Python:

**Encapsulamiento**

Permite **empaquetar** el código dentro de una unidad (objeto) donde se puede determinar el ámbito de actuación.

**Abstracción**

Permite **generalizar** los tipos de objetos a través de las clases y simplificar el programa.

**Herencia**

Permite **reutilizar** código al poder heredar atributos y comportamientos de una clase a otra.

**Polimorfismo**

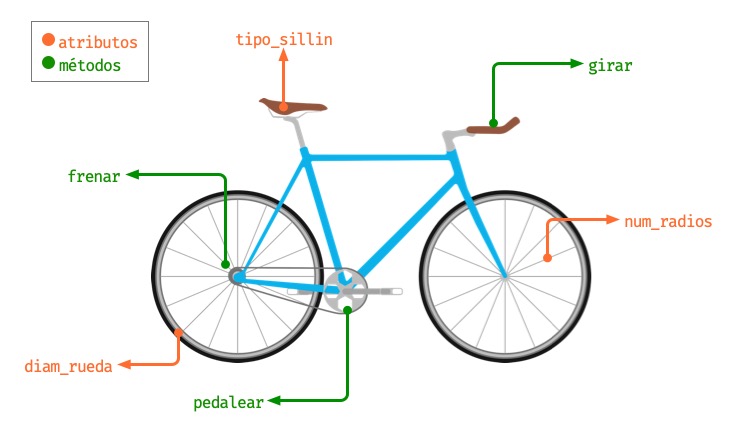
Permite **crear** múltiples objetos a partir de una misma pieza de código flexible.

¿Qué es un objeto?

Un **objeto** es una **estructura de datos personalizada** que contiene **datos** y **código**:

| **Elementos** | **¿Que Son?** | **¿Cómo se llaman?** | **¿Cómo se identifican?** |
| --- | --- | --- | --- |
| datos | Variables | Atributos | nombres |
| codigo | funciones | Métodos | verbos |

Un objeto representa **una instancia única** de alguna entidad a través de los valores de sus atributos e interactúan con otros objetos (o consiguen los mismos) a través de sus métodos.



¿Qué es una clase?

Para crear un objeto primero debemos definir la clase que lo contiene. Podemos pensar en la **clase** como el **molde** con el que crear nuevos objetos de ese tipo.

En el **proceso de diseño** de una clase hay que tener en cuenta – entre otros – el **principio de responsabilidad única**, intentando que los atributos y los métodos que contengan estén enfocados a un objetivo único y bien definido.

Creando objetos

Empecemos por crear nuestra **primera clase**.

Para ello usaremos la palabra reservada class seguido del nombre de la clase:

Consejo: Los nombres de clases se suelen escribir en formato CamelCase y en singular.

**class** **StarWarsDroid**:

**pass**

Existen multitud de droides en el universo StarWars. Una vez que hemos definido la clase genérica podemos crear **instancias/objetos** (droides) concretos:

c3po = StarWarsDroid()

r2d2 = StarWarsDroid()

bb8 = StarWarsDroid()

type(c3po)

\_\_main\_\_.StarWarsDroid

type(r2d2)

\_\_main\_\_.StarWarsDroid

type(bb8)

\_\_main\_\_.StarWarsDroid

Añadiendo atributos

Un **atributo** no es más que una variable, un nombre al que asignamos un valor, con la particularidad de vivir dentro de una clase o de un objeto.

Los atributos – por lo general – se suelen autorizar durante la creación de un objeto, pero también es posible añadirlos a posteriori:

blue\_droid = StarWarsDroid()

golden\_droid = StarWarsDroid()

golden\_droid.name = 'C-3PO'

blue\_droid.name = 'R2-D2'

blue\_droid.height = 1.09

blue\_droid.num\_feet = 3

blue\_droid.partner\_droid = golden\_droid *# otro droide como atributo*

Una vez creados, es muy sencillo acceder a los atributos:

golden\_droid.name

'C-3PO'

blue\_droid.num\_feet

3

Hemos definido un droide «socio». Veremos a continuación que podemos trabajar con él de una forma totalmente natural:

type(blue\_droid.partner\_droid)

\_\_main\_\_.StarWarsDroid

blue\_droid.partner\_droid.name *# acceso al nombre del droide socio*

'C-3PO'

blue\_droid.partner\_droid.num\_feet *# aún sin definir!*

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'StarWarsDroid' object has no attribute 'num\_feet'

blue\_droid.partner\_droid.num\_feet = 2

Añadiendo métodos

Un **método** es una función que forma parte de una clase o de un objeto. En su ámbito tiene acceso a otros métodos y atributos de la clase o del objeto al que pertenece.

La definición de un método (de instancia) es análoga a la de una función ordinaria, pero incorporando un primer parámetro self que hace referencia a la instancia real del objeto.

Una de las acciones más sencillas que se pueden hacer sobre un droide es encenderlo o apagarlo. Vamos a implementar estos dos métodos en nuestra clase:

**class** **Droid**:

**def** switch\_on(self):

print("Hi! I'm a droid. Can I help you?")

**def** switch\_off(self):

print("Bye! I'm going to sleep")

k2so = Droid()

k2so.switch\_on()

Hi! I'm a droid. Can I help you?

k2so.switch\_off()

Bye! I'm going to sleep

Inicializacion

Existe un **método especial** que se ejecuta cuando creamos una instancia de un objeto. Este método es \_\_init\_\_ y nos permite asignar atributos y realizar operaciones con el objeto en el momento de su creación. También es muy conocido como el **constructor**.

Veamos un ejemplo de este método con nuestros droides en el que únicamente guardaremos el nombre del droide como un atributo del objeto:

**class** **Droid**:

# Definición del constructor.

**def** \_\_init\_\_(self, name):

# Atributo del constructor

self.name = name

droid = Droid('BB-8')

# Creación del objeto (y llamada implícita al constructor)

droid.name

# Acceso al atributo name creado previamente en el constructor.

'BB-8'

Atributos

Acceso directo

En el siguiente ejemplo vemos que, aunque el atributo name se ha creado en el constructor de la clase, también podemos modificarlo desde «fuera» con un acceso directo:

**class** **Droid**:

**def** \_\_init\_\_(self, name):

self.name = name

droid = Droid('C-3PO')

droid.name

'C-3PO'

droid.name = 'waka-waka' *# esto sería válido!*

Propiedades

Como hemos visto previamente, los atributos definidos en un objeto son accesibles públicamente. Esto puede parecer extraño a personas desarrolladoras de otros lenguajes. En Python existe un cierto «sentido de responsabilidad» a la hora de programar y manejar este tipo de situaciones.

Una posible solución «pitónica» para la privacidad de los atributos es el uso de propiedades. La forma más común de aplicar **propiedades** es mediante el uso de decoradores:

* @property para leer el valor de un atributo.
* @name.setter para escribir el valor de un atributo.

Veamos un ejemplo en el que estamos ofuscando el nombre del droide a través de propiedades:

**class** **Droid**:

**def** \_\_init\_\_(self, name):

self.hidden\_name = name

**@property**

**def** name(self):

print('inside the getter')

**return** self.hidden\_name

**@name**.setter

**def** name(self, name):

print('inside the setter')

self.hidden\_name = name

droid = Droid('N1-G3L')

droid.name

inside the getter

'N1-G3L'

droid.name = 'Nigel'

inside the setter

droid.name

inside the getter

'Nigel'

En cualquier caso, seguimos pudiendo acceder directamente a .hidden\_name:

droid.hidden\_name

'Nigel'

VALORES CALCULADOS

Una propiedad también se puede usar para devolver un **valor calculado** (o computado).

A modo de ejemplo, supongamos que la altura del periscopio de los droides astromecánicos se calcula siempre como un porcentaje de su altura. Veamos cómo implementarlo:

**class** **AstromechDroid**:

**def** \_\_init\_\_(self, name, height):

self.name = name

self.height = height

**@property**

**def** periscope\_height(self):

**return** 0.3 \* self.height

droid = AstromechDroid('R2-D2', 1.05)

droid.periscope\_height *# podemos acceder como atributo*

0.315

droid.periscope\_height = 10 *# no podemos modificarlo*

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: can't set attribute

La ventaja de usar valores calculados sobre atributos simples es que el cambio de valor en un atributo no asegura que actualicemos otro atributo, y además siempre podremos modificar directamente el valor del atributo, con lo que hemos obtenido efectos colaterales indeseados.

Ocultando atributos

Python tiene una convención sobre aquellos atributos que queremos hacer **«privados»** (u ocultos): comenzar el nombre con doble subguión.

**class** **Droid**:

**def** \_\_init\_\_(self, name):

self.\_\_name = name

droid = Droid('BC-44')

droid.\_\_name *# efectivamente no aparece como atributo*

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Droid' object has no attribute '\_\_name'

Lo que realmente ocurre tras el telón se conoce como « name mangling » y consiste en modificar el nombre del atributo incorporado la clase como un prefijo. Sabiendo esto podemos acceder al valor del atributo supuestamente privado:

droid.\_Droid\_\_name

'BC-44'

Atributos de clase

Podemos asignar atributos a las clases y serán heredados por todos los objetos instanciados de esa clase.

A modo de ejemplo, en un principio, todos los droides están diseñados para que obedezcan a su dueño. Esto lo conseguiremos a nivel de clase, salvo que ese comportamiento se sobrescriba:

**class** **Droid**:

obeys\_owner = **True** *# obedece a su dueño*

good\_droid = Droid()

good\_droid.obeys\_owner

True

t1000 = Droid()

t1000.obeys\_owner = **False** *# T-1000 (Terminator)*

t1000.obeys\_owner

False

Droid.obeys\_owner *# el cambio no afecta a nivel de clase*

True

Métodos

Un método de instancia es un método que modifica el comportamiento del objeto al que hace referencia. Recibe self como primer parámetro, el cual se convierte en el propio objeto sobre el que estamos trabajando. Python envía este argumento de forma transparente.

Veamos un ejemplo en el que, además del constructor, creamos un método de instancia para desplazar un droide:

**class** **Droid**:

**def** \_\_init\_\_(self, name): *# método de instancia -> constructor*

self.name = name

self.covered\_distance = 0

**def** move\_up(self, steps): *# método de instancia*

self.covered\_distance += steps

print(f'Moving ***{***steps***}*** steps')

droid = Droid('C1-10P')

droid.move\_up(10)

Moving 10 steps

Métodos de clase

Un método de clase es un método que modifica el comportamiento de la clase a la que hace referencia. Recibe cls como primer parámetro, el cual se convierte en la propia clase sobre la que estamos trabajando. Python envía este argumento de forma transparente. La identificación de estos métodos se completa aplicando el decorador @classmethod a la función.

Veamos un ejemplo en el que implementaremos un método de clase que lleva la cuenta de los droides que hemos creado:

**class** **Droid**:

count = 0

**def** \_\_init\_\_(self):

Droid.count += 1

**@classmethod**

**def** total\_droids(cls):

print(f'***{***cls.count***}*** droids built so far!')

droid1 = Droid()

droid2 = Droid()

droid3 = Droid()

Droid.total\_droids()

3 droids built so far!

Métodos estáticos

Un método estático es un método que no modifica el comportamiento del objeto ni de la clase. No recibe ningún parámetro especial. La identificación de estos métodos se completa aplicando el decorador @staticmethod a la función.

Veamos un ejemplo en el que creamos un método estático para devolver las categorías de droides que existen en StarWars:

**class** **Droid**:

**def** \_\_init\_\_(self):

**pass**

**@staticmethod**

**def** get\_droids\_categories():

**return** ['Messeger', 'Astromech', 'Power', 'Protocol']

Droid.get\_droids\_categories()

['Messeger', 'Astromech', 'Power', 'Protocol']

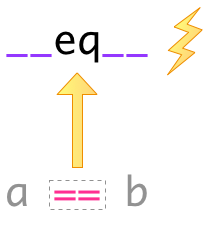
Metodos magicos

Cuando escribimos ¿cómo sabe el objeto lo que debe hacer para multiplicarse con el objeto entero? O dicho de otra forma, ¿cuál es la implementación del operador para «strings» y enteros? En valores numéricos puede parecer evidente (siguiendo los operadores matemáticos), pero no es así para otros objetos. La solución que proporciona Python para estas (y otras) situaciones son los métodos mágicos .'hello world' \* 3'hello world'3\*

Los métodos mágicos empiezan y terminan por doble subguión \_\_(es por ello que también se les conoce como «dunder-methods»). Uno de los «dunder-methods» más famosos es el constructor de una clase: \_\_init\_\_().

Digamos que los métodos mágicos se «disparan» de manera transparente cuando utilizamos ciertas estructuras y expresiones del lenguaje.

Para el caso de los operadores, existe un método mágico asociado (que podemos personalizar). Por ejemplo, la comparación de dos objetos se realiza con el método \_\_eq\_\_():



Extrapolando esta idea a nuestro universo StarWars, pudimos establecer que dos droides son iguales si su nombre es igual, independientemente de que tengan distintos números de serie:

**class** **Droid**:

**def** \_\_init\_\_(self, name, serial\_number):

self.serial\_number = serial\_number

self.name = name

**def** \_\_eq\_\_(self, droid):

**return** self.name == droid.name

droid1 = Droid('C-3PO', 43974973242)

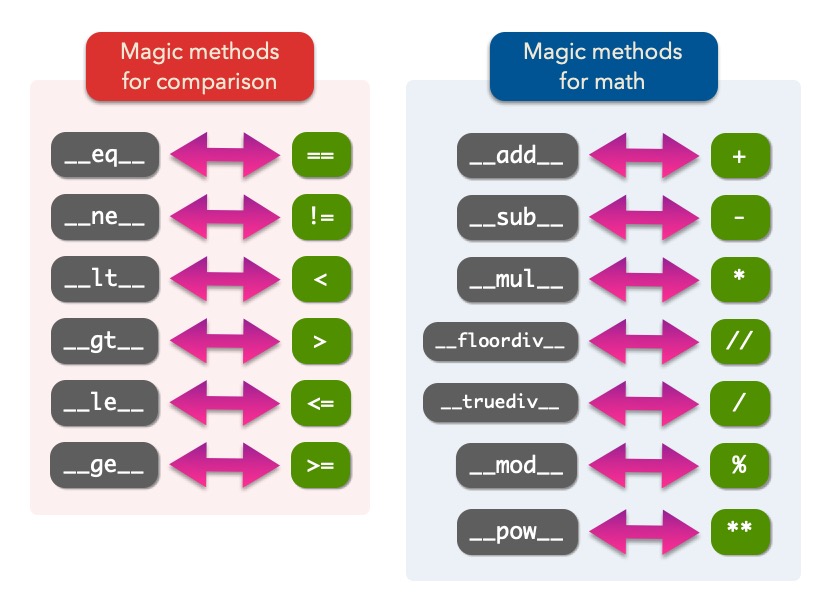
droid2 = Droid('C-3PO', 85094905984)

droid1 == droid2 *# llamada implícita a \_\_eq\_\_*

True

droid1.\_\_eq\_\_(droid2)

True



Los métodos mágicos no sólo están restringidos a operadores de comparación o matemáticos. Existen muchos otros en la documentación oficial de Python, donde son llamados métodos especiales.

Veamos un ejemplo en el que «sumamos» dos droides. Esto se podria ver como una fusion. Supongamos que la suma de dos droides implica:

a) que el nombre del droide resulta es la concatenación de los nombres de los droides;

b) que la energía del droide resultó es la suma de la energía de los droides:

**class** **Droid**:

**def** \_\_init\_\_(self, name, power):

self.name = name

self.power = power

**def** \_\_add\_\_(self, droid):

new\_name = self.name + '-' + droid.name

new\_power = self.power + droid.power

**return** Droid(new\_name, new\_power) *# Hay que devolver un objeto de tipo Droid*

droid1 = Droid('C3PO', 45)

droid2 = Droid('R2D2', 91)

droid3 = droid1 + droid2

print(f'Fusion droid:**\n*{***droid3.name***}*** with power ***{***droid3.power***}***')

Fusion droid:

C3PO-R2D2 with power 136

\_\_STR\_\_

Uno de los métodos mágicos más utilizados es \_\_str\_\_que permite establecer la forma en la que un objeto está representado como cadena de texto :

**class** **Droid**:

**def** \_\_init\_\_(self, name, serial\_number):

self.serial\_number = serial\_number

self.name = name

**def** \_\_str\_\_(self):

**return** f'🤖 Droid "***{***self.name***}***" serial-no ***{***self.serial\_number***}***'

droid = Droid('K-2SO', 8403898409432)

print(droid) *# llamada a droid.\_\_str\_\_()*

🤖 Droid "K-2SO" serial-no 8403898409432

str(droid)

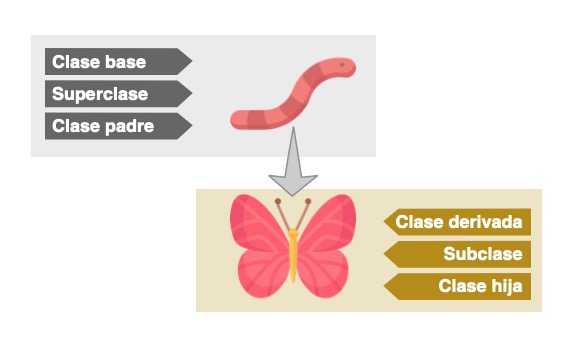
'🤖 Droid "K-2SO" serial-no 8403898409432'

f'Droid -> ***{***droid***}***'

'Droid -> 🤖 Droid "K-2SO" serial-no 8403898409432'

Herencia

La herencia consiste en crear una nueva clase partiendo de una clase existente, pero que añade o modifica ciertos aspectos. Se considera una buena práctica tanto para reutilizar código como para realizar generalizaciones.



Cuando se utiliza herencia, la clase derivada, de forma automática, puede usar todo el código de la clase base sin necesidad de copiar nada limpio.

Heredar desde una clase base

Para que una clase «herede» de otra, basta con indicar la clase base entre paréntesis en la definición de la clase derivada.

Sigamos con el ejemplo. Una de las grandes categorías de droides en StarWars es la de droides de protocolo. Vamos a crear una herencia sobre esta idea:

**lass** **Droid**:

*''' Clase Base '''*

**pass**

**class** **ProtocolDroid**(Droid):

*''' Clase Derivada '''*

**pass**

issubclass(ProtocolDroid, Droid) *# comprobación de herencia*

True

r2d2 = Droid()

c3po = ProtocolDroid()

Vamos a añadir un par de métodos a la clase base, y analizar su comportamiento:

**class** **Droid**:

**def** switch\_on(self):

print("Hi! I'm a droid. Can I help you?")

**def** switch\_off(self):

print("Bye! I'm going to sleep")

**class** **ProtocolDroid**(Droid):

**pass**

r2d2 = Droid()

c3po = ProtocolDroid()

r2d2.switch\_on()

Hi! I'm a droid. Can I help you?

c3po.switch\_on() *# método heredado de Droid*

Hi! I'm a droid. Can I help you?

r2d2.switch\_off()

Bye! I'm going to sleep

Sobre escribir un metodo

Como hemos visto, una clase derivada hereda todo lo que tiene su clase base. Pero en muchas ocasiones nos interesa modificar el comportamiento de esta herencia.

En el ejemplo vamos a modificar el comportamiento del método switch\_on() para la clase derivada:

**class** **Droid**:

**def** switch\_on(self):

print("Hi! I'm a droid. Can I help you?")

**def** switch\_off(self):

print("Bye! I'm going to sleep")

**class** **ProtocolDroid**(Droid):

**def** switch\_on(self):

print("Hi! I'm a PROTOCOL droid. Can I help you?")

r2d2 = Droid()

c3po = ProtocolDroid()

r2d2.switch\_on()

Hi! I'm a droid. Can I help you?

c3po.switch\_on() *# método heredado pero sobreescrito*

Hi! I'm a PROTOCOL droid. Can I help you?

Añadir un método

La clase derivada también puede agregar métodos que no estaban presentes en su clase base. En el siguiente ejemplo vamos a añadir un método translate() que permita a los droides de protocolo traducir cualquier mensaje:

**class** **Droid**:

**def** switch\_on(self):

print("Hi! I'm a droid. Can I help you?")

**def** switch\_off(self):

print("Bye! I'm going to sleep")

**class** **ProtocolDroid**(Droid):

**def** switch\_on(self):

print("Hi! I'm a PROTOCOL droid. Can I help you?")

**def** translate(self, msg, from\_language):

*''' Translate from language to Human understanding '''*

print(f'***{***msg***}*** means "ZASCA" in ***{***from\_language***}***')

r2d2 = Droid()

c3po = ProtocolDroid()

c3po.translate('kiitos', 'Huttese') *# idioma de Watoo*

kiitos means "ZASCA" in Huttese

r2d2.translate('kiitos', 'Huttese') *# droide genérico no puede traducir*

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Droid' object has no attribute 'translate'

Accediendo a la clase base

Puede darse cuenta de la situación en la que tengamos que acceder desde la clase derivada a métodos o atributos de la clase base. Python ofrece super() como mecanismo para ello.

Veamos un ejemplo más elaborado con nuestros droides:

**class** **Droid**:

**def** \_\_init\_\_(self, name):

self.name = name

**class** **ProtocolDroid**(Droid):

**def** \_\_init\_\_(self, name, languages):

super().\_\_init\_\_(name) *# llamada al constructor de la clase base*

self.languages = languages

droid = ProtocolDroid('C-3PO', ['Ewokese', 'Huttese', 'Jawaese'])

droid.name *# fijado en el constructor de la clase base*

'C-3PO'

droid.languages *# fijado en el constructor de la clase derivada*

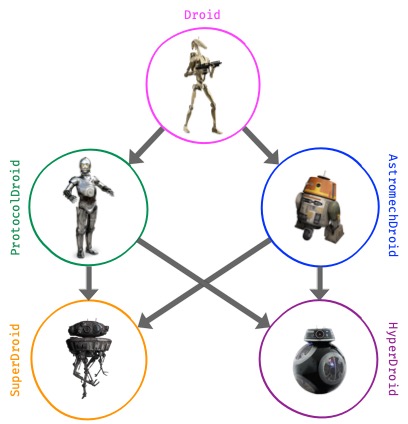
['Ewokese', 'Huttese', 'Jawaese']

Herencia múltiple

Aunque no está disponible en todos los lenguajes de programación, Python sí permite que los objetos puedan heredar de múltiples clases base.

Si en una clase se hace referencia a un método o atributo que no existe, Python lo buscará en todas sus clases base. Es posible que exista una colisión en caso de que el método o el atributo buscado esté, a la vez, en varias clases base. En este caso, Python resuelve el conflicto a través del orden de resolución de métodos.

Supongamos que queremos modelar la siguiente estructura de clases con herencia múltiple:



**class** **Droid**:

**def** greet(self):

**return** 'Here a droid'

**class** **ProtocolDroid**(Droid):

**def** greet(self):

**return** 'Here a protocol droid'

**class** **AstromechDroid**(Droid):

**def** greet(self):

**return** 'Here an astromech droid'

**class** **SuperDroid**(ProtocolDroid, AstromechDroid):

**pass**

**class** **HyperDroid**(AstromechDroid, ProtocolDroid):

**pass**

Todas las clases en Python disponen de un método especial llamado mro() que devuelve una lista de las clases que se visitarían en caso de acceder a un método o un atributo. También existe el atributo \_\_mro\_\_ como una tupla de esas clases:

SuperDroid.mro()

[\_\_main\_\_.SuperDroid,

\_\_main\_\_.ProtocolDroid,

\_\_main\_\_.AstromechDroid,

\_\_main\_\_.Droid,

object]

HyperDroid.\_\_mro\_\_

(\_\_main\_\_.HyperDroid,

\_\_main\_\_.AstromechDroid,

\_\_main\_\_.ProtocolDroid,

\_\_main\_\_.Droid,

object)

Veamos el resultado de la llamada a los métodos definidos:

super\_droid = SuperDroid()

hyper\_droid = HyperDroid()

super\_droid.greet()

'Here a protocol droid'

hyper\_droid.greet()

'Here an astromech droid'

Todos los objetos en Python heredan, en primera instancia, de object. Esto se puede comprobar con el mro()correspondiente:

int.mro()

[int, object]

str.mro()

[str, object]

float.mro()

[float, object]

tuple.mro()

[tuple, object]

list.mro()

[list, object]

bool.mro()

[bool, int, object]

mezclas

Hay situaciones en la que nos interesa incorporar una clase base «independiente» de la jerarquía establecida, y sólo a efectos de tareas auxiliares . Esta aproximación podría ayudar a evitar colisiones en métodos o atributos reducir la ambigüedad que añade la herencia múltiple. Estas clases auxiliares reciben el nombre de «mixins» .

Veamos un ejemplo en el que usamos un «mixin» para mostrar las variables de un objeto:

**class** **Instrospection**:

**def** dig(self):

print(vars(self)) *# vars devuelve las variables del argumento*

**class** **Droid**(Instrospection):

**pass**

droid = Droid()

droid.code = 'DN-LD'

droid.num\_feet = 2

droid.type = 'Power Droid'

droid.dig()

{'code': 'DN-LD', 'num\_feet': 2, 'type': 'Power Droid'}

Agregacion y composicion

Aunque la herencia de clases nos permite modelar una gran cantidad de casos de uso en términos de « is-a » ( es un ), existen muchas otras situaciones en las que la agregación o la composición son una mejor opción. En este caso una clase se compone de otros casos: hablamos de una relación « has-a » (tiene un).

Hay una sutil diferencia entre agregación y composición:

La composición implica que el objeto utilizado no puede «funcionar» sin la presencia de su propietario.

La agregación implica que el objeto utilizado puede funcionar por sí mismo.



Veamos un ejemplo de agregación en el que añadimos una herramienta a un droide:

**class** **Tool**:

**def** \_\_init\_\_(self, name):

self.name = name

**def** \_\_str\_\_(self):

**return** self.name.upper()

**class** **Droid**:

**def** \_\_init\_\_(self, name, serial\_number, tool):

self.name = name

self.serial\_number = serial\_number

self.tool = tool *# agregación*

**def** \_\_str\_\_(self):

**return** f'Droid ***{***self.name***}*** armed with a ***{***self.tool***}***'

lighter = Tool('lighter')

bb8 = Droid('BB-8', 48050989085439, lighter)

print(bb8)

Droid BB-8 armed with a LIGHTER

ESTRUCTURAS DE DATOS

Creando listas

Una lista está compuesta por cero o más elementos. En Python debemos escribir estos elementos separados por comas y dentro de corchetes. Veamos algunos ejemplos de listas:

mpty\_list = []

languages = ['Python', 'Ruby', 'Javascript']

fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]

data = ['Tenerife', {'cielo': 'limpio', 'temp': 24}, 3718, (28.2933947, -16.5226597)]

Una lista puede contener tipos de datos heterogéneos, lo que la hace una estructura de datos muy versátil.

Conversión

Para convertir otros tipos de datos en una lista podemos usar la función list():

*# conversión desde una cadena de texto*

list('Python')

['P', 'y', 't', 'h', 'o', 'n']

Si nos fijamos en lo que ha pasado, al convertir la cadena de texto Python se ha creado una lista con 6 elementos, donde cada uno de ellos representa un caracter de la cadena. Podemos extender este comportamiento a cualquier otro tipo de datos que permita ser iterado (iterables).

Lista vacía

Existe una manera particular de usar list() y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en una lista, con lo que obtendremos una lista vacía:

Operaciones con listas

Obtener un elemento

Igual que en el caso de las cadenas de texto, podemos obtener un elemento de una lista a través del índice (lugar) que ocupa. Veamos un ejemplo:

shopping = ['Agua', 'Huevos', 'Aceite']

shopping[0]

'Agua'

shopping[1]

'Huevos'

shopping[2]

'Aceite'

shopping[-1] *# acceso con índice negativo*

'Aceite'

El índice que usemos para acceder a los elementos de una lista tiene que estar comprendido entre los límites de la misma. Si usamos un índice antes del comienzo o después del final obtendremos un error (excepción):

Trocear una lista

El troceado de listas funciona de manera totalmente análoga al troceado de cadenas. Veamos algunos ejemplos:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping[0:3]

['Agua', 'Huevos', 'Aceite']

shopping[:3]

['Agua', 'Huevos', 'Aceite']

shopping[2:4]

['Aceite', 'Sal']

shopping[-1:-4:-1]

['Limón', 'Sal', 'Aceite']

*# Equivale a invertir la lista*

shopping[::-1]

['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']

En el troceado de listas, a diferencia de lo que ocurre al obtener elementos, no debemos preocuparnos por acceder a índices inválidos (fuera de rango) ya que Python los restringirá a los límites de la lista:

shopping

['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping[10:]

[]

shopping[-100:2]

['Agua', 'Huevos']

shopping[2:100]

['Aceite', 'Sal', 'Limón']

Invertir una lista

Python nos ofrece, al menos, tres mecanismos para invertir los elementos de una lista:

Conservando la lista original:

Mediante troceado de listas con step negativo:

shopping

['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping[::-1]

['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']

Conservando la lista original:

Mediante la función reversed():

shopping

['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

list(reversed(shopping))

['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']

Modificando la lista original:

Utilizando la función reverse() (nótese que es sin «d» al final):

shopping

['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.reverse()

shopping

['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']

Añadir al final de la lista

Una de las operaciones más utilizadas en listas es añadir elementos al final de las mismas. Para ello Python nos ofrece la función append(). Se trata de un método destructivo que modifica la lista original:

shopping = ['Agua', 'Huevos', 'Aceite']

shopping.append('Atún')

shopping

['Agua', 'Huevos', 'Aceite', 'Atún']

CREANDO DESDE VACÍO

Una forma muy habitual de trabajar con listas es empezar con una vacía e ir añadiendo elementos poco a poco. Se podría hablar de un patrón creación.

Supongamos un ejemplo en el que queremos construir una lista con los números pares del 1 al 20:

even\_numbers = []

**for** i **in** range(20):

**if** i % 2 == 0:

even\_numbers.append(i)

even\_numbers

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

Añadir en cualquier posición de una lista

Ya hemos visto cómo añadir elementos al final de una lista. Sin embargo, Python ofrece una función insert() que vendría a ser una generalización de la anterior, para incorporar elementos en cualquier posición. Simplemente debemos especificar el índice de inserción y el elemento en cuestión. También se trata de una función destructiva:

shopping = ['Agua', 'Huevos', 'Aceite']

shopping.insert(1, 'Jamón')

shopping

['Agua', 'Jamón', 'Huevos', 'Aceite']

shopping.insert(3, 'Queso')

shopping

['Agua', 'Jamón', 'Huevos', 'Queso', 'Aceite']

El índice que especificamos en la función insert() lo podemos intepretar como la posición delante (a la izquierda) de la cual vamos a colocar el nuevo valor en la lista.

Al igual que ocurría con el troceado de listas, en este tipo de inserciones no obtendremos un error si especificamos índices fuera de los límites de la lista. Estos se ajustarán al principio o al final en función del valor que indiquemos:

shopping = ['Agua', 'Huevos', 'Aceite']

shopping.insert(100, 'Mermelada')

shopping

['Agua', 'Huevos', 'Aceite', 'Mermelada']

shopping.insert(-100, 'Arroz')

shopping

['Arroz', 'Agua', 'Huevos', 'Aceite', 'Mermelada']

Aunque es posible utilizar insert() para añadir elementos al final de una lista, siempre se recomienda usar append() por su mayor legibilidad:

values = [1, 2, 3]

values.append(4)

values

[1, 2, 3, 4]

values = [1, 2, 3]

values.insert(len(values), 4) *# don't do it!*

values

[1, 2, 3, 4]

Combinar listas

Python nos ofrece dos aproximaciones para combinar listas:

Conservando la lista original:

Mediante el operador + o +=:

shopping = ['Agua', 'Huevos', 'Aceite']

fruitshop = ['Naranja', 'Manzana', 'Piña']

shopping + fruitshop

['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']

Modificando la lista original:

Mediante la función extend():

shopping = ['Agua', 'Huevos', 'Aceite']

fruitshop = ['Naranja', 'Manzana', 'Piña']

shopping.extend(fruitshop)

shopping

['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']

Hay que tener en cuenta que extend() funciona adecuadamente si pasamos una lista como argumento. En otro caso, quizás los resultados no sean los esperados. Veamos un ejemplo:

shopping = ['Agua', 'Huevos', 'Aceite']

shopping.extend('Limón')

shopping

['Agua', 'Huevos', 'Aceite', 'L', 'i', 'm', 'ó', 'n']

El motivo es que extend() «recorre» (o itera) sobre cada uno de los elementos del objeto en cuestión. En el caso anterior, al ser una cadena de texto, está formada por caracteres. De ahí el resultado que obtenemos.

Se podría pensar en el uso de append() para combinar listas. La realidad es que no funciona exactamente como esperamos; la segunda lista se añadiría como una sublista de la principal:

shopping = ['Agua', 'Huevos', 'Aceite']

fruitshop = ['Naranja', 'Manzana', 'Piña']

shopping.append(fruitshop)

shopping

['Agua', 'Huevos', 'Aceite', ['Naranja', 'Manzana', 'Piña']]

Modificar una lista

Del mismo modo que se accede a un elemento utilizando su índice, también podemos modificarlo:

shopping = ['Agua', 'Huevos', 'Aceite']

shopping[0]

'Agua'

shopping[0] = 'Jugo'

shopping

['Jugo', 'Huevos', 'Aceite']

MODIFICAR CON TROCEADO

No sólo es posible modificar un elemento de cada vez, sino que podemos asignar valores a trozos de una lista:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping[1:4]

['Huevos', 'Aceite', 'Sal']

shopping[1:4] = ['Atún', 'Pasta']

shopping

['Agua', 'Atún', 'Pasta', 'Limón']

La lista que asignamos no necesariamente debe tener la misma longitud que el trozo que sustituimos.

Borrar elementos

Python nos ofrece, al menos, cuatro formas para borrar elementos en una lista:

Por su índice:

Mediante la función del():

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

**del**(shopping[3])

shopping

['Agua', 'Huevos', 'Aceite', 'Limón']

Por su valor:

Mediante la función remove():

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.remove('Sal')

shopping

['Agua', 'Huevos', 'Aceite', 'Limón']

Por su índice (con extracción):

Las dos funciones anteriores del() y remove() efectivamente borran el elemento indicado de la lista, pero no «devuelven» 3 nada. Sin embargo, Python nos ofrece la función pop() que además de borrar, nos «recupera» el elemento; algo así como una extracción. Lo podemos ver como una combinación de acceso + borrado:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.pop()

'Limón'

shopping

['Agua', 'Huevos', 'Aceite', 'Sal']

shopping.pop(2)

'Aceite'

shopping

['Agua', 'Huevos', 'Sal']

Si usamos la función pop() sin pasarle ningún argumento, por defecto usará el índice -1, es decir, el último elemento de la lista. Pero también podemos indicarle el índice del elemento a extraer.

Por su rango:

Mediante troceado de listas:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping[1:4] = []

shopping

['Agua', 'Limón']

BORRADO COMPLETO DE LA LISTA

Python nos ofrece, al menos, dos formas para borrar una lista por completo:

1 Utilizando la función clear():

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.clear() # Borrado in-situ

shopping

[]

2 «Reinicializando» la lista a vacío con []:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping = [] *# Nueva zona de memoria*

shopping

[]

La diferencia entre ambos métodos tiene que ver con cuestiones internas de gestión de memoria y de rendimiento.

Encontrar un elemento

Si queremos descubrir el índice que corresponde a un determinado valor dentro la lista podemos usar la función index() para ello:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.index('Huevos')

Si buscamos un valor que existe más de una vez en una lista, la función index() sólo nos devolverá el índice de la primera ocurrencia.

Pertenencia de un elemento

Si queremos comprobar la existencia de un determinado elemento en una lista, podríamos buscar su índice, pero la forma pitónica de hacerlo es utilizar el operador in:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

'Aceite' **in** shopping

True

'Pollo' **in** shopping

False

Número de ocurrencias

Para contar cuántas veces aparece un determinado valor dentro de una lista podemos usar la función count():

sheldon\_greeting = ['Penny', 'Penny', 'Penny']

sheldon\_greeting.count('Howard')

0

sheldon\_greeting.count('Penny')

3

Convertir lista a cadena de texto

Dada una lista, podemos convetirla a una cadena de texto, uniendo todos sus elementos mediante algún separador. Para ello hacemos uso de la función join() con la siguiente estructura:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

','.join(shopping)

'Agua,Huevos,Aceite,Sal,Limón'

' '.join(shopping)

'Agua Huevos Aceite Sal Limón'

'|'.join(shopping)

'Agua|Huevos|Aceite|Sal|Limón'

Esta función join() es realmente la opuesta a la de split() para dividir una cadena.

Ordenar una lista

Python proporciona, al menos, dos formas de ordenar los elementos de una lista:

Conservando lista original:

Mediante la función sorted() que devuelve una nueva lista ordenada:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

sorted(shopping)

['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']

Modificando la lista original:

Mediante la función sort():

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.sort()

shopping

['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']

Ambos métodos admiten un parámetro «booleano» reverse para indicar si queremos que la ordenación se haga en sentido inverso:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

sorted(shopping, reverse=**True**)

['Sal', 'Limón', 'Huevos', 'Agua', 'Aceite']

Longitud de una lista

Podemos conocer el número de elementos que tiene una lista con la función len():

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

len(shopping)

5

Iterar sobre una lista

Al igual que hemos visto con las cadenas de texto, también podemos iterar sobre los elementos de una lista utilizando la sentencia for:

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

**for** product **in** shopping:

print(product)

Agua

Huevos

Aceite

Sal

Limón

ITERAR USANDO ENUMERACIÓN

Hay veces que no sólo nos interesa «visitar» cada uno de los elementos de una lista, sino que también queremos saber su índice dentro de la misma. Para ello Python nos ofrece la función enumerate():

shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

**for** i, product **in** enumerate(shopping):

print(i, product)

0 Agua

1 Huevos

2 Aceite

3 Sal

4 Limón

ITERAR SOBRE MÚLTIPLES LISTAS

Python ofrece la posibilidad de iterar sobre múltiples listas en paralelo utilizando la función zip():

shopping = ['Agua', 'Aceite', 'Arroz']

details = ['mineral natural', 'de oliva virgen', 'basmati']

**for** product, detail **in** zip(shopping, details):

print(product, detail)

Agua mineral natural

Aceite de oliva virgen

Arroz basmati

En el caso de que las listas no tengan la misma longitud, la función zip() realiza la combinación hasta que se agota la lista más corta.

Dado que zip() produce un iterador, si queremos obtener una lista explícita con la combinación en paralelo de las listas, debemos construir dicha lista de la siguiente manera:

shopping = ['Agua', 'Aceite', 'Arroz']

details = ['mineral natural', 'de oliva virgen', 'basmati']

list(zip(shopping, details))

[('Agua', 'mineral natural'),

('Aceite', 'de oliva virgen'),

('Arroz', 'basmati')]

Cuidado con las copias

Las listas son estructuras de datos mutables y esta característica nos obliga a tener cuidado cuando realizamos copias de listas, ya que la modificación de una de ellas puede afectar a la otra.

Veamos un ejemplo sencillo:

original\_list = [4, 3, 7, 1]

copy\_list = original\_list

original\_list[0] = 15

original\_list

[15, 3, 7, 1]

copy\_list

[15, 3, 7, 1]

A través de Python Tutor se puede ver claramente el motivo de por qué ocurre esto. Dado que las variables «apuntan» a la misma zona de memoria, al modificar una de ellas, el cambio también se ve reflejado en la otra.

Una posible solución a este problema es hacer una «copia dura». Para ello Python proporciona la función copy():

original\_list = [4, 3, 7, 1]

copy\_list = original\_list.copy()

original\_list[0] = 15

original\_list

[15, 3, 7, 1]

copy\_list

[4, 3, 7, 1]

Veracidad múltiple

Si bien podemos usar sentencias condicionales para comprobar la veracidad de determinadas expresiones, Python nos ofrece dos funciones «built-in» con las que podemos evaluar si se cumplen todas las condiciones all() o si se cumple alguna condición any(). Estas funciones trabajan sobre iterables, y el caso más evidente es una lista.

Supongamos un ejemplo en el que queremos comprobar si una determinada palabra cumple las siguientes condiciones:

Su longitud total es mayor que 4.

Empieza por «p».

Contiene, al menos, una «y».

Veamos la versión clásica:

word = 'python'

**if** len(word) > 4 **and** word.startswith('p') **and** word.count('y') >= 1:

print('Cool word!')

**else**:

print('No thanks')

Cool word!

Veamos la versión con veracidad múltiple usando all(), donde se comprueba que se cumplan todas las expresiones:

word = 'python'

enough\_length = len(word) > 4 *# True*

right\_beginning = word.startswith('p') *# True*

min\_ys = word.count('y') >= 1 *# True*

is\_cool\_word = all([enough\_length, right\_beginning, min\_ys])

**if** is\_cool\_word:

print('Cool word!')

**else**:

print('No thanks')

Cool word!

Veamos la versión con veracidad múltiple usando any(), donde se comprueba que se cumpla alguna expresión:

word = 'yeah'

enough\_length = len(word) > 4 *# False*

right\_beginning = word.startswith('p') *# False*

min\_ys = word.count('y') >= 1 *# True*

is\_fine\_word = any([enough\_length, right\_beginning, min\_ys])

**if** is\_fine\_word:

print('Fine word!')

**else**:

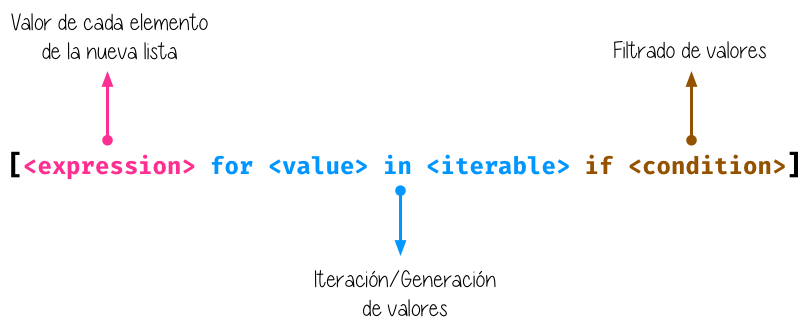
print('No thanks')

Fine word!

Este enfoque puede ser interesante cuando se manejan muchas condiciones o bien cuando queremos separar las condiciones y agruparlas en una única lista.

Listas por comprensión

Las listas por comprensión establecen una técnica para crear listas de forma más compacta basándose en el concepto matemático de conjuntos definidos por comprensión.



En primer lugar veamos un ejemplo en el que convertimos una cadena de texto con valores numéricos en una lista con los mismos valores pero convertidos a enteros. En su versión clásica haríamos algo tal que así:

alues = '32,45,11,87,20,48'

int\_values = []

**for** value **in** values.split(','):

int\_value = int(value)

int\_values.append(int\_value)

int\_values

[32, 45, 11, 87, 20, 48]

Ahora veamos el código utilizando una lista por comprensión:

values = '32,45,11,87,20,48'

int\_values = [int(value) **for** value **in** values.split(',')]

int\_values

[32, 45, 11, 87, 20, 48]

Condiciones en comprensiones

También existe la posibilidad de incluir condiciones en las listas por comprensión.

Continuando con el ejemplo anterior, supongamos que sólo queremos crear la lista con aquellos valores que empiecen por el dígito 4:

values = '32,45,11,87,20,48'

int\_values = [int(v) **for** v **in** values.split(',') **if** v.startswith('4')]

int\_values

[45, 48]

Anidamiento en comprensiones

Nivel avanzado

En la iteración que usamos dentro de la lista por comprensión es posible usar bucles anidados.

Veamos un ejemplo en el que generamos todas las combinaciones de una serie de valores:

values = '32,45,11,87,20,48'

svalues = values.split(',')

combinations = [f'***{***v1***}***x***{***v2***}***' **for** v1 **in** svalues **for** v2 **in** svalues]

combinations

['32x32',

'32x45',

'32x11',

'32x87',

'32x20',

'32x48',

'45x32',

'45x45',

'48x45',

'48x11',

'48x87',

'48x20',

'48x48']

Funciones matemáticas

Python nos ofrece, entre otras 4, estas tres funciones matemáticas básicas que se pueden aplicar sobre listas.

Suma de todos los valores:

Mediante la función sum():

data = [5, 3, 2, 8, 9, 1]

sum(data)

28

Mínimo de todos los valores:

Mediante la función min():

data = [5, 3, 2, 8, 9, 1]

min(data)

1

Máximo de todos los valores:

Mediante la función max():

data = [5, 3, 2, 8, 9, 1]

max(data)

9

Listas de listas

Nivel intermedio

Como ya hemos visto en varias ocasiones, las listas son estructuras de datos que pueden contener elementos heterogéneos. Estos elementos pueden ser a su vez listas.

A continuación, planteamos un ejemplo del mundo deportivo. Un equipo de fútbol suele tener una disposición en el campo organizada en líneas de jugadores. En aquella alineación con la que España ganó la copa del mundo en 2010 había una disposición 4-3-3 con los siguientes jugadores:

Veamos una posible representación de este equipo de fútbol usando una lista compuesta de listas. Primero definimos cada una de las líneas:

goalkeeper = 'Casillas'

defenders = ['Capdevila', 'Piqué', 'Puyol', 'Ramos']

midfielders = ['Xabi', 'Busquets', 'X. Alonso']

forwards = ['Iniesta', 'Villa', 'Pedro']

Y ahora las juntamos en una única lista:

team = [goalkeeper, defenders, midfielders, forwards]

team

['Casillas',

['Capdevila', 'Piqué', 'Puyol', 'Ramos'],

['Xabi', 'Busquets', 'X. Alonso'],

['Iniesta', 'Villa', 'Pedro']]

Podemos comprobar el acceso a distintos elementos:

team[0] *# portero*

'Casillas'

team[1][0] *# lateral izquierdo*

'Capdevila'

team[2] *# centrocampistas*

['Xabi', 'Busquets', 'X. Alonso']

team[3][1] *# delantero centro*

'Villa'

Tuplas

El concepto de tupla es muy similar al de lista. Aunque hay algunas diferencias menores, lo fundamental es que, mientras una lista es mutable y se puede modificar, una tupla no admite cambios y por lo tanto, es inmutable.

Creando tuplas

Podemos pensar en crear tuplas tal y como lo hacíamos con listas, pero usando paréntesis en lugar de corchetes:

empty\_tuple = ()

tenerife\_geoloc = (28.46824, -16.25462)

three\_wise\_men = ('Melchor', 'Gaspar', 'Baltasar')

Tuplas de un elemento

Hay que prestar especial atención cuando vamos a crear una tupla de un único elemento. La intención primera sería hacerlo de la siguiente manera:

one\_item\_tuple = ('Papá Noel')

one\_item\_tuple

'Papá Noel'

type(one\_item\_tuple)

str

Realmente, hemos creado una variable de tipo str (cadena de texto). Para crear una tupla de un elemento debemos añadir una coma al final:

one\_item\_tuple = ('Papá Noel',)

one\_item\_tuple

('Papá Noel',)

type(one\_item\_tuple)

tuple

Tuplas sin paréntesis

Según el caso, hay veces que nos podemos encontrar con tuplas que no llevan paréntesis. Quizás no está tan extendido, pero a efectos prácticos tiene el mismo resultado. Veamos algunos ejemplos de ello:

one\_item\_tuple = 'Papá Noel',

three\_wise\_men = 'Melchor', 'Gaspar', 'Baltasar'

tenerife\_geoloc = 28.46824, -16.25462

Modificar una tupla

Como ya hemos comentado previamente, las tuplas con estructuras de datos inmutables. Una vez que las creamos con un valor, no podemos modificarlas. Veamos qué ocurre si lo intentamos:

three\_wise\_men = 'Melchor', 'Gaspar', 'Baltasar'

three\_wise\_men[0] = 'Tom Hanks'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

Conversión

Para convertir otros tipos de datos en una tupla podemos usar la función tuple():

shopping = ['Agua', 'Aceite', 'Arroz']

tuple(shopping)

('Agua', 'Aceite', 'Arroz')

Esta conversión es válida para aquellos tipos de datos que sean iterables: cadenas de caracteres, listas, diccionarios, conjuntos, etc. Un ejemplo que no funciona es intentar convertir un número en una tupla:

tuple(5)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'int' object is not iterable

El uso de la función tuple() sin argumentos equivale a crear una tupla vacía:

tuple()

()

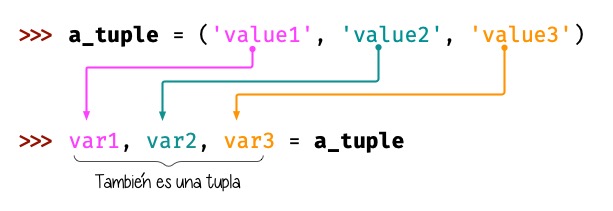
Para crear una tupla vacía, se suele recomendar el uso de () frente a tuple(), no sólo por ser más pitónico sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

Operaciones con tuplas

Con las tuplas podemos realizar todas las operaciones que vimos con listas salvo las que conlleven una modificación «in-situ» de la misma:

* reverse()
* append()
* extend()
* remove()
* clear()
* sort()

Desempaquetado de tuplas

El desempaquetado es una característica de las tuplas que nos permite asignar una tupla a variables independientes:

Veamos un ejemplo con código:

three\_wise\_men = ('Melchor', 'Gaspar', 'Baltasar')

king1, king2, king3 = three\_wise\_men

king1

'Melchor'

king2

'Gaspar'

king3

'Baltasar'

Python proporciona la función «built-in» divmod() que devuelve el cociente y el resto de una división usando una única llamada. Lo interesante (para el caso que nos ocupa) es que se suele utilizar el desempaquetado de tuplas para obtener los valores:

quotient, remainder = divmod(7, 3)

quotient

2

remainder

1

Intercambio de valores

A través del desempaquetado de tuplas podemos llevar a cabo el intercambio de los valores de dos variables de manera directa:

value1 = 40

value2 = 20

value1, value2 = value2, value1

value1

20

value2

40

A priori puede parecer que esto es algo «natural», pero en la gran mayoría de lenguajes de programación no es posible hacer este intercambio de forma «directa» ya que necesitamos recurrir a una tercera variable «auxiliar» como almacén temporal en el paso intermedio de traspaso de valores.

ranking = ('G', 'A', 'R', 'Y', 'W')

**>>>** head, \*body, tail = ranking

**>>>** head

'G'

**>>>** body

['A', 'R', 'Y']

**>>>** tail

'W'

Desempaquetado genérico

El desempaquetado de tuplas es extensible a cualquier tipo de datos que sea iterable. Veamos algunos ejemplos de ello.

Sobre cadenas de texto:

oxygen = 'O2'

**>>>** first, last = oxygen

**>>>** first, last

('O', '2')

**>>>** text = 'Hello, World!'

**>>>** head, \*body, tail = text

**>>>** head, body, tail

('H', ['e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd'], '!')

Sobre listas:

writer1, writer2, writer3 = ['Virginia Woolf', 'Jane Austen', 'Mary Shelley']

**>>>** writer1, writer2, writer3

('Virginia Woolf', 'Jane Austen', 'Mary Shelley')

**>>>** text = 'Hello, World!'

**>>>** word1, word2 = text.split()

**>>>** word1, word2

('Hello,', 'World!')

¿Tuplas por comprensión?

Los tipos de datos mutables (listas, diccionarios y conjuntos) sí permiten comprensiones pero no así los tipos de datos inmutables como cadenas de texto y tuplas.

Si intentamos crear una tupla por comprensión utilizando paréntesis alrededor de la expresión, vemos que no obtenemos ningún error al ejecutarlo:

myrange = (number **for** number **in** range(1, 6))

Sin embargo no hemos conseguido una tupla por comprensión sino un generador:

myrange

<generator object <genexpr> at 0x10b3732e0>

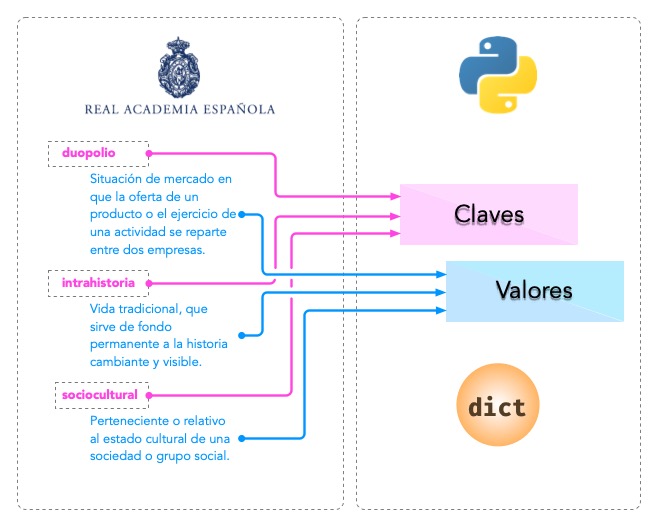
Tuplas vs Listas

Aunque puedan parecer estructuras de datos muy similares, sabemos que las tuplas carecen de ciertas operaciones, especialmente las que tienen que ver con la modificación de sus valores, ya que no son inmutables. Si las listas son más flexibles y potentes, ¿por qué íbamos a necesitar tuplas? Veamos 4 potenciales ventajas del uso de tuplas frente a las listas:

* Las tuplas ocupan menos espacio en memoria.
* En las tuplas existe protección frente a cambios indeseados.
* Las tuplas se pueden usar como claves de diccionarios (son «hashables»).
* Las namedtuples son una alternativa sencilla a los objetos.

Diccionarios

Podemos trasladar el concepto de diccionario de la vida real al de diccionario en Python. Al fin y al cabo un diccionario es un objeto que contiene palabras, y cada palabra tiene asociado un significado. Haciendo el paralelismo, diríamos que en Python un diccionario es también un objeto indexado por claves (las palabras) que tienen asociados unos valores (los significados).



Analogía de un diccionario en Python

Los diccionarios en Python tienen las siguientes características:

* Mantienen el orden en el que se insertan las claves.
* Son mutables, con lo que admiten añadir, borrar y modificar sus elementos.
* Las claves deben ser únicas. A menudo se utilizan las cadenas de texto como claves, pero en realidad podría ser cualquier tipo de datos inmutable: enteros, flotantes, tuplas (entre otros).
* Tienen un acceso muy rápido a sus elementos, debido a la forma en la que están implementados internamente.

Nota

En otros lenguajes de programación, a los diccionarios se les conoce como arrays asociativos, «hashes» o «hashmaps».

Creando diccionarios

Para crear un diccionario usamos llaves {} rodeando asignaciones clave: valor que están separadas por comas. Veamos algunos ejemplos de diccionarios:

empty\_dict = {}

**>>>** rae = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa'

**...** }

**>>>** population\_can = {

**...**  2015: 2\_135\_209,

**...**  2016: 2\_154\_924,

**...**  2017: 2\_177\_048,

**...**  2018: 2\_206\_901,

**...**  2019: 2\_220\_270

**...** }

En el código anterior podemos observar la creación de un diccionario vacío, otro donde sus claves y sus valores son cadenas de texto y otro donde las claves y los valores son valores enteros.

Conversión

Para convertir otros tipos de datos en un diccionario podemos usar la función dict():

*# Diccionario a partir de una lista de cadenas de texto*

**>>>** dict(['a1', 'b2'])

{'a': '1', 'b': '2'}

**>>>** *# Diccionario a partir de una tupla de cadenas de texto*

**>>>** dict(('a1', 'b2'))

{'a': '1', 'b': '2'}

**>>>** *# Diccionario a partir de una lista de listas*

**>>>** dict([['a', 1], ['b', 2]])

{'a': 1, 'b': 2}

Si nos fijamos bien, cualquier iterable que tenga una estructura interna de 2 elementos es susceptible de convertirse en un diccionario a través de la función dict().

Diccionario vacío

Existe una manera particular de usar dict() y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en un diccionario, con lo que obtendremos un diccionario vacío:

dict()

{}

Para crear un diccionario vacío, se suele recomendar el uso de {} frente a dict(), no sólo por ser más pitónico sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

Creación con dict()

También es posible utilizar la función dict() para crear dicionarios y no tener que utilizar llaves y comillas:

Supongamos que queremos transformar la siguiente tabla en un diccionario:

Atributo Valor

Name Guido

Surname Van Rossum

Job Python creator

Utilizando la construcción mediante dict podemos pasar clave y valor como argumentos de la función:

person = dict(

**...**  name='Guido',

**...**  surname='Van Rossum',

**...**  job='Python creator'

**...** )

**>>>** person

{'name': 'Guido', 'surname': 'Van Rossum', 'job': 'Python creator'}

El inconveniente que tiene esta aproximación es que las claves deben ser identificadores válidos en Python. Por ejemplo, no se permiten espacios:

person = dict(

**...**  name='Guido van Rossum',

**...**  date of birth='31/01/1956'

File "<stdin>", line 3

date of birth='31/01/1956'

^

SyntaxError: invalid syntax

Es posible crear un diccionario especificando sus claves y un único valor de «relleno»:

dict.fromkeys('aeiou', 0)

{'a': 0, 'e': 0, 'i': 0, 'o': 0, 'u': 0}

Es válido pasar cualquier «iterable» como referencia a las claves.

Operaciones con diccionarios

Obtener un elemento

Para obtener un elemento de un diccionario basta con escribir la clave entre corchetes. Veamos un ejemplo:

rae = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa'

**...** }

**>>>** rae['anarcoide']

'Que tiende al desorden'

Si intentamos acceder a una clave que no existe, obtendremos un error:

rae['acceso']

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'acceso'

USANDO GET()

Existe una función muy útil para «superar» los posibles errores de acceso por claves inexistentes. Se trata de get() y su comportamiento es el siguiente:

1 Si la clave que buscamos existe, nos devuelve su valor.

2 Si la clave que buscamos no existe, nos devuelve None salvo que le indiquemos otro valor por defecto, pero en ninguno de los dos casos obtendremos un error.

rae

{'bifronte': 'De dos frentes o dos caras',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa'}

# Equivalente a rae['bifronte'].

**>>>** rae.get('bifronte')

# La clave buscada no existe y obtenemos None.

'De dos frentes o dos caras'

**>>>** rae.get('programación')

# La clave buscada no existe y nos devuelve el valor que hemos aportado por defecto.

**>>>** rae.get('programación', 'No disponible')

'No disponible'

Añadir o modificar un elemento

Para añadir un elemento a un diccionario sólo es necesario hacer referencia a la clave y asignarle un valor:

Si la clave ya existía en el diccionario, se reemplaza el valor existente por el nuevo.

Si la clave es nueva, se añade al diccionario con su valor. No vamos a obtener un error a diferencia de las listas.

Partimos del siguiente diccionario para ejemplificar estas acciones:

rae = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa'

**...** }

Vamos a añadir la palabra enjuiciar a nuestro diccionario de la Real Academia de La Lengua:

rae['enjuiciar'] = 'Someter una cuestión a examen, discusión y juicio'

**>>>** rae

{'bifronte': 'De dos frentes o dos caras',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa',

'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'}

Supongamos ahora que queremos modificar el significado de la palabra enjuiciar por otra acepción:

rae['enjuiciar'] = 'Instruir, juzgar o sentenciar una causa'

**>>>** rae

{'bifronte': 'De dos frentes o dos caras',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa',

'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}

CREANDO DESDE VACÍO

Una forma muy habitual de trabajar con diccionarios es utilizar el patrón creación partiendo de uno vacío e ir añadiendo elementos poco a poco.

Supongamos un ejemplo en el que queremos construir un diccionario donde las claves son las letras vocales y los valores son sus posiciones:

VOWELS = 'aeiou'

**>>>** enum\_vowels = {}

**>>> for** i, vowel **in** enumerate(VOWELS):

**...**  enum\_vowels[vowel] = i + 1

**...**

**>>>** enum\_vowels

{'a': 1, 'e': 2, 'i': 3, 'o': 4, 'u': 5}

Hemos utilizando la función enumerate() que ya vimos para las listas en el apartado: Iterar usando enumeración.

Pertenencia de una clave

La forma pitónica de comprobar la existencia de una clave dentro de un diccionario, es utilizar el operador in:

'bifronte' **in** rae

True

**>>>** 'almohada' **in** rae

False

**>>>** 'montuvio' **not** **in** rae

False

El operador in siempre devuelve un valor booleano, es decir, verdadero o falso.

Obtener todos los elementos

Python ofrece mecanismos para obtener todos los elementos de un diccionario. Partimos del siguiente diccionario:

rae

{'bifronte': 'De dos frentes o dos caras',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa',

'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}

Obtener todas las claves de un diccionario:

Mediante la función keys():

rae.keys()

dict\_keys(['bifronte', 'anarcoide', 'montuvio', 'enjuiciar'])

Obtener todos los valores de un diccionario:

Mediante la función values():

rae.values()

dict\_values([

'De dos frentes o dos caras',

'Que tiende al desorden',

'Campesino de la costa',

'Instruir, juzgar o sentenciar una causa'

])

Obtener todos los pares «clave-valor» de un diccionario:

Mediante la función items():

rae.items()

dict\_items([

('bifronte', 'De dos frentes o dos caras'),

('anarcoide', 'Que tiende al desorden'),

('montuvio', 'Campesino de la costa'),

('enjuiciar', 'Instruir, juzgar o sentenciar una causa')

])

Para este último caso cabe destacar que los «items» se devuelven como una lista de tuplas, donde cada tupla tiene dos elementos: el primero representa la clave y el segundo representa el valor.

Longitud de un diccionario

Podemos conocer el número de elementos («clave-valor») que tiene un diccionario con la función len():

rae

{'bifronte': 'De dos frentes o dos caras',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa',

'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}

**>>>** len(rae)

4

Iterar sobre un diccionario

En base a los elementos que podemos obtener, Python nos proporciona tres maneras de iterar sobre un diccionario.

Iterar sobre claves:

**for** word **in** rae.keys():

**...**  print(word)

**...**

bifronte

anarcoide

montuvio

enjuiciar

Iterar sobre valores:

**for** meaning **in** rae.values():

**...**  print(meaning)

**...**

De dos frentes o dos caras

Que tiende al desorden

Campesino de la costa

Instruir, juzgar o sentenciar una causa

Iterar sobre «clave-valor»:

**for** word, meaning **in** rae.items():

**...**  print(f'***{***word***}***: ***{***meaning***}***')

**...**

bifronte: De dos frentes o dos caras

anarcoide: Que tiende al desorden

montuvio: Campesino de la costa

enjuiciar: Instruir, juzgar o sentenciar una causa

En este último caso, recuerde el uso de los «f-strings» para formatear cadenas de texto.

Combinar diccionarios

Dados dos (o más) diccionarios, es posible «mezclarlos» para obtener una combinación de los mismos. Esta combinación se basa en dos premisas:

1 Si la clave no existe, se añade con su valor.

2 Si la clave ya existe, se añade con el valor del «último» diccionario en la mezcla.

Python ofrece dos mecanismos para realizar esta combinación. Vamos a partir de los siguientes diccionarios para ejemplificar su uso:

rae1 = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'

**...** }

**>>>** rae2 = {

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa',

**...**  'enjuiciar': 'Instruir, juzgar o sentenciar una causa'

**...** }

Sin modificar los diccionarios originales:

Mediante el operador \*\*:

{\*\*rae1, \*\*rae2}

{'bifronte': 'De dos frentes o dos caras',

'enjuiciar': 'Instruir, juzgar o sentenciar una causa',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa'}

A partir de Python 3.9 podemos utilizar el operador | para combinar dos diccionarios:

rae1 | rae2

{'bifronte': 'De dos frentes o dos caras',

'enjuiciar': 'Instruir, juzgar o sentenciar una causa',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa'}

Modificando los diccionarios originales:

Mediante la función update():

rae1.update(rae2)

**>>>** rae1

{'bifronte': 'De dos frentes o dos caras',

'enjuiciar': 'Instruir, juzgar o sentenciar una causa',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa'}

Tener en cuenta que el orden en el que especificamos los diccionarios a la hora de su combinación (mezcla) es relevante en el resultado final. En este caso el orden de los factores sí altera el producto.

Borrar elementos

Python nos ofrece, al menos, tres formas para borrar elementos en un diccionario:

Por su clave:

Mediante la sentencia del:

rae = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa'

**...** }

**>>> del**(rae['bifronte'])

**>>>** rae

{'anarcoide': 'Que tiende al desorden', 'montuvio': 'Campesino de la costa'}

Por su clave (con extracción):

Mediante la función pop() podemos extraer un elemento del diccionario por su clave. Vendría a ser una combinación de get() + del:

rae = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa'

**...** }

**>>>** rae.pop('anarcoide')

'Que tiende al desorden'

**>>>** rae

{'bifronte': 'De dos frentes o dos caras', 'montuvio': 'Campesino de la costa'}

**>>>** rae.pop('bucle')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'bucle'

Borrado completo del diccionario:

La diferencia entre ambos métodos tiene que ver con cuestiones internas de gestión de memoria y de rendimiento.

Utilizando la función clear():

rae = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa'

**...** }

**>>>** rae.clear()

**>>>** rae

{}

«Reinicializando» el diccionario a vacío con {}:

rae = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa'

**...** }

**>>>** rae = {}

**>>>** rae

{}

Cuidado con las copias

Al igual que ocurría con las listas, si hacemos un cambio en un diccionario, se verá reflejado en todas las variables que hagan referencia al mismo. Esto se deriva de su propiedad de ser mutable. Veamos un ejemplo concreto:

original\_rae = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa'

**...** }

**>>>** copy\_rae = original\_rae

**>>>** original\_rae['bifronte'] = 'bla bla bla'

**>>>** original\_rae

{'bifronte': 'bla bla bla',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa'}

**>>>** copy\_rae

{'bifronte': 'bla bla bla',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa'}

Una posible solución a este problema es hacer una «copia dura». Para ello Python proporciona la función copy():

original\_rae = {

**...**  'bifronte': 'De dos frentes o dos caras',

**...**  'anarcoide': 'Que tiende al desorden',

**...**  'montuvio': 'Campesino de la costa'

**...** }

**>>>** copy\_rae = original\_rae.copy()

**>>>** original\_rae['bifronte'] = 'bla bla bla'

**>>>** original\_rae

{'bifronte': 'bla bla bla',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa'}

**>>>** copy\_rae

{'bifronte': 'De dos frentes o dos caras',

'anarcoide': 'Que tiende al desorden',

'montuvio': 'Campesino de la costa'}

En el caso de que estemos trabajando con diccionarios que contienen elementos mutables, debemos hacer uso de la función deepcopy() dentro del módulo copy de la librería estándar.

Diccionarios por comprensión

De forma análoga a cómo se escriben las listas por comprensión, podemos aplicar este método a los diccionarios usando llaves { }.

Veamos un ejemplo en el que creamos un diccionario por comprensión donde las claves son palabras y los valores son sus longitudes:

words = ('sun', 'space', 'rocket', 'earth')

**>>>** words\_length = {word: len(word) **for** word **in** words}

**>>>** words\_length

{'sun': 3, 'space': 5, 'rocket': 6, 'earth': 5}

También podemos aplicar condiciones a estas comprensiones. Continuando con el ejemplo anterior, podemos incorporar la restricción de sólo incluir palabras que no empiecen por vocal:

words = ('sun', 'space', 'rocket', 'earth')

**>>>** words\_length = {w: len(w) **for** w **in** words **if** w[0] **not** **in** 'aeiou'}

**>>>** words\_length

{'sun': 3, 'space': 5, 'rocket': 6}

Se puede consultar el PEP-274 para ver más ejemplos sobre diccionarios por comprensión.

Objetos «hashables»

Nivel avanzado

La única restricción que deben cumplir las claves de un diccionario es ser «hashables» 7. Un objeto es «hashable» si se le puede asignar un valor «hash» que no cambia en ejecución durante toda su vida.

Para encontrar el «hash» de un objeto, Python usa la función hash(), que devuelve un número entero y es utilizado para indexar la tabla «hash» que se mantiene internamente:

hash(999)

999

**>>>** hash(3.14)

322818021289917443

**>>>** hash('hello')

-8103770210014465245

**>>>** hash(('a', 'b', 'c'))

-2157188727417140402

Para que un objeto sea «hashable», debe ser inmutable:

hash(['a', 'b', 'c'])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unhashable type: 'list'

De lo anterior se deduce que las claves de los diccionarios, al tener que ser «hasheables», sólo pueden ser objetos inmutables.

La función «built-in» hash() realmente hace una llamada al método mágico \_\_hash\_\_() del objeto en cuestión:

hash('spiderman')

-8105710090476541603

**>>>** 'spiderman'.\_\_hash\_\_()

-8105710090476541603

Conjuntos

Un conjunto en Python representa una serie de valores únicos y sin orden establecido, con la única restricción de que sus elementos deben ser «hashables». Mantiene muchas similitudes con el concepto matemático de conjunto 1

Creando conjuntos

Para crear un conjunto basta con separar sus valores por comas y rodearlos de llaves {}:

lottery = {21, 10, 46, 29, 31, 94}

**>>>** lottery

{10, 21, 29, 31, 46, 94}

La excepción la tenemos a la hora de crear un conjunto vacío, ya que, siguiendo la lógica de apartados anteriores, deberíamos hacerlo a través de llaves:

wrong\_empty\_set = {}

**>>>** type(wrong\_empty\_set)

dict

Si hacemos esto, lo que obtenemos es un diccionario vacío.

La única opción que tenemos es utilizar la función set():

empty\_set = set()

**>>>** empty\_set

set()

**>>>** type(empty\_set)

set

Conversión

Para convertir otros tipos de datos en un conjunto podemos usar la función set() sobre cualquier iterable:

set('aplatanada')

{'a', 'd', 'l', 'n', 'p', 't'}

**>>>** set([1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5])

{1, 2, 3, 4, 5}

**>>>** set(('ADENINA', 'TIMINA', 'TIMINA', 'GUANINA', 'ADENINA', 'CITOSINA'))

{'ADENINA', 'CITOSINA', 'GUANINA', 'TIMINA'}

**>>>** set({'manzana': 'rojo', 'plátano': 'amarillo', 'kiwi': 'verde'})

{'kiwi', 'manzana', 'plátano'}

Importante

Como se ha visto en los ejemplos anteriores, set() se suele utilizar en muchas ocasiones como una forma de extraer los valores únicos de otros tipos de datos. En el caso de los diccionarios se extraen las claves, que, por definición, son únicas.

Nota

El hecho de que en los ejemplos anteriores los elementos de los conjuntos estén ordenados es únicamente un «detalle de implementación» en el que no se puede confiar.

Operaciones con conjuntos

Obtener un elemento

En un conjunto no existe un orden establecido para sus elementos, por lo tanto no podemos acceder a un elemento en concreto.

De este hecho se deriva igualmente que no podemos modificar un elemento existente, ya que ni siquiera tenemos acceso al mismo. Python sí nos permite añadir o borrar elementos de un conjunto.

Añadir un elemento

Para añadir un elemento a un conjunto debemos utilizar la función add(). Como ya hemos indicado, al no importar el orden dentro del conjunto, la inserción no establece a priori la posición donde se realizará.

A modo de ejemplo, vamos a partir de un conjunto que representa a los cuatro integrantes originales de The Beatles. Luego añadiremos a un nuevo componente:

*# John Lennon, Paul McCartney, George Harrison y Ringo Starr*

**>>>** beatles = set(['Lennon', 'McCartney', 'Harrison', 'Starr'])

**>>>** beatles.add('Best') *# Pete Best*

**>>>** beatles

{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}

Borrar elementos

Para borrar un elemento de un conjunto podemos utilizar la función remove(). Siguiendo con el ejemplo anterior, vamos a borrar al último «beatle» añadido:

beatles

{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}

**>>>** beatles.remove('Best')

**>>>** beatles

{'Harrison', 'Lennon', 'McCartney', 'Starr'}

Longitud de un conjunto

Podemos conocer el número de elementos que tiene un conjunto con la función len():

beatles

{'Harrison', 'Lennon', 'McCartney', 'Starr'}

**>>>** len(beatles)

4

Iterar sobre un conjunto

Tal y como hemos visto para otros tipos de datos iterables, la forma de recorrer los elementos de un conjunto es utilizar la sentencia for:

**for** beatle **in** beatles:

**...**  print(beatle)

**...**

Harrison

McCartney

Starr

Lennon

Consejo

Como en el ejemplo anterior, es muy común utilizar una variable en singular para recorrer un iterable (en plural). No es una regla fija ni sirve para todos los casos, pero sí suele ser una buena práctica.

Pertenencia de elemento

Al igual que con otros tipos de datos, Python nos ofrece el operador in para determinar si un elemento pertenece a un conjunto:

beatles

{'Harrison', 'Lennon', 'McCartney', 'Starr'}

**>>>** 'Lennon' **in** beatles

True

**>>>** 'Fari' **in** beatles

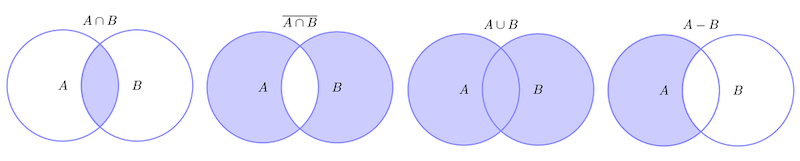
False

Teoría de conjuntos

Vamos a partir de dos conjuntos A={1,2} y B={2,3} para ejemplificar las distintas operaciones que se pueden hacer entre ellos basadas en los Diagramas de Venn y la Teoría de Conjuntos:

A = {1, 2}

**>>>** B = {2, 3}



Intersección

A∩B – Elementos que están a la vez en A y en B:

A | B

{1, 2, 3}

**>>>** A.union(B)

{1, 2, 3}

Diferencia

A−B – Elementos que están en A y no están en B:

A - B

{1}

**>>>** A.difference(B)

{1}

Diferencia simétrica

A∩B – Elementos que están en A o en B pero no en ambos conjuntos:

A ^ B

{1, 3}

**>>>** A.symmetric\_difference(B)

{1, 3}

Conjuntos por comprensión

Los conjuntos, al igual que las listas y los diccionarios, también se pueden crear por comprensión.

Veamos un ejemplo en el que construimos un conjunto por comprensión con los aquellos números enteros múltiplos de 3 en el rango [0,20):

m3 = {number **for** number **in** range(0, 20) **if** number % 3 == 0}

**>>>** m3

{0, 3, 6, 9, 12, 15, 18}

Conjuntos inmutables

Python ofrece la posibilidad de crear conjuntos inmutables haciendo uso de la función frozenset() que recibe cualquier iterable como argumento.

Supongamos que recibimos una serie de calificaciones de exámenes y queremos crear un conjunto inmutable con los posibles niveles (categorías) de calificaciones:

marks = [1, 3, 2, 3, 1, 4, 2, 4, 5, 2, 5, 5, 3, 1, 4]

**>>>** marks\_levels = frozenset(marks)

**>>>** marks\_levels

frozenset({1, 2, 3, 4, 5})

Veamos qué ocurre si intentamos modificar este conjunto:

marks\_levels.add(50)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'frozenset' object has no attribute 'add'

Nota

Los frozenset son a los sets lo que las tuplas a las listas: una forma de «congelar» los valores para que no se puedan modificar.

Ficheros

Aunque los ficheros encajarían más en un apartado de «entrada/salida» ya que representan un medio de almacenamiento persistente, también podrían ser vistos como estructuras de datos, puesto que nos permiten guardar la información y asignarles un cierto formato. 1

Un fichero es un conjunto de bytes almacenados en algún dispositivo. El sistema de ficheros es la estructura lógica que alberga los ficheros y está jerarquizado a través de directorios (o carpetas). Cada fichero se identifica unívocamente a través de una ruta que nos permite acceder a él.

Lectura de un fichero

Python ofrece la función open() para «abrir» un fichero. Esta apertura se puede realizar en 3 modos distintos:

* Lectura del contenido de un fichero existente.
* Escritura del contenido en un fichero nuevo.
* Añadido al contenido de un fichero existente.

Veamos un ejemplo para leer el contenido de un fichero en el que se encuentran las temperaturas máximas y mínimas de cada día de la última semana. El fichero está en la subcarpeta (ruta relativa) files/temps.dat y tiene el siguiente contenido:

29 23

31 23

34 26

33 23

29 22

28 22

28 22

Lo primero será abrir el fichero:

f = open('files/temps.dat')

La función open() recibe como primer argumento la ruta al fichero que queremos manejar (como un «string») y devuelve el manejador del fichero, que en este caso lo estamos asignando a una variable llamada f pero le podríamos haber puesto cualquier otro nombre.

Nota

Es importante dominar los conceptos de ruta relativa y ruta absoluta para el trabajo con ficheros. Véase este artículo de DeNovatoANovato.

Hay que tener en cuenta que la ruta al fichero que abrimos (en modo lectura) debe existir, ya que de lo contrario obtendremos un error:

f = open('foo.txt')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: 'foo.txt'

Una vez abierto el fichero ya podemos proceder a leer su contenido. Para ello Python nos ofrece la posibilidad de leer todo el fichero de una vez o bien leerlo línea a línea.

Lectura completa de un fichero

Siguiendo con nuestro ejemplo de temperaturas, veamos cómo leer todo el contenido del fichero de una sola vez. Para esta operación, Python nos provee, al menos, de dos funciones:

read()

Devuelve todo el contenido del fichero como una cadena de texto (str):

f = open('files/temps.dat')

**>>>** f.read()

'29 23\n31 23\n34 26\n33 23\n29 22\n28 22\n28 22\n'

readlines()

Devuelve todo el contenido del fichero como una lista (list) donde cada elemento es una línea:

f = open('files/temps.dat')

**>>>** f.readlines()

['29 23\n', '31 23\n', '34 26\n', '33 23\n', '29 22\n', '28 22\n', '28 22\n']

Importante

Nótese que, en ambos casos, los saltos de línea \n siguen apareciendo en los datos leídos, por lo que habría que «limpiar» estos caracteres. Para ello se recomienda utilizar las funciones ya vistas de cadenas de texto.

Lectura línea a línea

Hay situaciones en las que interesa leer el contenido del fichero línea a línea. Imaginemos un fichero de tamaño considerable (varios GB). Si intentamos leer completamente este fichero de sola una vez podríamos ocupar demasiada RAM y reducir el rendimiento de nuestra máquina.

Es por ello que Python nos ofrece varias aproximaciones a la lectura de ficheros línea a línea. La más usada es iterar sobre el propio manejador del fichero:

f = open('files/temps.dat')

**>>> for** line **in** f: *# that easy!*

**...**  print(line)

**...**

29 23

31 23

34 26

33 23

29 22

28 22

28 22

Truco

Igual que pasaba anteriormente, la lectura línea por línea también incluye el salto de línea \n lo que provoca un «doble espacio» entre cada una de las salidas. Bastaría con aplicar line.split() para eliminarlo.

Escritura en un fichero

Para escribir texto en un fichero hay que abrir dicho fichero en modo escritura. Para ello utilizamos un argumento adicional en la función open() que indica esta operación:

f = open('files/canary-iata.dat', 'w')

Nota

Si bien el fichero en sí mismo se crea al abrirlo en modo escritura, la ruta hasta ese fichero no. Eso quiere decir que debemos asegurarnos que las carpetas hasta llegar a dicho fichero existen. En otro caso obtenemos un error de tipo FileNotFoundError.

Ahora ya podemos hacer uso de la función write() para enviar contenido al fichero abierto.

Supongamos que queremos volcar el contenido de una lista en dicho fichero. En este caso partimos de los códigos IATA de aeropuertos de las Islas Canarias.

canary\_iata = ("GCFV", "GCHI", "GCLA", "GCLP", "GCGM", "GCRR", "GCTS", "GCXO")

**for** code **in** canary\_iata:

# Escritura de cada código en el fichero. La función write() no incluye el salto de línea por defecto,

# así que lo añadimos de manera explícita.

f.write(code + '**\n**')

# Cierre del fichero con la función close(). Especialmente en el caso de la escritura de ficheros, se

# recomienda encarecidamente cerrar los ficheros para evitar pérdida de datos.

f.close()

Advertencia

Siempre que se abre un fichero en modo escritura utilizando el argumento 'w', el fichero se inicializa, borrando cualquier contenido que pudiera tener.

Añadido a un fichero

La única diferencia entre añadir información a un fichero y escribir información en un fichero es el modo de apertura del fichero. En este caso utilizamos 'a' por «append»:

f = open('more-data.txt', 'a')

En este caso el fichero more-data.txt se abrirá en modo añadir con lo que las llamadas a la función write() hará que aparezcan nueva información al final del contenido ya existente en dicho fichero.

Usandos contextos

Python ofrece gestores de contexto como una solución para establecer reglas de entrada y salida a un determinado bloque de código.

En el caso que nos ocupa, usaremos la sentencia with y el contexto creado se ocupará de cerrar adecuadamente el fichero que hemos abierto, liberando así sus recursos:

**with** open('files/temps.dat') **as** f:

**...**  **for** line **in** f:

**...**  max\_temp, min\_temp = line.strip().split()

**...**  print(max\_temp, min\_temp)

**...**

29 23

31 23

34 26

33 23

29 22

28 22

28 22

Línea 1

Apertura del fichero en modo lectura utilizando el gestor de contexto definido por la palabra reservada with.

Línea 2

Lectura del fichero línea a línea utilizando la iteración sobre el manejador del fichero.

Línea 3

Limpieza de saltos de línea con strip() encadenando la función split() para separar las dos temperaturas por el caracter espacio. Ver limpiar una cadena y dividir una cadena.

Línea 4

Imprimir por pantalla la temperatura mínima y la máxima.

Nota

Es una buena práctica usar with cuando se manejan ficheros. La ventaja es que el fichero se cierra adecuadamente en cualquier circunstancia, incluso si se produce cualquier tipo de error.

Hay que prestar atención a la hora de escribir valores numéricos en un fichero, ya que el método write() por defecto espera ver un «string» como argumento:

lottery = [43, 21, 99, 18, 37, 99]

**>>> with** open('files/lottery.dat', 'w') **as** f:

**...**  **for** number **in** lottery:

**...**  f.write(number + '**\n**')

**...**

Traceback (most recent call last):

File "<stdin>", line 3, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Importante

Para evitar este tipo de errores, se debe convertir a str aquellos valores que queramos usar con la función write() para escribir información en un fichero de texto.