()	Conjugado del número complejo c.	
divmod(x, y)	La tupla (x // y, x % y)	Deprecado para números complejos.
pow(x, y)	x elevado a la potencia de *y*	pow(0, 0) está definido como 1.
x ** y	x elevado a la potencia de *y*	0 ** 0 está definido como 1.

Tabla 3 – Operaciones y funciones numéricas

Ejemplo de operaciones con números:

```
a = 24
b = float(2.5)
c = 0.1e-7
d = a + b
e = b / c
f = pow(2, 8)
g = 2 ** 8

print a, b, c, d, e, f, g
```

Código 5 – Operaciones numéricas

Las operaciones que están definidas para todos los tipos de datos reales (entero, entero largo y punto flotante) son las siguientes[6]:

Operación	Resultado	Notas
`math.trunc(x)`	x truncado a la parte entera	
`round(x[, n])`	x redondeado a n dígitos, redondeando hacia arriba. Si se omite n se considera 0.	
`math.floor(x)`	La parte entera más grande, menor o igual que x.	
`math.ceil(x)`	La parte entera más pequeña, mayor o igual que x.	

Tabla 4 – Operaciones numéricas para tipos reales

[6] Algunas de ellas están definidas en el módulo `math`, más adelante veremos cómo funcionan los módulos

Ejemplos de operaciones con reales:

```
import math

b = float(2.54321)

math.trunc(b)
round(b, 2)
math.floor(b)
math.ceil(b)
```

Código 6 – Operaciones numéricas para tipos reales

También se definen operaciones de a nivel de bits, estas operaciones solo tienen sentido para enteros. Los enteros negativos se tratan como su complemento a dos.

Las operaciones a nivel de bit definidas son las siguientes, hay que tener cuidado en no confundir los operadores booleanos *and*, *or* y *not* con las correspondientes operaciones a nivel de bit, ya que el resultado no tiene por ser el mismo:

Operación	Resultado	Notas
x | y	Operación OR bit a bit de x e y	
x ^ y	OR exclusivo bit a bit de x e y	
x & y	Operación AND bit a bit de x e y	
x << n	x desplazamiento a la izquierda de *n* bits	Equivalente a multiplicar por pow(2, n)
x >> n	x desplazamiento a la derecha de *n* bits	Equivalente a dividir por pow(2, n)
~x	Los bits de x invertidos	

Tabla 5 – Operaciones a nivel de bit

Ejemplo con operaciones de nivel de bit.

```
x = 100
y = 1

x | y
x & y
x ^ y
```

Código 7 – Operaciones a nivel de bit

3.3. Secuencias

En Python hay siete tipos de secuencias. Cadenas de caracteres o strings (`str`), cadenas de caracteres en Unicode (`unicode`), arreglos de bytes (`bytearray`), listas (`list`), tuplas (`tuple`) y objetos de xrange (`xrange`). Como podemos ver en Python, a diferencia de otros lenguajes, las cadenas de caracteres se consideran una secuencia y como tales utilizan las mismas funciones que el resto de secuencias.

Los literales del tipo *string* se escriben entre comillas simples o dobles `'abcd'` o `"xyz"`; en el caso de las cadenas en Unicode se utiliza la misma sintaxis, pero se añade como prefijo una `u`. por ejemplo: `u'abc'` o `u"xyz"`. Las listas se construyen utilizando los corchetes, separando los elementos con comas: `[a, b, c]`. Las tuplas construyen utilizando únicamente el operador coma y pueden estar rodeadas por paréntesis o no. Por ejemplo, `a, b, c` o `()`. La principal diferencia entre listas y tuplas es que la tupla es inmutable y una vez creada no podrá ser modificada. Los arreglos de bytes y los buffer se crean respectivamente con sus funciones `bytearray()` y `buffer()`. Los objetos del tipo xrange se pueden crear a partir de la función `xrange()` y sirven para generar un rango sin almacenar todo el rango en memoria, pero al ser un generador el uso de las funciones `in`, `not in`, `min()` y `max()` es ineficiente.

3.3.1. Operaciones con secuencias

Las siguientes operaciones están definidas para todos los tipos de secuencias:

Operación	Resultado	Notas
x in s	Devuelve True si un elemento de la secuencia s es igual a x, sino devuelve False.	En el caso de str y unicode se comporta como un test de substring.
x not in s	Devuelve False si un elemento de la secuencia s es igual a x, sino devuelve True.	En el caso de str y unicode se comporta como un test de substring.
s + t	La concatenación de *s* y *t*.	Para concatenaciones de str que tengan en cuenta el rendimiento es aconsejable usar str.join()
s * n, n * s	Equivalente a añadir a la secuencia s ella misma *n* veces	Los valores no se copian solo se referencian[7].
s[i]	Devuelve el elemento en la posición *i*, empezando por *0*	Si el índice i es negativo se considera que la posición inicial es el final de la secuencia.
s[i:j]	Devuelve la porción de la secuencia s desde el índice *i* al *j*	Si los índices son negativos se considera que la posición inicial para ese índice es el final de la secuencia.
s[i:j:k]	Devuelve la porción de la secuencia s desde el índice *i* al *j* con un salto *k*	Si los índices son negativos se considera que la posición inicial de ese índice es el final de la secuencia. El signo de k indica la dirección en al que se recorre la cadena.
len(s)	Longitud de la secuencia s	
min(s)	El elemento más pequeño de la secuencia s.	

[7] Para más información sobre efectos que se producen debidos a que la copia sea por referencia en vez de por valor consultar la siguiente FAQ: https://docs.python.org/2/faq/programming.html#faq-multidimensional-list

max(s)	El elemento más grande de la secuencia s.	
s.index(x)	La posición de la primera aparición del elemento x en la secuencia s	
s.count(x)	El número total de apariciones del elemento x en la secuencia s	

Tabla 6 – Operaciones y funciones predefinidas con secuencias

Ejemplo de operaciones con secuencias:

```
abc = "abcdefghijklmnopqrstuvwxyz"

abc[5 - 1]
len(abc)
abc[26 - 2]
abc[-2]
abc.index('n')

abc[13 + 1:13 + 1 + 5]
abc[15:5:-2]

abc.center(30, '=')
```

Código 8 – Operaciones con secuencias

En este apartado nos vamos a centrar solamente en los tipos de secuencias más habituales y recomendamos consultar la documentación de Python si queremos ampliar la información sobre el resto de tipos.

3.3.2. Métodos con Cadenas de Texto

Además de los tipos comunes a todas las secuencias, las cadenas de caracteres tienen definidas una gran cantidad de métodos propios, en la siguiente lista podemos ver los más interesantes:

Métodos	Resultado
str.capitalize()	Devuelve una copia de la cadena con la primera letra en mayúsculas.
str.center(*n*[, *fillchar*])	Devuelve una cadena de longitud n con la cadena centrada en ella. El carácter de relleno por defecto es el espacio.
str.count(*sub*[, *start*[, *end*]])	Devuelve el número de ocurrencias de la subcadena *sub* dentro de la cadena. Los parámetros *start* y *end* indican la subcadena en la que se busca.
str.decode([*encoding*[, *errors*]])	Decodifica la cadena utilizando el códec *encoding*. El parámetro *errors* nos permite configurar la forma de tratar los errores[8].
str.encode([*encoding*[, *errors*]])	Codifica la cadena utilizando el códec *encoding*. El parámetro *errors* nos permite configurar la forma de tratar los errores.
str.endswith(*suffix*[, *start*[, *end*]])	Devuelve True si la cadena termina en suffix sino devuelve False. Los parámetros *start* y *end* indican la subcadena en la que se busca.
str.expandtabs([*tabsize*])	Convierte los tabuladores de la cadena en espacios. El parámetro *tabsize* nos indica cuantos espacios equivalen a un tabulador.
str.find(*sub*[, *start*[, *end*]])	Devuelve la posición de la primera ocurrencia de la subcadena *sub* dentro de la cadena. Los parámetros *start* y *end* indican la subcadena en la que se busca. Devuelve -1 si no encuentra la subcadena.
str.format(**args*, ***kwargs*)	Realiza una operación de formateo. El primer argumento es una cadena de caracteres junto con campos a reemplazar rodeados por

[8] Para ver con más detalle cómo funcionan consultar el siguiente enlace https://docs.python.org/2/library/codecs.html

	{}. El resto de argumentos son los valores de los campos a reemplazar.
str.index(*sub*[, *start*[, *end*]])	Igual que find, pero devuelve una excepción si no encuentra la cadena.
str.isalnum()	Devuelve True si la cadena es alfanumérica y al menos contiene un carácter, sino devuelve False.
str.isalpha()	Devuelve True si la cadena es alfabética y al menos contiene un carácter, sino devuelve False.
str.isdigit()	Devuelve True si la cadena es numérica y al menos contiene un carácter, sino devuelve False.
str.islower()	Devuelve True si la cadena es esta en minúsculas y al menos contiene un carácter que sea distinto en mayúsculas y minúsculas, sino devuelve False.
str.isspace()	Devuelve True si la cadena son todo espacios y al menos contiene un carácter, sino devuelve False.
str.istitle()	Devuelve True si la cadena tiene formato de título inglés y al menos contiene un carácter, sino devuelve False.
str.isupper()	Devuelve True si la cadena está en mayúsculas y al menos contiene un carácter que sea distinto en mayúsculas y minúsculas, sino devuelve False.
str.join(*iterable*)	Devuelve una cadena formada por la concatenación de las cadenas que forman parte del iterador *iterable*. El separador entre cadenas es la cadena *str* a la que se le aplica este método.
str.ljust(*width*[, *fillchar*])	Devuelve una cadena justificada a la izquierda. El relleno se hace utilizando el carácter *fillchar*, por defecto es un espacio. Si la longitud es menor que la longitud de la cadena devuelve una copia de la misma cadena.
str.lower()	Devuelve una copia de la cadena con todos los caracteres en minúscula.

str.lstrip([*chars*])	Devuelve una copia de la cadena eliminando del inicio los caracteres especificados en la lista. Si no se especifica el carácter o es None se utilizara el espacio.
str.partition*(sep)*	Devuelve una tupla con tres elementos, la cadena hasta la primera aparición de *sep*, el mismo separador y la cadena detrás del separador. Si el separador no aparece en la cadena devolverá ('str', '', '').
str.replace(*old*, *new*[, *count*])	Devuelve una copia de la cadena reemplazando todas las apariciones de la cadena *old* por la cadena *new*. Si se añade el argumento *count* solo las *count* primeras apariciones se reemplazaran.
str.rfind(*sub*[, *start*[, *end*]])	Devuelve la posición más alta de la ocurrencia de la subcadena *sub* dentro de la cadena. Los parámetros *start* y *end* indican la subcadena en la que se busca. Devuelve -1 si no encuentra la subcadena.
str.rindex(*sub*[, *start*[, *end*]])	Igual que rfind, pero devuelve una excepción si no encuentra la cadena.
str.rjust(*width*[, *fillchar*])	Igual que str.ljust() pero justificando por la derecha.
str.rpartition(*sep*)	Devuelve una tupla con tres elementos, la cadena hasta la última aparición de *sep*, el mismo separador y la cadena detrás del separador. Si el separador no aparece en la cadena devolverá ('', '', 'str').
str.rsplit([*sep*[, *maxsplit*]])	Devuelve una lista de las subcadenas usando el separador *sep*. Si el parámetro *maxsplit está definido solo se devuelven las subcadenas más a la derecha.*
str.rstrip([*chars*])	Devuelve una copia de la cadena eliminando del final los caracteres especificados en la lista. Si no se especifica el carácter o es None se utilizará el espacio.
str.split([*sep*[, *maxsplit*]])	Devuelve una lista de las subcadenas usando el separador *sep*. Si el parámetro *maxsplit está definido solo se devuelven las maxsplit subcadenas.*

	Si el separador no se define o es None *se utiliza un algoritmo especial, agrupando todos los espacios en blanco como un único separador y eliminando las cadenas vacías de la secuencia.*
str.splitlines([*keepends*])	Devuelve una lista con las líneas contenidas en la cadena. Utilizando todos los separadores de línea que existen. Si el parámetro *keepends* existe y es True se conserva el separador de línea, en cualquier otro caso se elimina.
str.startswith(*prefix*[, *start*[, *end*]])	Devuelve True si la cadena empieza por *prefix* sino devuelve False. Los parámetros *start* y *end* indican la subcadena en la que se busca.
str.strip([*chars*])	Devuelve una copia de la cadena eliminando del principio y del final los caracteres especificados en la lista. Si no se especifica el carácter o es None se utilizara el espacio.
str.swapcase()	Devuelve una copia de la cadena cambiando las mayúsculas por las minúsculas y viceversa.
str.title()	Devuelve una cadena puesta en formato título, esto es separa las palabras de la cadena y pone el primer carácter en mayúscula. Hay que tener en cuenta que no se basa en ningún idioma y cada palabra será un grupo de letras separada por caracteres que no sean letras.
str.translate(*table*[, *deletechars*])	Devuelve una copia de la cadena donde los caracteres se reemplazan utilizando la cadena *table* como tabla de reemplazo. Esta cadena tiene que tener las 256 posiciones que se utilizan para mapear los caracteres.
str.upper()	Devuelve una copia de la cadena con todos los caracteres en mayúscula.
str.zfill(*width*)	Devuelve una copia de la cadena numérica *str* rellenando con ceros las posiciones necesarias hasta que el tamaño de la cadena sea *width*. Si la longitud de la cadena es mayor se devolverá la cadena entera.

Tabla 7 – Métodos de la clase str

Ejemplos con la clase str:

```
abc = "abcdefghijklmnopqrstuvwxyz"

abc.center(30, '=')
abc.upper()
abc.find('opq')

lista = abc.split('i')
print lista

'-'.join(lista)
```

Código 9 – Métodos de la clase str

Además de los métodos definidos para `str` los objetos del tipo `unicode` tienen definidos de forma adicional los siguientes métodos.

Métodos	Resultado
unicode.isnumeric()	Devuelve True si la cadena es numérica y al menos contiene un carácter, sino devuelve False. Los caracteres Unicode que son numéricos incluyen todos aquellos que tengan la propiedad de numérico.
unicode.isdecimal()	Devuelve True si la cadena es decimal y al menos contiene un carácter, sino devuelve False. Los caracteres Unicode decimales que son todos dígitos y todos aquellos que puedan usarse para formar un sistema de numeración de base 10.

Tabla 8 – Métodos de clase Unicode

Ejemplos con la clase unicode:

```
uni = u"abcdefghijklmnopqrstuvwxyz”
type(uni)
```

Código 10 – Métodos de la clase unicode

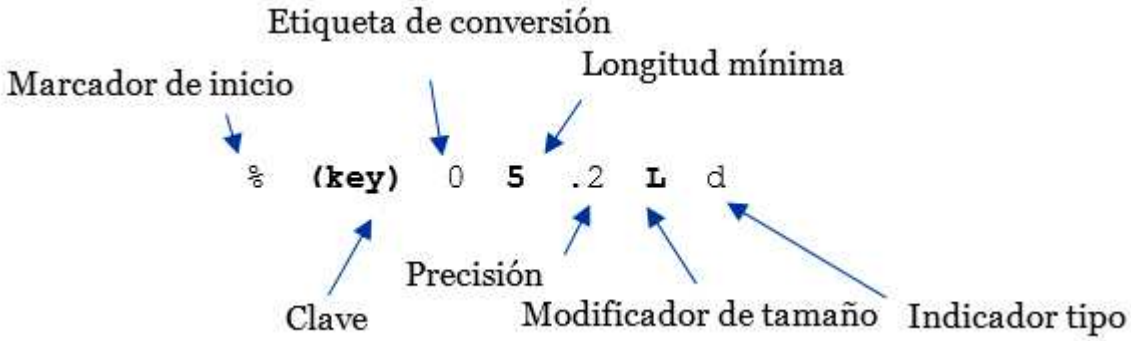
3.3.2.1. Formatear cadenas de texto

Los tipos de datos cadena de texto tienen un operador adicional, el operador % que sirve para dar formato a las cadenas de caracteres.

Métodos	Resultado
format % values	Donde *format* es una cadena de tipo str o unicode. Y *values* es una tupla o un diccionario con los valores de las variables que están definidas en la cadena a formatear.

Tabla 9 – Operación de formateo de cadenas

Esta cadena admite unos marcadores específicos que serán reemplazados por los valores que se pasan en el operando de la derecha. La definición de estos marcadores es la siguiente:



De todos estos marcadores, los únicos obligatorios son el marcador de inicio y el indicador del tipo de conversión, el resto son opcionales. El modificador de tamaño puede incluirse, pero no tiene ningún efecto.

Las diferentes etiquetas que existen se muestran en la siguiente tabla y cada una de ellas nos permite modificar la forma en la que se realiza la conversión.

Etiqueta	Resultado
'#'	El resultado de la conversión utilizara el formato alternativo. Ver en la siguiente tabla.
'0'	El resultado de la conversión utilizará ceros para rellenar los caracteres necesarios para ocupar la longitud mínima.

'-'	El resultado de la conversión se alinea a la izquierda. Tiene preferencia sobre la etiqueta '0'.
' '	Se añade un espacio en blanco a la izquierda de un número positivo (o de una cadena vacía) para un formato con signo.
'+'	Un carácter de signo ('+' o '-') precede al resultado de la conversión. Tiene preferencia sobre la etiqueta ' '.

Tabla 10 – Etiquetas de conversión

Tanto el campo longitud de la conversión como el de precisión son dos campos numéricos. En ambos casos se puede utilizar el carácter especial '*' para indicar que ese valor no es fijo sino variable. En ese caso se leerá el valor de la precisión de la posición actual y el valor a formatear será el siguiente elemento de la tupla.

Tipo	Resultado	Alternativo
'd'	Decimal con signo.	
'i'	Decimal con signo.	
'o'	Valor octal con signo.	El formato alternativo añade un cero '0' a la izquierda del número formateado después de los caracteres de relleno. Si el resultado es 0 no añade nada.
'u'	Obsoleto, es idéntico a 'd'.	
'x'	Valor hexadecimal con signo (en minúsculas).	El formato alternativo añade un cero '0x' a la izquierda del número formateado después de los caracteres de relleno. Si el resultado es 0 no añade nada.
'X'	Valor hexadecimal con signo (en mayúsculas).	El formato alternativo añade un cero '0X' a la izquierda del número formateado después de los caracteres de relleno. Si el resultado es 0 no añade nada.
'e'	Formato exponencial en punto flotante (en minúsculas). Por defecto la precisión es 6.	El formato alternativo siempre muestra el punto decimal incluso si no está seguido por un carácter.

'E'	Formato exponencial en punto flotante (en mayúsculas). Por defecto la precisión es 6.	El formato alternativo siempre muestra el punto decimal incluso si no está seguido por un carácter.
'f'	Formato decimal en punto flotante. Por defecto la precisión es 6.	El formato alternativo siempre muestra el punto decimal incluso si no está seguido por un carácter.
'F'	Formato decimal en punto flotante. Por defecto la precisión es 6.	El formato alternativo siempre muestra el punto decimal incluso si no está seguido por un carácter.
'g'	Formato de punto flotante. Usa el formato exponencial si el exponente es menor que -4 o no es menor que la precisión, sino usa el formato decimal (en minúsculas). Por defecto la precisión es 6.	El formato alternativo siempre muestra el punto decimal incluso si no está seguido por un carácter. Tampoco se eliminaran los ceros aunque normalmente lo hicieran.
'G'	Formato de punto flotante. Usa el formato exponencial si el exponente es menor que -4 o no es menor que la precisión, sino usa el formato decimal (en mayúsculas). Por defecto la precisión es 6.	El formato alternativo siempre muestra el punto decimal incluso si no está seguido por un carácter. Tampoco se eliminaran los ceros aunque normalmente lo hicieran.
'c'	Formato para un único carácter, acepta enteros o cadenas de caracteres de una única posición.	
'r'	Devuelve una cadena de caracteres usando la función repr(). La precisión determina el número máximo de caracteres a mostrar.	
's'	Devuelve una cadena de caracteres usando la función str(). La precisión determina el número máximo de caracteres a mostrar.	
'%'	No se convierte ningún argumento, el resultado es siempre un %.	

Tabla 11 – Tipos de formato de cadenas

Ejemplo de formato de cadenas:

```
"formato %0.2f" % 5.4
```

Código 11 – Formato de cadenas

3.3.4. Operaciones con listas

Además de las operaciones definidas para todas las secuencias, puesto que las listas son un tipo de datos mutable se pueden realizar las siguientes operaciones para modificar sus valores.

Operación	Resultado
s[i] = x	El elemento en la posición *i* de la lista *s* es remplazado por x.
s[i:j] = t	La porción de *s* desde *i* hasta *j* se reemplaza por el contenido del iterable *t*.
del s[i:j]	Es equivalente a s[i:j] = [] es decir elimina los elementos de *s* desde la posición *i* a la *j*.
s[i:j:k] = t	Los elementos de s[i:j:k] son reemplazados por los del iterador *t*. En este caso el iterador *t* debe tener la misma longitud que la porción de *s*.
del s[i:j:k]	Elimina los elementos de s[i:j:k] de la lista.
s.append(x)	Añade el elemento x al final de la lista. Es equivalente a s[len(s):len(s)] = [x]
s.extend(t) o bien s += t	Añade al final de la lista el iterador t. También se puede escribir como s[len(s):len(s)] = t
s *= n	Añade a la lista s su contenido n veces. Los elementos no se copian solo se añade su referencia.
s.count(x)	Devuelve el número de ocurrencias del elemento x. Esta función también funciona para tuplas.
s.index(x[, i[, j]])	Devuelve el k más pequeño que cumple que s[k] == x y i <= k < j. Esta función también funciona para tuplas.
s.insert(i, x)	Añade el elemento x en la posición i de la lista s. Es equivalente a s[i:i] = [x]
s.pop([i])	Saca el elemento en la posición i de la lista. Es equivalente a x = s[i]; del s[i]; return x
s.remove(x)	Elimina la primera aparición de elemento x en la lista s. Es equivalente a del s[s.index(x)]
s.reverse()	Invierte el orden de la lista. Sería equivalente a s = s[::-1]
s.sort([cmp[, key[, reverse]]])	Ordena los elementos de la lista.

Tabla 12 – Operaciones con listas

Ejemplos de operaciones con listas y tuplas:

```
mi_lista = [1, 2, 3, 4]
mi_lista = list([1, 12])

mi_tupla = (1, 2, 3, 4)
mi_tupla = tuple((1, 12))

mi_lista.append(5)
mi_lista.extend((6, 7))
```

Código 12 – Operaciones con listas y tuplas

3.4. Conjuntos

Un conjunto es una colección no ordenada de objetos distintos sobre los que se puede aplicar la función predefinida `hash()`. Al ser una colección no ordenada no se almacena el orden de inserción o la posición de cada elemento, por tanto, no se podrán utilizar las operaciones habituales en las listas como operaciones para trocear el conjunto u obtener un elemento por su posición.

Para crear un conjunto se puede utilizar el constructor `set(iterable)` o bien una lista de elementos separados por comas y rodeados por llaves `{a, b, c}`. Hay que tener en cuenta que si la lista está vacía el objeto que se creará será del tipo `dict`[9].

Al ser un tipo de datos mutable, existen diversas funciones en las que no se podría usar este tipo de datos. Para poder usar un conjunto en estas funciones existe un tipo de conjuntos que es inmutable. Este tipo de conjunto se crea utilizando el constructor `frozenset(iterable)`.

[9] Podemos ver el tipo de datos `dict` en el capítulo 0

3.4.1. Operaciones y métodos con conjuntos

A continuación, se muestra la lista de operaciones aplicables a un conjunto, tanto en su versión mutable como inmutable.

Operación	Resultado
len(s)	Devuelve el número de elementos del conjunto s.
x in s	Comprueba si el elemento x está en el conjunto s. Devuelve True si esta y False en cualquier otro caso.
x not in s	Comprueba si el elemento x no está en el conjunto s. Devuelve False si esta y True en cualquier otro caso.
set.isdisjoint(other)	Devuelve True si no hay ningún elemento en común entre el conjunto *set* y el otro conjunto *other*. En cualquier otro caso devuelve False.
set.issubset(other) set <= other	Devuelve True si el conjunto *set* es un subconjunto del conjunto *other*. En cualquier otro caso devuelve False.
set < other	Devuelve True si el conjunto *set* es un subconjunto propio de *other*. Esto es set <= other and set != other.
set.issuperset(other) set >= other	Devuelve True si el conjunto *set* es un superconjunto del conjunto *other*. En cualquier otro caso devuelve False.
set > other	Devuelve True si el conjunto *set* es un superconjunto propio de *other*. Esto es set >= other and set != other.
set.union(other, ...) set | other | ...	Devuelve un conjunto que contiene todos los elementos del conjunto *set* y de todos los conjuntos *other*.
set.intersection(other, ...) set & other & ...	Devuelve un conjunto que contiene todos los elementos comunes del conjunto *set* y de todos los conjuntos *other*.
set.difference(other, ...) set - other - ...	Devuelve un conjunto que contiene todos los elementos del conjunto *set* y que no están en cualquiera de todos los conjuntos *other*.
set.symmetric_difference(other) set ^ other	Devuelve un conjunto que contiene todos los elementos en el conjunto *set* o en el conjunto *other* pero que no están en ambos.

set.copy()	Devuelve una copia del conjunto *set*.
set == other	Devuelve True si todos los elementos del conjunto *set* están en el conjunto *other* y viceversa. Devuelve False en cualquier otro caso.

Tabla 13 – Operaciones y métodos con conjuntos

Hay que tener en cuenta que los métodos que aceptan un parámetro aceptan no solo conjuntos sino también cualquier tipo de iterable.

En el caso de mezclar en una misma operación objetos del tipo `set` con objetos del tipo `frozenset`, si esa operación devuelve un conjunto, este será del mismo tipo del objeto que este más a la izquierda en la operación.

3.4.2. Operaciones y métodos de conjuntos mutables

Además de las operaciones descritas en el apartado anterior existen operaciones que permiten modificar los elementos de un conjunto, como es obvio estas operaciones no pueden aplicarse a un conjunto de tipo `frozenset` puesto que es inmutable.

Operación	Resultado
set.add(elem)	Actualiza el conjunto *set* añadiendo el elemento *elem*.
set.remove(elem)	Actualiza el conjunto *set* eliminando el elemento *elem*. Si *elem* no está en el conjunto este método lanzara una excepción.
set.discard(elem)	Actualiza el conjunto *set* eliminando el elemento *elem* si existe.
set.pop()	Devuelve un elemento arbitrario del conjunto *set* y después actualiza el conjunto set eliminando ese elemento. Lanza una excepción si el conjunto está vacío.
set.clear()	Actualiza el conjunto set eliminado todos los elementos.
set.update(other, ...) set |= other | ...	Actualiza el conjunto *set* con los todos los elementos de los conjuntos *other*.

set.intersection_update(other, ...) set &= other & ...	Actualiza el conjunto *set* manteniendo únicamente los elementos que aparecen en todos los conjuntos.
set.difference_update(other, ...) set -= other | ...	Actualiza el conjunto *set* eliminado los elementos que aparecen en todos los conjuntos *other*.
set.symmetric_difference_update(other) set ^= other	Actualiza el conjunto *set* manteniendo únicamente los elementos que únicamente aparecen en un conjunto pero no están en ambos.

Tabla 14 – Operaciones con conjuntos de tipo set

Al igual que en la tabla anterior todos los métodos que aceptan un parámetro aceptan no solo conjuntos sino también cualquier tipo de iterable.

Ejemplos de operaciones con conjuntos:

```
mi_conjunto = {25, 28}
otro_conjunto = frozenset(mi_conjunto)

mi_conjunto.add(26)
mi_conjunto.add(25)
print(mi_conjunto)

25 in mi_conjunto

mi_conjunto.pop()
print(mi_conjunto)

25 in mi_conjunto

mi_conjunto == otro_conjunto
```

Código 13 – Operaciones con conjuntos

3.5. Diccionarios

Un diccionario es un objeto de tipo mapa en los que se relaciona una clave, que tiene que ser un objeto sobre él se puede aplicar la función predefinida `hash()`, con un valor que puede ser un objeto de cualquier tipo. Los diccionarios se consideran objetos mutables, lo que significa, como ya hemos visto antes, que su contenido puede modificarse.

Un diccionarios puede crease colocando entre llaves una lista de pares *clave:valor* separados por coma. Por ejemplo: `{"one": 1, "two": 2, "three": 3}`. Otra forma de crear un diccionario es utilizar el constructor de la clase `dict`. Este constructor nos permite crear diccionarios de varias formas. Si se proporciona un argumento en forma de *keyword* (clave=valor) se añadirán al diccionario esos pares de clave valor. Si el argumento es un iterable, cada uno de los elementos de ese iterable debe ser a su vez otro iterable con dos valores. Ejemplos: `dict([('two', 2), ('one', 1), ('three', 3)])` `dict(['ab', 'cd', 'ef'])`

3.5.1. Operaciones y métodos con diccionarios

Los diccionarios contienen los siguientes métodos.

Operación	Resultado
d == other	Comparación entre diccionarios, devuelve True si todos los elementos del diccionario son iguales, en caso contrario devuelve False.
len(d)	Devuelve el número de elementos del diccionario *d*.
d[key]	Devuelve el valor asociado a la clave *key*. Si la clave no existe lanza una excepción.
d[key] = value	Añade al diccionario la clave *key* con el valor *value*, si la clave ya existe modifica el valor
del d[key]	Elimina la clave *key*. Si la clave no existe lanza una excepción.
key in d	Comprueba si la clave *key* está en el diccionario *k*. Devuelve True si está presente y False en caso contrario.
key not in d	Comprueba si la clave *key* no está en el diccionario *k*. Devuelve False si está presente y True en caso contrario.

iter(d)	Devuelve el iterador sobre el objeto diccionario *d*.
d.clear()	Elimina todos los elementos del diccionario.
d.copy()	Devuelve una copia superficial del diccionario d.
d.fromkeys(seq[, value])	Crea un nuevo diccionario a partir de la secuencia de claves *seq*. Estas claves se inicializarán con el valor *value*. Si no proporciona valor por defecto este será None.
d.get(key[, default])	Devuelve el valor asociado a la clave *key*. Si la clave no existe devuelve default. Si no se proporciona el valor default devolverá None.
d.has_key(key)	Comprueba si la clave *key* está en el diccionario *k*. Devuelve True si está presente y False en caso contrario. Este método esta deprecado y es preferible usar el operador in.
d.items()	Devuelve una lista de tuplas compuestas por los pares clave valor que forman el diccionario.
d.iteritems()	Devuelve un iterador sobre las tuplas clave valor del diccionario. Añadir o eliminar elementos en el diccionario mientras se está recorriendo este iterador puede causar una excepción o no recorrer todos los elementos.
d.iterkeys()	Devuelve un iterador sobre las claves del diccionario. Añadir o eliminar elementos en el diccionario mientras se está recorriendo este iterador puede causar una excepción o no recorrer todos los elementos.
d.itervalues()	Devuelve un iterador sobre los valores del diccionario. Añadir o eliminar elementos en el diccionario mientras se está recorriendo este iterador puede causar una excepción o no recorrer todos los elementos.
d.keys()	Devuelve una lista con las claves del diccionario.
d.pop(key[, default])	Si la clave *key* está en el diccionario devuelve el valor correspondiente, sino devuelve el valor *default*. Si el valor *default* no se define y la clave no está en el diccionario lanzara una excepción.
d.popitem()	Devuelve un elemento del diccionario en forma de tupla clave valor y después lo elimina del diccionario.

	Si no hay elementos en el diccionario lanza una excepción.
d.setdefault(key[, default])	Si la clave *key* está en el diccionario devuelve su valor. Sino añade la clave al diccionario con el valor del parámetro *default*. Si este valor no está definido utiliza None como valor.
d.update([other])	Actualiza el diccionario añadiendo los elementos de *other*. Estos valores pueden ser una lista de *keywords* o bien otro diccionario. En caso de que alguna de las claves exista actualiza su valor.
d.values()	Devuelve una lista con los valores del diccionario.
d.viewitems()	Devuelve un objeto vista con los elementos del diccionario, es similar a la función d.items()pero el contenido de la vista se actualiza al actualizar el diccionario.
d.viewkeys()	Devuelve un objeto vista con las claves del diccionario, es similar a la función d.keys()pero el contenido de la vista se actualiza al actualizar el diccionario.
d.viewvalues()	Devuelve un objeto vista con los valores del diccionario, es similar a la función d.values()pero el contenido de la vista se actualiza al actualizar el diccionario.

Tabla 15 – Operaciones y métodos con diccionarios

Ejemplo de operaciones con diccionarios:

```
mi_diccionario = {'Enero': 31, 'Febrero': 28}
mi_diccionario.get('Enero')

mi_diccionario['Enero']
mi_diccionario['Marzo'] = 31

'Enero' in mi_diccionario

otro_diccionario = dict([('Enero', 31), ('Febrero', 28), ('Marzo', 31)])
```

Código 14 – Operaciones y métodos con diccionarios

3.6. Ficheros

Muchos programas necesitan leer ficheros del sistema de ficheros local, para hacer esto tenemos una función predefinida, la función `open()` para abrir los ficheros y una clase, la clase `file` para poder manejar los ficheros abiertos.

3.6.1. Función predefinida open()

La función open nos permite abrir un fichero para utilizarlo en nuestro código.

Función	Resultado
open(name[, mode[, buffering]])	Devuelve un objeto de tipo *file*. El parámetro *name* es una cadena de caracteres con el nombre del fichero, el parámetro *mode* es modo de apertura por defecto se abre en modo lectura. Y por último el parámetro *buffering* para el tamaño del buffer que puede ser: 0 para no usar buffer, 1 para un buffer de línea, un valor negativo para usar el valor por defecto del sistema y un valor mayor que 1 para indicar la cantidad de bytes.

Tabla 16 – Función predefinida open()

Los modos de apertura pueden ser los siguientes:

- `r`, para abrir el fichero en modo lectura.
- `w`, para abrir el fichero en modo escritura, si el fichero existía se sobrescribe.
- `a`, para abrir el fichero en modo añadir al final.

A parte de los modos básicos se puede añadir el valor `b` si queremos abrir el fichero en modo binario y/o el valor + para añadir el fichero en modo lectura/escritura.

3.6.2. Objetos de tipo file

Los objetos de tipo fichero nos permiten manejar los ficheros de nuestro sistema de archivos. Estos objetos suele obtenerse por medio de la función `open()` pero también pueden obtenerse de otras funciones predefinidas y clases. Los ficheros tienen definidos los siguientes atributos.

Atributo	Resultado
file.closed	Un atributo booleano indicando el estado actual del fichero.
file.encoding	Una cadena de caracteres con la codificación del fichero.
file.errors	El manejador de errores de Unicode asociado al fichero.
file.mode	Una cadena con el modo en que está abierto el fichero.
file.name	Una cadena de caracteres con el nombre del fichero que está abierto.
file.newlines	Una tupla con todos los valores de fin de línea encontrados en el fichero, None si todavía no ha encontrado ninguno.
file.softspace	Valor booleano que indica si hay que añadir un espacio en blanco delante de ciertos caracteres cuando se escribe usando *print*.

Tabla 17 – Atributos del objeto file

Los siguientes métodos están definidos para los objetos de tipo file.

Métodos	Resultado
file.close()	Cierra el fichero. Un fichero cerrado no puede ser modificado. Se puede cerrar un fichero varias veces sin que se produzca una excepción.
file.flush()	Vacía el buffer del fichero escribiendo los datos en el fichero.
file.fileno()	Devuelve el número entero que representa el descriptor del fichero.
file.isatty()	Devuelve un booleano indicando si el fichero es un terminal TTY (o se comporta como uno).

file.next()	Un objeto fichero es un iterador y como tal puede usarse. Ver el capítulo 0 para más información sobre iteradores.
file.read([size])	Lee del fichero hasta que llega al final del mismo. Si se especifica el parámetro *size* y no es negativo lee esa cantidad de bytes
file.readline([size])	Devuelve una línea del fichero. Si se especifica el parámetro *size* se lee hasta el final de línea o hasta que lee una cantidad la de bytes indicada en *size*.
file.readlines([sizehint])	Devuelve una lista de líneas del fichero utilizando la función file.readline(). Si se especifica el parámetro *sizehint* se lee esa cantidad de bytes, de forma aproximada, o hasta que se alcanza el final de fichero.
file.xreadlines()	Deprecado. Esta función funciona igual que iter(file).
file.seek(offset[, whence])	Sirve para posicionarse en el fichero, el parámetro *offset* indica el desplazamiento que se hace desde el principio del fichero. El parámetro whence indica desde donde se lee, puede adoptar los siguientes valores: os.SEEK_SET para indicar el incio del fichero, os.SEEK_CUR para indicar la posición actual u os.SEEK_END para indicar desde el final del fichero.
file.tell()	Devuelve la posición actual en el fichero.
file.truncate([size])	Trunca el tamaño del fichero. Por defecto usa la posición actual. Si se añade el parámetro *size* se trunca el fichero a ese tamaño (aproximado).
file.write(str)	Escribe la cadena *str* en el fichero. Puesto que hay un buffer el fichero no reflejara los cambios hasta que se use flush() o close().
file.writelines(sequence)	Escribe la secuencia de cadenas de caracteres *sequence* en el fichero, pero aunque el nombre indique que escribe líneas no añade caracteres de fin de línea.

Tabla 18 – Métodos del objeto file

Ejemplo de uso de ficheros:

```
archivo = open('file.dat')
linea = archivo.readline()
print linea
archivo.close()
```

Código 15 – Uso de ficheros

4. Herramientas de control de flujo[10]

Los elementos que vamos a ver a continuación son los que se encargan de controlar el flujo del programa. Como en otros lenguajes de programación tenemos sentencias condicionales, que nos permiten ejecutar ciertas partes del código dependiendo de una condición. Sentencias iterativas que nos permiten ejecutar de forma repetida algunas líneas de código. Además de las sentencias de control de flujo, también hablaremos de las sentencias de control de excepciones que nos permiten manejar las excepciones que se generen en el código. Por último, en este capítulo trataremos la definición de funciones en Python.

4.1. Sentencias condicionales

La sentencia condicional por antonomasia es el *if – else*. Esta sentencia está formada por tres bloques distintos.

El primer bloque es la parte del *if*, que está formada por la palabra reservada *if* seguida de una condición y por el carácter `':'`. Si la condición es evaluada a `True` nos indica que las sentencias dentro del bloque *if* se ejecutaran.

```
if x > 0:
    print 'positivo'
```

Código 16 – Sentencia if

[10] Fuente: https://docs.python.org/2/tutorial/controlflow.html

El segundo bloque es el bloque *else*. Este bloque es opcional y siempre que esté presente debe estar a continuación de un bloque *if*. Está formado por la palabra reservada *else* y por el carácter `':'`. Es el bloque que se ejecutará si no se cumple la condición que aparece en el bloque del *if* (es decir se evalúa a `False`)

```
if x > 0:
    print 'mayor que cero'
else:
    print 'otro'
```

Código 17 – Sentencia if - else

El último bloque es el bloque *elif*. Este bloque es opcional y es una combinación de un bloque *else* con un bloque *if*. Está formado por la palabra reservada *elif*, una condición y por el carácter `':'`. Su funcionamiento es equivalente a la combinación que sustituye y su principal utilidad es acortar la notación. En una misma sentencia *if* pueden existir múltiples bloques *elif*. Esto nos servirá, por tanto, para escribir las sentencias *switch* de otros lenguajes.

```
if x > 0:
    print 'positivo'
elif x == 0:
    print 'cero'
else:
    print 'negativo'
```

Código 18 – Sentencia elif

Además de la forma normal existe una forma más compacta de escribir la sentencia *if* que es la siguiente:

```
print 'sentencia si' if cond else 'sentencia si no'
```

Código 19 – Sentencia if compacta

En este caso la sentencia a ejecutar, si se cumple la condición *cond,* se escribe delante de la palabra reservada *if*. A continuación, se escribe la condición y por último se escribe la parte del *else*. Esa notación abreviada suele usarse en los generadores y está pensada para parecerse a la notación matemática.

4.2. Sentencias iterativas

Las sentencias iterativas nos permiten escribir código que realiza múltiples iteraciones sin tener que escribir cada una de las iteraciones. Además de las sentencias habituales para crear bucles, *while* y *for*, tenemos un tipo especial de estructura que nos permite generar listas de una forma más compacta, las listas de comprensión. Y para completar las sentencias iterativas tenemos los generadores que nos permiten ir generando los valores de una estructura de uno en uno sin tener que almacenar toda la estructura en memoria.

4.2.1. Bucles: while

La sentencia de bucle *while* se utiliza para repetir un bloque de código mientras una condición se cumpla.

```
while True:
     page = exec()
     if not page:
        break
```

Código 20 – Bucle while

La sentencia está compuesta por la palabra reservada *while* una condición, el carácter `':'` y un bloque de código que será el que se repita. Además del bloque *while* se puede añadir un bloque *else* (de forma opcional) como en la sentencia condicional *if*. Este bloque se ejecutará en el momento que se evalúa la condición del bucle *while* y esta es `False`.

```
i = 0
while i < -10:
    i += 1
    print i
else:
    print 'out while'
```

Código 21 – Bucle while con else

Además de estos dos bloques existen dos sentencias especiales que solo pueden aparecer dentro del bloque que se repite. La sentencia *break* que detendrá la ejecución de la sentencia *while* (incluido el bloque *else*, si existe) y continuará ejecutando el resto del código. Y la sentencia *continue* que detiene la ejecución de la iteración actual del bucle y continua con la siguiente iteración en caso de que la condición siga cumpliéndose.

4.2.2. Bucles: for

El bucle *for* es algo distinto al de otros lenguajes de programación, en Python está diseñado para recorrer una secuencia de forma iterativa. La sentencia *for* está compuesta por la palabra reservada *for* seguido de una instrucción *in* que itera sobre una secuencia, por el carácter `':'` y por último por un bloque de código que será el que se repita.

```
for x in ['cero', 'uno', 'dos', 'tres']:
    print x
```

Código 22 – Bucle for

Además de este bloque básico se puede añadir un bloque *else*, este bloque se ejecutará cuando la lista se haya completado.

```
for x in ['cero', 'uno', 'dos', 'tres']:
    print x
else:
    print 'out for'
```

Código 23 – Bucle for con else

Al igual que en el bucle *while*, las palabras reservadas *break* y *continue* funcionan de la misma forma que en la sentencia *while*.

Hay que tener en cuenta que iterar sobre una secuencia con un bucle for no hace una copia de la secuencia y por lo tanto cualquier modificación que se haga en la secuencia durante la iteración puede producir sucesos inesperados. Una forma de evitar esto es realizar una copia de la secuencia (usando por ejemplo `list[:]`) para iterar sobre ella e ir modificando la lista original.

```
#forma erronea
sec = ['a', 'b', 'c']
for x in sec:
    sec.remove(x)
print sec

#forma correcta
sec = ['a', 'b', 'c']
for x in sec[:]:
    sec.remove(x)
print sec
```

Código 24 – Modificar una secuencia con un bucle

4.2.2.1. Función range()

La función predefinida *range* está creada teniendo en mente el funcionamiento del bucle for. Puesto que este bucle solo nos sirve para recorrer una secuencia y es bastante habitual necesitar recorrer una lista de enteros la función *range* se creó para generar una secuencia de números enteros. Existen dos formatos para la función *range*, son los siguientes:

Método	Resultado
range(stop)	Devuelve una lista de enteros comenzando en cero y terminando en el parámetro *stop*.
range(start, stop[, step])	Devuelve una lista de enteros comenzando en el parámetro *start* y terminando en *stop*. El incremento entre elementos de la lista se define en el parámetro *step*. Si no se define el parámetro *step* se utiliza el valor por defecto que es 1.

Tabla 19 – Formato de la función range

Un ejemplo de la función *range* en conjunción de un bucle *for*:

```
# números del 0 al 9
for i in range(10):
    print i
```

Código 25 – Función range

4.2.3. List comprehensions[11]

Las *list comprehensions* son una forma rápida de generar listas. En las aplicaciones típicas, las listas se crean aplicando una operación a todos los elementos de una lista o filtrando los elementos de una lista que cumplen una determinada condición.

Están formadas por una expresión seguida de una sentencia *for*, esta a su vez está seguida por cero o más sentencias *for* y/o sentencias *if* y todo ello rodeado por corchetes. Por ejemplo, para generar una lista con las potencias de dos hasta el 16 podemos hacer lo siguiente:

```
potencias = [x**2 for x in range(16)]
```

Código 26 – List comprehension

Un ejemplo un poco más complejo podría ser calcular las tablas de multiplicar:

```
tabla = ["%dx%d=%d" % (x, y, x*y) for x in range(10) for y in range(10)
\
if x != 0 and y != 0]
```

Código 27 – List comprehension 2

Como podemos ver en este ejemplo en caso de que la expresión sea un tupla deberá estar obligatoriamente rodeada por paréntesis o se lanzará una excepción.

[11] Para una información más detallada de las listas de compresión consultar https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions

4.2.4. Iteradores

El tipo de datos iterador está definido para dar soporte al patrón de diseño Iterador. Todos los tipos de secuencias, los diccionarios y los elementos del paquete *Collections* implementan esta funcionalidad. A todos los objetos que implementan este método se le denomina iterables.

Los métodos que hacen falta para implementar este patrón son los siguientes:

Método	Resultado
container.__iter__()	Este método se aplica en el contenedor que queremos iterar y nos devuelve el iterador con el que poder recorrer los elementos del contenedor.
iterator.__iter__()	Este método es igual al que se aplica al contenedor y sirve para que se pueda utilizar tanto el contenedor como el propio iterador en un bucle.
iterator.next()	Devuelve el siguiente elemento del iterador. En caso de no quedar elementos por recorrer en el contenedor, lanzará una excepción.

Tabla 20 – Métodos de un iterador

Un ejemplo donde podemos ver cómo funciona internamente un iterador:

```
iterator = range(10).__iter__()
while True:
    try:
        x = iterator.next()
        print x
    except StopIteration as e:
        break
```

Código 28 - Iterador

En el ejemplo anterior se utiliza la sentencia de control de excepciones que veremos en este mismo capítulo más adelante.

4.2.5. Generadores

Los generadores son una forma de iterar sobre elementos, pero sin tener que ocupar todo el espacio del contenedor en memoria. Un generador nos permite recorrer el contenedor de una forma más eficiente, pero tiene una contrapartida, una vez que hemos utilizado un elemento este se consume y no podremos volver atrás. Una forma de definirlos es utilizar las expresiones de generación [12] que es una generalización de las *list comprehensions* más eficiente en términos de memoria. Estas expresiones se definen igual que las *list comprehensions* pero utilizando paréntesis en vez de corchetes.

```
#creamos un tupla en vez de una lista
generator = (x**2 for x in range(16))
t = tuple(generator)
print t
```

Código 29 - Generadores

Como los generadores son un tipo de iterador se pueden utilizar tanto en bucles como utilizando la función `next()`.

```
# imprimimos la lista de potencias de 2
generator = (x**2 for x in range(16))
for i in generator:
    print i
```

Código 30 - Generador en un bucle

Además de utilizando las expresiones de generación si necesitamos crear un generador más complejo, podemos definir una función y utilizar la palabra reservada *yield*. Más adelante veremos cómo se define una función y como se utiliza la sentencia *yield*.

[12] Las expresiones de generación están definidas en el documento PEP-289, para más información consultar https://www.python.org/dev/peps/pep-0289/

4.2.5.1. Función xrange()

La función predefinida *xrange* tiene la misma forma que la función *range* pero en vez de devolver una lista, devuelve un generador que nos permite generar la lista. En términos de rendimiento no notaremos una gran mejoría con respecto a la función range, puesto que ambas funciones tienen que crear los valores. Sin embargo, si usamos la función *xrange* sí se notará una mejora de rendimiento en listas muy grandes puesto que estas no ocuparan mucho espacio en memoria.

Método	Resultado
xrange(stop)	Devuelve el generador de una lista de enteros comenzando en cero y terminando en el parámetro *stop*.
xrange(start, stop[, step])	Devuelve el generador de una lista de enteros comenzando en el parámetro *start* y terminando en *stop*. El incremento entre elementos de la lista se define en el parámetro *step*. Si no se define el parámetro *step* se utiliza el valor por defecto que es 1.

Tabla 21 – Formato de la función xrange

4.3. Sentencias de control de excepciones

Como ya hemos comentado anteriormente en algunas ocasiones se producirá un error al tratar de ejecutar una instrucción, a estos errores en tiempo de ejecución se les llama excepción. Cuando ocurre una excepción podemos tratarla o bien el programa terminará con un error.

4.3.1. Sentencia try

Para tratar estas excepciones debemos usar la sentencia *try*. La forma básica de la sentencia *try* está formada por dos bloques, el primero formado por la palabra reservada *try*, por el carácter `':'` y por un bloque de código. El segundo bloque es el bloque *except* que puede formularse de varias formas, pero en todas ellas se incluye la palabra clave *except* seguida por el carácter `':'`. Aquí tenemos un ejemplo:

```
try:
    print int(i)
except Exception:
    print "exception"
```

Código 31 - Sentencia try - except

En este ejemplo podemos ver como al intentar ejecutar la sentencia `print` esta genera una excepción que es capturada y tratada en el bloque del *except*.

Para poder tratar de forma más precisa las excepciones es posible incluir varios bloques *except* distintos y además existen otras formas de definir el bloque *except* lo que nos permite definir de muchas formas distintas el control de excepciones. Todo ello lo podemos ver en el siguiente ejemplo.

```
try:
    print int(i)
except (ZeroDivisionError, FloatingPointError), e:
    print "excepcion 1"
except IndexError, e:
    print "excepcion 2"
except NameError as e:
    print "excepcion 3 " + str(e)
except:
    print "excepcion 4"
```

Código 32 – Diferentes formatos de except

En el caso de definirse varios bloques *except* se irá comprobado la excepción que se captura y solo se ejecutará el primer bloque que coincida. Es posible definir una clausula *except* que no incluya ninguna excepción y que sirva para tratar cualquier excepción que ocurra. Este último caso conviene utilizarlo con mucho cuidado puesto que puede ocultar errores que deberían ser tratados de otra forma.

La sentencia try tiene dos bloques opcionales que nos permiten aún más flexibilidad en el control de excepciones. El primero es el bloque del *else*, este bloque como en otros casos se define por la palabra *else* seguido por el carácter ':' y debe ir situado después de todos los bloques *except* que estén definidos.

Este bloque se ejecutará cuando se termine la ejecución del bloque *try* y siempre que no se produzca una excepción. En el caso de producirse una excepción en el bloque *else* esta no será tratada por las sentencias *except* que la preceden.

```
try:
    print int(i)
except:
    print "excepcion"
else:
    print "else"
```

Código 33 – Sentencia try con else

El último bloque que puede añadirse es el bloque *finally*, este bloque opcional debe situarse detrás de todos los bloques *except* y *else* que puedan existir y puede ser el único bloque definido aparte del bloque *try*. La función de este bloque es ejecutarse siempre, independientemente de que se produzca una excepción en alguno de los bloques anteriores. Cuando se ha producido se almacena, se ejecuta el bloque *finally* y después se lanza la excepción. En el caso de producirse una excepción en el bloque *finally* cualquier excepción almacenada se descarta y se lanza la excepción producida en el bloque *finally*.

```
try:
    print int(i)
finally:
    print "final"
```

Código 34 – Sentencia try con finally

4.3.2. Sentencia raise

La sentencia *raise* permite al usuario lanzar excepciones concretas en cualquier punto del código. El único argumento de la función raise indica que excepción hay que lanzar que podría ser una excepción predefinida o bien una clase de excepción creada por el usuario.

```
raise NameError('Ejemplo')
```

Código 35 – Raise exception

Un caso típico en el que se utiliza esta sentencia es cuando estamos tratando una excepción y solo queremos saber que se ha producido, pero no queremos tratarla. En ese caso lo que se hace es capturar la excepción con un *except* y después volver a lanzarla con *raise*.

```
try:
    raise NameError('Ejemplo')
except NameError:
    print 'Una excepción de ejemplo'
    raise
```

Código 36 – Raise exception

4.3. Sentencia with

Existe otra forma más de abrir un fichero, este método nos permite abrir el fichero de tal forma que no tengamos que preocuparnos en cerrarlo. Para ello se utiliza la palabra reservada `with` seguida de una función predefinida `open()`, de la palabra reservada `as`, el carácter `':'` y por ultimo de un bloque de código. Dentro de este bloque de código el fichero estará abierto y cuando terminemos se cerrará automáticamente.

```
with open('workfile', 'r') as f:
    read_data = f.read()
```

Código 37 – Sentencia with

La sentencia `with` puede utilizarse con otras funciones aparte de `open()` para crear un contexto en el que esos métodos esta definidos.

4.4. Definición de funciones[13]

Una función es un bloque de código aislado que puede ser llamado en cualquier otra parte del código del programa para realizar una determinada tarea.

Para definir una función utilizamos la palabra reservada *def* seguida por el identificador de la función, por una lista de parámetros separadas por comas y rodeada de paréntesis, por el carácter ':' y por un bloque de código.

En este ejemplo podemos ver como se calcula la sucesión de Fibonacci[14] utilizando una función:

```
def fib(n):
    a, b = 0, 1
    while a < n:
        print a,
        a, b = b, a + b
```

Código 38 – Definición de una función, sucesión de Fibonacci

La ejecución de esta nueva función es igual al resto de funciones predefinidas en el sistema. Solo hay que poner el nombre de la función seguido por la lista de parámetros rodeados por paréntesis.

```
fib(300)
```

Código 39 – Ejecución de una función

Cuando ejecutamos una función se crea una nueva tabla de símbolos para almacenar todas las variables locales de la función. Cuando se declara una variable dentro de una función se almacena su identificador en la tabla de símbolos local. Cuando de hace referencia a una variable primero se mira en la tabla de símbolos local, después en la tabla de símbolos global y por último en la tabla de símbolos predefinidos. Estas tablas de símbolos son las que van

[13] Para más información de cómo se define una función consultar https://docs.python.org/2/tutorial/controlflow.html#defining-functions
[14] https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci

a gestionar nuestro alcance y que variable estamos utilizando en cada momento.

Todos los argumentos de una función se almacenan en la tabla de símbolos local y cuando se llama a una función se pasan por referencia lo que permite modificar los valores del parámetro. Un ejemplo de cómo funciona todo esto es el siguiente:

```
t = [1, 2, 3]
def change(tx):
    print t
    tx += ["4"]
    print tx

change(t)
print str(t)
```

Código 40 – Ejemplo con el alcance de una función

Cuando se define una función el identificador de la misma también se introduce en la tabla de símbolos, esto permite que ese identificador sea asignado a otro identificador o que se pase como parámetro de otra función.

Otra sentencia relacionada con las funciones es la sentencia *return*, esta sentencia permite devolver un valor desde la función al punto en el que se llamó.

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

Código 41 – Sentencia return

Hay que tener en cuenta que si la sentencia *return* no va acompañada de ninguna expresión se devolverá el valor `None`. También puede darse el caso de que no se añade una sentencia *return* y la función termine su ejecución, en ese caso también se devolverá el valor `None`.

4.5.1. Argumentos de una función

Además de la forma habitual de definir los parámetros como una lista, pueden definirse parámetros de varias formas interesantes.

La primera forma es definir valores por defecto en algún parámetro esto nos permite llamar a la función con menos parámetros, ya que si no se utiliza algún parámetro este tomará su valor por defecto.

```
def f(uno, dos=2, tres=3):
    return uno + dos + tres

#devuelve 6
print f(1)

#devuelve 8
print f(2,1,5)
```

Código 42 – Argumentos de una función, valores por defecto

Hay que tener en cuenta que la asignación de un valor por defecto solo se realiza una única vez en el momento de la asignación y después se almacena en la tabla de signos. Esto puede producir un efecto extraño cuando se asigna a un parámetro un objeto mutable (por ejemplo, una lista) ya que si dentro de la función modificamos ese parámetro en ejecuciones posteriores nos encontraremos con que ese parámetro se ha modificado. Podemos verlo en este ejemplo en el que vamos añadiendo valores a una lista.

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Código 43 – Argumentos de una función, valor por defecto mutable

Otra forma de definir los argumentos es utilizar como palabra clave el identificador del parámetro. Esto nos permite pasar los parámetros sin tener en cuenta el orden en que están definidos, la única restricción a esto es que los parámetros que se pasan por posición siempre deben ser los primeros. Como es lógico en los parámetros que se pasan por palabra clave, esta debe

coincidir con alguno de los parámetros de la función y tampoco puede pasarse un parámetro dos o más veces. Un ejemplo de cómo pueden utilizarse estos parámetros:

```
def f(uno, dos=2, tres=3):
    return uno + dos + tres

print f(1, tres=5)
print f(2, tres=1, dos=5)
```

Código 44 – Argumentos de una función, palabra clave

La tercera forma de definir parámetros es las listas arbitrarias de parámetros, estas listas permiten definir funciones que reciben un número arbitrario de parámetros. Para definir una lista arbitraria solo es necesario añadir un asterisco `(*)` delante del identificador del parámetro de la función. Es habitual utilizar `*args` como identificador, aunque no es obligatorio. Las listas arbitrarias de parámetros también pueden utilizarse para los parámetros por palabra clave, en este caso hay que añadir `(**)` delante del identificador. En este caso suele utilizarse como identificador `**kwargs`.

```
def funcion(l, *args):
    l.extend(args)
    return l

lista = []
print funcion(lista, "1", "2", "3")
```

Código 45 – Argumentos de una función, número arbitrario de argumentos

Relacionado con esta última forma está la posibilidad de desempaquetar listas de parámetros para pasarlas a las funciones de la forma apropiada. El asterisco sirve para convertir listas o tuplas en el formato de parámetros que acepta una función. El doble asterisco se puede utilizar para convertir un diccionario en el formato de palabra clave de los parámetros.

```
def f(uno, dos=2, tres=3):
    return uno + dos + tres

dic = {'uno': 1, 'dos': 6, 'tres': 3}
print f(**dic)
```

Código 46 – Argumentos de una función, desempaquetado

4.5.2. Funciones lambda

A veces cuando estamos escribiendo el código necesitamos definir una función, pero esta función es muy sencilla y no vamos a volver a utilizarla. Por lo tanto, utilizar la forma habitual de definir funciones no es lo más óptimo. Para definir las funciones estas funciones utilizaremos la palabra reservada lamba, seguida por una lista de parámetros separados por comas, por el carácter `':'` y por una única expresión. Esta expresión se corresponde con el cuerpo de la función. Estas funciones pueden utilizarse en cualquier caso en el que se requiera un objeto de tipo función. Podemos ver el siguiente ejemplo donde se define una función lambda que calcula el doble de un número.

```
def aplica_funcion(func, x):
    return func(x)

print aplica_funcion(lambda x: x * 2, 7)
```

Código 47 - Función lambda

4.5.3. Sentencia yield

En el apartado de los generadores ya veíamos que la sentencia *yield* se podía utilizar para crear una función de generación que fuese más compleja. La palabra reservada *yield* solo puede utilizarse en el cuerpo de una función y al utilizarla hace que la función se transforme en una función de generación. Su uso es similar al de la palabra reservada *return*.

```
def potencias_de_dos(x):
    for i in range(x):
        yield 2**i

potencias = potencias_de_dos(16)

print potencias.next()
print potencias.next()
print potencias.next()
```

Código 48 - Función yield

Lo más interesante de estas funciones de generación es que al utilizar la sentencia *yield* después de devolver el valor de la expresión el estado de ejecución de la función se *'congela'*. Esto significa que el valor de todas las variables se almacena y se mantiene hasta que el generador vuelva a

llamarse. En el momento en que esto suceda (en un bucle o con el método `next()`) la ejecución de la función continua desde el punto en que se congelo. Al ejecutar este ejemplo podemos ver cuál es el flujo de la ejecución.

```
def foo():
     print "begin"
     for i in range(3):
         print "before yield", i
         yield i
         print "after yield", i
     print "end"

f = foo()
f.next()
f.next()
f.next()
```

Código 49 – Función yield, funcionamiento interno

4.5.4. Funciones predefinidas15 por el interprete

El intérprete de Python tiene predefinidas multitud de funciones interesantes que pueden utilizarse en cualquier momento, a lo largo de este manual se han explicado unas cuantas de estas funciones relacionadas con alguno otro concepto. En este apartado se van a explicar algunas de las más útiles que no están relacionadas con otros conceptos.

4.5.4.1. Función predefinida filter

La función `filter` acepta dos parámetros aquí tenemos su signatura `filter(function, iterable)`, el primer parámetro es una función que devuelve un valor booleano. El segundo parámetro es un tipo de dato o clase que es iterable. La función devuelve una lista con el resultado de recorrer el iterable y aplicar la función a cada elemento y si el resultado es `True` lo devolverá. Para la función que hay que pasar de argumento es habitual usar una función lambda.

[15] Para ver todas las funciones predefinidas del intérprete de Python consultar https://docs.python.org/2/library/functions.html

```
lista = xrange(100)
# devuelve los numeros pares
print filter(lambda x: True if x % 2 == 0 else False, lista)
```

Código 50 – Función predefinida filter

4.5.4.2. Función predefinida map

La función `map` acepta dos o más parámetros y su signatura es la siguiente `map(function, iterable, ...)`. El primer parámetro es una función si la función es `None` se asume que es la función identidad. El segundo parámetro y siguientes son iterables. La función devuelve una lista con el tamaño del iterable más grande (los más pequeños se asume que se completan con `None`) después de aplicar la función pasándole como argumento elemento a elemento de cada iterable.

```
lista_in = xrange(10)
lista_in_2 = xrange(40, 50)

salida = map(lambda x, y: x * y, lista_in, lista_in_2)
print salida
```

Código 51 – Función predefinida map

4.5.4.3. Función predefinida reduce

La función `reduce` acepta dos o más parámetros su signatura es la siguiente `reduce(function, iterable [, initializer])`. El primer parámetro es la función que se aplicara a los elementos, el segundo parámetro es un iterable y el tercer parámetro, opcional, es un elemento que se utilizara para inicializar el proceso. La función reduce es algo más compleja puesto que en vez de devolver una lista de elementos devolverá un único valor este valor se obtiene aplicando la función de forma recursiva a los elementos del iterable. La función tiene que aceptar dos parámetros. El primero será el valor acumulado de las operaciones anteriores, empezando por el primer elemento del iterable, si se ha definido el parámetro *initializer* se utilizará este elemento. El segundo parámetro de la función será el elemento correspondiente del iterable. Aquí tenemos un ejemplo que calcula la suma y el máximo.

```
suma = str(reduce(lambda a, b: a + b, xrange(10)))
maximo = str(reduce(lambda a, b: max(a, b), xrange(10)))

print suma
print maximo
```

Código 52 – Función predefinida reduce

4.5.4.4. Función predefinida zip

Esta función acepta un numero variable de iterables, aquí está su signatura `zip([iterable, ...])`. Esta función devuelve una lista de tuplas del tamaño del iterable más pequeño y como su nombre indica, funciona como una cremallera, creando una tupla por cada elemento en la misma posición de cada iterable. Si el argumento es una lista de tuplas podemos utilizar el carácter `'*'` para realizar la función inversa.

```
x = (1, 2, 3)
y = (4, 5, 6)
zipped = zip(x, y)

x2, y2 = zip(*zipped)
```

Código 53 – Función predefinida zip

5. Clases y módulos

En este capítulo vamos a ver las dos formas más comunes de agrupar el código que son clases y paquetes.

5.1. Clases[16]

Las clases en Python proporcionan todos los mecanismos necesarios para la programación orientada a objetos, un mecanismo de herencia que permite heredar de múltiples clases la clase derivada puede reescribir cualquier método de la clase base, un método de una clase derivada puede utilizar el método de la clase base con el mismo nombre. Siguiendo la filosofía de Python, las clases pueden definirse en tiempo de ejecución y pueden modificarse durante la misma.

5.1.1. Definición de clases

La definición más sencilla de una clase utiliza la palabra reservada class un identificador seguido por el carácter `':'` y después un bloque de sentencias. Como en el resto de casos el sangrado de la línea nos indica hasta donde llega este bloque de código.

```
class Persona:
    nombre = ''

    def lee(self, libro):
        print self.nombre + " lee " + libro
```

Código 54 – Definición de una clase

[16] Fuente: https://docs.python.org/2/tutorial/classes.html

Una vez definida una clase para utilizarla tendríamos que instanciarla para crear un objeto de esa clase. Para hacer esto utilizaremos la misma notación que para llamar una función.

```
class Persona:
    nombre = ""

p1 = Persona()
```

Código 55 – Instanciar una clase

Las sentencias que suelen definirse dentro de una clase son definiciones de funciones y asignaciones a variables. Aunque siguiendo la nomenclatura de la programación orientada a objetos a las funciones se les denominan métodos y a las variables, atributos.

Los atributos se definen de la misma forma que las variables y como estas no necesitan ser declaradas, sino que existen a partir del momento en que ejecutan. Es por tanto posible declarar una clase y después realizar una asignación sobre un parámetro que no exista para añadirlo a la instancia de la clase.

```
class Persona:
    nombre = ""

p1 = Persona()
p1.nombre = "Maria"
p1.apellido = "Garcia"

print p1.nombre
print p1.apellido
```

Código 56 – Definición de atributos

Los métodos funcionan igual que las funciones, excepto por un detalle. Todos los métodos tienen reservado el primer parámetro que se define, para pasar una referencia al objeto desde el que se invoca el método. Esta referencia sirve para poder utilizar atributos y métodos desde el propio método. Una convención recomendable es utilizar el identificador `self` para indicar esta referencia, aunque podría utilizarse cualquier identificador. Aunque se añade en la declaración del método no hay que añadir este parámetro cuando se invoca el método.

```
class Persona:
    nombre = ""

    def __init__(self, name, age=0):
        self.nombre = name
        self.edad = age

    def lee(self, libro):
        print self.nombre + " lee " + libro
```

Código 57 – Definición de métodos

Además de los métodos que define el usuario existen algunos métodos predefinidos que podemos utilizar en nuestras clases, uno de los más habituales es el método __init__() que será llamado automáticamente cuando se crea una instancia de la clase. Como cualquier otro método podemos definir cualquier cantidad de parámetros, estos parámetros tendrán que usarse en la instanciación de cada objeto.

Al igual que las funciones los métodos se pueden asignar a una variable, hacer esto dentro de una clase creará dos métodos, con el mismo código, que podrían ser invocados desde una instancia del objeto.

En una clase podemos definir dos tipos de atributos. Los atributos que se definen dentro de un método de clase utilizando la referencia al objeto de la clase son los atributos de instancia y los atributos que se definen fuera de un método de la clase son los atributos de clase. La diferencia entre uno y otro es que los atributos de clase se comparten para todas las instancias y los atributos de instancia pertenecen a cada instancia. Esto puede producir efectos no esperados si definimos atributos de clase mutables ya que un cambio en una instancia afectaría a todas las instancias de ese objeto.

```
class Persona:
    nombre = ''
    estudios = []

    def __init__(self, nombre):
        self.nombre = nombre

a = Persona('Juan')
b = Persona('Pepe')
a.estudios.append('colegio')
b.estudios.append('universidad')

print b.estudios
print a.estudios
```

Código 58 – Atributos de clase mutables

En este ejemplo ambas instancias comparten la variable *estudios* y añadir un elemento en esa variable, en cualquier instancia, lo añadirá en todas.

Hay que tener cuidado al definir los métodos y atributos de la clase ya que definir ambos con el mismo identificador hará que el método sea sobrescrito y deje de poder utilizarse. Para evitar esto suelen utilizarse diversas convecciones como definir los atributos con un sustantivo y los métodos con un verbo.

El concepto de atributo o método 'privado', aquel que no puede invocarse desde fuera de la clase, no existe en Python. Aun así, existe un método para ocultar los miembros de una clase del exterior, cualquier atributo o método que comience por dos caracteres de subrayado '__' y no termine por más de un carácter de subrayado no podrá invocarse de forma normal. Esto se debe a que ese miembro será renombrado y se le concatenará el nombre de la clase precedido por un subrayado '_'.

```
class Persona:
    __oculto = 'hide'


a = Persona()

print dir(a)
print a._Persona__oculto
```

Código 59 – Atributo 'privado'

En caso de que no sepamos qué miembros tiene una clase, podemos usar la función predefinida `dir()` para que nos muestre todos los miembros que pertenecen a esa clase.

5.1.1. Herencia

Como todo lenguaje orientado a objetos Python permite herencia entre clases. La sintaxis para definir la herencia solo implica añadir una lista de clases rodeada por paréntesis detrás del nombre de la clase.

```
class Vehiculo():
    def mover(self):
        print "Me muevo"

class Autobus(Vehiculo):
    pass


moto = Vehiculo()
moto.mover()

bus = Autobus()
bus.mover()
```

Código 60 – Herencia entre clases

Cuando se intenta resolver un método de una clase, primero se busca en la clase derivada, si no se encuentra en esta se busca en la clase base y se sigue haciendo esto de forma recursiva hasta encontrar el método o llegar a la última clase. Es perfectamente posible que una clase derivada acabe sobrescribiendo un método de una clase base. Esto podría hacer que un método de la clase base acabe llamando a un método de una clase derivada cuando intentaba invocar otro método de la clase base. En el siguiente ejemplo redefinimos el método *update* lo que hace que el constructor intente llamar a un método *update* que ha sido redefinido y produce un error.

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

class MappingSubclass(Mapping):
    def update(self, keys, values):
        for item in zip(keys, values):
            self.items_list.append(item)

a = MappingSubclass('error')
```

Código 61 – Error al redefinir un método de una clase

5.2. Módulos[17]

Cuando escribimos un programa queremos poder guardar el código que tenemos hecho para reutilizarlo más adelante. Para hacer esto podemos guardar el código en un fichero con una extensión '.py'. Cuando hacemos esto lo habitual es acabar creando varios ficheros para facilitar su mantenimiento. A cada uno de esos ficheros se llama módulo y Python está preparado para poder utilizar el código de un fichero en otro.

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

def resta(a, b):
    return a - b
```

Código 62 – Definición del módulo funciones

[17] Fuente: https://docs.python.org/2/tutorial/modules.html

5.2.1. Definición y uso

Para definir un módulo solo tenemos que crear un fichero donde incluiremos nuestras definiciones y sentencias. El nombre del fichero, sin la extensión, será el nombre del módulo.

Existen dos formas de utilizar un módulo una es importar el módulo para utilizarlo dentro de otro módulo y la otra es utilizar el módulo como un script y ejecutarlo directamente. Para importar los datos de un módulo y después utilizar sus definiciones, utilizaremos el nombre del módulo.

```
import funciones

print funciones.suma(1,2)
```

Código 63 – Importar un módulo

Al importar un módulo todas las definiciones de funciones almacenadas en ese módulo se guardan en una tabla de símbolos distinta. Como vemos en el ejemplo anterior, para poder acceder a las definiciones del módulo tenemos que utilizar el nombre del módulo. En el caso de las sentencias que existan en el módulo estas se ejecutaran en el momento en que el nombre del módulo aparezca por primera vez en una sentencia de *import*, estas sentencias están para inicializar el módulo.

Existe otra forma de importar definiciones de funciones de un módulo de una forma más precisa. Utilizando la palabra reservada '*from*' podemos elegir que funciones utilizamos de un módulo.

```
from funciones import resta as r

print r(2,4)
```

Código 64 – Importar una función específica de un módulo

Además, se puede añadir un alias, usando la palabra reservada '*as*' a esta sentencia de importación que se añadirá en la tabla de símbolos local.

La segunda forma es ejecutar directamente el módulo llamando a Python desde la línea de comandos y pasándole como argumentos el nombre del fichero que contiene el módulo.

```
$> python funciones.py
```

Si queremos hacer que un módulo se pueda ejecutar tanto ejecutándolo desde la línea de comandos como importándolo desde otro módulo hay que tener en cuenta que las sentencias que tratan los argumentos de entrada solo se pueden utilizar desde el módulo principal. Para saber esto podemos añadir a nuestro módulo lo siguiente:

```
def main(argv):
    print 'main'

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

Código 65 – Añadir función main

Esta es la forma habitual de hacerlo, consta de dos partes, la primera es una función definida para ejecutar las instrucciones necesarias para ejecutar el script y la segunda es una sentencia condicional que nos permite leer los parámetros de entrada y llamar a la función que procesa la entrada.

5.2.1.1. Orden de importación

Cuando se importa un módulo existe una forma determinada en la que se va buscando en distintas rutas el fichero del módulo. Estas rutas están contenidas en la variable `sys.path` que se inicializa con los siguientes valores, el directorio donde está el script actual que se está ejecutando, las rutas almacenadas en la variable de entorno `PYTHON_PATH` y la ruta de instalación de Python. Una vez inicializada esta variable podría modificarse por un programa de Python. Al buscar en primer lugar en el directorio donde está el fichero en ejecución es posible que algún módulo del sistema no sea ejecutado a favor de otro fichero. Esto podría causar errores inesperados a menos que este reemplazo sea premeditado.

5.2.2. Módulos estándar

Cualquier instalación de Python incluye una librería estándar de módulos de Python, estos módulos están construidos dentro del intérprete, aunque no forman parte del núcleo del lenguaje se han construido por defecto multitud de módulos que pueden utilizarse para múltiples utilidades. Estos módulos dependen de la plataforma en la que se han instalado y alguno como por ejemplo el módulo `winreg` solo están definidos en las distribuciones para Windows. A continuación, vamos a ver unos cuantos ejemplos de módulos estándar.

5.2.2.1. Módulo pickle

El módulo *pickle* implementa un algoritmo para serializar y de-serializar la estructura de un objeto de Python. El formato de datos usado por *pickle* es específico para Python así que no tiene las restricciones que podría tener otro estándar. Hay tres protocolos distintos que pueden utilizarse con pickle

Protocolo	Descripción
0	Protocolo que escribe el objeto en ASCII es compatible con las versiones antiguas de Python pero ocupa más espacio que los protocolos binarios.
1	Protocolo binario clásico, es compatible con las versiones antiguas de Python.
2	Protocolo binario, fue introducido en Python 2.3 y es más eficiente para las clases del nuevo estilo.

Tabla 22 – Módulo pickle, protocolos

Atributos del módulo pickle:

Métodos	Resultado
pickle.HIGHEST_PROTOCOL	Versión más alta del protocolo disponible

Tabla 23 – Módulo pickle, atributos

El módulo pickle define los siguientes métodos:

Métodos	Resultado
pickle.dump(obj, file[, protocol])	Escribe una representación del objeto *obj* en el fichero *file*, que debe estar abierto. Si se omite el parámetro *protocol* se utiliza el 0 como valor por defecto. Si se indica un valor negativo se utiliza la versión más alta del protocolo.
pickle.load(file)	Lee una cadena del fichero *file*, que debe estar abierto, lo interpreta y lo convierte en un objeto.
pickle.dumps(obj[, protocol])	Devuelve la representación del objeto *obj* como una cadena de texto en vez de escribirla en un fichero. Si se omite el parámetro *protocol* se utiliza el 0 como valor por defecto. Si se indica un valor negativo se utiliza la versión más alta del protocolo.
pickle.loads(string)	Lee una cadena de texto, la interpreta y la convierte en un objeto.

Tabla 24 – Módulo pickle, métodos

Además de los atributos y los métodos el módulo pickle define las siguientes excepciones para controlar los errores que puedan producirse al utilizar este módulo:

Excepción	Descripción
pickle.PickleError	Excepción base desde la que heredan las otras dos.
pickle.PicklingError	Excepción que se lanza en el método dump cuando un objeto que no puede ser convertido se pasa como argumento.
pickle.UnpicklingError	Excepción que se lanza cuando se produce un error al intentar convertir en objeto un *dump*.

Tabla 25 – Módulo pickle, excepciones

Ejemplo de funcionamiento del módulo:

```
import pickle

mi_lista = range(1000)
pickle.dump(mi_lista, open('./mi_lista.dat', 'wb'))

mi_lista2 = pickle.load(open('./mi_lista.dat', 'rb'))
print mi_lista2
```

Código 66 – Módulo pickle

5.2.2.2. Módulo re

Este módulo añade el uso de expresiones regulares (RE) al lenguaje de Python. El carácter de escape utilizado dentro de las expresiones regulares es la barra invertida `'\'` lo que colisiona con el carácter de escape de las cadenas de caracteres en Python por lo tanto para poner una barra invertida en una cadena hay que poner `'\\\\'` cada pareja se transforma en una barra al convertirse en `str` y después hacen falta dos barras para poner la barra de la expresión regular.

Además de parear caracteres normales las expresiones regulares permiten parear una cantidad de caracteres especiales. A continuación, se muestra una lista.

Carácter especial	Descripción
'.'	Representa cualquier carácter menos el fin de línea.
'^'	Representa el inicio de la cadena.
'$'	Representa el fin de la cadena o justo antes del carácter de fin de línea.
'*'	Representa cero o más repeticiones de la RE anterior.
'+'	Representa una o más repeticiones de la RE anterior.
'?'	Representa cero o una repeticiones de la RE anterior.
*?, +?, ??	Sirve para evitar que las expresiones '*', '+' y '?' consuman el máximo número de caracteres posible y devuelvan el mínimo posible.
{m}	Representa exactamente m repeticiones de la RE anterior.
{m,n}	Representa entre m y n repeticiones de la RE anterior. Intentará consumir el mayor número posible de repeticiones.

{m,n}?	Representa entre m y n repeticiones de la RE anterior. Intentará consumir el mínimo número posible de repeticiones.
'\'	Carácter de escape.
[]	Se utiliza para indicar un conjunto de caracteres. Los caracteres especiales pierden su significado en el interior. Se pueden incluir rangos utilizando el carácter '-'. Se pueden incluir complementos usando el carácter '^' en la primera posición. Estos caracteres especiales se escapan usando '\'.
'|'	Si A y B son dos RE crea un patrón que casa con A o con B. Se pueden concatenar cualquier número de RE con este operador, pero una vez que parea con la más a la izquierda no sigue comprobando.
(...)	Comprueba la RE dentro de los paréntesis y además indica un grupo. El contenido de cada grupo puede obtenerse después de realizar la función match. Los grupos empiezan a contarse a partir del 1.
(?iLmsux)	Una o más letras del conjunto {'i', 'L', 'm', 's', 'u', 'x'} cada una se corresponde con un parámetro. Este grupo de parámetros se aplica a toda la expresión. Ver más adelante parámetros para el significado de cada bandera.
(?:...)	Comprueba la RE dentro de los paréntesis pero no guarda el contenido y este no podrá recuperarse.
(?P<name>...)	Al igual que los paréntesis comprueba la RE y guarda el grupo. Este grupo podrá ser recuperado utilizando como identificador el valor de *name*.
(?P=name)	Indica una referencia a un grupo definido anteriormente etiquetado con la etiqueta *name*.
(?#...)	Un comentario dentro de la RE.
(?=...)	RE *lookahead*, la expresión regular casa con el valor entre los paréntesis pero no consume ningún carácter de la cadena.
(?!...)	RE *lookahead negado*, la expresión regular casa si no coincide con el valor entre los paréntesis, pero no consume ningún carácter de la cadena.
(?<=...)	RE *lookbehind,* la expresión regular casa si está precedida por la RE contenida en los paréntesis. Como

	en los casos anteriores no consume la cadena contenida entre los paréntesis.
(?<!...)	RE *lookbehind negado,* la expresión regular casa si no está precedida por la RE contenida en los paréntesis. Como en los casos anteriores no consume la cadena contenida entre los paréntesis.
(?(id)yes-pattern | no-pattern)	ER similar a una condición. Si la expresión identificada con el identificador id está presente intenta casar con la ER contenida en yes-pattern sino lo intenta con la parte del no-pattern. La parte del no-pattern se puede omitir.

Tabla 26 – Módulo re, caracteres especiales

Al igual que los caracteres, también existen una cantidad de secuencias que tienen un significado especial, aquí las podemos ver.

Secuencia especial	Descripción
\number	Esta secuencia sirve para identificar un grupo definido anteriormente identificado por *number*. La RE casará si coincide con la RE definida en el grupo *number*.
\A	Casa solo con el inicio de la cadena.
\b	Casa con la cadena vacía, pero solo al comienzo o al final de una palabra. (Palabra está definido como una secuencia de caracteres alfanuméricos o subrayados).
\B	Casa con la cadena vacía pero solo cuando no está al comienzo o al final de una palabra.
\d	Casa con cualquier digito decimal. Es equivalente a [0-9].
\D	Casa con cualquier carácter que no sea un digito decimal. Es equivalente a [^0-9].
\s	Casa con cualquier carácter blanco. Es equivalente a [\t\n\r\f\v].
\S	Casa con cualquier carácter que no sea un blanco. Es equivalente a [^ \t\n\r\f\v].
\w	Casa con cualquier carácter alfanuméricos o subrayado. Es equivalente a [a-zA-Z0-9_].
\W	Casa con cualquier carácter que no sea alfanuméricos o un subrayado. Es equivalente a [^a-zA-Z0-9_].
\Z	Casa solo con el final de la cadena.

Tabla 27 – Módulo re, secuencias especiales

Este módulo define los siguientes atributos:

Constantes	Descripción
re.DEBUG	Muestra información de debug sobre la RE.
re.IGNORECASE re.I	Modifica el casado para que no tenga en cuenta las mayúsculas y las minúsculas.
re.LOCALE re.L	Modifica las secuencias especiales para que tengan en cuenta el LOCALE.
re.MULTILINE re.M	Modifica los caracteres especiales '^' y '$' para que detecten el inicio de línea y el final de línea respectivamente.
re.DOTALL re.S	Modifica el carácter especial '.' para que case con todos los caracteres incluido el final de línea.
re.UNICODE re.U	Modifica las secuencias especiales para que tengan en cuenta los caracteres especiales de UNICODE.
re.VERBOSE re.X	Modo verboso. Modifica la forma de escribir las RE para que puedan ser más legibles. Se ignoran los caracteres blancos al final de línea y se pueden añadir comentarios para clarificar las expresiones regulares.

Tabla 28 – Módulo re, constantes

Y las siguientes funciones:

Métodos	Descripción
re.compile(pattern, flags=0)	Compila el patrón *pattern* y devuelve un objeto de la clase re.RegexObject. Se puede modificar la forma de casar los patrones utilizando el parámetro *flags*. A este parámetro se pueden pasar cualquier cantidad de constantes usando el operador '|'.
re.search(pattern, string, flags=0)	Busca en la cadena *string* la primera ocurrencia del patrón *pattern* y devuelve un objeto del tipo re.MatchObject. En caso contrario, devuelve None. Se puede modificar la forma de casar los patrones utilizando el parámetro *flags*.
re.match(pattern, string, flags=0)	Busca en la cadena *string* si el patrón *pattern* corresponde con el incio de la cadena y devuelve un objeto del tipo re.MatchObject. En caso contrario devuelve None. Se puede modificar la forma de casar los patrones utilizando el parámetro *flags*.
re.split(pattern, string, maxsplit=0, flags=0)	Divide la cadena *string* utilizando el patrón *pattern*, devuelve una lista con los valores obtenidos. Si se

	incluyen grupos en el patrón también se incluirán en la lista que se devuelve. Si se incluye el parámetro *maxsplit* se limita el número máximo de divisiones por este número. Se puede modificar la forma de casar los patrones utilizando el parámetro *flags*.
re.findall(pattern, string, flags=0)	Busca en la cadena *string* todas las ocurrencias del patrón *pattern*. Devuelve una lista con estas ocurrencias. En caso de definirse grupos se devolverá una lista de tuplas con los elementos que casan de cada grupo. Se puede modificar la forma de casar los patrones utilizando *flags*.
re.finditer(pattern, string, flags=0)	Devuelve un iterador de objetos re.MatchObject por cada una de las coincidencias del patrón *pattern* en la cadena *string*. Se puede modificar la forma de casar los patrones utilizando *flags*.
re.sub(pattern, repl, string, count=0, flags=0)	Devuelve una cadena con el resultado de reemplazar cada uno de las coincidencias del patrón *pattern* en la cadena *string*. El reemplazo se hará con el parámetro *repl*, que puede ser una cadena o una función. En el caso de la cadena se sustituyen todos los caracteres de escape y se reemplaza por todas las ocurrencias. Si se define una función se llamará a la función una vez por cada vez que coincida. El parámetro *count* indica el número de reemplazos que pueden hacerse. Se puede modificar la forma de casar los patrones utilizando el parámetro *flags*.
re.subn(pattern, repl, string, count=0, flags=0)	Igual que re.sub() pero devuelve una tupla con dos valores, la cadena modificada y el número de reemplazos realizados.
re.escape(string)	Devuelve una cadena con todos los elementos de la cadena *string* precedidos por una barra invertida.
re.purge()	Borra la cache de expresiones[18].

Tabla 29 – Módulo re, métodos

[18] Esta cache guarda el objeto compilado `re.RegexObject` de la última expresión ejecutada usando `re.match()`, `re.search()` o `re.compile()` para mejorar el rendimiento de la siguiente ejecución.

En caso de producirse un error se lanzará la siguiente excepción:

Excepción	Descripción
`re.error`	Esta excepción se lanza cuando la expresión regular pasada no es válida, o cuando se produce algún error.

Tabla 30 – Módulo re, excepción

5.2.2.2.1. Clase re.RegexObject

Esta clase permite realizar las mismas funciones que la clase base, pero utilizando una expresión regular compilada anteriormente lo que mejora el rendimiento. Esta clase define los siguientes atributos.

Atributos	Descripción
Flags	Todas las banderas que se han pasado a la expresión regular tanto por parámetro como usando (?...).
Groups	El número de grupos que captura la expresión.
Groupindex	Un mapeo de todos los grupos que se han definido con un nombre con su identificador.
Pattern	La cadena desde la que se ha compilado la expresión regular.

Tabla 31 – Módulo re, Clase re.RegexObject atributos

Se definen los siguientes métodos:

Métodos	Descripción
search(string[, pos[, endpos]])	Funciona igual que el método re.search()pero añade dos parámetros adicionales, una posición inicial *pos* desde la que empezar el casado. Y una posición final *endpos* que es hasta donde llegará la búsqueda.
match(string[, pos[, endpos]])	Funciona igual que el método re.match()pero añade dos parámetros adicionales igual que search().
split(string, maxsplit=0)	Funciona igual que el método re.split().
findall(string[, pos[, endpos]])	Funciona igual que el método re.findall()pero añade dos parámetros adicionales igual que search().

finditer(string[, pos[, endpos]])	Funciona igual que el método re.finditer()pero añade dos parámetros adicionales igual que search().
sub(repl, string, count=0)	Funciona igual que el método re.sub().
subn(repl, string, count=0)	Funciona igual que el método re.subn().

Tabla 32 – Módulo re, Clase re.RegexObject métodos

5.2.2.2.2. Clase re.MatchObject

La clase re.MatchObject es la encargada de gestionar las coincidencias obtenidas a partir de un patrón. Estos son los atributos que define:

Atributos	Descripción
Pos	Contiene el valor de posición inicial *pos* pasado a la función search() o match() que ha generado el objeto.
Endpos	Contiene el valor de posición final *endpos* pasado a la función search() o match() que ha generado el objeto.
Lastindex	El entero que se corresponde con el último grupo que ha casado. Si no ha casado ningún grupo devolverá None.
Lastgroup	El nombre del último grupo que ha casado que tiene nombre. Si no ha casado ningún grupo o ninguno tiene nombre devolverá None.
Re	La expresión regular que se ha utilizado en la función search() o match() que ha generado el objeto.
String	La cadena que se ha pasado a la función search() o match() que ha generado el objeto.

Tabla 33 – Módulo re, Clase re.MatchObject atributos

Los siguientes métodos se definen en esta clase:

Métodos	Descripción
expand(template)	Devuelve una cadena utilizando la cadena *template* con todos los caracteres de escape reemplazados por su valor y todas las referencias, numéricas o con una etiqueta, a un grupo sustituidas por el valor al que hacen referencia.
group([group1, ...])	Devuelve un grupo o una tupla de grupos. Si algún índice es negativo o supera el número de grupos definidos lanzará una excepción. Si el grupo está definido en una parte de la expresión que no ha casado el valor devuelto será None. Se pueden usar etiquetas en los parámetros pero si no han sido definida se lanzará una excepción.
groups([default])	Devuelve los valores de todos los grupos definidos. El parámetros *default* se utiliza como valor para los grupos que no han participado en el pareado, por defecto es None.
groupdict([default])	Devuelve los valores de todos los grupos con etiqueta definidos. El parámetros *default* se utiliza como valor para los grupos que no han participado en el pareado, por defecto es None.
start([group])	Devuelve la posición inicial de la cadena donde el grupo *group* ha realizado el pareado. Si no se define el parámetro *group* por defecto será cero, que corresponde a la expresión al completo.
end([group])	Devuelve la posición final de la cadena donde el grupo *group* ha realizado el pareado. Si no se define el parámetro *group* por defecto será cero, que corresponde a la expresión al completo.
span([group])	Devuelve una tupla con los valores start() y end() obtenidos como en los dos métodos anteriores.

Tabla 34 – Módulo re, Clase re.MatchObject métodos

5.2.2.2.3. Ejemplo

Aquí podemos ver un ejemplo de cómo funciona las expresiones regulares en Python.

```
import re

m = re.search('Don (?=Quijote)', 'Don Quijote de la mancha')
if m:
    m.group(0)

pc = re.compile('Don (?=Quijote)')
m = pc.match('Don Quijote de la mancha')
if m:
    m.group(0)

# devuelve los valores separados
re.split('\W+', 'Uno, dos, tres.', maxsplit=1)
# devuelve los valores separados junto con los separadores
re.split('(\W+)', 'Uno, dos, tres.', maxsplit=2)
# encuentra todas las ocurrencias de la expresion y devuelve una tupla
re.findall('(\d)(\w)', '3s un e5cr1t0r')

s = '3s un e5cr1t0r.'
r = re.search('(\d\w).*?(?P<uno>\d\w).*(\d\w)|(?P<dos>asdf)', s)

# Devuelve los valores de los grupos
r.group(1, 3)
r.groups('')
r.groupdict()

# inicio de la coincidencia
r.start(0)
# fin de la coincidencia
r.end(0)
# (inicio, fin) de la coincidencia
r.span(0)
```

Código 67 – Expresiones regulares

6. Estilo de programación

En Python dispone de una lista de PEP (propuestas para mejorar Python, por sus siglas en ingles). Estas propuestas recogen diversas utilidades, entre ellas las relacionadas con el estilo de programación.

6.1. PEP – 8: Estilo de codificación

Python define una guía de estilo para escribir código, se denomina PEP-8 y nos permite

6.1.1. Instalación

Para instalar pep8 podemos utilizar el comando *pip* desde línea de comandos.

```
$> pip install pep8
```

O bien utilizar la herramienta que incorpora pycharm. Buscamos el intérprete de pycharm seleccionamos el intérprete de Python en el que queremos instalar el paquete y pulsamos instalar.

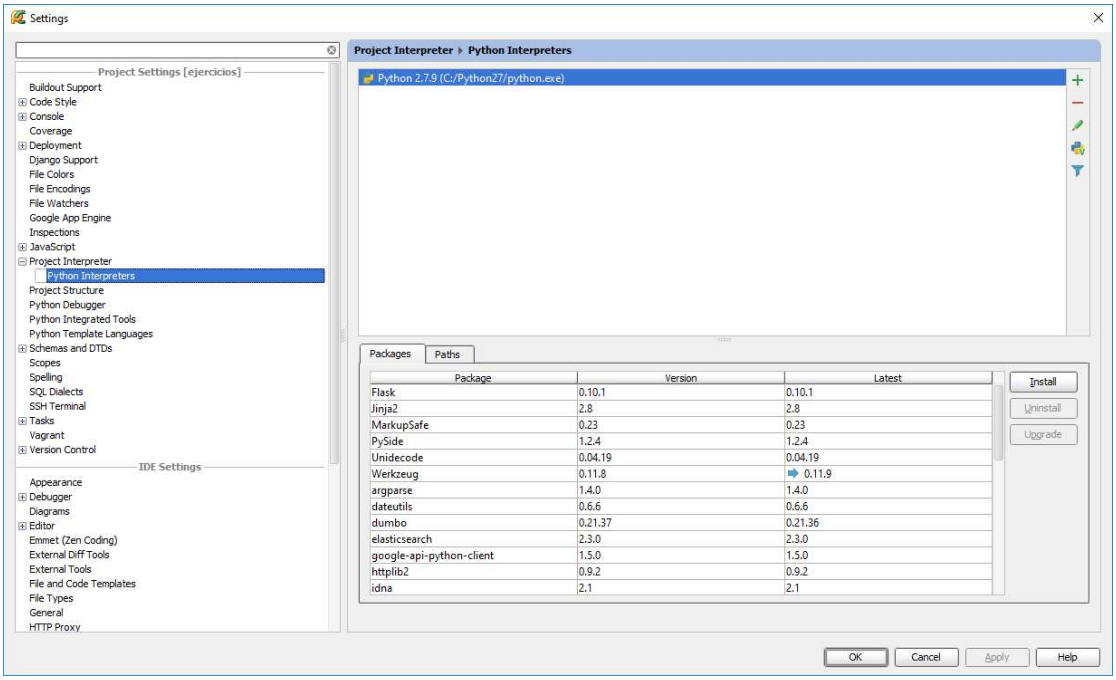


Ilustración 20 – Instalar un paquete 1

Podemos filtrar para buscar el paquete que queremos instalar, en este caso pep8 y le damos a instalar.

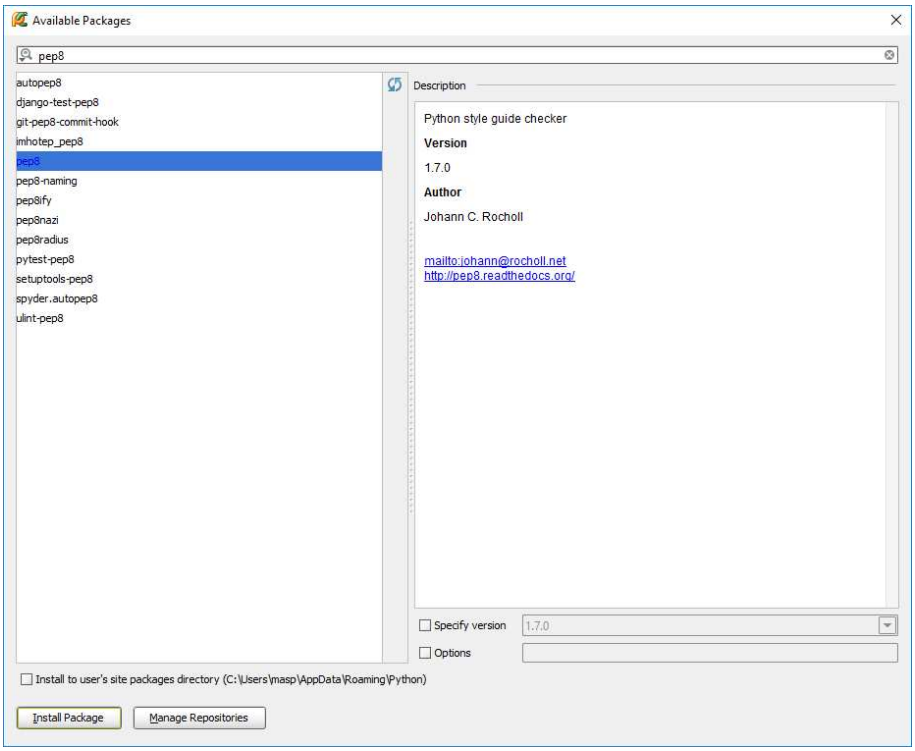


Ilustración 21 – Instalar un paquete 2

Si el paquete se instala correctamente veremos un mensaje que lo indica.

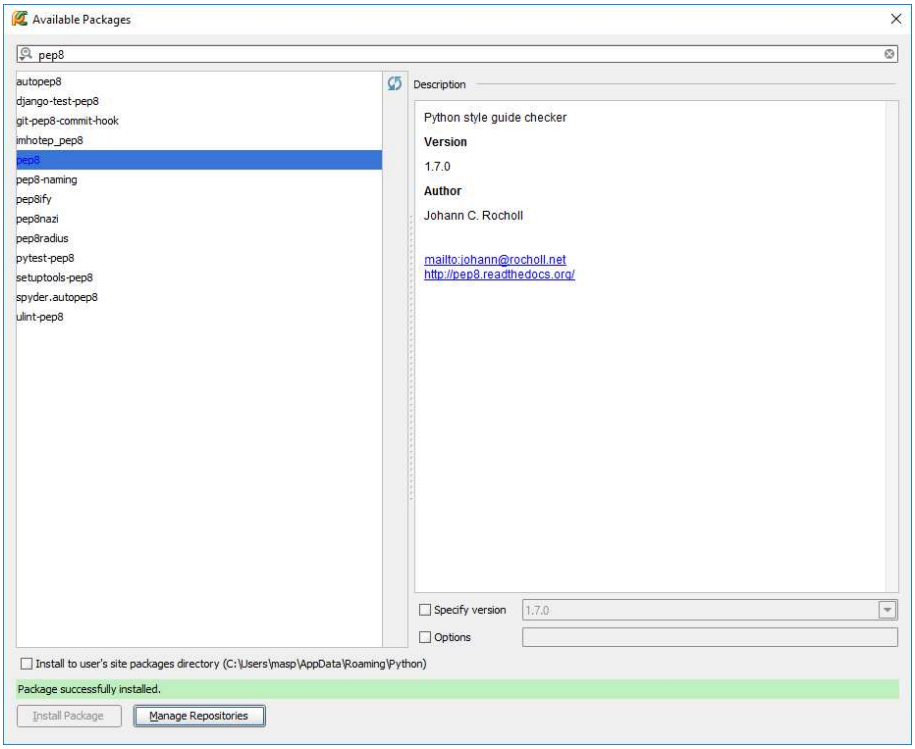


Ilustración 22 – Instalar un paquete 3

6.1.2. Reglas

Algunas de las reglas que se aplican son las siguientes:

- Sangrar (Indent) las líneas con 4 espacios por nivel.
- No mezclar nunca espacios con tabuladores. (Es preferible usar espacios).
- Limitar a 79 caracteres por línea.
- Separar las definiciones de funciones y las definiciones de clase con dos líneas en blanco.
- Separar los métodos definidos dentro de una clase por una línea en blanco.
- Se puede usar una línea en blanco dentro de una función, para separar secciones lógicas.
- Las sentencias de importación, por lo general, deben estar en líneas separadas.
- Se permite importar usando *from* en una única línea.
- Evitar usar espacios en blanco:
  - Dentro de paréntesis, llaves, o corchetes.
  - Antes de una coma, punto y coma y dos puntos.
  - Delante del paréntesis de apertura de la lista de argumentos de una función.

- No usar más de un espacio alrededor de una asignación.
- Usar espacios alrededor de los operadores aritméticos.
- Usar un espacio detrás de la coma.
- No utilizar espacios alrededor del signo `'='` cuando se usa para indicar un argumento o un valor de parámetros, o alrededor el signo `':'` en una operación de partición.
- No se recomienda usar las sentencias compuestas (varias instrucciones en la misma línea), aunque se permite usar las sentencias *if, for* o *while* con una pequeña instrucción en la misma línea.

6.1.3. Ejecución

Aunque con el tiempo escribiremos el código directamente con el formato recomendado existen varias herramientas que nos hacen esto más sencillo. Para verificar si un programa está escrito con el estilo correcto solo tenemos que ejecutar el programa pep8 para ver si tenemos algún error.

```
$> pep8 programa.py
```

Este programa nos permite modificar que errores queremos tener en cuenta en nuestro código por ejemplo si queremos permitir líneas más largas.

```
$> pep8 --max-line-length=99 programa.py
```

Otra opción es usar pycharm, esta herramienta incorpora no solo la validación del código mientras escribimos, sino que además podemos aplicar todas las reglas de estilo de forma automática. Solo tenemos que ir al menú `code -> Auto-Indent lines` y la herramienta arreglará de forma automática los errores de estilo.

6.2. PEP – 257: Estilo de documentación

Cuando escribimos el código una de las tareas que hay que hacer es escribir la documentación para que otras personas puedan utilizar nuestros programas. La especificación de Python tiene reservada la primera línea después de la definición de una función o clase para añadir la cadena de documentación. La documentación de una línea está reservada para métodos muy sencillos.

```
def foo():
    """Return the value of foo."""
    return foo
```

Código 68 – Documentación de una línea

Se utilizan las triples comillas para indicar este tipo de comentarios, y se pondrán ambas comillas en la misma línea que el texto. No debe dejar una línea en blanco ni delante ni detrás del comentario de documentación.

Lo normal será que en una única línea no podamos describir el comportamiento de una función algo más compleja. En este caso se utiliza la documentación multilínea.

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
```

Código 69 – Documentación multilínea

En este caso se utiliza la primera línea para dar la descripción de la función, una línea en blanco y después una descripción más detallada de la función. Para esta documentación de varias líneas se usa la triple comilla, pero en este caso el cierre está en una línea separada. En la documentación de funciones es habitual incluir los parámetros de la función. En las de la clase suelen incluirse un resumen con los métodos. Los módulos incluyen todas las clases que han definido, así como las funciones del api público.

7. Paquetes interesantes

Existen multitud de paquetes en Python muchos son de la librería estándar, pero otros muchos están desarrollados por terceras partes.

7.1. Paquete Collections

Este paquete de la librería estándar contiene distintos tipos de datos contenedores. Que sirven como una alternativa a las clases básicas de Python. Vamos a ver un ejemplo de una clase para modificar los diccionarios.

7.1.1. Counter

La clase `Counter` está pensada para manejar diversas formas de cuentas. Se trata de una subclase de `dict` y como tal, solo puede almacenar objetos que pueden usar la función predefinida `hash()`, el valor puede ser cualquier entero incluso con valores negativos. El constructor de esta clase permite crear un `Counter` a partir de cualquier iterable o mapa (incluso otro `Counter`).

```
from collections import Counter
c = Counter(a=1, b=2, c=3)
```

Código 70 – Creación de un counter

Un `Counter` utiliza las mismas funciones que un diccionario normal con algunas diferencias. No lanza una excepción cuando se intenta acceder a un elemento con una clave que no existe. La función `fromkeys()` no está implementada en la clase `Counter` y la función `update()` en vez de modificar los valores del contador los que hace es acumular los dos valores. Además de estos cambios la clase `Counter` incluye estos métodos.

Métodos	Descripción
elements()	Devuelve un iterador sobre los elementos de la clase y devuelve una copia de cada elemento por cada valor del contador. Si algún elemento tiene un contador negativo no se devolverá ningún elemento.
most_common([n])	Devuelve los elementos ordenados por su contador, si dos elementos tienen el mismo valor del contador se ordenarán de forma arbitraria. Si se incluye el parámetro *n* solo se devuelven los *n* elementos más frecuentes.
subtract([iterable-or-mapping])	Actualiza el diccionario utilizando un iterador o un mapa, restando los valores del iterador de los elementos del counter.

Tabla 35 – Métodos de Counter

Un ejemplo de los nuevos métodos de `Counter`.

```
from collections import Counter
c = Counter(a=4, b=2, c=0, d=-2)
d = Counter(a=1, b=2, c=3, d=4)
c.subtract(d)

print c
```

Código 71 – Counters, función subtract

7.2. Paquete argparse[19]

Este paquete que nos permite procesar los parámetros de entrada de un programa Python y los transforma en un `Namespace` que contiene un atributo por cada parámetro de entrada.

El Paquete `argparse` incluye la clase `argparse.ArgumentParser` que es la que se encarga de procesar la entrada. Esta clase tiene el método `add_argument()` que es el que se encarga de configurar los parámetros de entrada y el método `parse_args()` que es el encargado de parsear la entrada en función de los parámetros configurados.

[19] Consultar la referencia completa del paquete argparse para más información https://docs.python.org/2/library/argparse.html

```
import argparse

parser = argparse.ArgumentParser(description="Este programa saluda ")
parser.add_argument("-n", "--nombre", default="unknown")
parser.add_argument("-e", "--edad", help="edad", type=int, default=0)

args = parser.parse_args(('-n', 'nombre', '--edad', '10'))
print "Hola " + args.nombre + ". Tienes " + str(args.edad)
```

Código 72 – Parseo de la entrada con argparse

Los parámetros del método `parser.add_argument()` son los siguientes.

Métodos	Descripción
name or flags	Define el nombre o la lista de etiquetas que se aceptaran para ese parámetro. Los parámetros definidos por nombre se seleccionan de la entrada por orden los que incluyen una etiqueta no tienen un orden especifico pero deben ir precedidos de la etiqueta.
action	El tipo de acción que hay que realizar cuando se encuentra el parámetro, el valor por defecto, es almacenarlos pero existen otras muchas opciones.
nargs	El número de argumentos que deben consumirse de la entrada.
const	Un valor constante que es necesario en ciertas acciones.
default	Un valor por defecto para asignar al parámetro si no está presente en la entrada.
type	El tipo al que el parámetro debe ser convertido.
choices	Un contenedor con todos los valores validos del parámetro.
required	Indica si el parámetro es opcional o no, por defecto considera que las etiquetas son opcionales y sirve para hacerlas obligatorias.
help	Una descripción de lo que hace el parámetro para imprimirla en la ayuda.
metavar	El nombre del parámetro dentro de la ayuda.

Tabla 36 – Parámetros de parser.add_argument()

El método `parse_args(args=None, namespace=None)` es el encargado de convertir la entrada, por defecto utiliza como parámetro de entrada `sys.argv` pero se puede utilizar una secuencia para hacer pruebas.

7.3. Paquete csv[20]

El formato de ficheros CSV (fichero separado por comas por sus siglas en inglés) es un formato bastante habitual para los dataset. En Python tenemos el paquete csv para leer de forma rápida estos ficheros. Este paquete está compuesto por unas funciones globales y dos clases, una para leer `DictReader` y otra para escribir `DictWriter`. Entre las funciones que existen, las más interesantes son estas dos:

Funciones	Descripción
csv.reader(csvfile, dialect='excel', **fmtparams)	Devuelve un objeto de tipo csv.reader que nos permite iterar sobre las filas del fichero csv. El parámetro *diaclect* nos permite modificar la forma de tratar el fichero csv y por defecto es 'excel'. Además de estos dos podemos pasarle diversos parámetros de formateo que nos permiten modificar los valores por defecto del dialecto.
csv.writer(csvfile, dialect='excel', **fmtparams)	Devuelve un objeto de tipo csv.writer que nos permite crear el fichero csv, habitualmente será un objeto de tipo file y es recomendable abrirlo en formato binario. Al igual que en la lectura tenemos el parámetro *diaclect* que nos permite modificar la forma de tratar el fichero csv. Y también podemos pasarle diversos parámetros de formateo que nos permiten modificar los valores por defecto del dialecto.

Tabla 37 – Paquete csv, funciones predefinidas

[20] Para más información del paquete csv consultar la documentación https://docs.python.org/2/library/csv.html

Un ejemplo de uso de la función de lectura.

```
import csv

with open('./titanic.csv') as f:
    reader = csv.reader(f, delimiter=',', quotechar='"')
    print "header: " + str(reader.next())
    for line in reader:
        print line
```

Código 73 – Paquete csv, lectura

Además de estas funciones predefinidas también disponemos de la clase `DictReader` que nos permitirá iterar sobre las filas, convirtiendo cada una de ellas en un diccionario. Y la función `DictWriter` que nos permite escribir un diccionario en un fichero. En este ejemplo podemos verlo.

```
import csv

with open('./t.csv', mode='wb') as fout:
    with open('./titanic.csv') as f:
        dictReader = csv.DictReader(f, delimiter=',', quotechar='"')
        dictWriter = csv.DictWriter(fout, delimiter=',',
                                    quotechar='"',
                                    fieldnames=dictReader.fieldnames)

        for line in dictReader:
            dictWriter.writerow(line)
```

Código 74 – Paquete csv, lectura/escritura con diccionarios

7.4. Paquete tweepy

Este paquete proporciona un recubrimiento de Python para el API pública de twitter y nos permitirá usar tweets para hacer nuestros análisis.

7.4.1. Instalación

La forma más sencilla de instalar este paquete es usar `pip`, este programa suele venir en la instalación por defecto y su uso es muy simple.

```
$> pip install tweepy
```

Otra opción es clonar el repositorio de Github y después realizar la instalación manualmente.

```
$> git clone https://github.com/tweepy/tweepy.git

$> cd tweepy

$> python setup.py install
```

Si estamos usando pycharm podemos utilizar el mismo método que vimos en la instalación del paquete pep8, ver en el capítulo 0.

7.4.2. Registro

Para poder utilizar el api de twitter es necesario tener una cuenta de twitter y crear una aplicación en el siguiente enlace:

https://apps.twitter.com/

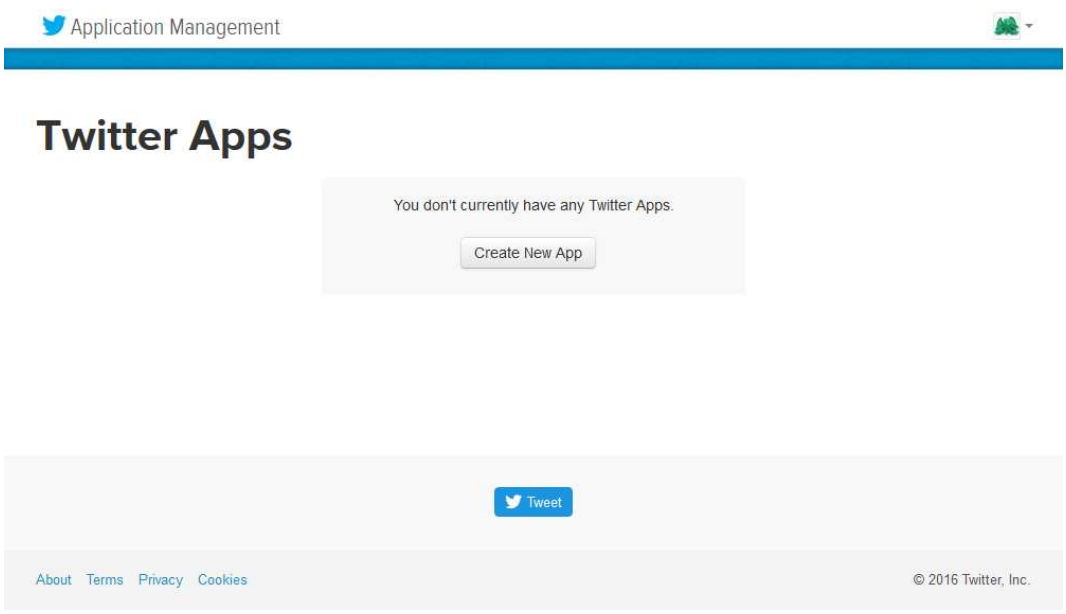


Ilustración 23 – Aplicación de twitter paso 1

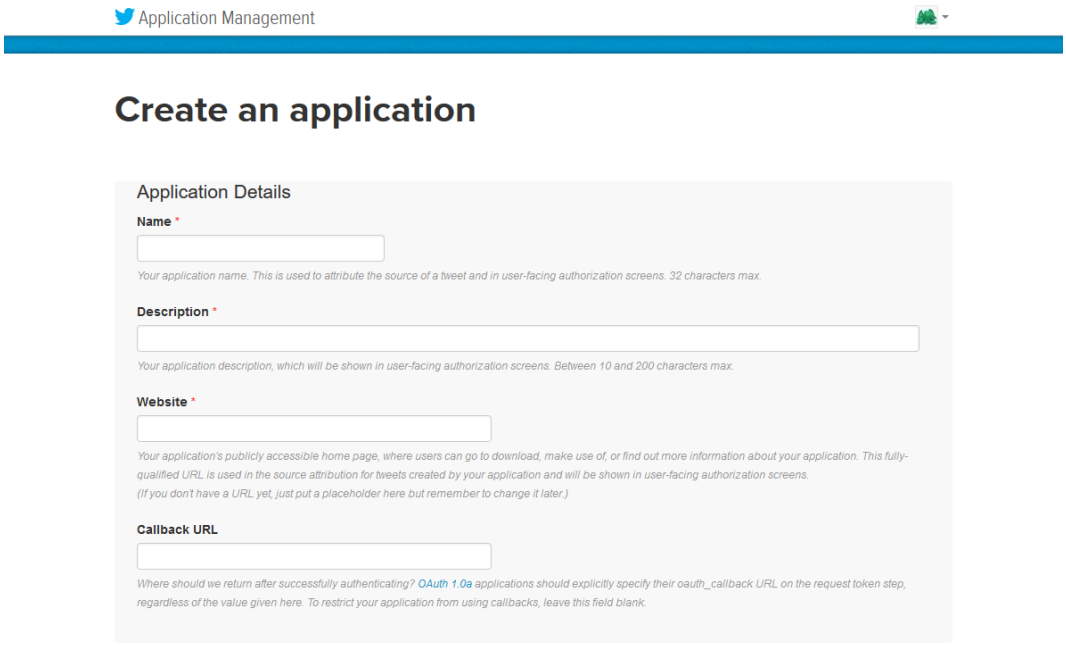


Ilustración 24 – Aplicación de twitter paso 2

Una vez completado el registro de la aplicación solo tenemos que ir a la pestaña *Keys and Access Tokens* y copiar nuestras claves de acceso a la aplicación.

7.4.3. Ejemplo

Aquí tenemos un ejemplo de código para acceder a nuestra cuenta de twitter y leer todos los tweets del *timeline,* hay que recordar cambiar los valores de las variables por los correspondientes a nuestra cuenta.

```
import tweepy

# config
consumer_key = 'XXXXXX'
consumer_secret = 'XXXXXX'

token = 'XXXXXX'
secret = 'XXXXXX'

auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(token, secret)

api = tweepy.API(auth)

public_tweets = api.home_timeline()
for tweet in public_tweets:
        print tweet.text
```

Código 75 – Leer tweets usando tweepy

Anexo 1. Índice de tablas

Anexo 2. Índice de ejemplos de código