Intr. al leguaje de programación Python

Tipos de Datos. Cadenas, listas, tuplas

Alejandro Roca Alhama

IES Cierva

Febrero de 2015

¿Qué vamos a ver? I

Introducción al leguaje de programación Python

¿Por dónde vamos?

Tipos numéricos básicos

En Python, los tipos de datos capaces de almacenar números son prácticamente idénticos a los que se pueden encontrar en otros lenguajes de programación. Y se utilizan para almacenar información como:

- Saldo actual de la cuenta bancaria.
- La distancia que nos separa de Marte.
- Número de visitas que hemos recibido en nuestra web.
- El número pi.
- ...

Tipos numéricos básicos

Varios tipos

- En Python, no hay un solo tipo de objeto numérico, sino varios:
  - Enteros y números en coma flotante.
  - Números complejos.
  - Números decimales de precisión fija.
  - Números fraccionarios.
  - Conjuntos.
  - Booleanos.
  - Enteros de precisión ilimitada.
  - Otros proporcionados por módulos.

Literales numéricos

Entre sus tipos básicos, Python proporciona enteros (positivos y negativos) así como números en coma flotante.

- Los enteros también se pueden escribir en otras bases, teniendo números en hexadecimal, octal y binario.
- Ofrece el tipo números complejos.
- Ofrece enteros con una precisión "ilimitada".

La siguiente tabla muestra cómo aparecen estos números cuando se escriben como literales:

Literal	Interpretación
1234, -24, 0, 99999999999999999	Enteros (tamaño ilimitado)
1.23, 2., 3.14e-10, 4e210, 4.0e+210	Números en coma fotante
0177, 0xff, 0b10000000	Octal, hexadecimal y binario (Python 2.7)
3+4j, 3.0+4.0j, 3J	Números complejos

Literales numéricos I

Cosas a destacar

Los literales numéricos en Python son muy sencillos de escribir, pero hay que remarcar:

- Literales enteros y en coma flotante.
  - Se escriben como cadenas de dígitos decimales.
  - Los números en coma flotante tienen un punto decimal y/o pueden escribirse con notación exponencial (número más una e (ó E) que introduce un exponente con signo).
  - Python usa aritmética en coma flotante siempre que sea necesario. Internamente lo hace con aritmética en 64 bits.
- Literales enteros en Python 2.7: normal y long.
  - Dos tipos de enteros: normal (artimética 32 bits) y long (precisión ilimitada).
  - Para forzar que un entero se trate como long, basta añadir al final una *l* ó una *L*.

Literales numéricos II

Cosas a destacar

  - Si un entero excede la representación en 32 bits (*overflow*), Python lo convierte automáticamente al tipo long.
- Literales en hexadecimal, octal y binario.
  - Los enteros se pueden codificar en base 10 (decimal), 16 (hexadecimal), 8(octal) y 2 (binario).
    - Hexadecimal: empiezan por 0x ó 0X y a continuación una cadena de dígitos hexadecimales (0-9, a-f, A-F).
    - Octal: empiezan por 0 seguido de una cadena de dígitos octales (0-7).
    - Binario: emppiezan por 0b ó 0B y a continuación una cadena de dígitos binarios (0, 1).
  - Internamente se codifican como enteros.
  - Se pueden convertir su valor entero a una representación de cadena concreta con: *hex(I)*, *oct(I)* y *bin(I)*.
  - Se puede convertir una cadena a un entero a través de la función *int(str, base)*.
- Números complejos.

Literales numéricos III

Cosas a destacar

- Se escriben de la forma *parte_real+parte_imaginaria*, donde la parte imaginaria acaba en 'j' ó 'J'. Internamente se eimplementan como un par de números flotantes, pero tienen definidas todas las operaciones de complejos. Se puede crear un número complejo a partir de la función *complex(real, imag)*.

- Otros.
  - El resto de tipos tienen sus propias reglas y sintaxis.

Herramientas numéricas en Python I

Para trabajar con números...

Para trabajar con números Python incorpora:

- Operadores para construir expresiones:
  - +, -, *, /, >>, **, &, etc.
- Funciones matemáticas internas:
  - pow, abs, round, int, hex, bin,...
- Módulos:
  - *math*: proporciona funciones como: *cos*, *sin*, *tan*, *sqrt*, *log*, *log10*, ... Y constantes como *pi* ó *e*.
  - Otras librerías como *random*, *fractions*, etc.
- En Python, todo son objetos, por lo que incluso los tipos básicos tienen métodos interesantes.

Expresiones I

¿Qué es una expresión?

Expresión

Una expresión es una fórmula que muestra cómo calcular un valor.

Las expresiones más sencillas están formadas por:

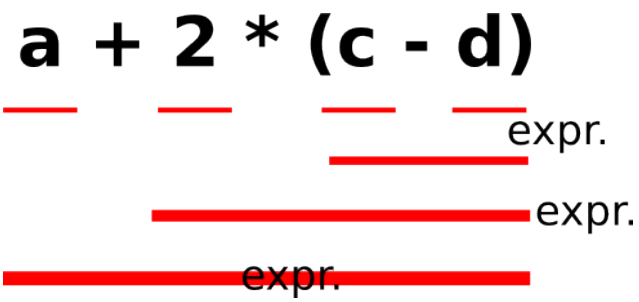
- Una constante (literal).
- Una variable.

Se pueden construir expresiones más complejas combinando expresiones con operadores aritméticos y lógicos.

Expresiones y operadores I

Un ejemplo

Una expresión más compleja podría ser:



La expresión "a + 2 * (c - d)" está formada por:

- La expresión a,
- la expresión 2 * (c - d),
- ... combinadas a través del operador +.

Expresiones y operadores I

Muchos operadores...

- Los operadores son las herramientas básicas para la construcción de expresiones.
- Python posee una gran cantidad de operadores, que se pueden clasificar en:
  - Operadores aritméticos.
  - Operadores relacionales.
  - Operadores lógicos.
  - Operadores de tratamiento de bits.
  - Etc.

Expresiones y operadores I

Operadores aritméticos

- Son los operadores que se utilizan en las operaciones aritméticas básicas: suma, resta, multiplicación, división y módulo.
- En Python son:

```
+    -    *    /    //    **    %
```

  - Son operadores binarios, necesitan dos operandos.
    - + y - también son unarios cuando señalan el signo.
  - El operador // es el operador de división entera:
    - / ->devuelve un real.
    - // ->división entera, devuelve un entero si dividimos números enteros.
  - % es el operador módulo, devuelve el módulo de una división entera.
    - Ejemplo: 8 % 3 devuelve 2.
  - ** es el operador potencia.
    - 8 ** 3 equivale a 8^3

Expresiones y operadores I

Precedencia de operadores

La precedencia de los operadores en Python es la siguiente:

```
x ** y
+    - (unarios)
*    /    //    %
+    -
==    !=
<    <=    >    >=
not x
x and y
x or y
```


Expresiones y operadores I

Precedencia de operadores: ejemplos

¿Qué operación se realiza primero?

A + B * C (aplicar precedencia)
(A + B) * C (paréntesis)
A * B + C * D (aplicar precendencia)
A + B + C (aplicar asociatividad)

Expresiones y operadores I

Mezcla de operadores y conversiones

¿Qué pasa en el siguiente código?

```
>>> a = 20
>>> b = 3.14
>>> print(a + b)
23.14
>>>
```

- ¿Por qué no se produce un error si los tipos son distintos?
- En Python, como en otros lenguajes hay promoción de tipos.
  - ... pero solo en tipos numéricos.

Expresiones y operadores I

Mezcla de operadores y conversiones

Regla en las conversiones

Python convierte el tipo más simple al tipo más complejo.

Para Python:

- Los números enteros son más simples que los números en coma flotante.
- Los números en coma flotante son más simples que los números complejos.

Además podemos realizar conversiones:

```
>>> int(3.1415926)
3
>>> float(89)
89.0
>>>
```

Otros tipos numéricos I

Hay más...

- Los tipos numéricos básicos en Python son:
  - *integer*, *floating point* y *complex*.
- Pero además se pueden utilizar los siguientes:
  - Decimal.
    - Muy similar a los números en coma flotante, pero permiten trabajar con un número fijo de decimales.
    - Muy útil para trabajar con cantidades de dinero o valores de cualquier tipo que necesiten de mayor exactitud.
    - Definido en el módulo *decimal*.
  - Fraction.
    - Permite trabajar con fracciones, números que constan de un numerador y un denominador.
    - Definido en el módulo *fractions*.
  - Set.
    - Definen un colección de objetos únicos, no ordenados e inmutables.

Otros tipos numéricos II

Hay más...

  - Son los conjuntos definidos en la teoría matemática de conjuntos, con las operaciones definidas en ella como unión, intersección, etc.
  - Booleans.
    - Define los valores definidos en el álgebra de Boole: *True*/*False*, más las operaciones lógicas y relacionales.
- Además existen una gran variedad de librerías de terceras partes para el cálculo y otras tareas matemáticas como:
  - NumPy.
  - SciPy.
  - ...

Introducción al leguaje de programación Python

¿Por dónde vamos?

2 Cadenas
- Literales
- Usando cadenas
- Métodos de cadena
- Formateo de cadenas

Cadenas I

Introducción

Definición

Una cadena o string es una colección ordenada de caracteres que se usa para almacenar y representar información basada en texto.

Las cadenas pueden ser utilizadas para almacenar cualquier tipo de información que se pueda codificar como texto:

- Palabras y símbolos (nombres, direcciones, teléfonos, ...).
- Contenidos de ficheros de texto cargados en memoria.
- Direcciones IP, direcciones de Internet (URLs), ...
- Programas Python.
- Código en otros lenguajes: C, C++, Java, (X)HTML, CSS, ...

Pero también permiten almacenar:

Cadenas II

Introducción

- Los valores binarios de un conjunto de bytes.
- Texto Unicode (multibyte).
- Etc.

Cadenas I

Más que en otros lenguajes

Aunque sean similares a las cadenas de otros lenguajes como C, en Python hay ciertas diferencias:

- Python no distingue entre cadenas de un solo carácter y cadenas de más caracteres.
  - Otros lenguajes como C sí hacen esta distinción.
  - En Python basta escribir: mi_caracter = 'F'
- Técnicamente, las cadenas son secuencias inmutables, eso quiere decir que una cadena no puede ser modificada una vez que se crea.
- Muchos operadores que se pueden utilizar con secuencias se pueden utilizar con cadenas.
- "" ó '' representa a la cadena vacía. La cadena vacía es una cadena que no contiene NINGÚN carácter.
- Las cadenas vienen acompañadas de un gran número de herramientas para su procesamiento:

Cadenas II

Más que en otros lenguajes

- Las cadenas permiten operaciones como concatenación, slicing, indexado, ...
- Existen varios métodos para procesarlas.
- Existen varios módulos para trabajar con cadenas (procesamiento de patrones, expresiones regulares, análisis XML, etc).

Cadenas I

Tipos de cadenas: Python 3 vs Python 2.X

En Python 2.X (2.6 y 2.7) existen dos tipos de cadenas:

- *unicode*: las cadenas *unicode* representan texto Unicode, normalmente en una codificación multibyte.
- *str*: permiten almacenar cadenas de texto con representación en 8 bits (ASCII, ISO-8859-XX, etc.) y datos binarios.

En Python 3 existen tres tipos:

- *str*: utilizado para texto Unicode y para cualquier otra codificación.
- *bytes*: datos binarios.
- *bytearray*: versión mutable de *bytes* (también disponible desde 2.6).

Cadenas I

Literales

Las cadenas son muy fáciles de utilizar en Python, hay muchas formas de definir una:

- Comillas simples.
- Comillas dobles.
- Comillas triples.
- Secuencias de escape.
- Raw strings.
- Cadenas Unicode (solo en 2.6).

Literales I

Comillas simples y comillas dobles

Son exactamente lo mismo y permiten hacer las mismas cosas. Es muy cómodo tener dos formas de especificar el inicio y el final de una cadena:

```
>>> print("hola")
hola
>>> print('hola')
hola
>>> dijo = 'Y me dijo: "Vete ahora mismo"'
>>> print(dijo)
Y me dijo: "Vete ahora mismo"
>>> heroes = "Superman" ' y ' "Batman"
>>> print(heroes)
Superman y Batman
>>>
```

Literales I

Secuencias de escape

Las secuencias de escape permiten introducir codificaciones de bytes especiales en una cadena.

- Se construyen con el carácter especial '\' seguido de uno o más caracteres. Destacan:

Escape	Significado
\newline	Ignora, permite continuar la línea más abajo
\\	Carácter \
\'	Carácter '
\"	Carácter "
\a	Bell
\n	Nueva línea
\t	Tabulador
\xNN	Carácter con el valor NN en hexadecimal
\oNN	Carácter con el valor NN en octal
\0	Null, caráter binario 0 (cero)
\other	No hay secuencia de escape, se mantiene \

Literales I

Secuencias de escape: ejemplo

```
>>> print("Superhéroes:\n\tSuperman\n\tBatman")
Superhéroes:
  Superman
  Batman
>>> print("Y me dijo: \"Fuera\"")
Y me dijo: "Fuera"
>>> print("C:\Windows\system32")     # Fuente de problemas
C:\Windows\system32
>>> print("C:\new\otro")
C:
ew\otro
>>> print(r"C:\new\otro")            # Usar cadenas raw
C:\new\otro
>>>
```

Literales I

\0 no finaliza la cadena

Hay dos formas de implementar cadenas:

- Al estilo de C
  - Las cadenas se almacenan en memoria como un conjunto de caracteres acabado por el carácter \0.
- Al estilo de Pascal... y de Python
  - Las cadenas se almacenan guardando la longitud seguida de los caracteres que la forman.
  - En Python el carácter \0 (null) NO FINALIZA una cadena.

Literales I

Comillas triples

Otra forma de encerrar las cadenas, también denominadas cadenas en bloque (*block string*), consiste en utilizar tres comillas (simples o dobles):

- Tres comillas + texto + tres comillas.
- Se pueden utilizar dentro tanto comillas simples como dobles sin necesidad de ser escapadas.
- Perfectas para cadenas multilínea ya que permiten mostrar en nuestros programas fragmentos de texto completos como ayuda, trozos de manual, mensajes de error, trozos de código XML ó (X)HTML, etc.
- También se utiliza en cadenas de documentación. Éstas cadenas permiten documentar el código de forma especial mientras se escribe.

Literales II

Comillas triples

```
>>> cita = '''
... "Disculpen si les llamo caballeros,
...  pero todavía no les conozco bien."
...                         Groucho Marx.
... '''
>>> print(cita)

"Disculpen si les llamo caballeros,
 pero todavía no les conozco bien."
                      Groucho Marx.

>>>
```


Cadenas en acción I

Operaciones básicas

Entre las operaciones básicas a realizar con las cadenas destacan:

- La concatenación (operador +).
- La repetición (operador *).
- Podemos saber la longitud (tamaño) de una cadena a través de la función *len*.

Cadenas en acción I

Operaciones básicas: ejemplo

```
>>> nombre = "Antonio"
>>> apellidos = "Sánchez Pérez"
>>> nombre_completo = nombre + " " + apellidos
>>> print(nombre_completo)
Antonio Sánchez Pérez
>>> print ("*" * 10)
**********
>>> print(len(nombre_completo))
21
>>> print(len(nombre))
7
>>> print(len(apellidos))
13
>>>
```


Cadenas en acción I

operaciones básicas: iteración

Podemos recorrer una cadena a través de un bucle for así:

```
cadena = "Iron Man"

for c in cadena:
    print(c)
```

Cadenas en acción I

Operaciones básicas: pertenencia

Podemos comprobar si una cadena es una subcadena de otra a través del operador *in*.

- También se puede hacer a través del método *str.find*.

```
>>> cadena = "Iron Man"
>>> 'a' in cadena
True
>>> 'X' in cadena
False
>>> 'Iron' in cadena
True
>>> if 'I' in cadena:
...       print("'I' es subcadena")
...
'I' es subcadena
>>>
```


Cadenas en acción I

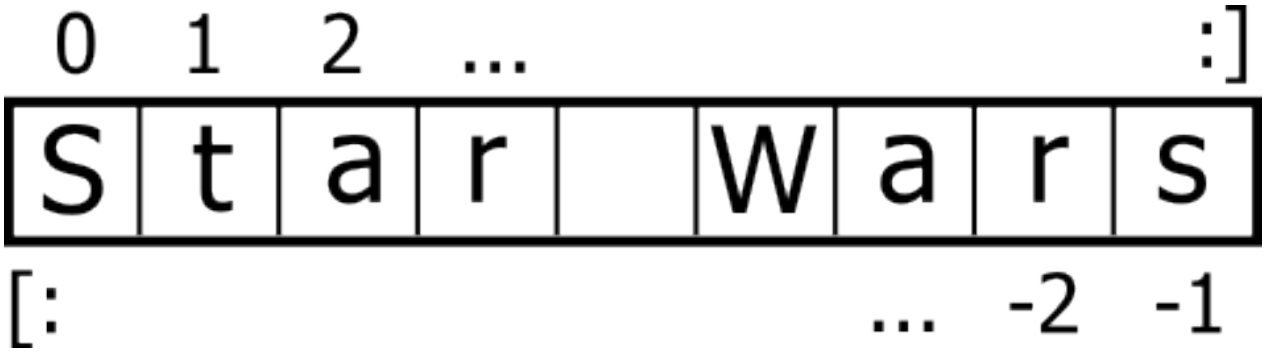
Operaciones básicas: indexado (indexing)

Como una cadena es una colección ordenada de caracteres, se puede acceder a cada uno de los caracteres que forman una cadena a través de sus posición.

- Como en C y el resto de lenguajes, el primer carácter de una cadena se encuentra en la posición 0.
- Python permite indexar caracteres con índices negativos, *-1* hace referencia al último carácter.
  - Técnicamente, Python suma el número negativo al número de caracteres de la cadena.

Cadenas en acción I

Operaciones básicas: indexing and slicing



Cadenas en acción I

Indexing: ejemplo

```
>>> cadena = 'Linux'
>>> cadena[0]
'L'
>>> cadena[4]
'x'
>>> cadena[-1]
'x'
>>> for i in range(len(cadena)):
...     print(cadena[i])
...
L
i
n
u
x
>>>
```

Cadenas en acción I

Operaciones básicas: extracción (slicing)

La operación de slicing permite extraer partes de una cadena, es una forma más general de indexar una cadena. Devuelve una sección completa en vez de un carácter.

- Su forma de uso es: **[index:index]**.
  - El primer número representa el primer carácter a tomar (incluido).
  - El segundo número representa el último carácter a tomar (no incluido).
- También se puede indicar: **[1:]**.
  - Toma todos los caracteres desde el primero (incluido).
- Y también: **[:-1]**.
  - Toma todos los caracteres hasta el último (no incluido).

Cadenas en acción I

Indexing/Slicing: un resumen

- **Indexing (S[i])**: toma elementos según un índice.
  - El primer item se encuentra en la posición 0.
  - Índices negativos indican que hay que contar desde el final.
  - S[0] devuelve el primer elemento.
  - S[-2] devuelve el segundo elemento desde el final (como S[len(S) - 2])
- **Slicing (S[i:j])**: extrae secciones continuas de una secuencia.
  - El límite superior no se incluye.
  - Los límites por defecto son cero y la longitud de la cadena.
    - S[::] == S[0:len(S)]
  - S[1:3] toma los elemento 1 y 2.
  - S[1:] toma los elementos desde 1 hasta la longitud de la cadena.
  - S[:3] toma los elemento 0, 1 y 2.
  - S[:-1] toma todos los elementos excepto el último elemento.
  - S[:] toma la cadena completa.

Cadenas en acción I

Indexing/Slicing: el tercer elemento

El slicing soporta un tercer elemento que indica el incremento a utilizar. La forma genérica es:

- **X[i:j:k]** que significa: toma todos elementos de X desde el elemento i al j - 1 de k en k elementos.
- Por defecto k es 1.
- Se pueden especificar incrementos negativos.
- Ejemplo:

Cadenas en acción II

Indexing/Slicing: el tercer elemento

```
>>> cadena = 'Programando en Python'
>>> cadena[0::2]
'Pormnoe yhn'
>>> cadena[0:12:2]
'Pormno'
>>> cadena[::-1]
'nohtyP ne odnamargorP'
>>> cadena[15::]
'Python'
>>> cadena[20:14:-1]
'nohtyP'
>>>
```

Cadenas en acción I

Modificando cadenas

- Las cadenas son secuencias *inmutables*, lo que significa que no se puede modificar una vez que se definan.

```
>>> cadena = 'Alex'
>>> cadena[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

- La única forma de modificar una cadena es construir otra a través de diversas operaciones y asignar, si se quiere, el mismo nombre.
- Hay un montón de formas y métodos para trabajar con cadenas: operadores, métodos, expresiones de formateo de cadenas, etc.

Cadenas en acción II

Modificando cadenas

```
>>> cadena = 'Alex'
>>> cadena = 'a' + cadena[1:]
>>> print(cadena)
alex
>>>
```

Métodos de cadenas I

Las cadenas son objetos

Además de los operadores, la clase cadena proporciona una gran cantidad de métodos para trabajar y manipular cadenas de texto.

- Sin llegar a introducir completamente la orientación a objetos podemos decir que un método es una forma diferente de realizar algo sobre un objeto en concreto, en nuestro caso una cadena.

Una función es una forma de empaquetar código, mientras que una llamada a un método combina dos operaciones en una (búsqueda de atributos y llamada):

- *Búsqueda de atributos*.
  - Una expresión de la forma **miobjeto.atributo1** significa "busca el valor del atributo 'atributo1' en el objeto 'miobjeto'.
- *Expresión de llamada*.

Métodos de cadenas II

Las cadenas son objetos

- Una expresión de la forma **funcion(argumentos)** significa "llama al código 'función' pasando cero o más argumentos separados por comas, y toma el valor devuelto por la función".

Métodos de cadenas I

Las cadenas son objetos (cont)

- Al llamar a un método de un objeto estamos haciendo esas dos cosas juntas.
- La forma genérica de invocación de métodos es: **objeto.metodo(argumentos)**
  1. Buscamos el método en el objeto.
  2. Lo llamamos pasándole los argumentos.
  3. Tomamos un resultado si el método devuelve uno.

Métodos de cadenas I

Métodos más importantes

Los métodos más utilizados con cadenas son:

S.capitalize() Devuelve la misma cadena con el primer carácter en mayúscula.

S.center(width[, fillchar])] Devuelve la cadena centrada en un tamaño de *width* caracteres. Por defecto se utilizar el carácter espacio como relleno a no ser que se especifique el carácter *fillchar*.

S.find(sub[, start[, end]]) Devuelve la posición de la primera ocurrencia de la subcadena *sub*. De forma optativa se pueden incluir (en formato slicing) las posiciones donde buscar. El método devuelve la posición -1 si no se encuentra.

S.join(iterable) Devuelve una cadena resultante de concatenar los elementos de la secuencia *iterable* separados con la cadena a la que se aplica el método.

Métodos de cadenas II

Métodos más importantes

S.partition(sep) Busca el separador *sep* en la cadena y devuelve una tupla formada por: los caracteres hasta el separador, el separador, los caracteres después del separador. Si no se encuentra, la tupla contendrá la cadena completa y dos cadenas vacías.

S.replace(old, new[, count]) Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena *old* por la cadena *new*. El parámetro *count* indica el número de ocurrencias como máximo a reemplazar.

S.split([sep[, maxsplit]]) Devuelve una lista conteniendo todas las subcadenas en las que se divide la cadena a través del separador *sep*. Por defecto se usa el espacio como separador. El argumento *maxsplit* especifica el número máximo de particiones a realizar.

Métodos de cadenas III

Métodos más importantes

Todos los métodos de cadena son:

```
S.capitalize    S.isalnum       S.join          S.rsplit
S.casefold      S.isalpha       S.ljust         S.rstrip
S.center        S.isdecimal     S.lower         S.split
S.count         S.isdigit       S.lstrip        S.splitlines
S.encode        S.isidentifier  S.maketrans     S.startswith
S.endswith      S.islower       S.partition     S.strip
S.expandtabs    S.isnumeric     S.replace       S.swapcase
S.find          S.isprintable   S.rfind         S.title
S.format        S.isspace       S.rindex        S.translate
S.format_map    S.istitle       S.rjust         S.upper
S.index         S.isupper       S.rpartition    S.zfill
```

Métodos de cadenas I

Ejemplo

Listing 1: Métodos de cadenas

```
#!/usr/bin/env python3

# Sustituyendo dentro de una cadena
cadena = 'Windows es un buen sistema operativo'
print("Longitud:", len(cadena))
print(cadena.replace('buen', 'mal'))
print()

# Buscando dónde se encuentra una subcadena
cadena = 'C, C++, Python, PHP'
print('Python se encuentra en la posición:', cadena.find('Python'))
print()

# Tenemos una lista de Sistemas Operativos y queremos mostrarlos así:
# Windows---Linux---Solaris---MacOS
l = ['Windows', 'Linux', 'Solaris', 'MacOS']
print('---'.join(l))
print()

# Separamos las diferentes partes de un dominio:
dominio = 'www.google.es'
print('Subdominios de', dominio + ':')
for parte in dominio.split('.'):
```


Métodos de cadenas II

Ejemplo

```
    print(parte)
print()

# Rellenamos cadenas con ceros para mostrar todos los número de 0 a 100
# con tres dígitos
for i in range(101):
  print(str(i).zfill(3), end = ' ')
```

Expresiones de formateo de cadenas I

Introducción

Con los métodos de manipulación de cadenas ya vistos se puede hacer de todo, aún así, Python proporciona un mecanismo más avanzado para la realización de tareas de procesamiento de cadenas: *string formatting*.

- Este mecanismo permite realizar en un solo paso varias sustituciones basadas en tipos.
- Dos "sabores":
  - Expresiones de formateo de cadenas.
    - Técnica original desde los principios de Python. Basada en el modelo definido por la función *printf* de C.
  - Llamadas a métodos de formateo de cadenas.
    - Añadida en Python 2.6 y Python 3.0. Es la recomendada actualmente y recoge gran parte de la funcionalidad de la anterior.

String Formatting Method Calls I

Lo básico

- El método *format* permite usar una cadena como una plantilla en la que se van sustituyendo valores representados por los argumentos que se pasan en el método.
- En la cadena se especifican los puntos en los que se hacen las sustituciones a través de llaves, bien por posición, $\{1\}$ o bien por nombre $\{edad\}$.
- Ejemplos:

String Formatting Method Calls II

Lo básico

```
>>> cadena = 'Ejemplos de sistemas operativos: {0}, {1} y {2}.'
>>> print(cadena.format('Linux', 'MacOS', 'Windows'))
Ejemplos de sistemas operativos: Linux, MacOS y Windows.
>>>
>>> print('En {0}, {1} tenía {2} años'.format(2013, 'Bill Gates',
    57))
En 2013, Bill Gates tenía 57 años
>>>
>>> cadena = 'Windows {version} vió la luz en el año {fecha}'
>>> print(cadena.format(version = '1.0', fecha = 1985))
Windows 1.0 vió la luz en el año 1985
>>>
```


String Formatting Method Calls I

Más cosas básicas

- Como argumentos al método format se puede pasar de todo: cadenas, números, listas, tuplas, diccionarios...

```
>>> print('The special one: {0[0]}'.format(['Linux', 'Windows',
     'MacOS']))
The special one: Linux
>>>
```

- Además se pueden utilizar especificadores de formato para modificar cómo se tienen que mostrar los parámetros. Se indican así:

{**fieldname!conversionflag:formatspec**}

- Indicando:
  - *fieldname*: índice o nombre del campo.
  - *conversionflag*: indican la conversión a realizar a través de las llamadas *repr* (r), *str* (s) ó *ascii* (a).

String Formatting Method Calls II

Más cosas básicas

- *formatspec*: especifica cómo tiene que presentarse el argumento en cuanto a ancho, alienación, relleno, posiciones decimales, etc.

String Formatting Method Calls I

formatspec

- *formatspec* toma la siguiente forma general:

  [[fill]align][sign][width][.precision][type]

  - fill: carácter de relleno, espacios por defecto.
  - align: izquierda(<), derecha(>), centrada(^), relleno antes del signo decimal(=).
  - sign: siempre(+), solo en negativos(-), dejar un espacio en vez del + (ESPACIO).
  - width: ancho.
  - precision: número de decimales.
  - type: enteros(d), binarios(b), octales(o), hexadecimales(x ó X), punto flotante(f, F, E, g ó G, n), cadenas(s), caracteres(c), ...

String Formatting Method Calls I

formatspec. Ejemplos

```
>>> print('-{0:<10}-'.format('Linux'))
-Linux     -
>>> print('-{0:>10}-'.format('Linux'))
-     Linux-
>>> print('-{0:^10}-'.format('Linux'))
-  Linux   -
>>> print('-{0:=^10}-'.format('Linux'))
-==Linux===-
>>>
>>> x = 255
>>> print('{0:10d}'.format(x))
       255
>>> print('{0:10x}'.format(x))
        ff
>>> print('{0:10b}'.format(x))
  11111111
>>>
>>> import math
>>> i = math.pi
```


String Formatting Method Calls II

formatspec. Ejemplos

```
>>> print('{0:10.2f}'.format(i))
      3.14
>>> print('{0:10.2e}'.format(i))
  3.14e+00
>>> print('{0:+10.2f}'.format(i))
     +3.14
>>> print('{0:+010.2f}'.format(i))
+000003.14
>>>
```

Introducción al leguaje de programación Python

¿Por dónde vamos?

Sobre las cadenas I

Hasta ahora solo números y cadenas

- Hasta ahora solo se han utilizado dos tipos de datos: números y cadenas.
- Muchos de los operadores ya vistos se utilizan en el resto de tipos.
- Tanto los números como las cadenas son tipos muy importantes, pero no son los más significativos que Python incorpora.

Tipos de datos principales en Python I

Solo tres :)

Los tipos de datos principales en Python son:

- **Números**.
  - Enteros, punto flotantes, fracciones, otros.
  - Soportan operaciones como la suma, la multiplicación...
- **Secuencias**.
  - Cadenas, listas y tuplas.
  - Soportan operaciones como la indexación (indexing), la extracción (slicing), la concatenación, etc.
- **Mapas**.
  - Diccionarios.
  - Soportan operaciones como la indexación por clave, etc.

Quedan un poco fuera de esta categoría los **conjuntos** que podrían tener una categoría propia.

Otra clasificación I

Mutables e inmutables

Básicamente podemos clasificar los tipos de datos (y por tanto los objetos) en:

- **Inmutables**
  - Estos objetos NO SE PUEDEN modificar una vez creados.
  - Se pueden construir objetos nuevos ejecutando expresiones sobre los existentes.
  - Ejemplos: números, cadenas, tuplas, frozensets.
- **Mutables**.
  - Se pueden modificar sin necesidad de crear nuevos objetos.
  - También se pueden realizar copias.
  - Ejemplos: listas, diccionarios, conjuntos.

Tipos de datos en Python I

Todos los incluidos en el lenguaje

- **None**
  - type(None)
- **Booleans**
  - bool
- **Numbers**
  - int
  - float
  - complex
- **Sequences**
  - str
  - list
  - tuple
  - range
- **Mapping**
  - dict
- **Sets**
  - set
  - frozenset

Listas I

Uso básico en Python

- Las *listas*, junto con los *diccionarios*, son los tipos de datos más utilizados en Python.
- Ambos representan colecciones de objetos y ambos son mutables.
- Estos tipos son los que permiten la construcción de estructuras de datos ricas y complejas en nuestros programas Python.

Listas I

Introducción

- Una lista representa una colección ordenada de objetos.
- Es un tipo de objetos muy flexible.
- Una sola lista puede contener un conjunto de objetos, todos los objetos no tienen por qué ser del mismo tipo.
- Son mutables, a una lista se pueden añadir más objetos, se pueden eliminar objetos de ella, etc.
- Permiten operaciones de indexado, slicing, etc.

Listas I

Características principales

Las listas representan un tipo de datos básico, la ventaja es que el propio lenguaje Python las incorpora, mientras que en otros lenguajes como C, hay que implementarlas (o utilizar una librería).
Las listas de Python son:

- *Colección ordenada de objetos arbitrarios*.
  - Representan un lugar para almacenar objetos. Todos eso objetos almacenados se tratan como un grupo.
  - Las listas son secuencias: los elementos permanecen ordenados de izquierda a derecha.
- *Acceso a través de un índice (offset)*.
  - El acceso a los elementos de una lista se realiza a través del operador [], igual que con las cadenas.
  - Al tratarse de colecciones ordenadas también son aplicables las operaciones slicing y concatenación.

Listas II

Características principales

- *De longitud variable, heterogéneas y anidables*
  - Las listas pueden crecer y decrecer según se vayan añadiendo o quitando elementos.
  - Pueden contener cualquier tipo de objetos: números, caracteres, cadenas, ... Y de distintos tipos al mismo tiempo.
  - Se puede tener listas de listas de listas de...
- *Secuencias mutables*
  - Se puede modificar una lista: cambiar un elemento por otro, añadir nuevos elementos, borrar elementos, etc.
  - Todas las operaciones que se pueden realizar sobre una cadena también se aplican a las listas: indexado, slicing, concatenación... La diferencia es que devuelven otra lista en vez de otra cadena.
- *Arrays de referencias a objetos*
  - Técnicamente una lista contiene cero o más referencias a otros objetos. Esta definición recuerda a los arrays de punteros de C.

Listas III

Características principales

- Python no implementa arrays como hacen otros lenguajes. Incorpora un módulo *array*, pero rara vez se usa. Las listas de Python son casi tan rápidas como los arrays de C.

Listas I

Resumen de las operaciones básicas

Operación	Interpretación
L = []	Una lista vacía
L = list()	Otra forma de crear una lista vacía
L = [0, 1, 2, 3]	Una lista con cuatro elementos
L = [[0, 1], [2, 3]]	Una lista con dos listas
L = list('Linux')	Creamos una lista a partir de una secuencia
L = list(range(10))	...o de otra secuencia
L[i]	Elemento de la lista en la posición i
L[i][j]	Elemento j de la lista de la posición i
L[i:j]	Slicing: elementos desde la posición i a j - 1
len(L)	Número de elementos de la lista
L1 + L2	Concatenación de listas
L * 3	Repetición de los elementos de una lista

Listas I

Resumen de las operaciones básicas (cont.)

Operación	Interpretación
for x in L: print(x)	Iteración por todos los elementos de una lista
8 in L	Pertenencia a la lista
L.append(8)	Añade un elemento nuevo a la lista
L.extend([0, 1, 2])	Añade elementos desde un iterable
L.insert(i, x)	Insertar el objeto x en la posición i
L.index('A')	Permite buscar elementos en la lista
L.count('A')	Cuentas las ocurrencias de un elemento en la lista
L.sort()	Ordena
L.reverse()	Invierte la lista de orden
del L[i]	Eliminar el objeto en la posición i de la lista
del L[i:j]	Eliminar todos los objetos desde la posición i hasta j - 1
L.pop()	Elimina el último elemento de la lista

Listas I

Resumen de las operaciones básicas (y último)

Operación	Interpretación
L.remove(x)	Elimina la primera ocurrencia del objeto x
L[i:j] = []	Borrar todos los elementos entre i y j - 1 sustituyéndolos por el objeto x
L[i] = 1	Asigna el objeto a la posición i
L[i:j] = [4, 5, 6]	Asignación por slicing

Listas I

Listado de todos los métodos

Todos los métodos de listas son:

```
L.append   L.copy     L.extend   L.insert   L.remove   L.sort
L.clear    L.count    L.index    L.pop      L.reverse
```

Listas en acción I

Creación de listas

- Podemos crear listas básicamente de dos formas:
  - Indicando directamente los elementos de la lista entre corchetes y separados por coma.
  - Utilizando el constructor de la clase lista.
    - Las listas son objetos de la clase *list*.

```
>>> l1 = [0, 1, 2, 3]                    # Lista con 4 números
>>> l2 = ['Linux', 'Windows', 'MacOS']   # Lista de cadenas
>>> l3 = []                              # Creamos una lista vacía
>>> l4 = list()                          # Creamos otra lista vacía
>>> l5 = list('Linux')                   # A partir de una cadena
>>> print(l5)
['L', 'i', 'n', 'u', 'x']
>>> l6 = list(range(10))                 # A partir de otro iterable
>>> print(l6)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```


Listas en acción I

Lists comprehensions (listas por comprensión)

- Lists Comprehensions es una expresión compacta utilizada para definir listas. Heredada en Python de los lenguajes funcionales.
- Ejemplos:
  - Construir una lista con todos los cuadrados de 0 a 10:
  - Sin lists comprehensions:

```
l = list()
for i in range(11):
  l.append(i * i)
```

  - Con lists comprehensions:

```
l = [i * i for i in range(11)]
```

Listas en acción I

Operaciones básicas

- Por ser secuencias, las listas soportan las mismas operaciones que puede soportar una cadena: operador + y operador *, pero con el significado de concatenar listas y repetir listas.
- Ejemplos:

```
>>> l = ['Linux', 'Windows', 'MacOS']
>>> len(l)
3
>>> l + ['Solaris', 'AIX']
['Linux', 'Windows', 'MacOS', 'Solaris', 'AIX']
>>> l * 2
['Linux', 'Windows', 'MacOS', 'Linux', 'Windows', 'MacOS']
>>>
```


Listas en acción I

Iteración sobre listas y pertenencia

Podemos recorrer una lista fácilmente así:

```
sistemas = ['Linux', 'Windows', 'MacOS']
for sistema in sistemas:
    print(sistema)
```

También podemos comprobar si un elemento forma parte de una lista así:

```
>>> sistemas = ['Linux', 'Windows', 'MacOS']
>>> 'Windows' in sistemas
True
>>> 'Solaris' in sistemas
False
>>>
```

Listas en acción I

Indexación

Por ser secuencias, las indexación funciona exactamente igual que con las cadenas:

```
>>> sistemas = ['Linux', 'Windows', 'MacOS', 'Solaris']
>>> sistemas[1] = 'Windows 7'
>>> print(sistemas)
['Linux', 'Windows 7', 'MacOS', 'Solaris']
>>> print(sistemas[1])
Windows 7
>>> print(sistemas[0])
Linux
>>> print(sistemas[len(sistemas) - 1])
Solaris
>>>
```


Listas en acción I

Slicing

Como en cualquier secuencia:

```
>>> sistemas = ['Linux', 'Windows 7', 'MacOS', 'Solaris']
>>> sistemas[-1]
'Solaris'
>>> sistemas[::2]
['Linux', 'MacOS']
>>> sistemas[::-1]
['Solaris', 'MacOS', 'Windows 7', 'Linux']
>>> sistemas[0:4]
['Linux', 'Windows 7', 'MacOS', 'Solaris']
>>>
```

Listas en acción I

Matrices

- Una matriz es un array de más de una dimensión, la única forma de crear matrices en Python es anidando listas. Es decir, creando una lista de listas.
- Podemos crear una matriz de 3 x 3 fácilmente:

```
>>> matrix = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> matrix2 = [ [1, 2, 3],
...             [4, 5, 6],
...             [7, 8, 9]]
>>> matrix
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> matrix2
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
```


Listas en acción II

Matrices

- A la hora de acceder, un solo índice devuelve una lista completa, un segundo índice devuelve un elemento.

```
>>> matrix[0][0]
1
>>> matrix[0][1]
2
>>> matrix[2][2]
9
>>>
```

- ¿Cómo podríamos recorrer toda la matriz?

Listas en acción I

Recorriendo una matriz

Listing 2: Recorrer una matriz 3x3

```
#!/usr/bin/env python3

matrix = [ [1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]]

for fila in matrix:
  for j in fila:
    print(j, end = ' ')
  print()
```


Listas en acción I

Modificando listas: indexado y slicing

Al tratarse de objetos mutables, las listas pueden modificarse a través de las operaciones de indexado y slicing:

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[9] = 100
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 100]
>>> l[0:2] = [0, 0]
>>> l
[0, 0, 2, 3, 4, 5, 6, 7, 8, 100]
>>> l[5:10] = []
>>> l
[0, 0, 2, 3, 4]
>>>
```

En el último ejemplo (slice assignment) la asignación se realiza en dos pasos:

1. *Borrado*. La sección indicada a la izquierda de la asignación se borra.

Listas en acción II

Modificando listas: indexado y slicing

2. *Inserción*. En la posición donde se borraron los elementos se realiza la inserción de los elementos indicados a la derecha de la asignación.

Listas en acción I

Modificando listas: métodos de listas

- Las operaciones de indexado y slicing no son las únicas formas de modificar una cadena. La clase *list* implementa varios métodos para la manipulación de listas.

Listing 3: Ejemplos de uso de métodos de lista

```
#!/usr/bin/env python3

# Creamos una lista y añadimos varios elementos
l = list()
l.append(10)
l.append(18)
l.append(24)
l.append(36)
l.append(2)
l.append(7)
print("Lista original:", l)
print()

# Invertimos la lista
l.reverse()
```

Listas en acción II

Modificando listas: métodos de listas

```
print(l)
print()

# Ordenamos la lista y la mostramos
l.sort()
print(l)
print()

# Podemos ordenar la lista en orden inverso así:
l.sort(reverse=True)
print(l)
print()

# Añadimos un nuevo elemento y lo quitamos
l.append(134)
print(l)
l.pop()
print(l)
print()

# Añadimos varios números al final
l.extend([100, 200, 300])
print(l)
print()

# Buscamos el número 200
i = l.index(200)
```


Listas en acción III

Modificando listas: métodos de listas

```
print("El número 200 está en la posición", i)
# En esa misma posición metemos el número 500 y lo eliminamos
l.insert(i, 500)
print(l)
l.remove(500)
print(l)
# Eliminamos también el elemento en la posición 0
l.pop(0)
# Otra vez
del l[0]
print(l)
print()
```

Diccionarios I

Uso básico (revisited)

- Los *diccionarios*, junto con las listas, son los tipos de datos más utilizados en Python.
- Ambos representan colecciones de objetos y ambos son mutables.
- Estos tipos son los que permiten la construcción de estructuras de datos ricas y complejas en nuestros programas Python.

Diccionarios I

Introducción

- Los diccionarios son, posiblemente, uno de los datos más flexibles que Python proporciona.
- Listas y diccionarios son los tipos de datos fundamentales:
  - Mientras que las listas son colecciones **ordenadas** de objetos...
  - ... los diccionarios son colecciones desordenadas.
  - La diferencia principal es que en un diccionarios los objetos son almacenados y recuperados a través de una clave (**key**), en vez de una posición.

Diccionarios I

Tipo de datos incorporado en Python

- Es un tipo incluido en el propio lenguaje.
- Los diccionarios sustituyen a estructuras de datos y algoritmos que en otros lenguajes hay que implementar de forma manual.
- Sustituyen a los registros, estructuras y tablas de símbolos de otros lenguajes.
- La indexación en un diccionario es una operación muy rápida.

Diccionarios I

Características principales

Entre sus características principales destacan:

- *Accedidas a través de una clave.*
  - También se les denomina arrays asociativos o hashes.
  - Asocian un conjunto de valores a través de una clave o *key*.
  - Acceso muy similar al de una lista, pero a través de dicha clave en vez de un desplazamiento.
- *Colección desordenada de objetos arbitrarios.*
  - Los valores almacenados en un diccionario no se almacenan siguiendo ningún orden particular. Las claves proporcionan la localización de cada elemento de forma simbólica.
- *De tamaño variable, heterogénea y anidable.*
  - Los diccionarios pueden crecer y decrecer en cualquier momento sin que se realice ninguna copia sobre ellos.
  - Pueden contener objetos de cualquier tipo.

Diccionarios II

Características principales

- Un diccionario puede contener un diccionario que contenga un diccionario que contenga...
- *Mapeos mutables.*
  - Un diccionario puede modificarse...
  - ... pero no soporta operaciones que soportan otros tipos secuenciales como cadenas o listas. Al ser objetos desordenados, las operaciones de concatenación ó slicing simplemente no tienen sentido.
  - Unico tipo de datos de la categoría "mapping" (mapean claves a valores).
- *Tablas de referencias a objetos (hash tables).*
  - Internamente se implementan como tablas hashes.
    - Una tabla hash es una estructura de datos que asocia claves con valores.
    - La principal operación que soportan de forma eficiente es la búsqueda.
  - Un diccionario almacena referencias a objetos.

Diccionarios I

Resumen de las operaciones básicas

Operación	Interpretación
D = {}	Un diccionario vacío
D = dict()	Otra forma de crear un diccionario vacío
D1 = {'cd': 8, 'dvd': 2}	Un diccionario con dos elementos
D2 = {'HP': {'A': 8, 'B': 9}}	Diccionarios anidados
D1['cd']	Indexando por clave
D2['HP']['A']	Indexando a través de dos diccionarios
'cd' in A	Pertenencia: ¿la clave 'cd' está presente?
D.keys()	Método que devuelve una lista con todas las claves
D.values()	Método que devuelve una lista con todos los valores
D.items()	Método que devuelve una lista de tuplas (key, value)
D.copy()	Método que devuelve una copia del diccionario

Diccionarios I

Resumen de las operaciones básicas (cont.)

Operación	Interpretación
D.get(key, default)	Método que devuelve el valor bajo key o default si key no existe
D.update(D2)	Método que actualiza el diccionario D con los valores de D2
D.pop(key)	Borra el valor bajo key
len(D)	Número de entradas almacenadas en el diccionario
D['cd'] = 42	Modifica el valor bajo la clave 'cd'
del D[key]	Borra una entrada por clave
list(D.keys())	Crea una lista formada por todas las claves del diccionario
D = {x: x * 2 for x in range(10)}	Diccionarios por comprensión (comprehension)

Diccionarios I

Listado de todos los métodos

Todos los métodos de diccionarios son:

```
D.clear       D.get         D.pop         D.update
D.copy        D.items       D.popitem     D.values
D.fromkeys    D.keys        D.setdefault
```

Diccionarios en acción I

Operaciones básicas

Se puede crear un diccionario a base de literales y acceder a él a través de claves:

```
>>> d = {'windows': 12, 'linux': 34, 'macos': 20}
>>> print(d['linux'])
34
>>> print(d)
{'windows': 12, 'macos': 20, 'linux': 34}
>>>
```

¡¡ El orden no es importante !!

Diccionarios en acción I

El orden no es importante

Del ejemplo anterior se deducen varias cosas:

- El orden no es importante.
- Para implementar una búsqueda de valores por clave (hashing) las claves necesitan ser reordenadas en memoria.
- El mapeo es muy eficiente en cuanto a búsquedas.
- Esta reordenación hace que las operaciones de concatenación y slicing no tengan sentido, ya que implican un orden.

Diccionarios en acción I

La función len devuelve el número de elementos

Listing 4: Función len

```
#!/usr/bin/env python3

peliculas = dict()

peliculas['Star Wars'] = 'George Lucas'
peliculas['Terminator'] = 'James Cameron'
peliculas['Blade Runner'] = 'Ridley Scott'
peliculas['Lord of the Rings'] = 'Peter Jackson'
peliculas['Robocop'] = 'Paul Verhoeven'

print(len(peliculas), 'peliculas en nuestra colección')
if 'Star Wars' in peliculas:
  print('Tenemos Star Wars')
print

print('Las películas son:')
for peli in list(peliculas.keys()):
  print('-', peli)

# Un diccionario es iterable, también lo podemos recorrer así:
print('Las películas son:')
for key in peliculas:
  print('-', key)
```


Diccionarios en acción II

La función len devuelve el número de elementos

Diccionarios en acción I

Modificando diccionarios

Lo diccionarios son mutables:

Listing 5: Mutabilidad

```
#!/usr/bin/env python3

peliculas = dict()

peliculas['Star Wars'] = 'George Lucas'
peliculas['Terminator'] = 'Pedro Almodóvar'

# Modificamos una de las entradas
peliculas['Terminator'] = 'James Cameron'

# Añadimos dos entradas nuevas
peliculas['Lord of the Rings'] = 'Peter Jackson'
peliculas['Alien'] = 'Ridley Scott'

# Eliminamos entradas del diccionario, dos formas:
del peliculas['Star Wars']
peliculas.pop('Lord of the Rings')

print('Las películas de nuestra colección son:')
for key in peliculas:
  print('- {0} ({1})'.format(key, peliculas[key]))
```


Diccionarios en acción II

Modificando diccionarios

Diccionarios en acción I

Métodos de diccionarios

Varios métodos interesantes:

Listing 6: Métodos

```
#!/usr/bin/env python3

peliculas = {'Star Wars': 'George Lucas', 'Terminator': 'James Cameron',
             'Lord of the Rings': 'Peter Jackson', 'Alien': 'Ridley Scott'}

print('Los directores son:')
print(peliculas.values())

# Podemos recuperar el diccionario como una lista de tuplas:
print(peliculas.items())
# Recorremos dicha lista:
for peli, director in peliculas.items():
  print('- {0} ({1})'.format(peli, director))

# Directores
print(peliculas.get('Star Wars'))
# Y si no existe...
print(peliculas.get('Blade Runner'))
print(peliculas.get('Blade Runner', 'No existe'))

# Más peliculas
```


Diccionarios en acción II

Métodos de diccionarios

```
nomegustan = {'Saw': 'James Wan', 'The Blair Witch Project': ['Eduardo Sánchez', 'Daniel
      Myrick']}
peliculas.update(nomegustan)
print('Pelis actualizadas:')
for peli, director in peliculas.items():
  print('- {0} ({1})'.format(peli, director))

# Borramos las que no nos gustan
peliculas.pop('Saw')
peliculas.pop('The Blair Witch Project')
print('Las que me gustan')
for key in peliculas:
  print('- {0} ({1})'.format(key, peliculas[key]))
```

Diccionarios en acción I

Notas sobre el uso de diccionarios

Los diccionarios son muy fáciles de utilizar pero hay que tener siempre presente:

- **Las operaciones sobre secuencias no funcionan.**
- **Las asignaciones sobre nuevas claves añaden entradas al diccionario.**
- **Las claves no tienen por qué ser cadenas, se permite cualquier otro objeto inmutable**.

Diccionarios en acción I

Uso de diccionarios como listas flexibles

Este código es ilegal:

```
>>> l = list()
>>> l[99] = 'Alex'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

Pero este otro no:

```
>>> l = dict()
>>> l[99] = 'Alex'
>>>
```

Diccionarios en acción I

Errores por claves no existentes

Es muy común en el uso diario de diccionarios acceder a valores a través de claves inexistentes. Posibles soluciones:

- Probar antes que la clave existe.
- Usar excepciones.
- Usar el método get, devuelve *None* por defecto.

Diccionarios en acción I

Uso de diccionarios como registros

- Es muy común el uso de registros en otros lenguajes de programación.
- Un registro permite agrupar datos relacionados y acceder a ellos a través del nombre del campo.
- Podemos simular los registros a través del uso de diccionarios.

Ejemplo: agenda telefónica.

Diccionarios en acción I

Una agenda telefónica

Listing 7: Agenda telefónica

```
#!/usr/bin/env python3

agenda = list()

alias = input("Introduza el alias del nuevo contacto (INTRO para terminar): ")
while alias != '':
  contacto = {}
  contacto['alias'] = alias
  contacto['nombre'] = input("Introduza el nombre completo: ")
  contacto['telefono'] = input("Introduza el número de teléfono: ")
  agenda.append(contacto)

  alias = input("Introduza el alias del nuevo contacto (INTRO para terminar): ")

print("Mostramos la lista de nuestros contactos:")
print(agenda)

print("La mostramos en formato lista:")
print('{0:^10} - {1:^30} - {2:^10}'.format("ALIAS", "NOMBRE", "TELÉFONO"))
for contacto in agenda:
  print('{0:>10} - {1:>30} - {2:^10}'.format(contacto['alias'], contacto['nombre'],
        contacto['telefono']))
```


Diccionarios en acción II

Una agenda telefónica

Diccionarios en acción I

Estructura de datos

El uso de diccionarios y listas (y tuplas) permite en Python la construcción de nuevas estructuras de datos. Esto permite poder utilizar de forma fácil información relacionada:

```
>>> contacto = {}
>>> contacto['nombre'] = 'antonio'
>>> contacto['edad'] = 26
>>> contacto['trabajo'] = 'programador/analista'
>>>
>>> print(contacto['nombre'])
antonio
>>>
```


Diccionarios en acción II

Estructura de datos

```
>>> antonio = {'alias': 'WarState',
...            'trabajo': ['programador', 'analista'],
...            'web': 'http://www.warstate.local',
...            'direccion': {'provincia': 'MURCIA', 'CP': '30162'}}
>>> print(antonio['alias'])
WarState
>>> print(antonio['trabajo'])
['programador', 'analista']
>>> print(antonio['trabajo'][0])
programador
>>> print(antonio['direccion']['CP'])
30162
>>>
```

Tuplas I

Introducción

- Una tupla es un tipo de datos que permite construir grupos de objetos.
- Funcionan casi exactamente igual que las listas, pero son inmutables.
- No proporcionan tantos métodos como las listas, pero sí comparten varias de sus propiedades.
- Se escriben entre paréntesis.

Tuplas I

Características principales

Las tuplas representan el último tipo de datos básico de Python. Entre sus características principales destacan:

- *Colección ordenada de objetos arbitrarios.*
  - Como las cadenas y las listas, las tuplas son colecciones de objetos ordenados por posición.
  - Pueden almacenar cualquier tipo de objeto.
- *Acceso a través de un índice (offset).*
  - El acceso a los elementos de una tupla se realiza a través de un índice, de la misma forma que se hace con las cadenas y con las listas.
  - Las tuplas soportan todas las operaciones que permite este acceso por desplazamiento, como son las operaciones de indexado y slicing.
- *Secuencias inmutables.*
  - Las tuplas son secuencias, igual que lo son las cadenas y las listas.

Tuplas II

Características principales

- Además son inmutables, no permiten ninguna operación de modificación.

- *De longitud fija, heterogéneas y anidables*
  - Como son inmutables, la única forma de modificar el tamaño de una tupla es crear una nueva a través de una operación de asignación, tal como se hacía con las cadenas.
  - Una tupla puede almacenar cualquier tipo de objetos: números, cadenas, listas, diccionarios, otras tuplas, ...
  - Se puede tener tuplas de tuplas de tuplas de...
- *Arrays de referencias a objetos*
  - Como las listas, una tupla se puede ver como un array de referencias a objetos.
  - Una tupla almacena punteros (referencias) a otros objetos. Esto hace que la indexación de una tupla sea una operación relativamente rápida.

Tuplas I

Resumen de las operaciones básicas

Operación	Interpretación
()	Una tupla vacía
T = (0,)	Una tupla de un solo elemento (no es una expresión)
T = (0, 'Python', 3.14)	Una tupla de tres elementos
T = 0, 'Python', 3.14	La misma tupla anterior
T = ('abc', ('def', 'ghi'))	Tuplas anidadas
T = tuple('Python')	Creando una tupla a partir de un objeto iterable
T[i]	Elemento de la tupla en la posición i
T[i][j]	Elemento j del objeto de la tupla en la posición i
T[i:j]	Slicing: elementos desde la posición i a j - 1
len(T)	Número de elementos de la tupla
T1 + T2	Concatenación de tuplas
T * 3	Repetición de los elementos de una tupla

Tuplas I

Resumen de las operaciones básicas (cont.)

Operación	Interpretación
for x in T: print x	Iteración por todos los elementos de una tupla
8 in T	Pertenencia a la tupla
[x ** 2 for x in T]	Listas por comprensión utilizando tuplas
T.index('A')	Permite buscar elementos en una tupla
T.count('A')	Cuentas las ocurrencias de un elemento en la tupla

Tuplas I

Listado de todos los métodos

Las tuplas cuentan con pocos métodos:

```
T.count  T.index
```

Tuplas en acción I

Operaciones básicas

Creación, concatenación, repetición, indexado y slicing:

```
>>> a = (1, 2)
>>> b = (3, 4)
>>> a + b
(1, 2, 3, 4)
>>> (7, 7) * 3
(7, 7, 7, 7, 7, 7)
>>> a = (0, 1, 2, 3, 4, 5)
>>> a[0]
0
>>> a[1:3]
(1, 2)
>>> a[-1]
5
>>>
```


Tuplas en acción I

Ciertas peculiaridades sintácticas

Los paréntesis permiten delimitar una expresión, por lo que hay que especificarle a Python cuándo unos paréntesis:

1. Determinan una expresión simple, o...
2. ... definen una tupla de un solo elemento.

```
>>> x = (64)          # Definimos un entero
>>> type(x)
<class 'int'>
>>> y = (64,)         # Definimos una tupla que contiene solo un
    entero
>>> type(y)
<class 'tuple'>
>>> y
(64,)
>>>
```

Tuplas en acción II

Ciertas peculiaridades sintácticas

Además Python permite omitir los paréntesis en todos los casos en los que no haya ambigüedad:

```
>>> x = 0, 1, 2, 3          # Definimos una tupla
>>> type(x)
<class 'tuple'>
>>>
```

Consejo:

Consejo: utilizar siempre los paréntesis para definir y utilizar tuplas.

Tuplas en acción I

Conversiones, métodos e inmutabilidad

Exceptuando sintaxis, las operaciones sobre tuplas son idénticas a las operaciones sobre listas y cadenas. Diferencias:

- Las operaciones +, * y slicing devuelven tuplas nuevas.
- Las tuplas no definen los mismos métodos que se definían para cadenas o listas.
  - Ejemplo: no existe un método sort para ordenar tuplas.
  - ¿Cómo se ordena una tupla?

```
>>> t = ('Windows 7', 'Linux', 'Solaris', 'AIX', 'MacOS')
>>> tmp = list(t)
>>> tmp.sort()
>>> t = tuple(tmp)
>>> t
('AIX', 'Linux', 'MacOS', 'Solaris', 'Windows 7')
>>>
>>> t = ('Windows 7', 'Linux', 'Solaris', 'AIX', 'MacOS')
>>> t = sorted(t)
```

Tuplas en acción II

Conversiones, métodos e inmutabilidad

```
>>> t
['AIX', 'Linux', 'MacOS', 'Solaris', 'Windows 7']
>>>
```


Tuplas en acción I

Métodos index y count

Las tuplas solo definen dos métodos:

- index: Devuelve la posición del elemento a buscar. Se genera una excepción si el elemento no se encuentra.
- count: Devuelve el número de ocurrencias del elemento a buscar en la tupla.

Tuplas en acción I

Métodos index y count: ejemplo

```
>>> t = (23, 46, 81, 23, 56, 35, 89, 2)
>>> t.index(46)
1
>>> t.index(23)
0
>>> t.index(100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
>>> t.count(23)
2
>>> t.count(100)
0
>>>
```


Tuplas en acción I

Inmutabilidad

La inmutabilidad de las tuplas afecta a la tupla en sí, no a sus contenidos:

```
>>> t = ('Lenguajes', ['python', 'C', 'C#'])
>>> t[0] = 'Programación'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t[1][2] = 'Java'
>>> t
('Lenguajes', ['python', 'C', 'Java'])
>>>
```

Tuplas en acción I

¿Por qué listas y tuplas?

- Las matemáticas están por todas partes :)
- Las tuplas proporcionan un tipo de datos que nunca va a ser modificado, lo que permite tener constantes.
- Una tupla puede ser usada en determinadas situaciones donde las listas no pueden ser utilizadas: claves en diccionarios, ciertas operaciones y funciones, etc...

Como regla general:

- Usar listas para cualquier colección ordenada que pueda ser modificada,
- ... para el resto: tuplas.

Bibliografía I

Para saber más...


González Duque, R.
Python para todos.
http://mundogeek.net/tutorial-python/


Lutz, M.
Learning Python.
Ed. O'Reilly, 4ª ed. 2009.


Python Programming Language. Official Site.
http://python.org/