Tema 1

En este tema

En el primer tema, vamos abordar algunos aspectos que, sin ser precisamente muy avanzados, son fundamentales para trabajar con Python seriamente y que suelen ser descuidados en otros cursos.

La primera parte, titulada "**Python básico**", no es más que un recordatorio de las ideas fundamentales de Pyhton. Se supone que el alumnado de este curso debería conocer ya todos ellos, y se mencionan de forma casi telegráfica para servir de recordatorio.

La segunda parte, "**Mutable e inmutable**", trata un tema bastante teórico pero fundamental: Una gran cantidad de errores de programación en Python derivan de no haber entendido correctamente la diferencia entre objetos mutables e inmutables y cómo los trata Python. Es mucho más importante de lo que parece para cualquier persona que tenga previsto trabajar con Python de forma más o menos seria.

Después, "**Formateo de cadenas**" nos servirá para ver algunos métodos de las cadenas de texto (porque en Python las cadenas, como todo, son objetos) que nos van a servir más adelante a la hora de mostrar textos (en python, históricamente, hay varias formas de formatear texto, pero format() es la más nueva, la más cómoda y flexible, y la recomendada actualmente). Además veremos un formato de cadenas que, por ser bastante novedoso (se incluyó a partir de Python 3.7) no suele verse en otros cursos.

En "**Slices**" se explica una sintaxis muy característica de Python para extraer listas a partir de subconjuntos de elementos de otras listas, así como para hacer lo mismo en otros iterables, como tuplas o cadenas de texto.

"**Control de errores**" es un capítulo muy importante, porque trata sobre cómo manejar interrupciones en nuestro código, que no sólo es una tarea muy importante en cualquier lenguaje de programación, sino que en Python, debido a su filosofía "EAFP", forma parte integral de cualquier flujo de programa.

Por último, en "**Manejo de ficheros**" veremos algunos aspectos fundamentales para crear, leer y escribir en ficheros, y cómo trabajar con su contenido.

Cómo se puede ver, ninguno de estos aspectos es especialmente avanzado, y es

probable que parte de ellos ya nos sean conocidos. Pero son conceptos fundamentales que necesitan ser conocidos antes de entrar en otros temas

Python básico

Esto son una serie de conceptos básicos de Python que se dan por conocidos, pero que se mencionan como recordatorio.

Tipos de dato simples

Texto

Se acota entre comiilas simples, dobles y triples.

```
"Soy un texto"

'Soy un texto'

"""Soy un texto
con varias líneas"""
```

El carácter de escape es \

Se puede crear con la función str()

Números enteros, reales y complejos

Enteros:

0
1
6
192356

Se convierte a entero con la función int()

Coma flotante (es decir, con decimales):

0.0

0.127546734
234235.678678
2.0

Con float()

Números complejos (con parte real y parte imaginaria):

(3+1j)
(2-2j)
(5-3j)

Con complex()

Booleanos (lógicos):

Sólo pueden valer True o False.

Fundamentales para condicionales y cualquier operación lógica.

Se crean con bool()

None

El dato que dice que no hay dato.

Tipos iterables

Los iterables son conjuntos de datos que pueden ser tomados uno a uno. Se pueden usar, por ejemplo, para alimentar un bucle for.

Listas:

Conjuntos de elementos a los que se accede por su posición. Recordemos que la primera posición es el cero.

```
lista = ['cosa', 2, 'otra', ['a', 'b']]
print(lista[1])
print(lista[3][0])
```

Se puede convertir algo a lista con list()

Tuplas:

Son como las listas, pero no se pueden modificar (no se pueden agregar elementos, quitarlos o cambiarlos).

```
tupla = 'cosa', 'otra cosa', 'y otra'
tupla = ('cosa', 'otra cosa', 'y otra')
print(tupla[0])
```

Con tuple()

Nota: Las cadenas de texto se pueden usar como si fueran tuplas de caracteres.

Diccionarios:

Parecidos a las listas, pero se accede a los elementos por su clave, no por su número de orden. La clave puede ser cualquier tipo de dato, siempre que sea inmutable.

```
diccionario = {'clave'; valor, 'otra clave': 'otro valor, 'y otra': 'y otro'}
diccionario = {'clave'; valor,
               'otra clave': 'otro valor,
               'y otra': 'y otro'}
```

Se crean con la función dict()

Sets:

Un set es un conjunto de elementos sin orden y sin clave. No admiten elementos repetidos.

```
set = {'cosa', 'otra', 'y otra'}
```

Frozensets:

Como los sets, pero inmutables.

Se crean con frozenset()

Operadores

Aparte de los operadores aritméticos clásicos (+, -, *, /, **...), los lógicos (and, or y not), y los de comparación (==, !=, >, <, >=, <=) python tiene algunos operadores peculiares, pero muy importantes:

is
is not

Identifica si un objeto es el mismo que otro objeto (o lo contrario, con not, claro)

in
not in

Identifica si un objeto está contenido en otro (o lo contrario, con not, claro). Elementos en listas, subcadenas en cadenas de texto...

El operador "+", aplicado a cadenas, las concatena.

Se puede multiplicar una cadena un número de veces con el operador "*".

Control de flujo

Salto condicional

Se hace el salto condicional usando:

if:
elif:
else;

Por ejemplo:

```
if a > b:
    print("a es mayor que b")
elif a < b:
    print("a es menor que b")
else:
    print("a y b son iguales")
```

Bucles

El bucle más simple es while, que se ejecuta mientras una condición es cierta:

```
i = 0

while i < 10:
    print(i)
    i += 1
```

Para recorrer elementos de un interable, usamos el bucle for:

```
letras = "abcdefghijklmnñopqrstuvwxyz"

for letra in letras:
    if letra == 'c':
        continue
    if letra == 'g':
        break
    print(letra)
```

En los bucles "continue" interrumpe la iteración actual y salta directamente a la siguiente, y "break" interrumpe el bucle completamente y pasa a la primera instrucción tras este.

Funciones:

Las funciones se define con def:

```
def enmarca(palabra):
    return '**' + palabra + '**'

enmarcado = enmarca('Tu nombre')
print(enmarcado)

print(enmarca('Tu nombre'))
```

Toda función retorna un valor. Si no se alcanza ninguna sentencia "return", retornará un valor None.

Clases y objetos

Las clases se definen con class.

Sus método se definen como funciones, con def, pero su primer argumento debe ser "self" (en realidad, podría tener cualquier nombre, pero usad siempre "self").

Sus atributos son como cualquier otra variable, pero precedidas por "self." para indicar que pertenecen al objeto.

```
class MiClase:

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

    def ma(self):
        return self.palabra.upper()

    def mi(self):
        return self.palabra.lower()

    def re(self):
        return self.palabra[::-1]

una = MiClase("Una palabra")
print(una.ma())
print(una.mi())
print(una.re())
```

Recordatorio: En Python todo son objetos. Por ejemplo, Las cadenas de texto, las funciones o los números enteros tiene métodos y atributos.

Mutable e inmutable

En otros lenguajes de programación, las variables se pueden ver como "cajas" en las que se guarda un valor (una cadena, un número, etc). De hecho, normalmente hablamos como si en Python fuera igual.

Esta imagen también sirve en Python como una primera aproximación, pero la cosa es un poco más compleja.

En realidad, en Python, las variables deberían ser vistas como etiquetas, más que como cajas: Cuando asignamos un valor a una variable, en realidad estamos creando una variable con un nombre interno asignado por python (que nosotros no necesitamos conocer, pero que se puede ver con la función "id()") y a esa variable le asignamos una "etiqueta" con el nombre que hayamos usado para nuestra variable (por eso en Python el nombre realmente correcto para los nombres de variable es "identificadores").

Esto sirve tanto para variables, como para funciones, objetos, etc.

Python actúa de este modo porque es más eficiente a la hora de gestionar recursos. Por ejemplo, cuando asignamos el valor de esa variable a otra (haciendo algo como

"variable2 = variable1") python no crea una variable nueva, sino que asigna una nueva "etiqueta" al valor inicial, de modo que tenemos un sólo dato almacenado en la memoria, pero referenciado por dos nombres distintos.

Pero ¿Qué pasa si modificamos el valor de una de las variables pero no el de la otra? Que Python elimina la etiqueta correspondiente a la variable modificada y crea una nueva variable (con un id interno distinto) con el nuevo valor.

Veamoslo con un ejemplo:

```
#/usr/bin/env python3

# Ejemplo de variables e IDs

# asignamos un valor a una variable:

primera_variable = 5

# copiamos ese valor en otra:

segunda_variable = primera_variable

# vamos a ver los id de cada una:

print("ANTES")
print("ID de la primera variable:")
print(id(primera_variable))

print("ID de la segunda variable:")
print(id(segunda_variable))
# como vemos, son el mismo.

# modificamos la segunda:

segunda_variable = segunda_variable + 1

# Y volvemos a ver los id:

print("DESPUES")
print("ID de la primera variable:")
print(id(primera_variable))

print("ID de la segunda variable:")
print(id(segunda_variable))

# como vemos, la segunda ha cambiado
```

Aunque este funcionamiento interno de Python es bastante distinto de otros lenguajes, esto no afecta al uso por nuestra parte, y podemos trabajar como con cualquier otro.

Excepto por un detalle.

Normalmente, cuando modificamos una variable, esta es destruida y creada una nueva, pero si se trata de, por ejemplo, una lista, esto no ocurre así.

Si modificamos una lista, el ID sigue siendo el mismo (porque, en realidad, hemos modificado uno o más elementos de la lista, no la lista en sí).

La consecuencia es que, si copiamos una lista en otra, y modificamos una de ellas ¡La otra también cambia!

Este ejemplo seguramente aclare las cosas:

```
#/usr/bin/env python3

# Ejemplo de listas e IDs

# creamos una lista:

primera_lista = [1, 2, 3]

# la copiamos en otra:

segunda_lista = primera_lista

# las mostramos:

print("primera lista:")
print(id(primera_lista))

print("segunda lista:")
print(id(segunda_lista))
# son iguales.

# modificamos el primer elemento de la segunda lista:

segunda_lista[0] = 1000

# y las mostramos otra vez:

print("DESPUES")

print("primera lista:")
print(id(primera_lista))

print("segunda lista:")
print(id(segunda_lista))
# Han cambiado ambas!!!
```

Si queremos copiar una lista en otra y que realmente sean listas independientes, podemos hacerlo explícitamente, usando la función "list()" de este modo:

```
#/usr/bin/python3
# -*- coding: utf-8 -*-

# Copiando listas

# creamos una lista:

primera_lista = [1, 2, 3]

# la copiamos en otra:

segunda_lista = list(primera_lista)
```

Dejo como ejercicio para cada uno el probar que, efectivamente, esto crea dos listas distintas.

Los objetos de Python que, como los números, son destruidos y creados de nuevo al modificarlos se llaman "inmutables". Los que pueden modificarse sin que cambie su ID son llamados "mutables".

Las listas, y diccionarios, por ejemplo, son mutables. Los números, cadenas o tuplas son inmutables.

Como clave de un diccionario, por ejemplo, podemos utilizar cualquier valor inmutable: podríamos usar números, cadenas, booleanos, tuplas... pero no listas o diccionarios, dado que son mutables.

Para los que tengan especial interés: Esto es así porque los diccionarios se implementan como tablas hash, y a la hora de introducir un nuevo par clave-valor en el diccionario se calcula el hash de la clave para después poder encontrar la entrada correspondiente rápidamente. Si se modificara el objeto clave después de haber sido introducido en el diccionario, evidentemente, su hash también cambiaría y no podría ser encontrado.

Formateo de cadenas

Métodos para trabajar con cadenas:

Para imprimir una cadena, normalmente hacemos algo como esto:

```
cadena = "Hola, soy Pablo y estoy de paso"
print(cadena)
```

```
> Hola, soy Pablo y estoy de paso
```

(En este tema voy a poner los resultados precedidos del signo > para que se vea más claro)

Pero en Python, como ya sabemos, todo son objetos, loo que incluye a las cadenas. Y las cadenas son objetos que tiene algunos métodos interesantes:

```
cadena = "Hola, soy Pablo y estoy de paso".title()
print(cadena)
```

```
> Hola, Soy Pablo Y Estoy De Paso
```

```
cadena = "Hola, soy Pablo y estoy de paso".upper()
print(cadena)
```

```
> HOLA, SOY PABLO Y ESTOY DE PASO
```

```
cadena = "Hola, soy Pablo y estoy de paso".lower()
print(cadena)
```

```
> hola, soy pablo y estoy de paso
```

```
cadena = "Hola, soy Pablo y estoy de paso".casefold()
print(cadena)
```

```
> hola, soy pablo y estoy de paso
```

```
cadena = "Hola, soy Pablo y estoy de paso".rjust(80)
print(cadena)
```

```
>                                                Hola, soy Pablo y estoy de
paso
```

```
cadena = "Hola, soy Pablo y estoy de paso".ljust(80)
print(cadena)
```

```
> Hola, soy Pablo y estoy de
paso
```

```
cadena = "Hola, soy Pablo y estoy de paso".center(60)
print(cadena)
```

```
>              Hola, soy Pablo y estoy de paso
```

El método format():

Normalmente, cuando queremos formatear un texto a partir de sus partes de forma "rápida y sucia", lo hacemos concatenando sus elementos con el signo "+" de un modo parecido este:

```
print("hola" + " " + "mundo")
```

> hola mundo

Pero el (bastante nuevo) método format() de las cadenas nos permite hacer este tipo de cosas de forma mocho más clara y sencilla:

```
cadena = "Hola, soy {}".format("Recesvinto")
print(cadena)
```

> Hola, soy Recesvinto

Veamoslo con un ejemplo un poco más complejo. Lo que, usando el operador "+" haríamos así:

```
nombre = "Pablo"
mascota = "un gato"
cadena = "Hola, soy " + nombre + " y mi mascota es " + mascota + "."
print(cadena)
```

> Hola, soy Pablo y mi mascota es un gato.

Podemos hacerlo con este método así:

```
nombre = "Pablo"
mascota = "un gato"
cadena = "Hola, soy {} y mi mascota es {}.".format(nombre, mascota)
print(cadena)
```

> Hola, soy Pablo y mi mascota es un gato.

Como podemos ver, en la cadena los pares de corchetes "{}" son reemplazados por los argumentos que le pasamos al método format() en el mismo orden en el que se los pasamos.

Podemos usar los argumentos en los corchetes en otro orden distinto, indicándolo

con su número de índice (0, el primero, 1 el segundo...) en los propios corchetes, de este modo:

```
nombre = "Pablo"
mascota = "un gato"
cadena = "Hola, soy {1} y mi mascota es {0}.".format(nombre, mascota)
print(cadena)
```

> Hola, soy un gato y mi mascota es Pablo.

También podemos usar nombres en lugar de números:

```
nombre = "Pablo"
mascota = "un gato"
cadena = "Hola, soy {yo} y mi mascota es {gato}.".format(yo= nombre, gato=
mascota)
print(cadena)
```

> Hola, soy Pablo y mi mascota es un gato.

f-strings:

Otra opción para formatear textos son las f-strings.

Las f-strings son cadenas de texto especiales, que se indican poniéndole una letra f delante de las comillas, y en las que el texto que vaya entre corchetes se interpreta como cualquier otro código de Python. Por ejemplo, reemplazándolo con el valor de la variable del mismo nombre:

```
nombre = "Pablo"
mascota = "un gato"
cadena = f"Hola, soy {nombre} y mi mascota es {mascota}."
print(cadena)
```

> Hola, soy Pablo y mi mascota es un gato.

O, en general, reemplazándolo por el retorno del código que contenga:

```
nombre = "Pablo"
mascota = "un gato"
cadena = f"Hola, soy {nombre.upper()} y mi mascota es {mascota[1:5]}."
print(cadena)
```

```
> Hola, soy PABLO y mi mascota es n ga.
```

Como ejemplo algo más avanzado de esto, veamos un programa simple que genera nombres al azar uniendo sílabas y los usa en una f-string:

```
import random
def al_azar(silabas):
     lista_silabas = ["zo","se","mo","na","tre","cas","lu","de"]
    i = 0
    nombre = ""
    while i < silabas:
        aleatorio = int(len(lista_silabas) * random.random())
        nombre += lista_silabas[aleatorio]
        i += 1
    return nombre.title()

mascota = "un gato"

cadena = f"Hola, soy {al_azar(4)} y mi mascota es {mascota}."
print(cadena)
```

```
> Hola, soy Modetrede y mi mascota es un gato.
```

Este código, cada vez que se ejecute, dará un resultado distinto.

Slices

Como ya sabemos, podemos acceder a cada elemento de una lista por medio de su índice.

Si, por ejemplo, tenemos una lista con las primeras ocho letras del alfabeto:

```
alfa = ["a","b","c","d","e","f","g","h"]
```

Podemos referirnos al cuarto elemento (la "d") indicando el índice 3 (porque el cero es el primero) de este modo:

```
alfa[3]
```

Más interesantes es que también podemos usar números negativos como índice, que nos retornará el elemento que ocupe la poción indicada contando desde atrás, de modo que para obtener respectivamente el último y el penúltimo elementos de

"alfa" haríamos esto:

```
ultimo = alfa[-1]
penultimo = alfa[-2]
```

(no, por razones lógicas, la cuenta negativa no comienza por cero)

Pero también disponemos de una notación especial llamada "slices" que nos permite extraer secciones de la lista de formas más interesantes.

Para empezar, las slices usan índices separados por el signo ":" (dos puntos) para indicar qué parte de la lista queremos obtener según este formato general

```
[inicio:parada]
```

Donde "inicio" es el índice del primer elemento que queremos obtener y "parada" el del elemento en que se detendrá nuestra selección (no incluido). Por ejemplo, para mostrar a los cuatro primeros elementos de la lista "alfa" que hemos definido más arriba, podríamos hacerlo así:

```
primeros = alfa[0:4]
print(primeros)
```

Esto nos imprimirá la lista ["a","b","c","d"], es decir, desde el elemento 0 (el primero) hasta el elemento 3 (el cuarto), porque el 4 no está incluido.

Del mismo modo, para obtener los elementos desde el tercero ("c") al sexto ("f") haríamos algo así:

```
alfa[3:6]
```

Se puede omitir alguno de los índices. Omitir el de inicio es equivalente a copiar la lista desde el comienzo, como si hubiéramos puesto un cero. De este modo, para imprimir los cuatro primeros elementos de la lista "alfa" como hicimos antes, podríamos haberlo escrito igualmente así:

```
primeros = alfa[:4]
print(primeros)
```

Del mismo modo, si omitimos el índice de parada, retornará la lista desde el elemento que le indiquemos en inicio hasta el final de la lista. Así que si queremos obtener una lista desde el quinto elemento en adelante, lo indicaríamos así:

```
alfa[5:]
```

De esto se deduce que una forma fácil y rápida de copiar una lista completa es hacerlo así:

```
alfa[:]
```

Porque esto retorna una lista con los elementos de "alfa" desde el primero hasta el último.

En los slices también podemos usar números negativos, de modo que podemos extraer los dos últimos elementos de "alfa" de este modo:

```
alfa[-2:]
```

Es decir: desde el elemento -2 (el penúltimo) hasta el final de la lista.

Si queremos lo contrario, obtener todos los elementos excepto los dos últimos, lo haremos así:

```
alfa[:-2]
```

O sea: desde el principio hasta el elemento -2 (el penúltimo).

Los slices admiten un tercer número (separado por otros dos puntos) para indicar en qué pasos se devolverán los elementos. Si indicamos "2", se retornará uno de cada dos, "3" para retornar uno de cada tres...

De este modo, el siguiente slice:

```
alfa[1:7:2]
```

Nos retornaría la lista ['b', 'd', 'f'].

Y con el siguiente:

```
alfa[::3]
```

obtendremos ['a', 'd', 'g'].

Además, se pueden usar números negativos para indicar que los pasos deben

contarse de atrás adelante, por lo que este ejemplo:

```
print(alfa[6:3:-1])
```

Nos retornará la secuencia ['g', 'f', 'e']. Es importante fijarse en que hemos puesto "6" como incoo y "3" como parada porque, al indicar -1 en los pasos, vamos desde atrás adelante.

De esto se deduce que la forma más simple de invertir una lista es, simplemente, hacer:

```
alfa[::-1]
```

Si un slice está construido de modo que no puede retornar ningún elemento o no tiene sentido, retornará una lista vacía. Por ejemplo:

```
print(alfa[7:2])
```

Como no se puede avanzar desde el 7 al 2 (a menos que le pongamos unos pasos negativos), esta slice retornará una lista vacía.

Los slices se pueden usar con las tuplas exactamente del mismo modo que hemos visto con el ejemplo de las listas, con la diferencia de que lo que se retornará será una tupla y no una lista.

Además, y dado que Python trata las cadenas de texto como tuplas de caracteres, todo esto es también aplicable a estas, con la diferencia de que se retornarán otras cadenas de texto.

Por ejemplo, podemos hacer cosas como estas:

```
texto = "Hola Mundo"
texto[5:]
texto[::2]
"123456789"[:5]
```

Que retornarán, respectivamente, "Mundo", "Hl ud" y "12345".

Control de errores

Cuando ejecutamos un programa hay muchas cosas que pueden fallar. Podemos intentar abrir un archivo que no hay ahí, o tratar de operar con un dato erróneo, que haya inconsistencias en la información que manejamos o que ocurran cosas

inesperadas. Muchos de esos errores son independientes del programa y no se pueden gestionar fácilmente (por ejemplo, que haya un error de hardware o un fallo eléctrico). Nuestra responsabilidad como programadores es tratar de controlar aquellos que estén en nuestra mano.

Alguno errores se deben a una programación deficiente, como este ejemplo:

```
if numero < 0:
    texto = "menor que cero"
if numero > 0:
    texto = "mayor que cero"

print(texto)
```

En este ejemplo, si "numero" tiene el valor cero tendremos un error (del tipo NameError), porque no hemos llegado definir la variable "texto" en ningún momento. La mejor forma de evitar esto es iniciar la variable al principio o contemplar en nuestro "if" la posibilidad de que "numero" valga cero.

Pero, incluso en un programa perfectamente ideado, se pueden dar errores inesperados, debido a multitud de causas.

Comencemos con un error clásico:

```
a = 1
b = 0
resultado = a / b
print("resultado: " + str(resultado))
```

Si ejecutamos esto en un script, el programa finalizará repentinamente al llegar a la tercera línea (donde hacemos "resultado = a / b") y nos retornará un mensaje de error "ZeroDivisionError: división by zero".

El problema no es simplemente que el programa se detenga y no haga lo que esperábamos que hiciese: Si nuestro programa hubiese tenido cosas como archivos, comunicaciones o bases de datos con datos a medio guardar o cualquier otra cosa de ese tipo, un error como este podría haber sido desastroso. Que nuestros programas sean capaces de controlar los errores y reaccionar a ellos es fundamental para cualquier utilidad medianamente compleja.

Para mantener controlado el ejemplo anterior, podríamos escribir una serie operaciones que comprobasen que la variable "b" no es un cero, que es un número, etc. De paso, también habría que comprobar que "a" también es un número, por ejemplo. Pero, en cuanto nuestro programa comenzase a crecer y a hacer cosas más complejas, todas esas comprobaciones a cada paso harían el código más y

más complicado.

La aproximación de Python a este problema se conoce con las siglas EAFP, por la frase de Grace Hopper “it’s easier to ask for forgiveness than permission” ("Es más fácil pedir perdón que pedir permiso").

La idea consiste en permitir que los errores ocurran, pero "capturarlos" y actuar en consecuencia a ellos.

Para ello usamos la estructura try...except, que funciona del siguiente modo:

```
try:
    Código que queremos controlar
except:
    Qué hacemos si hay un error
else:
    Qué hacemos si no hay ningún error
```

Como se ve en el esquema de arriba, dentro del bloque "try" temeos el código Python que queremos ejecutar y del que queremos manejar los errores.

En caso de que ocurra un error, en lugar de detener el programa repentinamente, se ejecuta el código que hay en el bloque "except".

Opcionalmente, podemos poner una cláusula "else" que se ejecutará si no hay ningún error.

En nuestro ejemplo de la división por cero, podríamos hacer algo así:

```
a = 1
b = 0
try:
    resultado = a / b
except:
    print("Ha habido un error")
else:
    print("resultado: " + str(resultado))
```

Naturalmente, en un ejemplo más complejo, usaríamos el "except" para, por ejemplo, cerrar ficheros que estuvieran abiertos, asignar valores por defecto a variables que no estuviesen definidas o lo que fuera necesario para que nuestro programa pudiese continuar operando o, al menos, el error no fuera desastroso.

También podemos añadir un bloque "finally", que se ejecutará siempre, haya error o no:

```
a = 1
b = 0
try:
    resultado = a / b
except:
    print("Ha habido un error")
else:
    print("resultado: " + str(resultado))
finally:
    print("Pues eso")
```

Si nos fijamos en el ejemplo inicial, cuando nuestro programa fallaba nos retornaba un mensaje indicando, entre otras cosas, el tipo de error, que en este caso era "ZeroDivisionError".

Podemos hacer que nuestro programa interrumpa de forma distinta los distintos tipos de error usando esos nombres de este modo:

```
a = 1
b = 0
try:
    resultado = a / b
except ZeroDivisionError:
    print("no puedes dividir por cero")
else:
    print("resultado: " + str(resultado))
```

Ahora "except" captura un tipo concreto de error y actúa de forma específica ante él (en nuestro caso, muestra el texto "no puedes dividir por cero").

En realidad, "ZeroDivisionError" es lo que llamamos una "excepción": Una clase en Python que hereda de la clase "Exception". Lo que hacemos es capturar los errores de esa clase.

Aquí, "ZeroDivisionError" es lo que llamamos una "excepción", una clase de objeto (que hereda de la clase "Exception") que nos retorna Python cuando hay un error, y lo que hacemos con "try...except" es capturar las excepciones. En realidad, se retornan excepciones en más contextos que no indican necesariamente errores. Por ejemplo, cuando un generador llega a su último elemento emite una excepción GeneratorExit (que, por ejemplo, "for" es capaz de interpretar como el final del loop) o, cuando pulsamos Ctrl-c para forzar la salida del programa, emitimos una excepción KeyboardInterrupt.

Pero capturar excepciones concretas tiene como contrapartida que dejará pasar el resto de errores sin capturarlos, como en este ejemplo:

```
a = 1
b = "0"
try:
    resultado = a / b
except ZeroDivisionError:
    print("no puedes dividir por cero")
else:
    print("resultado: " + str(resultado)
```

Si ejecutamos este código (que sólo ha cambiado en las comillas del cero, de modo que "b" es ahora una cadena y no un número) fallará de nuevo, mostrando un error TypeError que no estamos capturando.

Afortunadamente, se pueden poner tantas clausulas "except" en un bloque "try" como sea necesario:

```
a = 1
b = "0"
try:
    resultado = a / b
except ZeroDivisionError:
    print("no puedes dividir por cero")
except TypeError:
    print("esto no es un número")
else:
    print("resultado: " + str(resultado))
```

Además podemos hacer que un "except" controle varios errores distintos incluyéndolos entre paréntesis y separados por comas:

```
a = 1
b = "0"
try:
    resultado = a / b
except ZeroDivisionError:
    print("no puedes dividir por cero")
except (TypeError, NameError):
    print("esto no es un número")
else:
    print("resultado: " + str(resultado))
```

La excepciones se interrumpen en el orden en que colocamos los excpet, de modo que podemos poner una última cláusula "except" sin indicar el error concreto, que capturará cualquier error que no haya sido capturado en los anteriores:

```
a = 1
b = "0"
try:
    resultado = a / b
except ZeroDivisionError:
    print("no puedes dividir por cero")
except (TypeError, NameError):
    print("esto no es un número")
except:
    print("error indefinido")
else:
    print("resultado: " + str(resultado))
```

Pero esto no es una buena práctica y no se recomienda.

Por ejemplo, cuando nuestro programa se bloquea (porque, por ejemplo, haya entrado en un bucle infinito) y pulsamos Ctrl-c para salir, lo que hacemos es generar una excepción del tipo KeyboardInterrupt que detiene el programa. Si hubiera un "except" que lo capturase, no podríamos salir.

Es muchos más seguro, cuando queremos capturar excepciones indefinidas, capturar las del tipo Except que, al ser la clase madre de la que dependen los errores, capturará a todos ellos.

```
a = 1
b = "0"
try:
    resultado = a / b
except ZeroDivisionError:
    print("no puedes dividir por cero")
except (TypeError, NameError):
    print("esto no es un número")
except Exception:
    print("error indefinido")
else:
    print("resultado: " + str(resultado))
```

Por último, también podemos asignar el error a una variable para, por ejemplo, obtener información más concreta, usando "as" en el "except" como en este ejemplo:

```
a = 1
b = 0
try:
    resultado = a / b
except ZeroDivisionError as e:
    print("Esto ha dado un error: " + str(e))
else:
    print("resultado: " + str(resultado)
```

Podemos forzar excepciones usando "raise". Por ejemplo, para generar directamente un ZeroDivisionError haríamos lo siguiente:

```
raise ZeroDivisionError
```

Además, podemos crear nuestros propios errores con nuestros propios mensajes de error fácilmente, instanciando a partir de la clase Exception:

```
x = -1
if x < 0:
  raise Exception("No se permiten negativos")
```

También podemos crear nuestra propia clase de excepciones haciendo que hereden de Exception:

```
class MiPropioError(Exception):
    pass
```

Se pueden encontrar todas las excepciones nativas de Python, junto con su jerarquía y una explicación de cada una de ellas en su página oficial https://docs.Python.org/3/library/exceptions.html#ZeroDivisionError

Manejo de ficheros

Lectura y escritura de ficheros

Abrir un fichero es tan simple como usar la función open():

```
fichero = open("ruta/al/fiichero")
```

La función open() toma como primer argumento una cadena de texto con la ruta del archivo, y retorna un objeto fichero que podemos usar para interaccionar con este.

Leer un archivo o escribir en él no tiene po qué ser lo mismo, de modo que la función open() admite también un segundo argumento opcional, que indica el modo en el que se abrirá el archivo.

El modo puede ser uno de lo siguientes:

'r' lectura (es el valor por defecto. Si el archivo no existe, dará un error)
'w' escritura (si el archivo no existe, se crea)
'a' añadir (si el archivo no existe, se crea)
'r+' lecto-escritura (tanto leer como escribir. Si el archivo no existe, se crea)

Si escribimos en un archivo que ya tenga información se sobrescribirá, perdiéndose el contenido anterior, a menos que lo hagamos abriendo el archivo con la opción "a", que hace que lo que escribamos se almacene a continuación de lo que hubiera antes.

A estos códigos se les puede añadir las letras "b" o "t" para abrir el archivo en modo binario (para operar con el contenido del archivo a nivel de bytes) o en modo texto (la opción por defecto).

Por ejemplo, para abrir un archivo en modo lectura de texto, podríamos usar la función open() del siguiente modo:

```
fichero = open('pruebas_python.txt', 'bt')
```

Pero, como tanto "b" como "r" son las opciones por defecto, es mucho más sencillo hacerlo simplemente así:

```
fichero = open('pruebas_python.txt')
```

Tras abrir un fichero y después de hacer con él las operaciones necesarias para nuestro programa, es necesario cerrarlo. Para ello usamos el método close() de este modo:

```
fichero.close()
```

Veamos ahora qué tipo de operaciones podemos hacer con un fichero.

Podemos usar la función print() para imprimir, en lugar de en pantalla, en un fichero (que hayamos abierto en un modo que permita escritura) usando el argumento file:

```
fichero = open('pruebas_python.txt', 'w')
print("Creamos una primera fila", file=fichero)
fichero.close()
```

Este código creará un fichero en el mismo directorio de nuestro programa con el nombre 'pruebas_python.txt' y el contenido que le hemos agregado.

Usando print() varias veces, exactamente igual que hacemos por pantalla, podemos añadir varias líneas:

```
fichero = open('pruebas_python.txt', 'w')

print("Una fila", file=fichero)
print("Otra fila", file=fichero)

fichero.close()
```

Como aquí hemos vuelto a abrir el mismo archivo del ejemplo anterior y hemos escrito en eĺ, el contenido anterior se ha perdido.

Si quisiéramos añadir contenido al archivo a continuación del lo que ya hubiese en él, usaríamos la opción "a":

```
fichero = open('pruebas_python.txt', 'a')

print("Una fila añadida", file=fichero)

fichero.close()
```

Para leer el contenido de un fichero abierto, disponemos de una serie de métodos que nos retornarán su contenido. El primero de ellos es el método read(), que nos devuelve una cadena de texto con el contenido del archivo:

```
fichero = open('pruebas_python.txt', 'r')

contenido = fichero.read()

fichero.close()

print(contenido)
```

Opcionalmente, el método read() permite indicar el número de caracteres que queremos tomar del archivo indicándolo como argumento de este modo:

```
fichero = open('pruebas_python.txt', 'r')

contenido = fichero.read(10)
```

```
fichero.close()

print(contenido)
```

A menudo, especialmente si el fichero es muy largo, no es práctico tomar todo su contenido de una sola vez. El método readlines() funciona igual que read(), pero retorna las líneas del fichero en forma de iterable (un objeto que nos permite tomar sus elementos uno a uno, como las listas o las tuplas):

```
fichero = open('pruebas_python.txt', 'r')

contenido = fichero.readlines()

fichero.close()

print(contenido)
```

Esto nos permite usar el contenido en un bucle for, por ejemplo:

```
fichero = open('pruebas_python_.txt')

for i in fichero.readlines():
    print(i)

fichero.close()
```

Gestores de contexto

Como hemos visto antes, si se trata de abrir un archivo que no existe dará un error, así que normalmente es necesario usar construcciones como esta (muy simplificada, claro está):

```
try:
    fichero = open('no_existe.txt')
except FileNotFoundError:
    pass
else:
    for i in fichero.readlines():
        print(i)

    fichero.close()
```

En este ejemplo, lo que estamos haciendo es tratar de abrir el archivo, trabajar con él (leer su contenido con un bucle for, en este caso) y cerrarlo. Pero sólo en caso de que el archivo exista y open() no retorne una excepción del tipo FileNotFoundError.

Esta estructura del tipo "abrir un archivo (u otro recurso) y, si no hay problemas, hacer lo que sea y después cerrarlo" es tan habitual que python dispone de una estructura, llamada "Gestor de contexto" para manejarla con más comodidad, usando la palabra reservada "with":

```
with open('pruebas_python.txt', 'r') as fichero:

    contenido = fichero.readlines()

print(contenido)
```

Esta estructura es equivalente a la anterior, con la diferencia de que aquí indicamos el nombre del fichero abierto con la cláusula "as" y no tenemos que preocuparnos de controlar si hemos podido abrir el recurso o de cerrar el archivo al terminar.

Aunque la entrada y salida de archivos es el uso más común de los gestores de contexto, cualquier objeto de python que se pueda manejar como un flujo con inicio y fin (Técnicamente, objetos que tengan definidos los métodos mágicos __enter__() y __exit__()) se pueden usar de este modo.

CSV

Un caso espeial y muy usado son los archivos CSV.

Un CSV no es más que un archivo de texto en el que se han definido filas (líneas en el archivo separadas por un retorno de carro) y columnas (subdivisiones de la fila, separadas por comas). Puede tener algunas complejidades más (comillas para distinguir texto de números, tabulaciones en lugar de comas y cosas así), pero básicamente es eso, texto separado por comas.

Sería relativamente sencillo crear una función que abra un archivo, lo lea línea a línea con readlines y separe cada línea por las comas, etc.

Pero python ya dispone de un módulo llamado "csv" que hace todo eso (y más) por nosotros, y que nos facilita mucho el trabajo:

```
import csv

with open('pruebas_csv.txt') as fichero:
    reader = csv.reader(fichero)
    for fila in reader:
        print(fila[2], fila[4])
```

Lo interesante de este ejemplo está en el uso de csv.reader(), que nos retorna una lista de filas que, a su vez, están compuestas por celdas, y que son accesibles por sus índices.

El método csv.DictReader() es muy parecido, pero cada fila retornada es un diccionario, accesible por el nombre de la columna (se asume por defecto que la primera fila del archivo son los nombres de las columnas):

```
with open('pruebas_csv.txt') as fichero:
    reader = csv.DictReader(fichero)
    for fila in reader:
        print(fila["Cabecera 2"], fila["Cabecera 4"])
```

Para usos más avanzados, en los que se requieren estructuras de datos mucho más complejas de lo que puede contener un fichero CSV, también tenemos un módulo para leer ficheros JSON.

Tema 2

En este tema

El segundo tema de este curso está dedicado a los entornos virtuales.

Los entornos virtuales son la forma más simple y práctica de tener el equivalente a múltiples instalaciones de Python diferentes.

Es importante meternos esto en la cabeza: Cualquier proyecto en Python debería empezar por la creación de un entorno virtual. Ya sea un pequeño script de prueba para practicar o un sofisticado módulo que pensamos distribuir, la primera instrucción que deberíamos usar en nuestro directorio de trabajo es algo parecido a "python3 -m venv venv".

Salvo las contadas excepciones en que un módulo no pueda instalarse en un entorno virtual, siempre deberíamos instalarlo en el entorno de nuestro proyecto, no en el entorno general.

Entornos virtuales con venv

En este capítulo se supone que ya tenemos instalados los módulos "pip" y "venv". Normalmente vienen instalados por defecto en casi todos los sistemas pero, si no es así, habrá que instalarlos antes de continuar.

Cuando programamos en Python lo normal (y lo razonable) es no reinventar la rueda y usar las funciones y método de módulos y paquetes para hacer la mayor parte posible del trabajo. Esto tiene el problema de que podemos ir acumulando paquetes innecesarios instalados (si, por ejemplo, borramos o modificamos un programa en el que estemos trabajando por no desinstalamos los paquetes que habíamos instalado para él), que desinstalemos paquetes que necesitábamos para otro proyecto, que dos proyectos distintos necesiten versiones incompatibles de paquetes o, en general, diversos problemas de organización.

Para evitar esto, se inventaron los entorno virtuales.

Un entorno virtual es, por decirlo de algún modo, un contexto seguro donde instalar paquetes de python. Consiste básicamente en un directorio con el ejecutable de python y todo lo que necesita para trabajar, acompañado de un estructura de directorios para instalar paquetes y algunas utilidades para facilitar su gestión. Podemos tener varios entornos virtuales en nuestro sistema, cada uno con su propio ejecutable de python y sos propios módulos instalados, y activar uno u

otro según nos convenga en cada momento. Además, cuando no necesitemos más uno de estos entornos, podemos eliminarlo sin afectar a los demás.

Los entornos virtuales se crean, como veremos enseguida, con el módulo de Python "venv".

Para ejecutar un módulo de python directamente llamamos a python usando el modificador -m seguido del nombre del módulo. De este modo, para ejecutar el módulo venv debemos hacerlo así:

```
python3 -m venv /ruta/al/entorno
```

donde "/ruta/al/entorno" es el directorio donde queremos que se almacenen los archivos de nuestro entorno virtual. Si no existe alguno de los directorios de la ruta indicada, se crearán automáticamente.

(dado que, en la mayoría de los sistemas actuales, la versión de python por defecto es la 2, debemos especificar que estamos usando "python3". En sistemas en que python3 es la instalación por defecto no será necesario especificarlo y sólo habría que poner "python", sin el "3")

En windows, dependiendo de la versión de Python concreta que tengamos y de dónde esté instalado, el comando sería algo parecido a esto:

```
c:\Python38\python -m venv c:\ruta\al\entorno
```

Suponiendo que "c:\Python38\python" fuese la ruta a la instalación de Python. O, si las variables de entorno PATH y PATHEXT están configuradas (y, por lo tanto, Windows puede encontrar por sí mismo la ruta a la instalación de Python), sería algo así:

```
python -m venv c:\ruta\al\entorno
```

Se puede ver cómo configurar las variables PATH y PATHEXT en la pagina ofcial https://docs.python.org/3/using/windows.html#using-on-windows

Independientemente del sistema operativo en el que trabajemos, lo normal es tener un entrono virtual independiente para cada proyecto. Y lo más usual es hacerlo creando, en el propio directorio raíz de nuestro proyecto, un subdirectorio llamado "venv".

De este modo, lo más simple es usar, en el directorio raíz del proyecto, el siguiente

comando:

```
python3 -m venv venv
```

Esto nos creará un directorio llamado "venv" donde se almacenarán los script de inicio de nuestro entorno virtual, los paquetes que instalemos en él y toda una serie de cosas más.

Pero sólo crear en entorno virtual no es suficiente. Una vez creado el entorno, para usarlo, debemos activarlo con el siguiente comando:

```
source /ruta/al/entorno/bin/activate
```

"source" es un comando unix que toma el contenido de un archivo y lo ejecuta en el entorno actual como si se tratase de una serie de comandos tecleados por el propio usuario. De este modo, lo que hacemos es ejecutar los comandos que hay en archivo "activate" del directorio "bin" de nuestro entorno virtual con nuetro usuario (leer el contenido de ese archivo y escribir manualmente ese contenido en nuestro terminal tendría el mismo efecto, pero no es nada práctico).

En windows, sin embargo, sería algo así:

```
c:\ruta\al\entorno\Scripts\activate.bat
```

Si usamos algún shell más "exótico", el comando puede ser ligeramente distinto, como puede verse en esta tabla extraída de la propia documentación del módulo:

Platform	Shell	Command to activate virtual environment
POSIX	bash/zsh	$ source <venv>/bin/activate
	fish	$ source <venv>/bin/activate.fish
	csh/tcsh	$ source <venv>/bin/activate.csh
	PowerShell Core	$ <venv>/bin/Activate.ps1
Windows	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerShell	PS C:\> <venv>\Scripts\Activate.ps1

Al activarlo, normalmente se nos indicará el cambio de entrono añadiendo el nombre de este entrono al principio del prompt de nuestro sistema, como se puede ver en la imagen de ejemplo:

Lo que estamos haciendo al activar el entorno virtual es indicarle a sistema que, a partir de ahora, el ejecutable de python que vamos a usar es el que se encuentra en el entorno virtual, y que la ruta a los módulos instalados también es la que corresponde a los del entrono virtual. Esto tiene como primera consecuencia que ya no debemos escribir "python3", si no que con "python" es suficiente.

Pero además, si instalamos algún módulo usando pip (tampoco hay que escribir pip3, ni aunque en nuestro sistema normalmente haga falta), estos se instalarán sólo en este entorno virtual, y serán accesibles a nuestros scripts (cuando hagamos un import) sólo si tenemos el entrono activado.

Naturalmente, esto se aplica sólo a la sesión en la que se ha activado el entorno virtual. Si abrimos otra sesión en otro terminal, por ejemplo, esa sesión tendrá el entorno de Python normal de nuestro sistema. Por supuesto, se pueden tener varios entornos virtuales distintos corriendo simultáneamente en diferentes sesiones sin ningún problema.

Para desactivar un entorno virtual sólo hay que escribir el siguiente comando:

```
deactivate
```

Y, a partir de ese momento, seguiremos usando python con nuestra instalación habitual.

Si hemos dejado de necesitar un entorno virtual y queremos eliminarlo, sólo hay que borrar el directorio en el que lo hemos creado (el directorio "venv" de nuestro proyecto, en nuestro ejemplo) y eliminaremos el entrono sin afectar ni a otros entornos virtuales ni a nuestra instalación de python.

El módulo venv tiene algunas opciones que pueden verse acudiendo a su ayuda con el comando:

```
python3 -m venv -h
```

Deberíamos utilizar venv para todas las prácticas y pruebas que hagamos, sobre todo si vamos a tener que instalar muchos paquetes. Pero también deberíamos usarlo en todos nuestros proyectos, para tener el control de los paquetes que

estamos usando.

Además, los entornos virtuales pueden usarse para el despliegue de aplicaciones y otros usos avanzados, pero eso va más allá de los objetivos de este tema.

Para más información, se puede consultar la documentación oficial de venv en https://docs.python.org/3/library/venv.html.

Instalación de paquetes

Una vez creado y activado nuestro entorno virtual, podemos comenzar a instalar los paquetes que necesitemos en nuestro proyecto.

El Python Package Index

Un lugar interesante para encontrar nuevos módulos es el Python Package Index.

El Python Package Index es un repositorio web con casi siete mil paquetes y módulos de Python listos para descargar e instalar.

En el Python Package Index es posible buscar módulos por palabras clave, leer su descripción, ver instrucciones y ejemplos de uso, y descargar e instalar aquellos que deseemos.

pip

El módulo pip forma parte de la librería estándar de Python, y nos ayuda a gestionar la instalación, mantenimiento, actualización y desinstalación de paquetes.

En los siguientes ejemplos (y en todos), si "pip3" no funciona en tu sistema, usa "pip" (o sea, sin el número) en su lugar.

Para usarlo, sólo hay que abrir un terminal y escribir en él:

```
pip3 install nombre_del_módulo
```

El programa pip se encarga descargar el módulo e instalarlo. Si ese módulo necesita a su vez de otros módulos (dependencias) los instala también de forma automática.

También podemos usar pip para buscar un módulo por medio de la opción search,

que permite buscar por términos en la base de datos del Python Package Index.

```
pip3 search nombre_del_paquete
```

Por ejemplo, para buscar el término "mysql" deberíamos escribir "pip3 search mysql".

Por último, pip también permite desinstalar módulos con la opción uninstall:

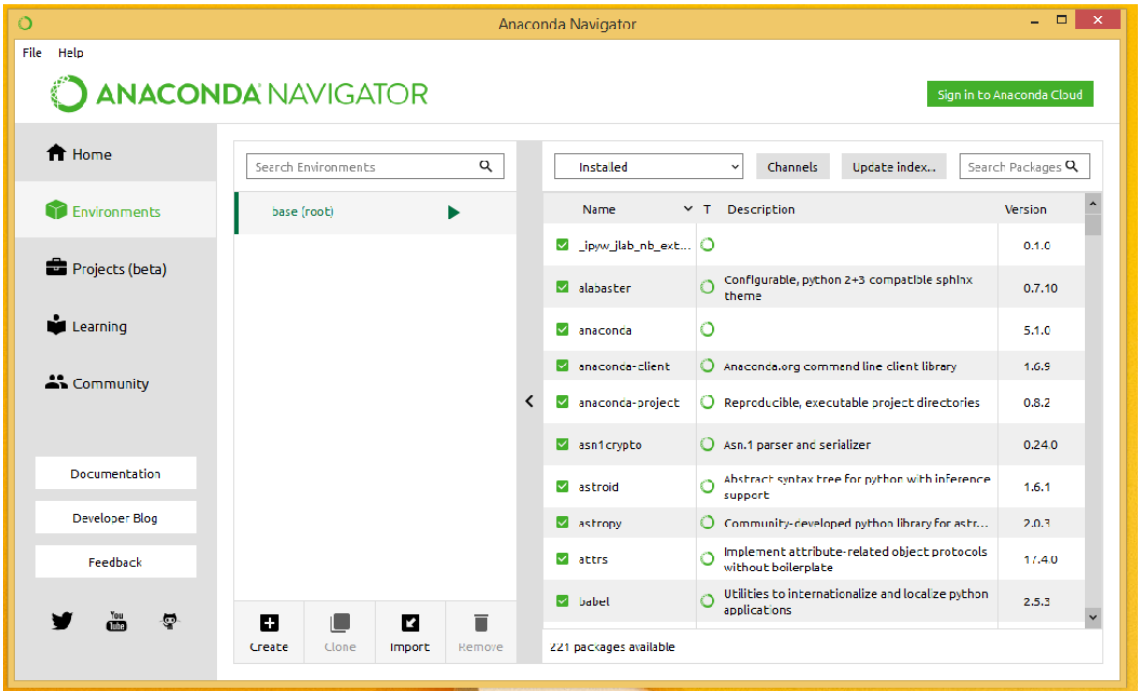
```
pip3 uninstall nombre_del_paquete
```

Entornos virtuales y paquetes en Anaconda

Si has instalado Python por medio de la distribución Anaconda, la información de esta página es para ti. Si no es el caso, puedes saltártela tranquilamente.

Anaconda tiene su propio sistema para gestionar tanto paquetes como entornos virtuales, que es bastante más cómodo que la línea de comandos.

Para acceder a él, debemos abrir Anaconda Navigator y, en el menú de la izquierdea, seleccionar el segundo apartado (“Environments”).



En la ventana que se nos muestra podemos ver dos secciones. La primera nos muestra qué entornos tenemos disponibles (Por defecto tenemos sólo uno, llamado “root”) y, además, pulsando el icono en forma de triángulo que el entorno tiene a la derecha de su nombre, nos permite iniciar cosas como un terminal o una sesión de Jupyter usando ese entorno.

Además, debajo de esa columna, unos botones nos permiten crear nuevos entornos, copiar uno que ya tengamos o importar uno nuevo.

A la derecha de esa columna, tenemos otra que nos indica qué paquetes tiene instalados el entorno que tenemos seleccionado. Un cuadro selector justo encima nos da varias opciones para seleccionar si queremos ver los paquetes instalados (es la opción por defecto), todos los paquetes disponibles, u otro subconjunto de estos.

A la derecha, hay un útil cuadro de “buscar”.

Si, por ejemplo, quisiéramos buscar un paquete nuevo para instalar, deberíamos seleccionar la vista de todos los paquetes disponibles en el selector y escribir el nombre del paquete en el buscador.

En la lista se nos mostrarán todos los que tiene un nombre parecido al que buscamos, y sólo necesitamos seleccionar el que queramos.

Se puede consultar la lista de paquetes disponibles para instalar por medio de Anaconda en su página web.

Tema 3

En este tema

Python es un lenguaje multiparadigma, pero fuertemente orientado a objetos. En Python solemos decir que "Todo son objetos", y eso tiene implicaciones profundas en cuanto a la forma en que podemos usarlo.

Las instancias son (obviamente) objetos. Y son objetos las listas o los diccionario, pero también son objetos los datos simples como los números o las cadenas de texto. E incluso son objetos las funciones o las propias clases.

Esto significa que sabiendo cómo funciona internamente los objetos y cómo cambiar su comportamiento podemos actuar sobre prácticamente todos los aspectos del lenguaje.

Por eso en este tema vamos a comenzar viendo los llamados "métodos mágicos", que nos permiten afectar al comportamiento de los objetos. Y veremos algunos casos concretos interesantes, como hacer objetos que se puedan usar en gestores de contexto, objetos que actúen como ejecutables u objetos iterables.

También veremos una forma eficiente y simple de crear listas y objetos generadores a partir de otros iterables.

Métodos mágicos

Consideremos la modesta pero útil función len().

La función len(), como ya sabemos retorna la longitud de lo que le pasemos como argumento.

Pero ¿Qué es la "longitud" de un objeto? Bueno, en realidad es el número de elementos de un contenedor, pero depende del objeto. Veamos este ejemplo:

```
cadena = "Hola holita"
lista = [1,2,3,4,5]
diccionario = {"uno": 1, "dos": 2}
numero = 42

print(len(cadena))
print(len(lista))
print(len(diccionario))
print(len(numero))
```

Si ejecutamos este ejemplo, veremos que la longitud de una cadena son los caracteres que tiene. La de una lista es el número de elementos, la de un diccionario es la cantidad de pares clave-valor y la de un número... es un error TypeError porque los números (según Python) no tienen longitud.

¿Cómo sabe Python qué es la longitud de cada cosa?

Pues gracias a un método especial que tienen algunos objetos, llamado __len__().

Esto es lo que llamaos un "Método mágico", y lo que lleva son dos guiones bajos delante y dos detrás. En adelante, todos los "Métodos mágicos" de los que vamos a hablar aquí son con dos guiones bajos delante y dos detrás. De hecho. también se les llama métodos "dunder", por "Double under". Pero "Métodos mágicos" mola más.

¿Significa esto que podemos crear nuestra propia clase y definirle la longitud que queramos? Sí. Significa justo eso. Veamos un ejemplo.

La clase más simple que podemos crear es la siguiente:

```
class MiClase:
    pass
```

Es decir: definimos la clase "MiClase" y no hacemos nada más. No agregamos métodos, ni atributos, ni nada.

Si instanciamos la clase y tratamos de medir la longitud del objeto tendremos el mismo error TypeError que vimos al intentarlo con los números, porque nuestro objeto no tiene longitud:

```
class MiClase:
    pass

objeto = MiClase()
print(len(objeto))
```

Para que tenga longitud debe tener un método __len__(), que será el que retorne esa longitud. Un ejemplo muy básico sería darle una longitud fija a todos los objetos de neutra clase, del siguiente modo:

```
class MiClase:
    def __len__(self):
        return 7
```

```
objeto = MiClase()
print(len(objeto))
```

Ahora todas las instancias de la clase MiClase tiene una longitud de 7, y es lo que retorna la función len() al aplicarla a ellas (y ya no hay TypeError).

Naturalmente, en nuestros programas reales lo normal será que el retorno de __len__() no sea un valor fijo, si no que se calcule en función de lo que nos interese a partir de los datos de nuestro objeto (en realidad, __len__() está pensado para que hagamos que nuestros contenedores retornen el número de elementos que contienen).

Seguramente ya has caído en la cuenta de que nuestro viejo y útil amigo __init__(), que es el que usamos para iniciar nuestros objetos es un método mágico, con sus dos guiones bajos delante y sus dos guiones bajos detrás.

Si __init__() es el método iniciador en Python, también tenemos un método finalizador, __del__(), que se ejecuta al destruir o finalizar un objeto (es decir, cuando no quedan referencias a ese objeto).

Y hay montones más.

Operadores

Por ejemplo, tenemos un método mágico (o, a veces más de uno) para cada operador:

__add__(self, Otro) para +

__sub__(self, Otro) para –

__mul__(self, Otro) para *

__truediv__(self, Otro) para /

__floordiv__(self, Otro) para //

__mod__(self, Otro) para %

__pow__(self, Otro) para ** (admite un tercer argumento opcional para el módulo)

Notarás que estos métodos, al contrario que __len__(), admiten un argumento además del "self". Esto es debido a que necesitan operar con otro objeto (es decir, si sumamos, sumamos una cosa a Otra). Veamos un ejemplo con la suma. Para

empezar, veamos la clase "Gente" que me acabo de inventar:

```
class Gente:
    def __init__(self, edad, nombre):
        self.edad = edad
        self.nombre = nombre

a = Gente(45, "Paco")
b = Gente(56, "María")
print(a + b)
```

Este ejemplo dará un error, porque el operando "+" no está definido para nuestra clase (es decir, no tenemos un método __add__()).

Ahora imaginemos que decido que la suma de dos personas es la suma de sus edades, para lo que haría esto:

```
class Gente:
    def __init__(self, edad, nombre):
        self.edad = edad
        self.nombre = nombre

    def __add__(self, otro):
        return self.edad + otro.edad

a = Gente(45, "Paco")
b = Gente(56, "María")
print(a + b)
```

O, supongamos que pienso que la suma de dos personas es la concatenación de sus nombres:

```
class Gente:

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

    def __add__(self, otro):
        return self.nombre + otro.nombre

a = Gente(45, "Paco")
b = Gente(56, "María")
print(a + b)
```

Aunque, bien pensado, creo que la suma de dos objetos de la clase Gente debería ser otro objeto de la clase Gente. Puedo hacer algo como esto:

```
class Gente:
```

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

    def __add__(self, otro):

        #Creamos un nuevo nombre a partir de los originales:
        nombre  = "{}-{} de Frankenstein".format(self.nombre[:2],
otro.nombre[:2])

        # Nos quedamos con la edad del mayor
        edad  = self.edad if self.edad > otro.edad else otro.edad

        return Gente(edad, nombre)

a = Gente(45, "Paco")
b = Gente(56, "María")

monstruo = a + b
print(monstruo.nombre)
print(monstruo.edad)
```

(Podeis imaginarme riendo como un maníaco y gritando "Está vivo").

Operadores con asignación

Las operaciones aritméticas con asignación también tiene sus métodos:

__iadd__(self, Otro) para +=
__isub__(self, Otro) para −=
__imul__(self, Otro) para *=
__itruediv__(self, Otro) para /=
__ifloordiv__(self, Otro) para //=
__imod__(self, Otro) para %=
__ipow__(self, Otro) para **= (admite un tercer argumento opcional para el módulo)

Dado que el operador con asignación normalmente hace la misma operación que el operador normal, estos métodos, en caso de no estar definidos, llaman al operador equivalente (es decir, __iadd__() usará __add__(), etc).

Comparación

Los operadores de comparación también tienen sus métodos mágicos:

__lt__(self, Otro) para <
__le__(self, Otro) para <=
__eq__(self, Otro) para ==
__ne__(self, Otro) para !=
__gt__(self, Otro) para >
__ge__(self, Otro) para >=

Estos métodos deben retornar True o False en función de sis se cumple la condición a la que corresponden o no. Por ejemplo, vamos Hacer que nuestra clase Gente compare a las personas por su edad salvo para el operador de igualdad, que siempre será cierto (porque la carta de derechos humanos dice que somos iguales) y el de desigualdad, que también será cierto siempre porque, ya se sabe, no hay dos personas iguales:

```
class Gente:
    def __init__(self, edad, nombre):
        self.edad = edad
        self.nombre = nombre

    def __add__(self, otro):

        #Creamos un nuevo nombre a partir de los originales:
        nombre  = "{}-{} de Frankenstein".format(self.nombre[:2],
otro.nombre[:2])

        # Nos quedamos con la edad del mayor
        edad  = self.edad if self.edad > otro.edad else otro.edad

        return Gente(edad, nombre)

    def __lt__(self, otro):
        return self.edad < otro.edad
    def __le__(self, otro):
        return self.edad <= otro.edad
    def __eq__(self, otro):
        return True
    def __ne__(self, otro):
        return True
    def __gt__(self, otro):
        return self.edad > otro.edad
    def __ge__(self, otro):
        return self.edad >= otro.edad
```

Nota que, como se ve en este ejemplo, podemos hacer que las comparaciones sean como queramos, incluso contradictorias.

Manejo de tipos y conversiones

Las operaciones de conversión de tipo también tiene sus métodos mágicos:

__int__(self) para la función int()
__float__(self) para la función float()
__complex__(self) para la función complex()
__oct__(self) para la función oct()
__hex__(self) para la función hex()
__index__(self) hace como __int__(self), pero aplicado a cuando el objeto se usa como índice o parte de un slice (debe retornar un entero).
__trunc__(self) para la función math.trunc()

También tenemos algunos métodos para los operadores unarios:

__pos__(self) para positivar (cuando hacemos +cosa).
__neg__(self) para negativizar (cuando hacemos -cosa)
__invert__(self) para invertir (cuando hacemos ~cosa).
__abs__(self) para la función abs()
__round__(self,n) para la función round()
__floor__(self) para la función math.floor()
__ceil__(self) para la función math.ceil()
__trunc__(self) para la función math.trunc()
__str__(self) para la función str()

Esta última (__str__()) es importante, porque es la representación en forma "humanamente legible" de nuestro objeto. Si este método no está implementado, al invocarlo se llamará a __repr__():

__repr__(self) Muestra la representación en cadena de texto de nuestro objeto, pero no orientado a ser leído por personas, sino a ser interpretable por el propio Python. En caso de que no esté implementado se hereda de la clase padre.
La implementación por defecto retornará algo como "<__main__.Gente object at 0x7fed25db3c88>", donde "__main__.Gente" es el módulo y la clase de nuestro objeto y "0x7fed25db3c88" su ID en hexadecimal.

Podemos agregar una representación en texto a nuestra clase así:

```
class Gente:
    def __init__(self, edad, nombre):
        self.edad = edad
        self.nombre = nombre

    def __add__(self, otro):

        #Creamos un nuevo nombre a partir de los originales:
```

```
        nombre  = "{}-{} de Frankenstein".format(self.nombre[:2],
otro.nombre[:2])

        # Nos quedamos con la edad del mayor
        edad  = self.edad if self.edad > otro.edad else otro.edad

        return Gente(edad, nombre)

    def __lt__(self, otro):
        return self.edad < otro.edad
    def __le__(self, otro):
        return self.edad <= otro.edad
    def __eq__(self, otro):
        return True
    def __ne__(self, otro):
        return True
    def __gt__(self, otro):
        return self.edad > otro.edad
    def __ge__(self, otro):
        return self.edad >= otro.edad

    def __str__(self):
        return "Persona llamada {} de {} años de edad".format(self.nombre,
self.edad)

persona = Gente(34, "Yolanda")

print(str(persona))
```

Gestores de contexto

La semana pasada, al hilo del uso de ficheros, comentamos los gestores de contexto.

Los gestores de contexto son esas estructuras que usábamos para abrir un archivo y, si es que se ha abierto correctamente, efectuar una serie de operaciones sobre él para luego cerrarlo.

Cada vez que tengamos un objeto en el que pueda resultarnos interesante hacer algo parecido (iniciar un contexto, hacer operaciones y cerrar el contexto) podemos hacerlo con una estructura del tipo:

```
with CONTEXTO as nombre:
    lo_que_sea
```

Para ello tenemos que definir en nuestro objeto dos nuevo métodos mágicos:

__enter__(self)
__exit__(self, exc_type, exc_value, exc_traceback)

El método __enter__() es el que se ocupa de iniciar el contexto (abrir un fichero, por ejemplo) y retornar el objeto que usaremos en ese contexto (el que asignaremos con "as", normalmente será el propio objeto) y el método __exit__() es el que gestiona la salida del contexto (cierra el fichero, por ejemplo) y además concreta lo que nuestro objeto hará en caso de error etc (por eso admite tantos argumentos). Si no hay erroes, los argumentos que reciba __exit__() serán todo None. Si ha habido algún error, contendrán la descripción de este:

exc_type: El tipo de excepción (Por ejemplo ZeroDivisionError)
exc_value La descripción de esta (Algo como 'integer division or modulo by zero')
exc_traceback El texto con la localización del error (Del estilo de 'File "mi_script.py", line 13, in __main__')

En estos tiempos de COVID podemos añadirle un gestor de contexto a nuestra clase Gente como este:

```
class Gente:
    def __init__(self, edad, nombre):
        self.edad = edad
        self.nombre = nombre

    def __add__(self, otro):

        #Creamos un nuevo nombre a partir de los originales:
        nombre  = "{}-{} de Frankenstein".format(self.nombre[:2],
otro.nombre[:2])

        # Nos quedamos con la edad del mayor
        edad  = self.edad if self.edad > otro.edad else otro.edad

        return Gente(edad, nombre)

    def __lt__(self, otro):
        return self.edad < otro.edad
    def __le__(self, otro):
        return self.edad <= otro.edad
    def __eq__(self, otro):
        return True
    def __ne__(self, otro):
        return True
    def __gt__(self, otro):
        return self.edad > otro.edad
    def __ge__(self, otro):
        return self.edad >= otro.edad

    def __str__(self):
        return "Persona llamada {} de {} años de edad".format(self.nombre,
self.edad)

    def __enter__(self):
        print("{} se pone la mascarilla".format(self.nombre))
```

```
        return self
    def __exit__(self, exc_type, exc_value, exc_traceback):
        print("{} se lava con gel hidroalcohólico".format(self.nombre))

with Gente(21, "José") as persona:
    print(persona)
```

De este modo, aseguramos la higiene antes y después de interaccionar con nuestro objeto.

Existen muchos más métodos mágicos, centrados en contextos más concretos como el acceso a métodos del objeto, trabajar directamente con clases, manejo de objetos que operan como listas etc. Cada vez que queramos cambiar la forma en que Python trabaja con nuestras clases, lo más probable es que haya uno o más métodos mágicos para ello. Puedes verlos en https://docs.python.org/3/reference/datamodel.html#special-method-names

En los siguientes capítulos veremos algunos más.

Objetos ejecutables

En Python, los paréntesis que hay detrás de una función son un operador, el operador de ejecución.

Es decir, en el siguiente código:

```
def saludo():
    return "Hola"

print(saludo())
print(saludo)
```

El primer print mostrará "Hola", porque imprime lo que retorna la función al ser ejecutado, pero el segundo mostrará algo como "<function saludo at 0x7f68a41641e0>", porque muestra la propia función (o, más exactamente, el resultado de su método __repr__())

Los objetos que responden al operador de ejecución se llama objetos "callables" (del inglés "call", "llamada").

El método __call__() es el que convierte los objetos de nuestra clase en "callables", esto es, en ejecutables como si fueran funciones. Al ejecutar el objeto, se llamará al método __call__().

Para crear objetos que emulen la función anterior, podemos crear una clase como esta:

```
class Saludo:
    def __call__(self):
        return "Hola"

saludo = Saludo()

print(saludo())
```

Naturalmente, __call__() es un método como otro cualquiera, y tiene acceso al resto de métodos y atributos de la clase. Esto nos permite hacer cosas muy interesantes:

```
class Incrementa:
    def __init__(self, numero);
        self.incremento = numero
    def __call__(self, valor):
        return valor + self.incremento

suma_uno = Incrementa(1)
suma_dos = Incrementa(2)

print(suma_uno(10))
print(suma_dos(10))
```

Con esta clase "Incrementa" estamos creando objetos que son ejecutables como funciones. Al instanciar cada objeto le pasamos como argumento un número que será el que, al ejecutar el objeto, se sume al que le pasemos.

En cierto modo, hemos convertido nuestra clase en una factoría que construye funciones a la medida.

Iteradores, generadores y funciones generadoras

Iteradores

Un iterador es un objeto capaz de retornar valores uno a uno y que pueden ser usados como un iterable (en un bucle for, por ejemplo). Concretamente, se trata de objetos que poseen los métodos mágicos __iter__() y __next__().

El método __iter__() es llamado por la función iter() y, en un iterador, normalmente retorna el propio objeto. El método __next__() es llamado por la función next() y debe retornar el siguiente elemento del iterable si lo hay, o una excepción

StopIteration cuando no quedan.

Veamos un ejemplo simple:

```
class Doble:
    def __init__(self):
        self.numero = 1
        self.maximo = 100
    def __iter__(self):
        return self
    def __next__(self):
        self.numero *= 2
        if self.numero > self.maximo:
            raise StopIteration
        else:
            return self.numero

for i in Doble():
    print(i)
```

Este iterador retorna a cada iteración un número el doble que el anterior (es decir, va retornando potencias de dos) hasta que sobrepasa el 100, que se detiene (y envía una interrupción StopIteration).

Le hemos puesto un límite para demostrar el uso de la interrupción, pero nada impide que un iterador retorne una sucesión infinita de resultados.

Esta es una de las ventajas de los iteradores: Podríamos haber obtenido el mismo resultado usando una función tradicional para crear una lista y usar esa lista en el bucle for, pero no podemos hacer una lista infinita.

Además de esto, los iteradores son muy eficientes respecto a las listas en cuanto al uso de memoria. Una lista necesita memoria para guardar sus elementos, y más memoria cuantos más elementos tenga. Pero un iterador no almacena todos los elementos, sino que los va retornando conforme son necesarios.

Una peculiaridad de los iteradores respecto a, por ejemplo, las listas, es que siempre recuerdan por donde vas. Si recorres parte de un iterador en un bucle y luego lo tratas de usar en otro bucle, continuaras por el siguiente elemento de la secuencia. Del mismo modo, un iterador se "agota" cuando llegamos a su último elemento.

Funciones generadoras

Como implementar los métodos __iter__() y __next__() (conocidos en conjunto como

el "protocolo generador") a veces puede ser un tanto complejo, Python dispone de una forma de hacer funciones que retornen un iterador.

Estas funciones se llaman "Funciones geradoras", y al tipo concreto de iterador que retorna se le llama "generador".

Alguna definiciones a grandes rasgos, porque los términos pueden liar un poco (además, la documentación de Python es un pelín inconsistente a este respecto):

Iterable: Algo de lo que se pueden sacar cosas una a una, y que se puede meter en un bucle for. Por ejemplo, una lista.

Iterador: Un objeto que tiene métodos __iter__() y __next__(). Un iterador es un iterable, pero no todos los iterables son iteradores.

Generador: Un iterador creado con una función generadora.

¿Cómo definimos u8na función generadora?

Para eso usamos la sentencia yield.

Vamos a comenzar con una función convencional, que retorne una lista de valores incrementados al doble como en el generador que hemos hecho más arriba:

```
def doble():
    lista = []
    numero = 1
    maximo = 50
    while numero < maximo:
        numero *= 2
        lista.append(numero)
    return lista

for i in doble():
    print(i)
```

Esta función retorna los mismos números que el generador de antes, pero no uno a uno, sino en forma de lista. La verdad es que es una forma horrible de crear una lista y en Python hay formas de hacer esto mucho mejor (y luego veremos algunas), pero nos vale como ejemplo.

Una función generadora (es decir, que retorne un generador) es muy parecida. Sólo necesitamos que retorne cada elemento (en lugar de toda una lista) usando la sentencia "yield" (en lugar de "return"):

```
def doble():
    numero = 1
```

```
    maximo = 50
    while numero < maximo:
        numero *= 2
        yield numero

for i in doble():
    print(i)
```

Como se puede ver, para convertir la función en una generadora, sólo hemos eliminado la lista (porque ya no la necesitamos) y reemplazado el lugar donde añadíamos un elemento a esta por un yield con ese elemento. Además, tampoco nos hace ya falta ese "return".

En realidad, pese a la apariencia, esta función no retorna los valores del yield uno a uno. Lo que retorna es un generador exactamente igual que el que hicimos al principio, que es el que retorna esos valores. Pero es una forma mucho más simple de programarlo y se ve mucho más claro lo que hace.

Comprensión

Listas por comprensión

Supongamos que tenemos una lista con los días de la semana en minúscula, y queremos crear, a partir de ella, otra que tenga la primera letra de cada día en mayúscula.

Podríamos hacerlo con un bucle for, como ya sabemos:

```
semana = ["lunes","martes","miércoles","jueves","viernes","sábado","domingo"]

semana_mayus = []

for dia in semana:
    semana_mayus.append(dia.capitalize())

print(semana_mayus)
```

En este ejemplo hemos creado una lista vacía (semana_mayus) y luego, usando un bucle for, hemos recorrido la lista original para aplicar una transformación a cada elemento de la lista (el método capitalize()) y añadir el resultado a la lista vacía por medio de append().

Supongamos ahora que tenemos una lista de números y queremos crear a partir de ella otra que contenga los mismos elementos, pero multiplicados por dos:

```
lista_original = [1,2,3,4,5,6,7,8,9]
lista_nueva = []
```

```
for x in lista_original:
    lista_nueva.append(x*2)
print(lista_nueva)
```

El procedimiento, como puede verse, es prácticamente el mismo aunque, naturalmente, en este caso concreto sería mucho mejor usar la función range, que nos permite evitar el uso de la lista original:

```
lista = []
for x in range(1,10):
    lista.append(x*2)

print(lista)
```

Esta estructura (crear una lista recorriendo los elementos de otra y haciendo alguna operación sobre ellos) es tan habitual que Python dispone de una sintaxis especial para hacerla más cómoda. Es lo que se conoce como "listas por comprensión".

El formato básico de una lista por comprensión se podría esquematizar de este modo:

```
lista = [OPERACIÓN for VARIABLE in ITERABLE]
```

Las listas por comprensión se declaran entre corchetes (después de todo, retornan una lista). Dentro de esos corchetes se define el bucle for de un modo muy parecido al que estamos acostumbrados (pero sin terminar en dos puntos) precedido de la operación que queremos aplicar a cada uno de los valores que retorne el bucle, y cuyo resultado será uno de los elementos de la lista final.

Veamos los mismos ejemplos que hemos usando arriba pero, esta vez, usando listas por comprensión:

```
semana = ["lunes","martes","miércoles","jueves","viernes","sábado","domingo"]
semana_mayus = [dia.capitalize() for dia in semana]
print(semana_mayus)

lista_original = [1,2,3,4,5,6,7,8,9]
lista_nueva = [x*2 for x in lista_original]
print(lista_nueva)

lista = [x*2 for x in range(1,10)]
print(lista)
```

Al igual que en el caso de los bucles for, se puede usar cualquier iterable para crear una lista por comprensión.

Por ejemplo, se puede definir una lista por extensión a partir de un dicionario:

```
dicc = {"a": "azul", "b": "blanco", "c": "cian", "d": "dorado"}
lista = [v.upper() for v in dicc.values()]
print(lista)
```

Se pueden filtrar los resultados indicando condiciones que deben cumplir con un if de este modo:

```
lista = [OPERACIÓN for VARIABLE in ITERABLE if CONDICIÓN]
```

Por ejemplo, para obtener una lista con los cuadrados de los números menores que 100 que sean divisibles por siete:

```
lista = [x ** 2 for x in range(1,100) if x % 7 == 0]
print(lista)
```

Generadores por comprensión

Además de las listas, podemos usar este mismo método obtener generadores.

El formato es exactamente el mismo, solo que usando paréntesis en lugar de corchetes:

Por ejemplo, podemos hacer los mismos ejemplos anteriores, pero creando generadores en lugar de listas:

```
semana = ["lunes","martes","miércoles","jueves","viernes","sábado","domingo"]
generador_mayus = (dia.capitalize() for dia in semana)

lista_original = [1,2,3,4,5,6,7,8,9]
generador_nuevo = (x*2 for x in lista_original)

generador_pares = (x*2 for x in range(1,10))

generador = (x ** 2 for x in range(1,100) if x % 7 == 0)
```

Tema 4

En este tema

Programación funcional

La programación funcional es un paradigma de programación (algo así como un estilo de programación) basado en el uso casi exclusivo de funciones. La programación funcional evita el uso de estructuras de control como los bucles (reemplazándolos por el uso de la recursividad, por ejemplo) y hace uso extensivo de funciones de orden superior (esto es: funciones que pueden aceptar una o más funciones como entrada y/o retornar una función como salida) como map o zip.

Python no es un lenguaje orientado a la programación funcional y, ni mucho menos, es un lenguaje funcional puro. Python es un lenguaje muy orientado a objetos, pero también es multiparadigma, y nos va apermitir sabar utilidad de los conceptos de la programación funcional.

La programación funcional es un concepto bastante complejo https://en.wikipedia.org/wiki/Functional_programming cuyas profundidades se escapan de los objetivos de este curso, pero veremos lo fundamental en lo que queda de tema.

Para empezar, como ya sabemos, en Python todo son objetos. Incluidas las funciones.

Esto significa que podemos, por ejemplo, asignar una función a una variable (o, lo que es lo mismo, darle un nombre) como a cualquier otro valor u objeto de Python:

```
imprime = print
imprime(“Hola Mundo”)
```

Aquí hemos asignado la función print() a la variable imprime (lo que es equivalente a decir que hemos creado un nuevo nombre para la función print()). No hemos llamado a la función con print(), sino que hemos puesto print sin el paréntesis final (el operador de ejecución, como ya vimos), con lo que esto nos retorna la propia función y no el resultado de su ejecución.

De este modo, podemos ejecutar “imprime” exactamente como si fuese la función print.

¿Tiene esto alguna utilidad? Sí, y lo vamos a ver en breve.

Funciones que actúan como bucles

Python tiene una serie de funciones cuyo uso es más o menos el mismo: Tomar una función y usarla de algún modo sucesivamente sobre los elementos de un iterable. Pueden ser muy útiles en contextos donde normalmente se usaría un bucle y, a veces pueden resultar más convenientes que estos porque evitan usar accesorios como contadores o variables temporales donde almacenar resultados intermedios.

También, a menudo, se pueden emplear para hacer el mismo trabajo que la comprensión. En estos casos, normalmente, es preferible esta última por claridad y eficiencia.

filter()

filter() aplica una función a los elementos de un iterable y retorna un iterador con sólo aquellos elementos para los que la función haya retornado True. Por ejemplo, hagamos un filtro para seleccionar sólo los elementos de una tupla que estén completamente en mayúsculas:

```
palabras = ("a", "SI", "Hola", "HOLA")

def es_mayus(texto):
    if texto == texto.upper():
        return True
    else:
        return False

print(list(filter(es_mayus, palabras)))
```

filterfalse()

El módulo "itertools" tiene la función filterfalse(), que funciona exactamente igual que filter(), pero sólo retorna los elementos para los que la función pasada como argumento haya retornado False:

```
palabras = ("a", "SI", "Hola", "HOLA")

from itertools import filterfalse
```

```
def es_mayus(texto):
    if texto == texto.upper():
        return True
    else:
        return False

print(list(filterfalse(es_mayus, palabras)))
```

map()

map() es una función que toma como argumentos una función y uno o más iterables, y retorna un iterador con el resultado de aplicar a esa función los elementos de los iterables como argumentos.

Veamoslo con un ejemplo:

```
lista1 = [1,2,3,4,5]

def duplica(n):
    return n * 2

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

lista2 = list(map(duplica,lista1))

for i in map(suma, lista2, lista1):
    print(i)
```

En este ejemplo hemos usado map() dos veces. La primera para duplicar todos los valores de la lista y (usando list()) crear otra lista con ellos. Como la función duplica() sólo toma un argumento, a map() le pasamos como argumentos la función y un solo iterable.

La segunda vez tomamos las dos listas y las sumamos con la función suma(). Como la función suma() toma dos argumentos, a map() le pasamos dos iterables además de la función.

En general, cuando tenemos que hacer un bucle para recorrer un iterable aplicando una función a cada uno de sus elementos, probablemente sea mejor usar map().

zip()

Aunque no usa funciones como argumentos como el resto que estamos viendo,

zip() se usa en mochos contextos junto con el resto de funciones de este tema. zip() toma varios iterables como argumento y retorna un iterador de tuplas. la primera de las tuplas estará formada por los primeros elementos de los iteradores, la segunda por los segundos, etc.

Por ejemplo:

```
letras = "abcdefghijklmnñopqrstuvwxyz"
numeros = [1,2,3,4,5,6,7,8,9]

print(zip(letras, numeros))
```

Como se puede ver en este ejemplo, la longitud del iterador resultante se trunca al más corto de los iterables originales.

reduce()

reduce() es otra función que toma como argumentos una función y un iterable. Es una función del módulo "functools", así que es necesario importar ese módulo completo o la función concreta antes de usarla.

La función que le pasemos a reduce() debe tomar dos argumentos. reduce() tomará los dos primeros elementos del iterable y los usará en la función. Después tomará el resultado y lo usará junto con el siguiente elemento del iterable, y así sucesivamente, retornando el valor final resultante.

Por ejemplo:

```
from functools import reduce
numeros = [1,2,3,4,5,6,7,8,9]

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

print(reduce(suma, numeros))
```

Además, reduce() admite un tercer argumento opcional. Si se le suministra, en lugar de tomar los dos primeros elementos del iterable en el primer paso, usará el valor de este argumento junto con el primer elemento del iterable.

Funciones que trabajan con funciones

Naturalmente, dentro de una función se pueden utilizar llamadas a otras funciones sin problema y, de hecho, suele ser lo normal:

```
def dobla(numero):
    return numero * 2

def cuadruplica(numero):
    return dobla(dobla(numero))
```

En el ejemplo de arriba vemos que la función "cuadruplica" usa en su cuerpo la función "dobla" que hemos definido previamente. De hecho, se limita a llamar dos veces a la función "dobla" (es decir, duplica el numero pasado como argumento, luego lo vuelve a duplicar y retorna el resultado).
Dado que dobla(numero) retorna el doble del número que le indiquemos, podemos usar esa expresión como argumento de otra función (otra llamada a "dobla", en nuestro caso) para multiplicar por cuatro el valor inicial.

Más interesante es que una función se puede llamar a sí misma en su propio cuerpo. Aunque es algo que hay que hacer con cuidado porque, al igual que vimos con los bucles, corremos el peligro de caer en un bucle infinito. Esto (que una función se llame a sí misma) es a lo que llamamos recursividad, y probablemente sea más fácil de ver con un ejemplo simple:

```
def cuenta_atras(numero):
    if numero >= 0:
        print(numero)
        cuenta_atras(numero - 1)
cuenta_atras(10)
```

En estos ejemplos las funciones están usando los retornos de otras funciones, pero, igual que hacemos con valores, listas o cualquier otro objeto, una función puede operar con otra función (o funciones), aceptarla como argumento o retornarla como resultado.

Por ejemplo, veamos esta pequeña función que acepta otra función como argumento e imprime algo de información sobre ella:

```
def inspecciona_funcion(fun):
    print('Esta es la función {}'.format(fun.__name__))
    print('Su ID es {}'.format(id(fun)))
    print('Y esta es su documentación:\n{}'.format(fun.__doc__))

inspecciona_funcion(print)
```

En este ejemplo podemos ver cómo, para referirnos a la función print(), hemos indicado su nombre sin añadirle detrás los paréntesis "()", como hicimos cuando la

asignamos a una variable. Como ya hemos visto, los paréntesis de una función (o de cualquier otro objeto callable) son el operador de llamada, que es el que indica que ese objeto debe ejecutarse. Para referirnos a la función en sí misma debemos indicarla sólo por su nombre, sin esos paréntesis.

Usamos los atributos __name__ y __doc__ (las funciones son objetos, y tienen atributos) para mostrar el nombre de la función y su docstring. También hemos usado la función id para obtener su identificador.

Pero, normalmente, cuando pasamos una función como argumento lo hacemos para usarla de algún modo. Veamos este otro ejemplo, en el que vamos a crear varias funciones:

```
def suma(num1, num2):
    print(num1 + num2)

def resta(num1, num2):
    print(num1 – num2)

def aplica(fun, num1, num2):
    fun(num1, num2)

aplica(suma, 4, 2)
aplica(resta, 4, 2)
```

En este ejemplo hemos creado dos funciones bastante triviales, suma y resta, que admiten dos argumentos cada una y hacen precisamente lo que dice su nombre; y una tercera función llamada aplica que es la que nos interesa aquí.

La función aplica admite tres argumentos, el primero debe ser una función y los otros dos son los valores sobre los que se aplicará esa función. Si se llama a la función aplica pasándole suma como primer argumento, sumará los valores de los otros argumentos; si se le pasa la función resta, los restará.

Las funciones filter() y map() que hemos visto en capítulos anteriores son ejemplos de funciones que aceptan otras funciones como argumento.

Recordemos que la función map(), concretamente, es una herramienta muy útil que toma una función y un iterable y retorna una lista de elementos, cada uno de los cuales es el resultado de aplicar la función al elemento correspondiente del iterable original. Vamos a hacer una versión simple de esta función:

```
def mi_map(fun, iterable):
    li = list()
    for i in iterable:
        li.append(fun(i))
    return li

def dobla(x):
```

```
    return 2 * x
```

print(mi_map(dobla,[1, 2, 3]))

Pero las funciones no sólo pueden pasarse como argumento. También pueden retornarse como resultado de otra función, aunque esto es un poco más complejo:

```
def externa():
    def interna():
        print('Hola Mundo')
    return interna

hola = externa()
hola()
```

En este ejemplo, una función llamada externa contiene la declaración de otra función de nombre interna (la cual, simplemente, imprime un “hola mundo”). Además, una sentencia return devuelve esta función.

Si asignamos el resultado de ejecutar externa a la variable hola, esta contiene la función interna, de modo que podemos usarla como esa función añadiéndole los paréntesis del operador de llamada.
Es importante recordar que el operador de llamada (los paréntesis al final del nombre de la función) es lo que hace que la función retorne un valor en lugar de la propia función.

En programación, a los elementos del lenguaje que pueden ser asignados, devueltos y manipulados libremente como cualquier otro tipo de dato se les llama “objetos de primera clase”. Esto quiere decir que, en Python, las funciones son objetos de primera clase.

En el ejemplo anterior la función retornada (y almacenada en la variable hola) no admite argumentos, pero podemos retornar una función que use argumentos del mismo modo:

```
def externa():
    def interna(nombre):
        print('Hola Mundo ' + nombre)
    return interna

hola = externa()
hola('Pablo')
```

Tener una función que nos devuelve otra función puede servirnos, por ejemplo, para crear algo así como una factoría de funciones que nos permita crear distintas funciones dependiendo de los argumentos que le asignemos a la factoría. Por

ejemplo:

```
def factoria(saludo):
    def interna(nombre):
        print(saludo + ' ' + nombre)
    return interna

saluda = factoria('Hola')

saluda2 = factoria('Saludos')

saluda('amigo')
saluda2('amigo')
```

En este caso, hemos construido una función factoria() que retorna otra función que imprimirá un saludo. Es importante fijarse en que la función factoria() admite un argumento (que se usará para construir el saludo) y que la función interna que retorna admite otro argumento (que usarán las funciones creadas para poner el nombre al que se saluda).

Pero existe un caso especial de funciones que retornan funciones que resulta muy interesante, y es el de las llamadas clausuras, que veremos en el próximo capítulo de este tema.

Clausuras

Las clausuras (del inglés closures), a veces llamadas también cerraduras, son funciones que definimos dentro de otras funciones, como las que hemos visto, pero que se usan para producir funciones con su propio entorno léxico.

¿Qué significa eso de "con su propio entorno léxico? Recordemos un par de cosas:

Las variables definidas dentro de una función no pueden ser accedidas desde fuera de esta (a menos que las marquemos como globales con la sentencia global).

Además, las variables definidas dentro de una función sólo existen mientras se ejecuta esa función; en el momento en que termina de ejecutarse, son eliminadas, por lo que, en principio, no es posible conservar un valor entre llamadas a la función usando variables definidas dentro de ella (para ello hace falta usar variables definidas fuera de la función como, por ejemplo, variables globales).

Una clausura es una función construida de modo que puede acceder a variables propias que conservan su valor incluso entre llamadas a la función.

Veamos un ejemplo simple:

```
def clausura():
    cosa = 0
    def interna():
        nonlocal cosa
        cosa += 1
        print(cosa)
    return interna

mi_funcion = clausura()

mi_funcion()
mi_funcion()
mi_funcion()
mi_funcion()
mi_funcion()
```

En este ejemplo hemos creado una función llamada clausura que define una variable llamada cosa y una función llamada interna, y luego retorna esa función como hemos visto en los ejemplos anteriores.

La función interna usa esa misma variable cosa, declarándola como nonlocal, lo que, como recordaremos, hace que el nombre así declarado se refiera a una variable ya existente en un ámbito superior (en este caso, la función clausura).

Lo único que hace esta función es incrementar en una unidad esa variable cosa y mostrar su valor en pantalla.

Si asignamos el resultado de ejecutar clausura a la variable mi_funcion() y después ejecutamos esta varias veces, vemos algo muy interesante:

Normalmente esperaríamos que, dado que una función no recuerda el valor de sus variables entre llamadas, cada vez que llamemos a mi_funcion() obtengamos el mismo resultado (el valor 1). Sin embargo, lo que ocurre es que el valor retornado se incrementa entre llamadas a la función.

De algún modo, el valor de cosa se está recordando entre llamadas a la función.

La clave de esto es la forma en la que hemos declarado la variable cosa. Al declararla en la función exterior (clausura en nuestro ejemplo) y luego declararla con nonlocal en la función interior (interna en este ejemplo), Python guarda su valor entre llamadas. Para ello, todas las variables que se hayan declarado de este modo son almacenadas en forma de tupla en el atributo __closure__ de la función.

```
def clausura():
    cosa = 0
    otra_cosa = 'hola'
    def interna():
        nonlocal cosa
        nonlocal otra_cosa
        cosa += 1
```

```
    print(cosa)
    return interna

mi_funcion = clausura()
print(mi_funcion.__closure__)
```

Las clausuras son una forma muy interesante de crear funciones con datos asociados a ellas y con una especie de espacio de nombres exclusivo. Son también una forma de evitar el uso de las desaconsejadas variables globales. En cierto modo, esta es una forma de construir algo parecido a objetos sólo con funciones.

Por otro lado, si necesitamos usar muchas variables de este modo, probablemente sea mejor crear una clase con los atributos necesarios en lugar de una clausura.

Decoradores

Si podemos crear funciones que toman funciones como argumento y podemos crear funciones que retornen funciones, entonces podemos crear funciones que tomen unas funciones y retornen otras funciones (no, no es un trabalenguas):

```
def modifica(func):

    def interna():
        print('~~~~~~~~~~')
        func()
        print('~~~~~~~~~~')

    return interna

def mi_funcion():
    print('Hola mundo')

otra_funcion = modifica(mi_funcion)

otra_funcion()
```

En este ejemplo hemos hecho una función modifica() que acepta una función como argumento y retorna otra función a partir de ella, que simplemente añade sendas líneas "~~~~~~~~~~" antes y después de esta. Además, hemos creado una función llamada mi_funcion que, simplemente, imprime el texto "Hola mundo".

Si pasamos mi_funcion como argumento a modifica() obtenemos una nueva función obtenida a partir de la primera.

Una interesante utilidad de esto es crear una especie de filtro que pueda transformar una función para añadirle alguna utilidad adicional. A esto se le llama "decorar" una función y a la función que usamos como filtro se le llama "decorador". Veamos un ejemplo basado en el anterior:

```
def decorador(func):

    def interna():
        print('~~~~~~~~~~')
        func()
        print('~~~~~~~~~~')

    return interna

def mi_funcion():
    print('Hola mundo')

print('Antes de decorar:')
mi_funcion()

mi_funcion = decorador(mi_funcion)

print('Despues de decorar:')
mi_funcion()
```

Aquí hemos hecho casi lo mismo que antes, pero esta vez sobreescribimos mi_funcion() con el resultado de pasar esa misma función por decorador.

Esto puede ser muy útil cuando, por ejemplo, necesitamos agregar una misma característica a varias funciones diferentes. Se puede usar un decorador para programar esa característica y añadirlo a cada una de ellas del modo que hemos visto.

Existe una notación especial que nos permite indicar la aplicación de un decorador de forma más simple y clara usando el nombre del decorador precedido del símbolo arroba (@) justo encima de la declaración de la función a decorar, del siguiente modo:

```
def decorador(func):

    def interna():
        print('~~~~~~~~~~')
        func()
        print('~~~~~~~~~~')

    return interna

@decorador
def mi_funcion():
    print('Hola mundo')

mi_funcion()
```

En este ejemplo puede verse como, en realidad, poner @decorador antes de la declaración de la función def mi_otra_funcion(): es exactamente lo mismo que declarar la función y después hacer mi_funcion = decorador(mi_funcion); se trata de una forma más breve y clara de escribirlo.

Se pueden acumular varios decoradores y aplicarlos uno sobre otro:

```
def decorador(func):

    def interna():
        print('~~~~~~~~~~')
        func()
        print('~~~~~~~~~~')

    return interna

def otro_decorador(func):

    def interna():
        print('**********')
        func()
        print('**********')

    return interna

@decorador
@otro_decorador
def mi_funcion():
    print('Hola mundo')

mi_funcion()
```

Como puede verse, primero se aplica sobre la función el decorador más próximo a ella (otro_decorador en este caso) y luego los siguientes que pudiera haber por orden de proximidad (es decir, desde abajo hacia arriba).

Dado que la función que retorna el decorador va a sustituir a la función "decorada", normalmente ambas deben aceptar los mismos argumentos y retornar el mismo tipo de resultado.

Los decoradores son una herramienta muy útil cando tratamos de crear utilidades que afecten a distintas funciones o métodos (recordad que un método no es más que una función un poco especial dentro de un objeto).

De hecho, Python trae ya "de fábrica" varios decoradores, de los que veremos alguno más adelante.

Funciones Lambda

Un caso especial de funciones de uso mucho más limitado son las funciones Lambda, también llamada funciones anónimas.

Se trata de funciones simples de una sola expresión que pueden ser utilizadas in situ.

Las funciones lambda se declaran con la palabra reservada “lambda” del siguiente modo:

```
lambda ARGUMENTOS : EXPRESIÓN
```

Donde ARGUMENTOS son los argumentos que admite la función y EXPRESIÓN es la expresión que se corresponde con la función en sí, y cuyo resultado será el valor retornado. Por ejemplo:

```
lambda alto , ancho : alto * ancho
```

Las funciones lambda no usan returrn, y están limitadas a una única sentencia (cuyo resultado es lo que se retorna, como hemos visto) por lo que no pueden ser demasiado complejas. Además, su abuso tiende a hacer el código poco legible, por lo que no se aconseja su uso extensivo.

Pero ¿Cómo usaríamos esta función?

La primera forma es dándole un nombre, lo que nos permite usarla como cualquier otra función:

```
superficie = lambda alto , ancho : alto * ancho
resultado = superficie(3,4)
```

El código anterior sería exactamente lo mismo que hacer lo siguiente:

```
def superficie(alto , ancho):
    return alto * ancho

resultado = superficie(3,4)
```

La segunda forma de usarla es directamente, in situ:

```
resultado = (lambda alto , ancho : alto * ancho)(3,4)
```

Es importante notar que usamos un paréntesis para agrupar la función lambda y, después, un segundo paréntesis (que es el operador de ejecución) en el que le pasamos los valores.

Pero el uso más habitual es como argumento para otras funciones (como map() u order()) o para ser retornado por funciones de orden superior.

Por ejemplo, como vimos antes, la función map() toma una función y un iterable y

retorna un iterador de elementos, cada uno de los cuales es el resultado de aplicar la función al elemento correspondiente del iterable original. Normalmente la usaríamos de este modo:

```
def adorna(cadena):
    return '*' + cadena + '*'

nombres = ['Ana','Benito','Carolina','David']

for nombre in map(adorna,nombres):
    print(nombre)
```

Pero podemos usar una función lambda para hacer lo mismo así:

```
nombres = ['Ana','Benito','Carolina','David']

for nombre in map(lambda x: '*' + x + '*',nombres):
    print(nombre)
```

Naturalmente, esto sólo tiene sentido si la función que necesitamos usar no es demasiado compleja, y sólo si no vamos a usarla en varios lugares distintos (en cuyo caso, necesitamos una función que tenga nombre para poder reutilizarla). En general, es posible usar una función lambda donde normalmente usaríamos el nombre de una función que hubiéramos definido previamente.

Tema 5

En este tema

Este es un tema muy breve, dedicado a repasar la herencia de clases en Python y a aclarar algunas cuestiones más avanzadas, como la forma de poder usar métodos heredados modificándolos y algunos aspectos de la herencia múltiple.

Herencia

En Python, como sabemos, unas clases pueden heredar de otras:

```
class Monstruo:
    def __init__(self, nombre):
        self.nombre = nombre
    def muerde(self):
        print("El monstruo {} te muerde. Duele".format(self.nombre))

class Vampiro(Monstruo):
    pass

m1 = Vampiro("Drácula")
m1.muerde()
```

En este ejemplo, la clase Vampiro hereda de la clase Monstruo, de modo que, aunque no le hemos definido ningún método, tiene los mismos que esa (__init__ y muerde).

Como ya sabemos, podemos añadir métodos y atributos adicionales a la clase hija que no estén en su clase madre. También podemos sobreescribir los métodos de la clase madre. Si hacemos un método nuevo con el mismo nombre, se usará ese en lugar de heredado:

```
class Monstruo:
    def __init__(self, nombre):
        self.nombre = nombre
    def muerde(self):
        print("El monstruo {} te muerde. Duele".format(self.nombre))

class Vampiro(Monstruo):
    def muerde(self):
        print("El vampiro {} te chupa la sangre".format(self.nombre))

m1 = Vampiro("Drácula")
m1.muerde()
```

Ahora, al llamar al método muerde() de la clase Vampiro, se usará el definido en la propia clase.

Pero podemos crear una nueva clase que sea heredera de Vampiro:

```
class Monstruo:
    def __init__(self, nombre):
        self.nombre = nombre
    def muerde(self):
        print("El monstruo {} te muerde. Duele".format(self.nombre))

class Vampiro(Monstruo):
    def muerde(self):
        print("El vampiro {} te chupa la sangre".format(self.nombre))

class VampiroVegano(Vampiro):
    def muerde(self):
        print("El vampiro {} muerde una cebolla".format(self.nombre))

m1 = VampiroVegano("Drácula Vegano")
m1.muerde()
```

La nueva clase hereda los métodos y atributos que no tenga definidos de su clase madre. Y si su clase madre no los tiene, los hereda de la clase madre de su clase madre (lo que vendría a ser su clase abuela) y así sucesivamente.

Pero hasta aquí las clases han heredado los métodos tal cual. Podemos heredarlos o podemos reescribirlos completamente. No podemos heredarlos y modificarlos.

Modificando la herencia

Supongamos que quiero que el método __init__ de Vampiro defina, además del atributo nombre, el atributo titulo_nobiliario. Pero supongamos que quiero hacerlo sin reescribir todo el método.

Puedo llamar al método de la clase madre del modo que vimos en la clase remota de la semana pasada:

```
class Monstruo:
    def __init__(self, nombre):
        self.nombre = nombre
    def muerde(self):
        print("El monstruo {} te muerde. Duele".format(self.nombre))

class Vampiro(Monstruo):
```

```
    def __init__(self, nombre, titulo_nobiliario):
        self.titulo_nobiliario = titulo_nobiliario
        Monstruo.__init__(self, nombre)

    def muerde(self):
        print("El vampiro {} te chupa la sangre".format(self.nombre))

m1 = Vampiro("Drácula", "Conde")
m1.muerde()
```

Con esto, estamos iniciado el valor de self.nombre a través del método de la clase Monstruo.

Pero esto, técnicamente, no es tomar el método de la clase madre. Estamos tomando el método de la clase Monstruo. Llamar a la clase madre por su nombre no es muy recomendable (salvo que por alguna razón queramos llamar al método de esa clase concreta). Por ejemplo, las clases que sean hijas o nietas de la clase vampiro seguirán heredando el método de la clase Monstruo, no de sus propias clases madre, que puede no ser lo que queramos.

La función Super()

La función super() nos permite acceder a los métodos y atributos de la clase madre, aunque hayan sido sobreescritos.

Por ejemplo, nuestro ejemplo anterior podríamos hacerlo así:

```
class Monstruo:
    def __init__(self, nombre):
        self.nombre = nombre
    def muerde(self):
        print("El monstruo {} te muerde. Duele".format(self.nombre))

class Vampiro(Monstruo):

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

    def muerde(self):
        print("El vampiro {} te chupa la sangre".format(self.nombre))

m1 = Vampiro("Drácula", "Conde")
m1.muerde()
```

Es importante fijarse que, al llamar al método __init__ con super(), ya no tenemos que pasarle el argumento self, como sí teníamos que ahcer al llamar a la clase con su nombre.

Herencia mútiple

En Python, las clases pueden heredar de más de una clase. Llamamos a esto "Herencia múltiple".

Por ejemplo, la clase HombreLobo puede heredar de Monstruo, pero también hereda de la clase Mascota:

```
class Monstruo:
    def __init__(self, nombre):
        self.nombre = nombre
    def muerde(self):
        print("El monstruo {} te muerde. Duele".format(self.nombre))

class Mascota:
    def pica(self):
        print("{} se rasca la oreja con la pata trasera".format(self.nombre))

class HombreLobo(Monstruo, Mascota):
    pass

m1 = HombreLobo("Lupin")
m1.muerde()
m1.pica()
```

Como vemos se puede heredar al mismo tiempo de dos o más clases madre. Pero ¿Qué pasa si un mismo método está definido en más de una de las clases madre? Que se hereda el de la primera que haya llamado en los argumentos al definir la clase (la de más a la izquierda):

```
class Monstruo:
    def __init__(self, nombre):
        self.nombre = nombre
    def muerde(self):
        print("El monstruo {} te muerde. Duele".format(self.nombre))

class Mascota:
    def pica(self):
        print("{} se rasca la oreja con la pata trasera".format(self.nombre))
    def muerde(self):
        print("La mascota {} te muerde. Duele un poco".format(self.nombre))

class HombreLobo(Monstruo, Mascota):
    pass

m1 = HombreLobo("Lupin")
m1.muerde()
m1.pica()
```

Dado que hemos creado la clase mascota con class HombreLobo(Monstruo,

Mascota), se usarán los métodos de la clase Monstruo con preferencia a los de la clase Mascota.

Pero hay más.

Porque nuestra clase puede heredar de varias clases, que a su vez heredan de varias clases, algunas de las cuales son comunes a varias. Es decir, algunas abuelas por parte de una clase pueden coincidir con otras abuelas por parte de otra madre, o incluso que algunas abuelas por parte de una madre coincidan con bisabuelas por parte de otra, o más complicado...

EL problema del diamante

En el esquema más simple, si una clase (D) hereda de otras dos (B y C) que a su vez, heredan de la misma (A) ¿A través de cual de las madres se hereda un método de la abuela? Es decir, ¿En qué orden se busca en las clases? Este problema, que se llama "El problema del diamante" es común a todos los lenguajes que admitan herencia múltiple.

(Puedes ver más sobre este problema del diamante en la wikipedia, de donde he copiado la imagen con el esquema de las clases)

Para resolver esto, Python sigue varios pasos:

Python crea una lista, cuyo primer elemento es la propia clase (D).

Añade a esa lista la primera clase madre (B)

Añade a esa lista, si la hay, la primera clase madre de la que acaba de agregar (A)

Añade la sigueinte clase madre (C)

Añade la primera clase madre de la que acaba de agregar (A, sí, otra vez)

Cuando ha terminado de recorrer todas las clases, tendrá una lista que puede tener montones de repeticiones. En nuestro caso, es DBACA.

Python elimina de esta lista todas las entradas repetidas de cadsa clase, salvo la última, de modo que nuestra lista qued así: DBCA.

Para buscar un método, se recorrerá esa lista en orden, dando prefereinca a los de la izquierda sobre los de la derecha.

Esta lista se almacena en el atributo de clase __mro__ (ojo: es un atributo de la clase, no de la instancia), de donde puede ser consultada:

```
class Monstruo:
    def __init__(self, nombre):
        self.nombre = nombre
    def muerde(self):
        print("El monstruo {} te muerde. Duele".format(self.nombre))

class Mascota:
    def pica(self):
        print("{} se rasca la oreja con la pata trasera".format(self.nombre))
    def muerde(self):
        print("La mascota {} te muerde. Duele un poco".format(self.nombre))

class HombreLobo(Monstruo, Mascota):
    pass

print(HombreLobo.__mro__)
```

Tema 6

En este tema

Las expresiones regulares son un potente lenguaje de descripción de cadenas. Son un lenguaje estándar (aunque con variaciones de unos intérpretes a otros) que es especialmente famoso por su uso en el lenguaje de programación Perl, aunque hoy día están extendidas en todos los lenguajes. El módulo re es el encargado en Python de tratar con estas expresiones.

Formato de las expresiones regulares

Introducción

Las "expresiones regulares", también conocidas por la contracción "regexps" (del inglés "regular expresions") son un sofisticado lenguaje de descripción (y de búsqueda y manipulación) de cadenas de texto. Son un estándar POSIX (aunque hay ciertas variaciones) que existe en cualquier lenguaje de programación decente y son una de las herramientas mas poderosas para manejar cadenas.

Pueden parecer un tanto áridas y extrañas al principio, pero seguramente acabarán haciéndose imprescindibles para tí.

El módulo re

Naturalmente, Python también tiene su propia aproximación a las expresiones regulares, y esta es por medio del módulo re (del inglés "Regular Expresions", también).

"re" es un módulo que provee a Python de una serie de funciones, clases y métodos para manejar cadenas por medio de expresiones regulares. Viene incluido por defecto en cualquier instalación de Python, por lo que no requiere de ninguna clase de preparativo especial antes de poder ser usado.

Antes de empezar a hablar de ello, veamos un pequeño ejemplo que nos servirá para, cambiando sus datos, ir probando los ejemplos que veamos a lo largo de este tema:

```
import re
patron= "mancha"
cadena="En un lugar de la mancha de cuyo nombre no quiero acordarme."
if re.search(patron, cadena):
```

```
    print( "Lo encontré, que cosas")
else:
    print("No aparece")
```

En este simple ejemplo estamos usando el método search para buscar un patrón (mancha) en una cadena (En un lugar de la mancha de cuyo nombre no quiero acordarme). No es muy sofisticado, y simplemente nos dice si aparece o no, pero nos servirá para ir empezando. Puedes (y deberías) probar a cambiar el patrón y/o la cadena para ver qué resultado te da.

En realidad, como veremos más adelante, el método search retorna un objeto MatchObject, que permite, entre otras cosas, ver cuántas apariciones de qué cadenas se han encontrado en qué posiciones, etc. Afortunadamente, el valor booleano de de este objeto es "True", con lo que también se puede usar en instrucciones más simples como la de arriba, que por ahora nos vale.

Para entender realmente los ejemplos que vamos a ir viendo deberías comprobarlos con este simple código, e incluso sería buena idea introducir tus propias variaciones.

Comodines

Los "comodines" son caracteres o grupos de estos que tienen una interpretación especial que sustituye a otros caracteres. Del mismo modo que un comodín de la baraja puede reemplazar a un As de corazones o un siete de picas, estos comodines se pueden interpretar como otros caracteres o grupos de estos.

Veamoslo con el más básico de los comodines:

Punto "."

El punto (.) reemplaza a un carácter cualquiera. De este modo, una búsqueda de la cadena "Pe.a", coincidirá con "Peta", "Pela" "Pesa", "PeÑa", "Pe-a", "Pe.a"...

La expresión que usamos para reemplazar o buscar cadenas se suele llamar el "patrón", y así lo haremos en adelante. De este modo, en el ejemplo anterior, "Pe.a" es el patrón que hemos usado para que coincida con todas esas cadenas.

Fíjate que no es necesario que sean caracteres alfanuméricos (admite también signos de puntuación y, en general, cualquier carácter) y que, evidentemente, también coincide con el propio signo ".". El único carácter con el que no coincide con "." (e incluso esto puede configurarse de otro modo con el uso del flag re.DOTALL) es con el "retorno de carro" o "nueva línea" (el clásico "\n").

Para poder poner en tu expresión regular un punto y que signifique sólo un punto y no otra cosa, hay que "escapar" el carácter anteponiéndole la barra "". De este modo, la expresión "Pe.a" sólo coincidiría con "Pe.a".

```
import re
patron = "r.ar.e."
cadena="En un lugar de la mancha de cuyo nombre no quiero acordarme."
if re.search(patron, cadena):
    print( "Lo encontré, que cosas")
else:
    print("No aparece")
```

¿Y si lo que buscamos es la propia cadena "Pe.a"? ¿Qué patrón debemos usar?

Para ello, de nuevo, debemos usar el carácter de escape "\", esta vez delante de la propia "\", para obtener algo así: "Pe\\\.a".

Nota que hay tres barras. La primera es para escapar la segunda, y la tercera es para escapar el punto (si sólo hubiésemos puesto dos, el punto se interpretaría como un comodín, tal como hemos visto arriba).

Sí, en ocasiones pueden acumularse montones de barras de escape.

Nueva línea "\n"

Cuando pulsas la tecla "Enter" o "Intro" para pasar a la línea siguiente, eso es un carácter de nueva línea. Como en casi todos los lenguajes y codificaciones, se representa mediante el código "\n".

Principio y fin de línea "^" y "$"

Otros dos caracteres especiales muy útiles son "^" y "$", que indican, respectivamente, el principio y el fin de una línea.

Por ejemplo, si en el código de arriba buscas el patrón "^acordarme.", no aparecerá (porque encontraría la palabra "acordarme" sólo en el caso de que esté al principio de línea). Sin embargo, el patrón "acordarme.$" (que busca la palabra acordarme" al final de la línea), sí dará un resultado.

En realidad, estos caracteres indican el principio o final de la variable. Para que realmente funcionen identificando todas las apariciones cuando hay varias líneas, hay que usar el flag "MULTILINE", que no es más que una constante predefinida en el módulo re, que indica qué tipo de comportamiento deben tener algunos de sus métodos.

Para hacer esto en nuestro ejemplo, cambiaríamos la llamada a search de este modo: "re.search(patron, cadena, re.MULTILINE):" (también se puede poner "re.M" en lugar de "re.MULTILINE", pero es mucho menos descriptivo).

De este modo, ya podemos hacer cosas como esta:

```
import re
patron= "mancha$"
cadena="En un lugar de la mancha\n de cuyo nombre no quiero acordarme."
```

```
if re.search(patron, cadena, re.M):
    print( "Lo encontré, que cosas")
else:
    print("No aparece")
```

Dígitos "\d" y "\D"

El comodín \d sustituye a cualquier dígito, y "\D" a cualquier cosa que no lo sea (letras, espacios, puntuación...).

Espacios "\s" y "\S"

El comodín "\s" identifica a cualquier carácter que implique un espacio. Esto incluye al espacio propiamente dicho, al tabulador, retorno de carro... El comodín "\S" es su opuesto.

Alfanumérico "\w" y "\W"

"\w" identifica caracteres alfanuméricos. Estos son las letras (mayúsculas y minúsculas), los números y el guión bajo "_". "\W" identificará los caracteres no alfanuméricos.

Corchetes “[]”

Por si esto fuera poco, los corchetes "[]" se usan para crear listas de caracteres. Si, por ejemplo, queremos que coincidan los números pares del 2 al 8, pondremos [2468]. De este modo, "a[bc]d" coincidirá tanto con "abd" como con "acd". En los corchetes pueden definirse secuencias separando el primer y el último elementos con un guión (-), como "[0-9]" para los números o "[a-h]" para las letras de la "a" a la "h". Pueden agruparse varas secuencias, de modo que [a-zA-Z] coincidirá con una letra mayúscula o minúscula.Dentro de unos corchetes se puede usar el símbolo "^" para indicar el conjunto opuesto al descrito. Por ejemplo, [^a-z] se referirá a cualquier carácter que no sea una letra minúscula.

Caracteres especiales

Además de los comodines que vimos en el tema anterior, hay otras herramientas que permiten modificar o afinar su utilidad, y que son las que le dan verdadera potencia.

Más “+”

Para empezar, el signo + detrás de un carácter hace que en el patrón coincidan ese carácter repetido una o más veces.

Por ejemplo, el patrón "jo+" encontrará las cadenas "jo", "joo", "jooo", etc. El patron "pa+c" coincidirá con "pac", "paac", "paaac"...

Del mismo modo, si queremos que el patrón coincida con cero, una o más apariciones, usaremos "". *Así, "abc*" coincidirá con "ac", "abc", "abbc" "abbbc"...

Y, si sólo buscamos cero o una apariciones, tenemos "?". En este caso, "ab?c" coincidirá con "ac" y "abc", pero no con "abbc".

Llaves “{}”

Si esto no es suficiente y necesitas más precisión, seguimos teniendo opciones.

"{n}" (donde n es un número), indica que buscamos n repeticiones del carácter que le precede. De este modo, "ab{3}c" es lo mismo que "abbbc",

Para tener algo más de margen, tenemos "{m,n}", que nos indica que buscamos un mínimo de m repeticiones y un máximo de n.

Paréntesis “()”

Hasta ahora, hemos aplicado estos modificadores a caracteres, pero también se pueden usar con cadenas completas encerrando estas entre paréntesis "()".

Por ejemplo, el patrón "ba(na)+" encontrará "bana", "banana", "bananana"...

Barra “|”

Se puede usar el signo | para separar opciones, de tal modo que "digo (hola|mundo)" coincidirá tanto con la cadena "digo hola" como con "digo mundo".

Naturalmente, todo esto se puede combinar para hacer cosas más complejas e interesantes. Por ejemplo, hagamos una expresión que reconozca números de DNI:

```
"^\d{8}[ -]?[a-zA-Z]$"
```

Con lo visto hasta ahora deberías poder justificar ese patrón. Al principio es un poco complicado de leer, pero parémonos a examinarla detenidamente. El DNI consta de un número de ocho dígitos y una letra. La letra, que puede ser mayúscula o minúscula, puede seguir al número directamente o estar separada por un espacio o un guión.Con el símbolo "^" al principio indicamos que queremos que la coincidencia coincida con el principio de la cadena, y con "$" al final hacemos que también coincida con el final. De este modo, buscamos que coincida la cadena completa, no una subcadena dentro de esta.

Algunos ejemplos más:

Número de teléfono: Tres grupos de tres dígitos separados (o no) por guiones:

```
"^[0-9]{3}-?[0-9]{3}-?[0-9]{3}$"
```

Nota que esto es lo mismo que escribir:

```
"^\d{3}-?\d{3}-?\d{3}$"
```
Si queremos un número de móvil (que vamos a suponer -erróneamente- que son los que empiezan por 6):

```
"^6[0-9]{2}-?[0-9]{3}-?[0-9]{3}$"
```
Una primera aproximación para identificar un correo electrónico "nombre@dominio.com" sería la siguiente:

```
"^([a-zA-Z0-9_-.]+)@([a-zA-Z0-9_-.]+).([a-zA-Z0-9_-.]{1,3})$"
```
Y para identificar hora y minutos, en formato "21:34":

```
"^[0-2]\d:[0-6]\d$"
```
Tabla de patrones.

Como referencia rápida tenemos:

https://help.libreoffice.org/Common/List_of_Regular_Expressions/es

Instalando Regular Expressions Tester

El módulo re

En el anterior tema hemos visto un montón de teoría sobre expresiones regulares y un pequeño ejemplo del módulo re por medio de search. Ahora vamos a ver algunas funciones que nos puedan servir para encontrar utilidad a lo aprendido:

Empecemos con search, que para algo ya la hemos usado:

```
re.search(Patrón, Cadena, [flags])
```
search busca en Cadena el Patrón que le indiquemos y, como comentamos entonces, retorna un objeto MatchObject que es el que contiene los datos de nuestra búsqueda y el que realmente usaremos.

Si no encuentra ninguna coincidencia, search retorna un valor booleano "falso".

Otra función muy similar a search es match:

```
re.match(Patrón, Cadena, [flags])
```
match se comporta igual que search, buscando el Patrón en la Cadena, pero sólo si coincide desde el inicio de Cadena. Más o menos, es un equivalente a hacer un search, pero iniciando el patrón con el carácter "^".

Una herramienta mucho más interesante es split:

```
re.split(Patrón, Cadena, [max], [flags])
```

split divide Cadena usando Patrón como referencia y retorna un array conteniendo las cadenas resultantes. Con el parámetro opcional max se le indica el número máximo de "trozos" que queremos obtener.

Veamoslo con este ejemplo que divide la cadena que se le proporciona en varias subcadenas numéricas usando cono separadores los caracteres no numéricos:

```
#!/usr/bin/python

# -*- coding: utf-8 -*-

import re

Resultados= re.split("\\D", "123a45x6-78.90")

print Resultados
```

Naturalmente, lo complejo del patrón sólo está limitado por tu imaginación y necesidades.

Otra herramienta muy útil es findall:

```
re.findall(Patrón, Cadena, [max], [flags])
```
Se comporta igual que find pero, en lugar de detenerse en la primera coincidencia, retorna un array que, por ejemplo, permite usar la salida de esta función en un bucle para explorar todas las coincidencias.

Las coincidencias se retornan en el orden en el que aparecen, y no se superponen (Es decir, que los caracteres que pertenezcan a una coincidencia no pueden pertenecer a otra).

Veamos un ejemplo simple:

```
#!/usr/bin/python

# -*- coding: utf-8 -*-

import re

Coincidencias= re.findall(".n", "En un lugar de la Mancha")

print Coincidencias
```

Hasta ahora sólo hemos usado las regexps para encontrar coincidencias, pero otra potente utilidad es la de reemplazar cadenas o partes de estas. Una utilidad para ello es sub:

```
re.sub(Patrón, Reemplazar, Cadena, [contador], [flags])
```
Esto es algo más complicado, pero fácil de entender. Usamos Patrón para buscar en Cadena y, las coincidencias que encontremos, las reemplazamos por Reemplazar. Si

se usa un número en el argumento opcional "contador", sólo se reemplazará ese número de apariciones como máximo, aunque hubiera más coincidencias.

Por ejemplo:

```
#!/usr/bin/python

# -*- coding: utf-8 -*-

import re

Cadena= "En un lugar de la Mancha fea y negra de los piratas"

Resultados= re.sub("a |a$", "as ", Cadena)

print Cadena

print Resultados
```

Igual que antes, las posibilidades de complejidad y sofisticación son ilimitadas

Un detalle interesante de sub es que Reemplazar puede ser una función, en lugar de una cadena. En este caso, la función será llamada a cada coincidencia, se le pasará la cadena encontrada como parámetro y su resultado será lo que se use como cadena para reemplazar.

En los ejemplos anteriores, cada vez que se usa un Patrón, Python tiene que compilar y preparar ese patrón antes de poder usarlo (Esto no es del todo cierto, porque python usa una caché para ello).

En realidad, esto no ocupa demasiados recursos pero, si se va a hacer un uso más o menos intensivo, es mejor compilar la expresión una sola vez y usar el objeto compilado como patrón. Para ello está compile:

```
re.compile(Patrón, [flags])
```

El resultado de esta función puede usarse como patrón en todas las expresiones que hemos visto antes, en lugar de usar una cadena de texto o una variable conteniéndola.

Flags

Ya hemos visto un par de flags que nos servía para cambiar el comportamiento por defecto de las funciones y los métodos del módulo “re”, pero hay algunos más que también resultan muy útiles. Se muestran tanto en la forma larga (y más comprensible) de una palabra y la forma breve, de una letra. Ambas formas de escribirlas tienen exactamente el mismo comportamiento, aunque se recomienda la forma larga, en beneficio de la claridad:

re.UNICODE

re.U

Necesaria para usar caracteres UNICODE (lo que no significa que no vayas a tener problemas con Unicode)

re.MULTILINE

re.M

Hace que "^" y "$" no se aplique sólo al principio y al fin de la cadena respectivamente, sino que se aplique a cada (salto de) línea dentro de ella.

re.IGNORECASE

re.I

Hace que los patrones no distingan mayúsculas de minúsculas

re.DOTALL

re.D

Hace que el comodín "." identifique TAMBIEN al carácter "\n"

Todos ellos tiene una versión larga y descriptiva y otra versión más corta y de significado ligeramente más hermético. En general, es recomendable la primera de ellas, por razones de claridad y legibilidad.

Clases

El módulo re provee de un par de clases, con sus métodos y atributos correspondientes, para tratar con expresiones regulares en POO. En realidad, su uso y propiedades son prácticamente iguales al visto en el caso de las funciones, pero se enlazan aquí sus descripciones a modo de referencia.

RegexObject

La clase RegexObject es el objeto de uso de las expresiones regulares, y admite métodos equivalentes a las funciones vistas anteriormente (find, match, split, sub)

MatchObject

La clase MatchObject, de la que ya hemos hablado anteriormente, es la que contiene el resultado de operaciones como search o match, y siempre tiene un valor booleano de true.

Tema 7

En este tema

El diseño de interfaces es una compleja profesión en sí misma, pero cualquier programa que escribamos tiene que interactuar con el usuario de algún modo. En este tema vamos a ver algunas herramientas para crear interfaces basados en texto, desde el uso de la línea de comandos hasta complejos sistemas de ventanas en modo texto.

Introducción al interfaz de texto

Un interfaz de usuario es el método que usamos para que nuestro programa interaccione con el usuario. Muy esquemáticamente, el usuario introduce datos en nuestro programa y nosotros a cambio le damos información. La forma más simple de hacer esto es con texto en un terminal.

Nuestro viejo y fiel input()

Cuando queremos que el usuario interactúe con nuestro programa en terminal, normalmente recurrimos a la función input().

Por ejemplo:

```
nombre = input("Escribe tu nombre")
print("Hola {}".format(nombre))
```

input() detiene el programa, espera a que el usuario introduzca un texto y pulse la tecla "Enter" y retorna ese texto en forma de string, continuando con la ejecución del programa.

Aunque a menudo es suficiente, esto tiene varios inconvenientes, como que nuestro programa necesita atención mientras se ejecuta, de modo que el usuario no puede ejecutarlo y desentenderse, lo que puede ser tedioso si el programa es lento. Además, un programa hecho así no puede usarse como parte de un script de terminal (como un programa bash o un .bat de Windows), porque no tiene forma de introducir los datos.

De modo que a veces necesitamos algo un poco más avanzado.

Argumentos

La forma típica de introducir datos en un programa en línea de comandos es por medio de argumentos. Si queremos que el comando cat (de Linux/Mac) nos muestre el contenido de un archivo, escribiremos lo siguiente:

```
cat nombre_del_archivo
```

También podemos encontrarnos algunos argumentos fijos, que comienzan por guiones. Por ejemplo, para consultar la ayuda de python escribimos lo siguiente:

```
python3 -h
```

O, también:

```
python3 --help
```

Estos comandos precedidos de guiones se suelen llamar "flags", y pueden preceder a modificadores. Por ejemplo, como ya sabemos, para que python ejecute un módulo directamente se emplea el flag -m de este modo:

```
python3 -m nombre_del_modulo
```

Lectura de argumentos con sys.argv

En los scripts de Python, los argumentos que le pasamos a un programa en línea de comandos se almacenan en la variable argv del módulo sys en forma de lista. El primer elemento de esa lista (sys.argv[0]) es el nombre del propio script, y los siguientes son los argumentos que le hayamos pasado, si los hay, en el mismo orden en que los hemos puesto.

Por ejemplo, supongamos que escribimos un script llamado "saluda.py" con este contenido:

```
import sys
print("Hola {}".format(sys.argv[1]))
```

Si ejecutamos en un terminal la siguiente orden:

```
python saluda.py Pablo
```

Nos imprimiría la cadena "Hola Pablo".

Hay que tener en cuenta que, en este ejemplo, si no introducimos ningún argumento, tendremos una excepción IndexError que deberíamos gestionar. Así que en nuestro ejemplo podríamos hacer algo así como esto:

```
import sys
try:
    print("Hola {}".format(sys.argv[1]))
except IndexError:
    print("Hola Persona")
```

Ahora, si hay un error IndexError, se imprimirá la neutr (y fílmica) cadena "Hola Persona". Probablemente habría que tener en cuenta otros posibles errores, pero esto nos vale para el ejemplo.

Naturalmente, podemos tener varios argumentos (y,de hecho, es lo normal):

```
import sys
for arg = sys.argv:
    print("Argumento {}".format(arg))
```

Tomando los valores en la lista sys.argv y trabajando con ellos es relativamente fácil usar los argumentos que se le pasan a nuestro script, siempre que no sea demasiado complejo. Para tareas más elaboradas, vamos a necesitar ayuda.

El módulo argparse

Cuando en nuestro programa tenemos que manejar varios argumentos, unos obligatorios, otros opcionales, algunos de los cuales pueden depender de otros, etc, la cosa empieza a complicarse. Además, como hemos visto. a veces los argumentos pueden estar en forma de "flag" de una letra (como el típico flag -h, que suele servir por convención para mostrar la ayuda de un programa) o de varias (como --help), o un argumento puede tener un predicado (como cuando se pone "-f archivo" o "--file archivo" para indicar un archivo) etc.

Aunque módulos clásicos de Ptyhon como getopt pueden facilitarnos mucho la tarea, Python introdujo en su versión 3.2 el módulo argparse, que nos ahorra gran parte del trabajo y lo hace todo más fácil.

Veamos un ejemplo:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('name', type=str, help='Nombre al que se saludará')
args = parser.parse_args()
print("Hola {}".format(args.name))
```

¿Qué hemos hecho aquí? Primero, tras importar el módulo argparse, creamos un objeto ArgumentParser al que llamamos "parser". Este objeto es el que vamos a usar para definir nuestros argumentos, tomar sus valores, etc.

Con el método add_argument() añadimos un argumento. Le decimos que se va llamar "name", que es de tipo string y le añadimos un texto de ayuda (porque, como veremos, argparse se ocupa de crear la ayuda).

Luego tomamos los argumentos en la variable args con el método parse_args() y usamos em atributo "name" de este objeto para obtener el valor de ese argumento.

Como vemos, argparse ha creado un atributo con el nombre que le definimos ("name") en el objeto que retorna parse_args() con el valor de ese argumento.

Además, si hay un error, como que el usuario no ponga el argumento, argparse se pucpa de detener el programa y mostrar un mensaje de ayuda al usuario. En nuestro caso, y suponiendo que el script se llame "saluda.py", el mensaje sería algo así:

```
usage: saluda.py [-h] name
saluda.py: error: the following arguments are required: name
```

Además, como se puede ver, argparse ha añadido el argumento opcional "-h". Si llamamos al script usando ese argumento, nos mostrará el menú de ayuda:

```
usage: si.py [-h] name

positional arguments:
  name        Nombre al que se saludará

optional arguments:
  -h, --help  show this help message and exit
```

Como se puede ver, ha añadido el texto de ayuda que definimos en add_argument().

Este menú de ayuda se puede personalizar de muchos modos. Por ejemplo, lo más

usual es agregar un texto describiendo el programa en la cabecera y/o al pié de ese menú. Lo podemos hacer con los argumentos opcionales description y epilog al crear el parser:

```
import argparse
help_txt = "Aquí debería poner una introducción a qué hace el programa."
epilog_txt = "Esto es el texto final tras la descripción de los comandos."
parser = argparse.ArgumentParser(description=help_txt, epilog=epilog_txt)
parser.add_argument('name', type=str, help='Nombre al que se saludará')
args = parser.parse_args()
print("Hola {}".format(args.name))
```

Podemos añadir cuantos argumentos queramos añadiendo más líneas con sucesivas instrucciones add_argument(). Además, podemos poner argumentos de tipo numérico, con flags, opcionales, etc, Vamos a ver un ejemplo un poco más complejo:

```
import argparse
help_txt = "Aquí debería poner una introducción a qué hace el programa."
epilog_txt = "Esto es el texto final tras la descripción de los comandos."
parser = argparse.ArgumentParser(description=help_txt, epilog=epilog_txt)
parser.add_argument('name', type=str, help='Nombre al que se saludará')
parser.add_argument('-n', '--number', type=int, dest='number', help='Número
de veces que se va a saludar')
parser.add_argument('-f', '--file', type=str, dest='file', help='nombre de
fichero con el que no hacemos nada, sólo mostrar que se pueden poner más
argumentos')
args = parser.parse_args()
count = args.number if args.number else 1
for i in range(count):
    print("{}: Hola {}".format(i, args.name))
```

Como nos informará la ayuda del programa, ahora tenemos dos argumentos más, con flags "--file" (que ta,bien puede escribirse "-f"), que es de tipo texto y "--number" (o "-n") que es de tipo numérico.

Además, si el usuario trata de introducir una cadena de texto en "--number", argparse se ocupará de advertirle del error, ahorrándonos tener que programar la gestión de ese aspecto.

El módulo argparse tiene muchas más opciones y posibilidades que pueden verse en su página de documentación oficial.

Trabajar con stdin, stdout y stderr

Un aspecto muy interesante de los comandos y programas de los sistemas operativos basados en UNIX es su filosofía de trabajo, consistente en hacer programas que hagan una sola cosa, pero la hagan bien que puedan interoperar

con otros programas y ficheros por medio de pipes (tuberías) y redirecciones.

Por ejemplo:

El comando cat nombre_de_archivo muestra por pantalla el contenido de ese archivo.
El comando sort nombre_de_archivo ordena las líneas del archivo y lo muestra por pantalla.
El comando uniq nombre_de_archivo elimina las líneas duplicadas del archivo (sólo si están una a continuación de la otra) y muestra el resultado por pantalla .
El comando wc nombre_de_archivo cuenta líneas, palabras y letras del archivo y muestra un resumen por pantalla .

Pero pueden usarse juntos para obtener el número de líneas distintas de un archivo y guardar ese resultado en otro archivo de este modo:

```
cat archivo.txt | sort | uniq | wc > resultado.txt
```

el carácter "pipe" (|) hace que cada comando actué sobre la salida del anterior, y el carácter ">" redirecciona el resultado al archivo resultado.txt.

Esto se consigue mediante las llamadas "salida estándar" y "entrada estándar" (y la salida de errores, que también veremos).

Puedes consultar esta página para saber más sobre estos conceptos.

stdout: La salida estándar

La salida estándar, normalmente, es el propio terminal. Cuando en Python hacemos print() sin indicar a qué archivo lo estamos haciendo (es decir, sin el argumento "file"), en realidad lo está mandando a la salida estándar.de forma que podemos hacer un script que muestre el texto "Hola mundo" como cualquiera de los que hemos ido probando en este curso y, si queremos contar sus palabras, redireccionarlo a wc del modo que hemos visto:

```
python3 mi_script.py | wc
```

que nos mostraría el siguiente resultado:

```
      1       2      11
```

(que significa una línea, dos palabras, 11 letras)

Es decir, que para usar la salida estándar no hay que hacer nada especial: Es la que usamos por defecto.

Pero, a veces, puede ser interesante enviar cosas a la salida estándar en programas que no la estén usando por defecto (algunos programas con interfaz web, por ejemplo) o por otras razones queramos asegurarnos de que es allí a donde va. Para ello el módulo sys dispone del atributo stdout, que contiene un objeto file al que podemos escribir como si de cualquier otro fichero abierto se tratase:

```
import sys
stdout = sys.stdout
stdout.write("Hola mundo\n")
```

stderr: La salida de errores

Otra salida similar es la salida de errores stderr, que se usa exactamente del mismo modo. Normalmente esta salida está redireccionada al terminal, exactamente igual que stodut, aunque en muchas ocasiones nos la encontraremos redireccionada a un log de errores. Podemos enviar aquí lo que queramos, claro, pero se supone que está para mandar mensajes de error o alerta si el programa ha fallado en lo que sea. De hecho, los mensajes de excepción que muestra nuestros scripts cuando fallan en realidad están yendo a la salida de error (que, como hemos dicho, está redireccionada a la pantalla del terminal).

Por ejemplo, aquí vamos capturar una excepción (que generamos nosotros mismos con raise) con un try y mostramos un mensaje de error por la salida de errores:

```
import sys
try:
    raise Exception
except Exception:
    print("Ha habido un problema", file=sys.stderr)
```

stdin: La entrada estándar

La entrada estándar de un programa, normalmente, es el teclado. Pero cuando redireccionamos un archivo con "<" o usamos un pipe ("|") con la salida de otro comando, será eso lo que se use como entrada estándar.

Para usar la entrada estándar tenemos otro atributo de sys, llamado stdin, que

también es un objeto file y que podemos usar como archivo de lectura. Por ejemplo:

```
import sys
stdin = sys.stdin
for line in stdin:
    if line == "salir\n":
        break
    print("La línea dice: {}".format(line))
```

Si ejecutamos este script redireccionándole un archivo o la salida de otro comando o programa, este mostrará un mensaje diciendo "La línea dice ..." por cada línea de ese programa, a menos que la línea diga "salida", en cuyo caso el programa terminará.

En caso de que no le redireccionemos nada, el programa se quedará esperando a que lo escribamos nosotros (porque la entrada estándar por defecto es el teclado), y hará lo mismo con las líneas que escribamos. Podemos escribir "salida" para que termine o, en general, pulsar Ctrl-C.

El paquete curses

El módulo curses es una envoltura (wrapper) alrededor de la famosa (y clásica) librería curses de C que permite a Python usar los métodos de esa librería. Posee herramientas para posicionar elementos en la terminal, redibujar la pantalla, ajustar colores, recibir entradas de texto, etc. Es una herramienta muy útil para hacer interfaces de tipo TUI (Terminal User Interface), algo un poco a medio camino entre las interfaces de línea de comando y los interfaces gráficos (o GUI).

NOTA: Curse no viene instalado por defecto en Windows. Para instalarlo con pip (o con pip3, ya sabes):

```
pip install windows-curses
```

Comencemos por lo básico, nuestro tradicional "Hola mundo":

```
import curses
window = curses.initscr()
window.addstr(10, 10, "Hola")
window.refresh()
window.getch()
window.addstr(11, 10, "Mundo")
window.refresh()
window.getch()
curses.endwin()
```

Tras importar el módulo curses creamos un objeto window, que es el que va a

gestionar nuestra ventana y cómo interaccionemos con ella.

Después añadimos un texto en las coordenadas que queramos con el método addstr(). Hay que tener en cuenta que, por razones históricas, curses pone primero la coordenada "y" (distancia en filas al borde superior) y después la coordenada "x" (distancia en caracteres al borde izquierdo), al revés que prácticamente cualquier otro sistema de coordenadas.

Después usamos el método refresh() para refrescar la pantalla. Este redibujado es algo que habrá que hacer cada vez que hagamos un cambio.

Y tras esto llamamos al método getch(), que detiene la ejecución del programa hasta que el usuario pulsa alguna techa.

Hacemos de nuevo el ciclo de añadir texto, redibujar y esperar la pulsación para imprimir el "Mundo" y, por último, con endwin() desactivamos las curses.

Esto último es muy importante. El módulo curses toma control completo del terminal, de lo que se muestra y de cómo se muestra y, si no desactivamos las curses antes de salir, lo más probable es que no podamos usar el terminal porque no se vea nada, o no se vea lo que escribimos, o no responda, o cualquier otro error extravagante y aleatorio que nos obligue a cerrar la terminal y abrir una nueva.

Esto también significa que, si nuestro programa falla y salta una excepción antes de que que cerremos las curses, volvemos a perder el control del terminal.

Esto de perder el terminal es tan habitual que curses tiene un método para evitarlo o, al menos, reducirlo todo lo posible. Se trata de wrapper().

El wrapper es una función de curses que toma como argumento una función y la "envuelve" en un entorno seguro para ejecutar nuestra ventana. De este modo, si hay un error o por la razón que sea salimos del programa sin cerrar las curses, el wrapper se encarga de que nuestra terminal vuelva a su estado original.

Para poder usar nuestro ejemplo con el wrapper tenemos que hacer un par de cambios. Primero tenemos que ponerlo todo en una función, porque la necesitamos para pasársela al wrapper como argumento. Además, esa función debe aceptar un argumento (que, por tradición se llama "stdscr" o "screen", pero que puede tener cualquier nombre) que es el objeto window que representa a nuestra pantalla y con el que trabajaremos:

```
import curses
```

```
def main_window(stdscr):
    stdscr.addstr(10, 10, "Hola")
    stdscr.refresh()
    stdscr.getch()
    stdscr.addstr(11, 10, "Mundo")
    stdscr.refresh()
    stdscr.getch()

curses.wrapper(main_window)
```

También podemos ver que ya no necesitamos cerrar las curses, porque wrapper se ocupa de ello.

Pero vamos a ver si arreglamos un poco la estética de nuestra ventana. Para empezar, ese cursor que aparece después del texto cuando está esperando a que pulsemos una tecla resulta muy feo. Podemos quitarlo con la función curses.curs_set(0), que cambia el modo del cursor (0 para invisible, 1 normal y 2 para resaltado). Cunado lo necesitemos podremos volver a mostrarlo.

Además, vamos a ponerle un borde a nuestra ventana con el método border() del objeto window:

```
import curses

def main_window(stdscr):
    curses.curs_set(0)
    stdscr.border()
    stdscr.addstr(10, 10, "Hola")
    stdscr.refresh()
    stdscr.getch()
    stdscr.addstr(11, 10, "Mundo")
    stdscr.refresh()
    stdscr.getch()

curses.wrapper(main_window)
```

Podemos crear nuevos objetos window dentro de la ventana principal con la función curses.newwin():

Por ejemplo, volvamos a mostrar nuestro Hola Mundo pero usando una ventana distinta para cada palabra:

```
import curses

def main_window(stdscr):
    curses.curs_set(0)
    stdscr.border()
```

```
    win1 = curses.newwin(5, 10, 10, 10)
    win1.border()
    win1.addstr(2, 3, 'Hola')
    stdscr.refresh()
    win1.refresh()

    stdscr.getch()

    win2 = curses.newwin(5, 10, 16, 10)
    win2.border()
    win2.addstr(2, 3, 'Mundo')
    stdscr.refresh()
    win1.refresh()
    win2.refresh()

    stdscr.getch()

curses.wrapper(main_window)
```

Como se ve en este ejemplo, las coordenadas de addrstr() se calculan respecto a la ventana en la que estamos añadiendo el texto.

El método getch() no sólo espera a que se pulse una tecla, sino que retorna el código ASCII del carácter pulsado. Muy similar a este, el método getkey() hace exactamente lo mismo, pero retornando un string con el carácter pulsado. Podemos usar esto para que nuestro programa actúe en función de la pulsación de una tecla (como en menús y ese tipo de interfaces):

```
import curses

def main_window(stdscr):
    curses.curs_set(0)
    stdscr.border()

    win1 = curses.newwin(5, 10, 10, 10)
    win1.border()
    win1.addstr(2, 3, 'Hola')
    stdscr.refresh()
    win1.refresh()

    stdscr.getch()

    win2 = curses.newwin(5, 10, 16, 10)
    win2.border()
    win2.addstr(2, 3, 'Mundo')

    stdscr.addstr(1, 1, 'Pulse la techas Q para salir')
    stdscr.refresh()
    win1.refresh()
    win2.refresh()

    while 1:
        k = stdscr.getkey()
        if k == "q":
```

```
            break

curses.wrapper(main_window)
```

Otra función parecida, y muy útil, es getstr(), que nos permite introducir una cadena de texto (terminada por un retorno de carro). Podemos usarla de este modo:

```
import curses

def main_window(stdscr):
    curses.curs_set(0)
    stdscr.border()

    win1 = curses.newwin(5, 30, 10, 10)
    win1.border()
    win1.addstr(2, 3, 'Nombre: ')
    stdscr.refresh()
    win1.refresh()

    text = win1.getstr()

    win2 = curses.newwin(5, 30, 16, 10)
    win2.border()
    win2.addstr(2, 3, 'Hola {}'.format(text.decode('utf-8')))

    stdscr.addstr(1, 1, 'Pulse la techas Q para salir')
    stdscr.refresh()
    win1.refresh()
    win2.refresh()

    while 1:
        k = stdscr.getkey()
        if k == "q":
            break

curses.wrapper(main_window)
```

Sin embargo, como podemos ver, no vemos la cadena de texto mientras la escribimos, lo que es bastante incómodo. Para arreglar esto tenemos que usar la función curses.echo() para que se muestre por pantalla el texto que escribimos (luego habrá que volver a ocultarlo con curses.noecho()).

También vamos a cambiar el cursor temporalmente para que se vea por donde escribimos.

Además, simplemente para que quede más bonito (y veamos lo que se puede hacer) le vamos a poner un título a cada ventana, simplemente poniendo el texto en la coordenada Y = 0, para que se superponga al borde:

```
import curses

def main_window(stdscr):
```

```
    curses.curs_set(0)
    stdscr.border()

    win1 = curses.newwin(5, 30, 10, 10)
    win1.border()
    win1.addstr(0, 1, 'ENTRADA:')
    win1.addstr(2, 3, 'Nombre: ')
    stdscr.refresh()
    win1.refresh()

    curses.echo()
    curses.curs_set(1)
    text = win1.getstr()
    curses.noecho()
    curses.curs_set(0)

    win2 = curses.newwin(5, 30, 16, 10)
    win2.border()
    win1.addstr(0, 1, 'SALIDA:')
    win2.addstr(2, 3, 'Hola {}'.format(text.decode('utf-8')))

    stdscr.addstr(1, 1, 'Pulse la techas Q para salir')
    stdscr.refresh()
    win1.refresh()
    win2.refresh()

    while 1:
        k = stdscr.getkey()
        if k == "q":
            break

curses.wrapper(main_window)
```

A getstr() podemos decirle dónde queremos que se muestre ese texto poniéndolo en sus argumentos (ya sabes, la y va antes que la x). Si no se lo indicamos, lo pondrá en el lugar donde se quedó el cursor (tras lo último que escribimos en pantalla.

Con esto sólo hemos rascado la superficie de este paquete. Curses es una librería muy compleja y potente. No es quizás la herramienta más fácil de usar (porque nos obliga a estar en todos los detalles) y puede ser muy farragosa, pero nos permite crear interfaces muy ricos y complejos.

Para más información: https://docs.python.org/3/library/curses.html

Tema 8

Ventanas y posicionamiento

Aunque hay varios paquetes que nos permiten crear interfaces gráficos para Python, tkinter es el estándar de facto para este tipo de tareas.

El concepto esencial en tkinter (y, en general, en cualquier sistema de interfaz gráfico) es el de "widget". un widget es un elemento gráfico que sirve como componente de nuestro interfaz. Así, un botón, un menú, una ventana o un cuadro de texto son ejemplos de widgets.

Crear un interfaz en tkinter significa "empaquetar" una serie de widgets a los que se les podrán vincular acciones.

Comencemos con una ventana básica con un texto:

```
import tkinter as tk
window = tk.Tk()
text = tk.Label(window, text="Hola mundo")
text.pack()
window.mainloop()
```

Tras importar el módulo tkinter y crear una ventana principal con el método Tk(), creamos una etiqueta (Label), un widget que sirve para mostrar texto. El primer argumento es el widget en el que se va a ubicar (en nuestro caso, la ventana) y en "text" le indicamos el texto que queremos que muestre.

Tras esto usamos un método que tiene casi todos los widgets, pack(), que sirve para ubicar el widget. Veremos más detalles enseguida, pero por ahora diremos que no basta con crear el elemento, hay que colocarlo.

Finalmente, iniciamos el bucle de eventos o mainloop, que mantendrá el programa en ejecución.

Como podemos ver, la ventana toma elementos del sistema operativo: El borde o los botones son los mismos que en el resto de ventanas del sistema (y se verán de forma distinta en otros sistemas operativos).

También vemos que la ventana es muy pequeña: Aunque se puede arrastrar desde las esquinas para cambiar su tamaño, por defecto se ha ajustado a su contenido. Podemos cambiar esto con el método geometry(). De paso, también podemos

ponerle un título a la ventana con el método tltle():

```
import tkinter as tk
window = tk.Tk()
window.title("Programa")
window.geometry("500x200")
text = tk.Label(window, text="Hola mundo")
text.pack()
window.mainloop()
```

Si queremos impedir que se pueda cambiar el tamaño de la ventana, tenemos el método resizable(), que admite dos argumentos booleanos y nos permite bloquear (con False) o permitir (con True) ajustar el tamaño a lo ancho y a lo alto independientemente:

```
window.resizable(False, False)
```

Si agregamos varios widgets tenemos que usar pack() con cada uno de ellos. Como hemos visto, pack sirve para "empaquetar" el widget y, por defecto, tkinter simplemente los irá situando uno debajo del otro (según el orden en que hagamos pack(), no el orden en que los definamos). Pero podemos decirle a pack() cómo queremos que los distribuya con el atributo "side". Para ello tkinter tiene unas constantes predefinidas que son TOP, LEFT, RIGHT y BOTTOM:

```
import tkinter as tk
window = tk.Tk()
window.title("Programa")
window.geometry("300x300")
text1 = tk.Label(window, text="uno")
text2 = tk.Label(window, text="dos")
text3 = tk.Label(window, text="tres")
text4 = tk.Label(window, text="cuatro")
text1.pack(side=tk.TOP)
text2.pack(side=tk.LEFT)
text3.pack(side=tk.RIGHT)
text4.pack(side=tk.BOTTOM)
window.mainloop()
```

Aunque pack() es la forma más fácil de ubicar nuestros widgets, también tenemos otras formas.

Para empezar, podemos situarlos en una "rejilla", indicándoles la columna y fila en que queremos que se ubique con el método grid():

```
import tkinter as tk
window = tk.Tk()
window.geometry("300x300")
```

```
for x in range(3):
    for y in range(3):
         entry = tk.Label(window, width=10, text="({}:{})".format(x, y))
         entry.grid(row = x, column = y)
window.mainloop()
```

Y también tenemos el método place(), que nos permite total control de la posición, al poder indicar las coordenadas exactas en que queremos poner nuestro widget:

```
import tkinter as tk
window = tk.Tk()
window.geometry("300x300")
txt_top_left = tk.Label(window, text="Arriba a la izquierda")
txt_top_left.place(x=10, y=10)
txt_bottom_right = tk.Label(window, text="Abajo a la derecha")
txt_bottom_right.place(x=150, y=200)
window.mainloop()
```

El método place también admite un argumento "height" que indica la altura del control.

Además, podemos agrupar nuestros widgets dentro de "marcos" para agruparlos de formas complejas gracias al widget Frame. En este ejemplo hemos creado un frame para poner dentro dos labels, mientras que dejamos otras dos fuera del frame. Nota que. al crar las labels, a dos de ellas se les ha pasado "frame" en lugar de "window" como argumento:

```
import tkinter as tk
window = tk.Tk()
window.title("Programa")
window.geometry("300x300")
frame = tk.Frame(window)
text1 = tk.Label(frame, text="uno")
text2 = tk.Label(frame, text="dos")
text3 = tk.Label(window, text="tres")
text4 = tk.Label(window, text="cuatro")
frame.pack()
text1.pack(side=tk.LEFT)
text2.pack(side=tk.RIGHT)
text3.pack(side=tk.TOP)
text4.pack(side=tk.TOP)
window.mainloop()
```

Es decir: Hemos ubicado con pack() dos de nuestras Labels dentro de un Frame que, a su vez, está colocado, también con pack(), en la ventana principal.

Si ahora hacemos exactamente lo mismo, pero colocando el frame en otra posición (cambiado de lugar su pack()), vemos que el resultado es muy distinto:

```
import tkinter as tk
window = tk.Tk()
window.title("Programa")
window.geometry("300x300")
frame = tk.Frame(window)
text1 = tk.Label(frame, text="uno")
text2 = tk.Label(frame, text="dos")
text3 = tk.Label(window, text="tres")
text4 = tk.Label(window, text="cuatro")
text1.pack(side=tk.LEFT)
text2.pack(side=tk.RIGHT)
text3.pack(side=tk.TOP)
text4.pack(side=tk.TOP)
frame.pack()
window.mainloop()
```

El posicionamiento de widgets puede complicarse mucho más, y tkinter tiene un montón de opciones de configuración y ajuste para ello, pero con esto ya es suficiente para nosotros.

En el siguiente tema pasaremos al siguiente paso en un interfaz: Conseguir que haga cosas.

Widgets y acciones

La idea detrás de un interfaz de usuario es servir de enlace entre este y nuestro programa. Para ello necesitamos algunos elementos con los que se pueda interaccionar, y que estos elementos respondan a las acciones del usuario. En un entorno gráfico estos elementos son widgets y, probablemente el rey de los widgets en los interfaces gráficos sea el botón.

En tkinter el botón se crea con Button. Como todos los widgets, necesita que le pasemos como argumento el widget donde se ubica. Le debemos pasar también el texto que queremos que muestre con el argumento "text" y, lo más importante, la acción que queremos que ejecute con el argumento "command".

La acción se indica mediante un "callback" que, en este contexto, no es más que el nombre de una función (o en general, cualquier "callable"):

```
import tkinter as tk
def adios():
    text["text"] = "Adiós"
window = tk.Tk()
window.title("Programa")
window.geometry("500x200")
text = tk.Label(window, text="Hola")
text.pack()
button = tk.Button(window, text="¡Me voy!", bg="green", fg="white",
command=adios)
button.pack()
```

```
window.mainloop()
```

Adicionalmente, hemos usado los argumentos "bg" y "fg", que admiten casi todos los widgets, para darle un poco de color.

El problema de llamar a la función con un callback es que no se le pueden pasar argumentos. Para hacer esto, normalmente se hace con una función intermedia que sea la que los pase como, por ejemplo, esta lambda:

```
import tkinter as tk
def adios(widget, text):
    widget["text"] = text
window = tk.Tk()
window.title("Programa")
window.geometry("500x200")
text = tk.Label(window, text="Hola")
text.pack()
button = tk.Button(window, text="¡Me voy!", bg="green", fg="white",
command=lambda: adios(text,"Chao"))
button.pack()
window.mainloop()
```

Nota que estamos usando ["text"] para acceder al atributo "text" de un widget como si fuera un elemento de un diccionario.

Meter una lambda ahí puede servirnos para cosa muy concretas, pero normalmente no resulta práctico ni elegante. Resulta mucho mejor utilizar clases y métodos para tener debidamente encapsulado nuestro trabajo:

```
import tkinter as tk
class my_window():
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Programa")
        self.window.geometry("500x200")
        self.text = tk.Label(self.window, text="Hola")
        button_hola = tk.Button(self.window, text="¡Hola!", bg="green",
fg="white", command=self._hello_btn)
        self.text.pack()
        button_hola.pack()
    def mainloop(self):
        self.window.mainloop()
    def _hello_btn(self):
        text = "Chao"
        self._show_text(self.text, text)
    def _show_text(self, widget_out, text):
        widget_out["text"] = text
ventana = my_window()
ventana.mainloop()
```

De este modo, lo más cómodo es que cada botón tenga su propio método con

toda la lógica necesaria, al que se llamará cada vez que se pulse.

Otro widget muy útil es Entry, que nos permite colocar el clásico cuadro de texto:

```
import tkinter as tk
class my_window():
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Programa")
        self.window.geometry("500x200")
        self.text = tk.Label(self.window, text="Dime tu nombre")
        self.name_input = tk.Entry(self.window)
        button_hola = tk.Button(self.window, text="¡Hola!", bg="green",
fg="white", command=self._hello_btn)
        button_adios = tk.Button(self.window, text="¡Me voy!", bg="red",
fg="white", command=self._bye_btn)
        self.text.pack()
        self.name_input.pack()
        button_hola.pack()
        button_adios.pack()
    def mainloop(self):
        self.window.mainloop()
    def _hello_btn(self):
        text = "Hola {}".format(self.name_input.get())
        self._show_text(self.text, text)
    def _bye_btn(self):
        text = "Adiós {}".format(self.name_input.get())
        self._show_text(self.text, text)
    def _show_text(self, widget_out, text):
        widget_out["text"] = text
ventana = my_window()
ventana.mainloop()
```

Como se puede ver, usamos el método get() de Entry para acceder al texto introducido por el usuario.

También podemos crear los típicos radio buttons (botones para seleccionar opciones) con Radiobutton. En este ejemplo, un poco más complejo, usamos una variable de tkinter creada con la función StringVar(), que nos permitirá acceder a los cambios que se hagan en ella dentro del bucle principal. Se la debemos pasar a los Radiobittons, y será esa variable donde guardemos el valor seleccionado (que se india en el value de cada Radiobutton). Además, podemos asignar con command una acción que se activará cada vez que pulsemos ese radiobutton (en lugar de tener que poner un botón exclusivamente para ello, que es lo que también podríamos haber hecho):

```
import tkinter as tk
class my_window():
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Programa")
```

```
        self.window.geometry("500x200")
        self.variable = tk.StringVar()
        radio_button1 = tk.Radiobutton(text="Opción 1",
variable=self.variable, value="uno", command=self._btn)
        radio_button2 = tk.Radiobutton(text="Opción 2",
variable=self.variable, value="dos", command=self._btn)
        radio_button3 = tk.Radiobutton(text="Opción 3",
variable=self.variable, value="tres", command=self._btn)
        self.text = tk.Label(self.window, text="")
        radio_button1.pack()
        radio_button2.pack()
        radio_button3.pack()
        self.text.pack()
    def mainloop(self):
        self.window.mainloop()
    def _btn(self):
        self.text["text"] = "Valor seleccionado:
{}".format(self.variable.get())
ventana = my_window()
ventana.mainloop()
```

Además de estos y otros widgets básico, tkinter dispone de controles más complejos en submódulos. Un ejemplo es filedialog, que es un submódulo de tkinter que nos provee de widgets para interaccionar con fichero, como el clásico diálogo para abrir un fichero:

```
import os
import tkinter as tk
from tkinter.filedialog import askopenfile
class my_window():
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Programa")
        self.window.geometry("500x200")
        button_file = tk.Button(self.window, text="Abrir archivo",
command=self._open_file)
        self.text = tk.Label(self.window, text="")
        button_file.pack()
        self.text.pack()
    def mainloop(self):
        self.window.mainloop()
    def _open_file(self):
        file = askopenfile(mode ="r", filetypes =[("Programillas Python",
"*.py")])
        file_name = os.path.basename(file.name)
        self.text["text"] = "Nombre del archivo: {}".format(file_name)
ventana = my_window()
ventana.mainloop()
```

La función askopenfile() crea un widget que permite navegar por el sistema de archivos y seleccionar un fichero. Además, permite agregar un filtro con el argumento filetipes (al que se le pasa una lista de tuplas con dos valores: Nombre y plantilla de fichero) para que se muestren sólo algunos tipos de archivo. También admite un argumento initialdir en el que indicar el directorio que se mostrará al abrir el widget.

El método retorna un objeto file que puede ser usado como ya hemos visto

anteriormente en este curso.

Para mantener organizado un gran conjunto de operaciones, el método más usual es crear un menú.

Para ello tkinter dispone del widget Menu.

Con este widget crearemos una barra de menú a la que podemos ir agregando conjuntos de opciones.

En tkinter, cada desplegable de un menú es un widget independiente que se crea como un menú contenido en este. De este modo, una barra con dos desplegables se crearía con un menú principal al que se agregarían otros dos menús, cada uno de los cuales tendrá sus elementos.

Cada elemento se añade a un menú con el método add_command() de este último. Con el argumento label le asignamos el texto que queremos que muestre, y con command le añadimos el consabido callback a la función que queremos que ejecute. También se puede añadir una línea de separación con add_separator().

Por último, agregamos el menú a nuestra ventana con el argumento menubar del método config().

```
import tkinter as tk
class my_window():
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Programa")
        self.window.geometry("500x200")
        menubar = tk.Menu(self.window)
        group1 = tk.Menu(menubar, tearoff=0)
        group1.add_command(label="Open", command=self._action)
        group1.add_command(label="New", command=self._action)
        group1.add_command(label="Save", command=self._action)
        menubar.add_cascade(label="Uno", menu=group1)
        group2 = tk.Menu(menubar, tearoff=1)
        group2.add_command(label="Otra cosa", command=self._action)
        group2.add_command(label="Y otra cosa", command=self._action)
        group2.add_separator()
        group2.add_command(label="Guerra nuclear total", command=self._tnw)
        menubar.add_cascade(label="Dos", menu=group2)
        self.window.config(menu=menubar)
        self.text = tk.Label(self.window, text="")
        self.text.pack()
    def mainloop(self):
        self.window.mainloop()
    def _action(self):
        self.text["text"] = "Hacemos algo"
    def _tnw(self):
        self.text["text"] = "¡Bum!"
        self.window.configure(bg="red")
ventana = my_window()
```

```
ventana.mainloop()
```
Una peculiaridad de los menús de tkinter (heredada de TCL/Tk) es el argumento "tearoff" que, si vale 1 (que es el valor por defecto) permite "despegar" el menú como una ventana independiente.

Y esto es sólo el principio de lo que se puede hacer. El paquete tkinter tiene un montón más de widgets, opciones de configuración y herramientas.

Puedes empezar a ver mucho más en su página de documentación oficial.

Tema 9

SQL: Trabajando con bases de datos

Una base de datos es un sistema en el que se almacena información de forma estructurada y se puede acceder a ella de formas complejas.

Existen diversos tipos de bases de datos y de formas de accederá ellas, pero las más extendidas son las bases de datos relacionales, y la forma más común de acceder a la información contenida en ellas es por medio del lenguaje SQL ("Structured Query Language").

Normalmente, una base de datos consiste en un programa que actúa como servidor y que responde a las peticiones que los clientes (personas o programas) le hacen en lenguaje SQL. Estos programas están optimizados para tener una gran eficiencia, rapidez y fiabilidad. Los más famosos son MySQL, MariaDB o PostgreSQL.

Las bases de datos más grandes residen en servidores dedicados, y pueden ser accedidas a través de internet.

Sin embargo, existe un motor de bases de datos llamado SQLite, que evita este modelo cliente-servidor y almacena la base de datos en un archivo local. Esto es muy útil en aplicaciones que necesiten guardar pequeñas cantidades de datos en local (como, por ejemplo, muchas aplicaciones para teléfono móvil). También es tremendamente útil para nosotros, que podemos practicar nuestro acceso a bases de datos sin tener que instalar y configurar un complejo servidor.

Así que para este tema vamos a usar SQLite, teniendo siempre en cuenta que el lenguaje SQL que vamos a ver es el mismo que se puede usar con los grandes servidores de bases de datos, y que la forma de trabajar es la misma.

las bases de datos relacionales organizan su información en tablas que contienen registros (podemos verlos como filas). Para ver cómo funciona, vamos a empezar creando una pequeña base de datos con una tabla:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute("CREATE TABLE biblioteca(Titulo text, Autor text)")
conexion.commit()
conexion.close()
```

Como siempre, comenzamos importando el módulo que vamos a necesitar (sqlite3, en este caso). Luego establecemos la conexión. En un motor como MySQL o PostgreSQL, deberíamos conectar con un servidor, indicando su dirección y su puerto. Pero en SQLite simplemente indicamos un nombre de fichero, donde se almacenará nuestra base de datos.

Después creamos un cursor con el método cursor(), que es, de un modo parecido a cuando leíamos ficheros, el objeto que usaremos para enviar y recibir información de muestra base de datos.

Ahora comenzamos con lo interesante: El método execute() de nuestro cursor nos permite escribir órdenes que luego podremos enviar al motor de bases de datos. Estas órdenes son cadenas de texto, y deben estar escritas en el lenguaje SQL que mencionamos antes. En este caso, la orden es la siguiente:

```
CREATE TABLE biblioteca(Titulo text, Autor text)
```

Esta orden (en SQL se les suele llamar "querys") indica que debe crear una tabla llamada "biblioteca" y que debe contener los campos "Titulo" (que, con "text", debe ser de tipo texto) y Autor (Que también será de tipo texto)

Si ya existiera una tabla con ese nombre, esta query la sobrescribiría. Podemos evitar esto especificando que sólo se debe crear la tabla si no existe ya, del siguiente modo:

```
CREATE TABLE IF NOT EXISTS biblioteca(Titulo text, Autor text)
```

Los comandos SQL pueden estar escritos en minúsculas, mayúsculas o la combinación de estas que prefiramos (no así los nombre de tablas, campos etc), pero es costumbre ponerlos en mayúsculas.

Normalmente, execute() ejecuta la sentencia y retorna el resultado automáticamente. Pero a veces, dependiendo de las circunstancias y del tipo de

query, puede que no ocurra así, y entonces la query no se ejecutará realmente hasta que usemos el método commit() de la conexión, que se asegura de enviarla al servidor y esperar su respuesta.

Por último, es muy importante cerrar la conexión a la base de datos cuando hayamos terminado de usarla con el método close(). Si no lo hacemos, puede haber pérdidas de datos o inconsistencias.

Con esto hemos creado una base de datos vacía, ahora hay que poblarla con algunos datos.

Para insertar registros en una base de datos usamos INSERT de este modo:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute("INSERT INTO biblioteca VALUES ('Cien años de Soledad','García
Marquez')")
conexion.commit()
conexion.close()
```

Nota que toda la parte de la conexión etc es exactamente igual. La única diferencia es la query SQL que estamos usando:

```
INSERT INTO biblioteca VALUES ('Cien años de Soledad','García Marquez')
```

Con esta orden insertamos en la tabla biblioteca un nuevo registro con los valores indicados entre paréntesis tras VALUES '(Cien años de Soledad' y 'García Marquez', en nuestro caso). A SQL, como a Python, no le importa si usamos comillas simples o dobles. Aquí estoy usando comillas simples porque la propia cadena está acotada entre comillas dobles.

Los valores se asignan a los campos que hemos definido al crear la tabla en el mismo orden. Pero, como no tenemos por qué saber el orden o no tenemos por qué usar siempre todos los campos, podemos indicarle los campos a los que asignamos valores poniéndolos entre paréntesis tras el nombre de la base de datos de este modo:

```
INSERT INTO biblioteca (Titulo, Autor) VALUES ('Cien años de Soledad','García
Marquez')
```

El resultado es exactamente el mismo, incluso aunque cambiemos el orden:

```
INSERT INTO biblioteca (Autor, Titulo) VALUES ('García Marquez', 'Cien años
de Soledad')
```

Podemos ejecutar varios comandos en una sola conexión:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute("INSERT INTO biblioteca VALUES ('Seis piezas fáciles','Richard
Feinnman')")
cur.execute("INSERT INTO biblioteca VALUES ('Python paso a paso','Pablo
Hinojosa')")
conexion.commit()
conexion.close()
```

Una vez creados unos cuantos registros, veamos cómo se leen:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute('SELECT * FROM biblioteca')
print(cur.fetchall())
conexion.commit()
conexion.close()
```

Con la query:

```
'SELECT * FROM biblioteca'
```

Estamos pidiendo todos los campos (eso es lo que significa el asterisco) de todos los registros de la tabla biblioteca.

Luego usamos el método fetchall() del objeto cursor para obtener todas las respuestas recibidas.

Si ejecutamos este script, nos mostrará algo como lo siguiente:

```
[('Cien años de Soledad', 'García Marquez'), ('Seis piezas fáciles', 'Richard
Feinnman'), ('Python paso a paso', 'Pablo Hinojosa')]
```

el método fetchall() ha retornado una lista de tuplas, cada una de las cuales se corresponde a un registro de la base de datos.

Podemos acceder a los resultado uno a uno con el método fetchone() de este modo:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute('SELECT * FROM biblioteca')
print(cur.fetchone())
print(cur.fetchone())
print(cur.fetchall())
conexion.commit()
conexion.close()
```

Si ejecutamos este script, la respuesta será algo como esto:

```
    ('Cien años de Soledad', 'García Marquez')
    ('Seis piezas fáciles', 'Richard Feinnman')
    [('Python paso a paso', 'Pablo Hinojosa')]
```

Nota que las dos primeras son las respuestas a los fetchone(), y han retornado sendas tuplas. El fetchall() final retorna, igual que antes, una lista con todos los registros restantes (que, en este caso, es solo uno) en forma de tupla.

Por otro lado, como fetchall nos retorna una lista, podemos usar un bucle for para recorrerla cómodamente.

Si en nuestra base de datos hay entradas repetidas, naturalmente, tendremos respuestas repetidas. Si queremos evitar esto, podemos usar DISTINCT en el SELECT, como se ve en este ejemplo:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute("INSERT INTO biblioteca VALUES ('Visiones peligrosas','Harlam
Ellison')")
cur.execute("INSERT INTO biblioteca VALUES ('Visiones peligrosas','Harlam
Ellison')")
cur.execute('SELECT DISTINCT * FROM biblioteca')
for fila in cur.fetchall():
        print(fila)
conexion.commit()
conexion.close()
```

Además, podemos indicar en qué campo basaremos el orden en el que queremos recibir las respuestas con ORDER BY. Por ejemplo, para que se muestren ordenadas por el campo Autor:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
```

```
cur.execute('SELECT DISTINCT * FROM biblioteca ORDER BY Titulo')
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

Pero, hasta ahora, estamos solicitando todos los registros. Una de las cosas más interesantes de SQL es que podemos pedir sólo registros que cumplan las condiciones que queramos. Por ejemplo, vamos a pedir sólo,los registros para los que el campo Autor sea "Harlam Ellison":

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute("SELECT * FROM biblioteca WHERE Autor = 'Harlam Ellison' ORDER BY
Autor" )
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

La cláusula WHERE indica que queremos sólo los campos que cumpla la condición que le sigue, en este caso, Autor = 'Harlam Ellison'.

Podemos también pedir sólo algunos campos del registro devuelto, en lugar de todos, indicándolos en el SELECT en lugar del asterisco que hemos estado usando hasta ahora. Además, los campos se retornarán en el orden que hayamos indicado:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute("SELECT Autor, Titulo FROM biblioteca WHERE Autor = 'Harlam
Ellison' ORDER BY Autor" )
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

También podemos modificar uno o más campos de uno o más registros mediante la orden UPDATE:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute("UPDATE biblioteca SET Titulo = 'Mi Libro' WHERE Autor = 'Pablo
Hinojosa'")
cur.execute('SELECT * FROM biblioteca')
for fila in cur:
        print(fila)
conexion.commit()
```

```
conexion.close()
```

En este ejemplo, le estamos diciendo a SQL que actualice la tabla biblioteca. En set indicamos, separados por comas,los nuevos pares campo = valor que queremos asignar y, al igual que hacíamos con los SELECT, la condición que deben cumplir los campos para que se aplique la query. En nuestro caso, la orden:

```
UPDATE biblioteca SET Titulo = 'Mi Libro' WHERE Autor = 'Pablo Hinojosa'
```

Vendría a decir "Actualiza la tabla biblioteca poniendo Mi Libro en el campo Título de todos aquellos campos cuyo campo Autor sea Pablo Hinojosa"

Si hubiéramos querido cambiar Tanto Autor como Titulo, deberíamos haber hecho algo así:

```
UPDATE biblioteca SET Titulo = 'Mi Libro', Autor = "Yo" WHERE Autor = 'Pablo
Hinojosa'")
```

Para borrar registros hacemos lo mismo, pero con DELETE:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute("DELETE FROM biblioteca WHERE Autor = 'García Marquez'")
cur.execute('SELECT * FROM biblioteca')
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

Como WHERE 1 es una condición que se cumple siempre, la siguiente orden borrará todos los registros de la base de datos:

```
DELETE FROM biblioteca WHERE 1
```
Lo que, de todos modos, es más fácil y eficiente hacer con TRUNCATE TABLE:

```
TRUNCATE TABLE biblioteca
```
Si lo que queremos es eliminar la tabla y no sólo borrar sus registros, lo podemos hacer con DROP TABLE:

```
DROP TABLE biblioteca
```

Usando campos que no existen

Hasta ahora hemos estado conectado a nuestra base de datos SQLite como un archivo en el mismo directorio que el programa, lo que no suele ser lo normal.

Aunque en los ejemplos seguiremos haciendo lo mismo, es evidente que podemos indicar la ruta que se necesaria en la conexión:

```
conexion =
sqlite3.connect('path/a/la/base/de/datos/donde/sea/que/la/tengamos')
```

Pero, además, podemos usar una base de datos en memoria. Este tipo de base de datos no usa el disco duro, no es permanente (cuando el programa finalice, desaparecerá) y su acceso es más rápido. En determinadas circunstancias puede ser muy útil. Se crean del rugiente modo:

```
conexion = sqlite3.connect(":memory:")
```

Vamos a comenzar creando una nueva tabla. Para ello vamos a usar esta lista de listas:

```
lista_usuarios =[
["Fulano", "Fulanez", 1.80, 90, 123123123],
["Mengano", "Menganez", 1.60, 50, 111111111],
["Zutano","Zutanez", 1.65, 85, 555555555]
]
```

Para empezar, creamos nuestra tabla y sus campos. Hay campos de texto, enteros y en coma flotante:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
query = "CREATE TABLE IF NOT EXISTS Usuarios(Nombre TEXT, Apellidos TEXT, Tlf
INT, Altura FLOAT, Peso FLOAT)"
cur.execute(query)
conexion.commit()
conexion.close()
```

Y, una vez creada, vamos a usar un bucle para poblarla:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
for i in lista_usuarios:
    valores = "'" + i[0] + "','" + i[1] + "', " + str(i[2]) + ", " +
str(i[3]) + ", " + str(i[4])
```

```
    query = "INSERT INTO Usuarios (Nombre, Apellidos, Altura, Peso, Tlf)
VALUES (" + valores + ")"
    print(query)
    cur.execute(query)
conexion.commit()
conexion.close()
```

En ese bucle construimos cada query concatenado valores (Hay un montón de formas mejores de construir esas querys, pero para nuestro ejemplo nos vale), luego los imprimimos (para mostrarlos) y ejecutamos la query como ya sabemos.

Y ya podemos comprobar que todo ha ido correctamente haciendo una query simple:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute('SELECT * FROM Usuarios')
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

Esto nos dará como respuesta algo parecido a esto:

```
('Fulano', 'Fulanez', 123123123, 1.8, 90.0)
('Mengano', 'Menganez', 111111111, 1.6, 50.0)
('Zutano', 'Zutanez', 555555555, 1.65, 85.0)
```

Es decir, los tres registros que hemos creado con exactamente los campos que hemos creado.

pero esto no es del todo cierto. SQLite crea automáticamente, para su propio uso, un campo índice que,normalmente, no nos muestra. Este campo se llama ROWID y podemos verlo así:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute('SELECT Nombre, ROWID FROM Usuarios')
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

SQL nos permite crear nuestros propios campos temporales en los que almacenar valores calculados a partir de otros.

Por ejemplo, nuestra base de datos tiene un campo Peso y un campo Altura, pero no tiene un campo para el "Índice de masa corporal" (Que vamos a suponer aquí que es el peso dividido por la altura).

Naturalmente, podemos obtener los valores de peso y altura para cada registro y hacer que nuestro programa los calcule. Pero también podemos pedirle esos datos a SQL directamente:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute('SELECT Nombre, Peso / Altura FROM Usuarios')
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

Lo que nos retornará ese valor para cada registro como si realmente fuera un campo más.

Pero, además, podemos asignarle un nombre a ese campo usando AS de este modo:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute('SELECT Nombre, Peso / Altura AS imc FROM Usuarios')
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

Esto tiene una gran utilidad en querys complejas, pero también puede servirnos como una forma simple y clara de filtrar resultados usando ese nombre en un WHERE:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute('SELECT Nombre, Peso / Altura AS imc FROM Usuarios WHERE imc =
50.0')
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

Nota que esto es lo mismo que si hubiésemos hecho directamente:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
cur.execute('SELECT Nombre, Peso / Altura AS imc FROM Usuarios WHERE Peso /
Altura')
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

pero nos ahorramos tener que calcular Peso / Altura dos veces y, además, se ve más claramente.

Uniendo tablas

Vamos a comenzar como siempre pero, esta vez, vamos a crear dos tablas más en nuestra base de datos usando dos listas:

```
lista_gente = [
["Fulano", "Fulanez", 1],
["Mengano", "Menganez", 2],
["Zutano","Zutanez", 3],
["Otro","Otrez", 3],
["Tipo","Tipez", 4]
]

lista_ciudades = [
[1,"Granada"],
[2,"Vladivostok"],
[3,"Pequín"],
[5,"Brasilia"]
]

import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
query = "CREATE TABLE IF NOT EXISTS Gente(Nombre TEXT, Apellidos TEXT,
cod_residencia INT)"
cur.execute(query)
query = "CREATE TABLE IF NOT EXISTS Poblaciones(Nombre TEXT, codigo INT NOT
NULL UNIQUE)"
cur.execute(query)
for i in lista_gente:
    valores = "'" + i[0] + "','" + i[1] + "', "  + str(i[2])
    query = "INSERT INTO Gente (Nombre, Apellidos, cod_residencia) VALUES ("
+ valores + ")"
    cur.execute(query)

for i in lista_ciudades:
    valores = "'" + i[1] + "', "  + str(i[0])
    query = "INSERT INTO Poblaciones (Nombre, codigo) VALUES (" + valores +
")"
    try:
        cur.execute(query)
    except:
        print("Has intentado repetir ID de ciudad, pero no pasa nada")
```

```
conexion.commit()
conexion.close()
```

Esto es un poco más largo, pero ya deberíamos poder reconocer la mayoría de lo que estamos haciendo.

Para empezar, hemos creado las tablas como ya sabemos, con la salvedad de que, en la tabla Poblaciones hay un campo codigo que, además de estar definido como INT, le añadimos NOT NULL UNIQUE.

Eso son dos órdenes de SQL:

La primera es NOT NULL, con la que le decimos que el valor no puede ser nulo (es decir, no podemos dejar de asignarle un valor).
La segunda es UNIQUE, con la que le decimos que no puede haber registros en esta tabla con valores repetidos en ese campo.

Si intentamos introducir un registro que tenga en ese campo un valor que ya exista en tro registro previo, no podremos hacerlo y a cambio tendremos una bonita excepción IntegrityError, para lo que hemos puesto ese try.

Una vez creadas ambas bases de datos, podemos comenzar aprobar una de las herramientas más poderosas (y complicadas) de SQL. Los JOINs.

En SQL podemos hacer querys complejas que abarquen varias tablas. Por ejemplo, en nuestro ejemplo, supongamos que queremos obtener los nombres y apellidos de las personas y su lugar de residencia.

Para ello, dado que en la tabla de Gente sólo tenemos los códigos de cuidad y no el nombre, primero debemos hacer una query en la tabla Gente para obtener el código de población de cada uno, luego hacer una query en la tabla de Poblaciones para encontrar los nombres de las ciudades y luego hacer que mi programa use un bucle o algo parecido para asignar el valor de cada una según su código.

Afortunadamente SQL permite hacer esto con una sola query:

```
import sqlite3
conexion = sqlite3.connect('mi_data_base1.db')
cur = conexion.cursor()
query = "SELECT Gente.Nombre, Apellidos, Poblaciones.Nombre FROM Gente INNER
JOIN Poblaciones ON Gente.cod_residencia = Poblaciones.codigo"
cur.execute(query)
for fila in cur:
        print(fila)
conexion.commit()
conexion.close()
```

Esto nos retornará algo como:

```
('Fulano', 'Fulanez', 'Granada')
('Mengano', 'Menganez', 'Vladivostok')
('Zutano', 'Zutanez', 'Pequín')
('Otro', 'Otrez', 'Pequín')
```

Como siempre, el meollo está en la query:

```
SELECT Gente.Nombre, Apellidos, Poblaciones.Nombre FROM Gente INNER JOIN
Poblaciones ON Gente.cod_residencia = Poblaciones.codigo
```

Nota que hay un par de campos "Nombre" a los que se llama precedidos del nombre de su tabla. Como existe un campo con ese nombre en ambas tablas, esto es necesario para diferenciarlos. Salvo ese detalle, estamos solicitando los campos en un SELECT del mismo modo que ya habíamos visto.

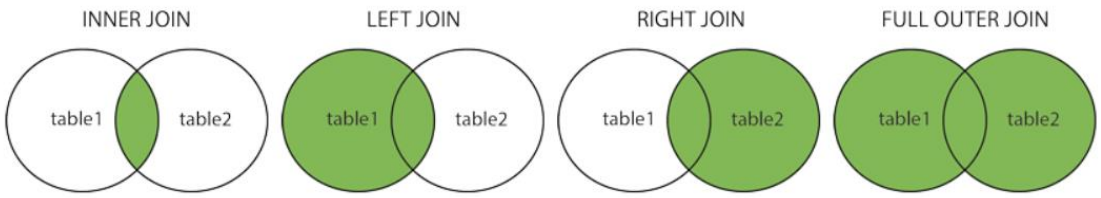
Sin embargo, en el FROM decimos "FROM Gente INNER JOIN Poblaciones". Con esto le estamos diciendo a SQL que queremos el resultado combinado de esas dos tablas (enseguida vamos a explicar esto un poco más). Luego, con ON, le decimos cuál ha de ser el criterio para emparejar los registros de ambas tablas. En nuestro caso, con ON Gente.cod_residencia = Poblaciones.codigo le decimos que el emparejamiento será de tal modo que coincida el cod_residencia de Gente con el codigo de Poblaciones.

Concretamente, INNER JOIN retorna de ambas tablas el subconjunto de registros que cumplen la condición. Es decir, los de la primera tabla que no cumplan la condición serán excluidos, los de la segunda que no la cumplan también serán excluidos (En el dibujo que hay al final de esta página se verá más claro). Es el tipo de JOIN más habitual, porque suele coincidir con el tipo de peticiones que necesitamos responder. Pero no es el único JOIN que existe.

Otras opciones son las siguientes:

```
SELECT Gente.Nombre, Apellidos, Poblaciones.Nombre FROM Gente INNER JOIN
Poblaciones ON Gente.cod_residencia = Poblaciones.codigo
SELECT Gente.Nombre, Apellidos, Poblaciones.Nombre FROM Gente LEFT JOIN
Poblaciones ON Gente.cod_residencia = Poblaciones.codigo
SELECT Gente.Nombre, Apellidos, Poblaciones.Nombre FROM Gente RIGHT JOIN
Poblaciones ON Gente.cod_residencia = Poblaciones.codigo
SELECT Gente.Nombre, Apellidos, Poblaciones.Nombre FROM Gente FULL OUTHER
JOIN Poblaciones ON Gente.cod_residencia = Poblaciones.codigo
```

Para aclarar el significado de cada tipo de JOIN, se suele usar una gráfica como la siguiente, en la que cada círculo representa una de las tablas, y el área verde el subconjunto de los valores que serán retornados por el JOIN.



SQL es un lenguaje extenso, que permite hacer tareas complejas de consulta y actualización de datos de forma rápida y eficiente. Aquí hemos mostrado sólo lo que puede responder al tipo de tareas más habituales, pero SQL es mucho más potente. Para aprender más sobre este lenguaje, La página de SQL del w3school puede ser un buen comienzo.