# Manejo de excepciones

Al momento de estar trabajando en el lenguaje Python, quizá nos hayamos encontrado con diferentes errores, los cuales muchas veces ocurren cuando escribimos mal una línea de código, es decir, nos faltó algún paréntesis, no pusimos los dos puntos después de una sentencia que define un bloque de código (:), errores de identación, entre algunos otros. Por ejemplo, observemos el siguiente código:

```
print("Hola mundo!"
```

Si nosotros intentamos ejecutar ese código, nos debe de mandar el siguiente error:

```
SyntaxError: unexpected EOF while parsing
```

Fijémonos en el nombre del error, el cual nos dice que se trata de un error sintáctico, es decir, que no escribimos bien la instrucción.

Algo similar ocurrirá si intentamos con el siguiente código

```
x = 3 * var * 5
```

Lo anterior nos arrojará lo siguiente:

```
NameError: name 'var' is not defined
```

Nos marca un error de nombres debido a que la variable '*var*' no está definida en el programa.

Así como estos, existen muchos tipos de errores, sin embargo, algunos ocurren en el momento de la interpretación del programa (justo como el error de sintáxis), el cual no nos deja completar la ejecución de nuestro programa. A pesar de esto, habrá muchas veces que no tengamos errores con la sintáxis y nuestro programa se ejecute a la perfección... Pero imagina el caso: *¿Qué pasaría si yo al momento de ejecutar una división, le paso un cero al denominador?* Para este caso, Python nos mandaría un `ZeroDivisionError` , pero vemos que este es, más que otra cosa, un error que se encuentra al momento de ejecución de nuestro programa y puede ser más causado por el usuario que por el propio programador.

Justamente para evitar este tipo de situaciones, utilizaremos *Excepciones* las cuales podemos definir como *errores en tiempo de ejecución*.

## Bloque try-except

Veamos el siguiente código para utilizarlo de ejemplo:

```
n = float(input("Introduce un número: "))
m = 2
print("{0}/{1}={2}".format(n,m,n/m))
```

Si lo ejecutamos, no debería marcar ningún tipo de error, pues sólo estamos efectuando una división entre dos números. Pero... *¿Has notado algo importante?* Cuando le pedimos al usuario que introduzca un número, él tiene la libertad de apretar cualquier letra de su teclado, es decir, a pesar de que nosotros le pidamos que introduzca un número, el usuario puede ser rebelde y escribir alguna letra. Si ese fuera el caso nos marcaría el siguiente error:

```
ValueError: could not convert string to float: 'a'
```

Nos dice que el caracter que introducimos, no puede convertirse a un tipo de dato float. Hay que darnos cuenta que este error fue justamente *en tiempo de ejecución*, es decir, podemos tratarla mediante una excepción. Además, podemos ver que cuando se genera el error termina el programa, y es con la excepción que podemos decirle que la ejecución continúe.

Para poder manejar una excepción, tendremos que añadir las cláusulas try-except dentro de nuestro código. Eso lo podemos lograr de la siguiente manera:

```
try:
        n = float(input("Introduce un número: "))
        m = 3
        print("{0}/{1}={2}".format(n,m,n/m))
except:
        print("Ha ocurrido un error, introduce bien el número")
```

Si traducimos el código a lenguaje humano, estaríamos diciendo lo siguiente: "Intenta el siguiente bloque de código, si se llega a producir un error durante la ejecución, imprime el siguiente mensaje: 'Ha ocurrido un error, introduce bien el número'". Es decir, si ocurre un error durante la ejecución de la división, me imprime el mensaje que está en el bloque except.

Si ejecutamos este código, ahora aunque nosotros metamos un caracter, nos imprimirá el mensaje que nosotros le implementamos.

```
Ha ocurrido un error, introduce bien el número
```

En este código, pareciera que también terminara el programa cuando se lanza la excepción, pero no es así. Para verlo un poco más claro, pongamos un ciclo while en el programa, en donde sólo finalice si es que no se lanza una excepción. Entonces el código quedaría así:

```
while True:
        try:
                n = float(input("Introduce un número: "))
                m = 4
                print("{0}/{1}={2}".format(n,m,n/m))
                break
        except:
                print("Ha ocurrido un error, introduce bien el número")
```

El código se seguirá ejecutando infinitamente debido al while True, y siempre intentará ejecutar el bloque de código; si introducimos un caracter, se lanza la excepción y termina la iteración, después se vuelve a ejecutar el bloque try. Si ahora introducimos un número, se ejecuta correctamente la operación y ejecuta el *break* para terminar el cliclo while. Es importante mencionar que ningún try puede ir sin su respectivo except, ya que de ser así, no sabría qué es lo que hay que ejecutar en caso de error.

# Cláusula Else

Además de las cláusulas try-except que ya conocemos, existirá una más llamada *else*. Esta cláusula *else* ejecutará un cierto bloque de código, en caso de que NO se lance ninguna excepción. Esta cláusula siempre irá después de la cláusula except, así como se muestra en el siguiente código:

```
while(True):
        try:
                n = float(input("Introduce un número: "))
                m = 4
                print("{0}/{1}={2}".format(n,m,n/m))
        except:
                print("Ha ocurrido un error, introduce bien el número")
        else: #Siempre se ejecuta si no ocurre una excepción
                print("Todo ha funcionado correctamente")
                break
```

Si nosotros introducimos un caracter, se lanzará la excepción y nos imprimirá el mensaje de error, pero en caso de que todo funcione correctamente y se ejecute la división, también se lanzará el mensaje del bloque *else*. El bloque else también puede leerse como: "En caso de que no ocurra ninguna excepción, ejecuta el siguiente código".

## Cláusula Finally

Todavía tendremos una cláusula más llamada *finally* la cual se ejecutará SIEMPRE, ocurra algún tipo de excepción o no. Por ejemplo, si se tiene el siguiente código similar a los anteriores:

```
while(True):
        try:
                n = float(input("Introduce un número: "))
                m = 4
                print("{0}/{1}={2}".format(n,m,n/m))
        except:
                print("Ha ocurrido un error, introduce bien el número")
        else: #Siempre se ejecuta si no ocurre una excepción
                print("Todo ha funcionado correctamente")
                break
        finally:
                print("Fin de la iteración")
```

La impresión de pantalla que dice "Fin de la iteración" siempre se ejecutará; a pesar de que nosotros ingresemos un caracter para la variable n, se va a ejecutar el bloque de código de la cláusula finally. Todas estas cláusulas deben siempre tener como complemento el try-except, ya que de otra manera el código no tendría sentido y puede marcar un error

## Raise

Habrá ocasiones en donde nosotros como programadores queramos que se ejecute un error a la fuerza, ya sea porque nosotros creamos nuestro propio error, o bien, porque en una instrucción, no detecta un error y nosotros queremos simularlo. Esto podemos lograrlo con la clausula *raise*. Por ejemplo, imaginemos que tenemos una función que hemos declarado con anterioridad y esta debe de recibir un argumento; en caso de que el argumento sea None, nosotros podemos poner una condicional en donde se fuerce a ejecutar una excepción, como puede ser el caso siguiente:

```
def funcion(algo = None):
    if algo == None:
        raise ValueError("Error! No se permite un valor nulo")
funcion()
```

Si ejecutamos este código, nos debe devolver lo siguiente:

```
ValueError: Error! No se permite un valor nulo
```

Vemos entonces que al utilizar la sentencia *raise*, nosotros podemos forzar un error y a la vez nosotros podemos incluir un mensaje al momento de llamarlo (Como es en el caso del ValueError).

## Jerarquía de excepciones

Cuando nos referimos a la jerarquía de excepciones, decimos entonces que muchos de los errores, provienen de otros. En realidad todos los errores que hemos estado manejando hasta el momento, no son más que clases, y al decir que provienen de otros, estamos diciendo que heredan de ciertas clases. Los diferentes tipos de errores se despliegan a continuación:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

Podemos observar que la clase padre de todas las excepciones es *BaseException*, y a la vez, esta contiene a la clase Exception (la cual contiene a la mayoría de los errores que nos hemos encontrado a lo largo del curso) y además algunos otros errores que se refieren a la interrupción de nuestro programa.

## Creación de nuestros errores

Si sabemos que todo en Python es un objeto (funciones, clases, tipos de datos), podemos imaginarnos que también los errores son objetos. Nosotros podríamos crear nuestro propio error a partir de una clase y ésta la podemos ejecutar mediante la cláusula *raise*. Para poder crear una clase que se comporte como una excepción, tendremos que heredar de una clase padre que se mostró dentro de la jerarquía de excepciones, la cual es *Exception*, pues esta contiene también a muchas de las excepciones que nos encontramos a la hora de codificar.

Nuestro error puede quedar de la siguiente manera:

```
class MiExcepcion(Exception):
    def __init__(self, mensaje):
        self.mensaje = mensaje
def funcion(algo = None):
    if algo == None:
        raise MiExcepcion("Esta es mi excepcion")
funcion()
```

Lo que obtenemos de ejecutar el código anterior es lo siguiente:

```
__main__.MiExcepcion: Esta es mi excepcion
```

Con esto queda comprobado que pude ejecutar mi error personalizado sin ningún problema.

Es importante que a partir de ahora todos nuestros programas utilicen excepciones, ya que de esta manera podemos evitarnos muchos problemas que puede llegar a causar al usuario en su afán de siempre hacer lo contrario a lo que dicen las instrucciones.

Si deseas obtener más información sobre cada una de las excepciones con las que cuenta Python, puedes hacer clic en el siguiente enlace para ir a la documentación: Built-in Exceptions

# Manejo de ficheros

## Módulos os y sys

### Módulo os

El módulo `os` nos permite acceder a funcionalidades dependientes del Sistema Operativo. Sobre todo, aquellas que nos refieren información sobre el entorno del mismo y nos permiten manipular la estructura de directorios (para leer y escribir archivos ver capítulo 9).

#### Archivos y directorios

El módulo `os` nos provee de varios métodos para trabajar de forma portable con las funcionalidades del sistema operativo. Veremos a continuación, los métodos más destacados de este módulo.

| Descripción | Método |
|---|---|
| Saber si se puede acceder a un archivo o directorio | os.access(path, modo_de_acceso) |
| Conocer el directorio actual | os.getcwd() |
| Cambiar de directorio de trabajo | os.chdir(nuevo_path) |
| Cambiar al directorio de trabajo raíz | os.chroot() |
| Cambiar los permisos de un archivo o directorio | os.chmod(path, permisos) |
| Cambiar el propietario de un archivo o directorio | os.chown(path, permisos) |
| Crear un directorio | os.mkdir(path[, modo]) |
| Crear directorios recursivamente | os.mkdirs(path[, modo]) |
| Eliminar un archivo | os.remove(path) |

| Descripción | Método |
|---|---|
| Eliminar un directorio | os.rmdir(path) |
| Eliminar directorios recursivamente | os.removedirs(path) |
| Renombrar un archivo | os.rename(actual, nuevo) |
| Crear un enlace simbólico | os.symlink(path, nombre_destino) |

### El módulo os y las variables de entorno

El módulo os también nos provee de un diccionario con las variables de entorno relativas al sistema. Se trata del diccionario `environ`:

```
import os
for variable, valor in os.environ.items():
    print("%s: %s" % (variable, valor)())
```

### os.path

El módulo `os` también nos provee del submódulo path `os.path` el cual nos permite acceder a ciertas funcionalidades relacionadas con los nombres de las rutas de archivos y directorios. Entre ellas, las más destacadas se describen en la siguiente tabla:

| Descripción | Método |
|---|---|
| Ruta absoluta | os.path.abspath(path) |
| Directorio base | os.path.basename(path) |
| Saber si un directorio existe | os.path.exists(path) |
| Conocer último acceso a un directorio | os.path.getatime(path) |
| Conocer tamaño del directorio | os.path.getsize(path) |
| Saber si una ruta es absoluta | os.path.isabs(path) |
| Saber si una ruta es un archivo | os.path.isfile(path) |
| Saber si una ruta es un directorio | os.path.isdir(path) |
| Saber si una ruta es un enlace simbólico | os.path.islink(path) |
| Saber si una ruta es un punto de montaje | os.path.ismount(path) |

## Módulo sys

El módulo `sys` es el encargado de proveer variables y funcionalidades, directamente relacionadas con el intérprete.

### Variables del módulo sys

Entre las variables más destacadas podemos encontrar las siguientes:

| Variable | Descripción |
|---|---|
| sys.argv | Retorna una lista con todos los argumentos pasados por línea de comandos. Al ejecutar python modulo.py arg1 arg2, retornará una lista: ['modulo.py', 'arg1', 'arg2'] |
| sys.executable | Retorna el path absoluto del binario ejecutable del intérprete de Python |

| Variable | Descripción |
|---|---|
| sys.maxint | Retorna el número positivo entero mayor, soportado por Python |
| sys.platform | Retorna la plataforma sobre la cuál se está ejecutando el intérprete |
| sys.version | Retorna el número de versión de Python con información adicional |

### Métodos del módulo sys

Entre los métodos más destacados del módulo `sys`, podemos encontrar los siguientes:

| Método | Descripción |
|---|---|
| sys.exit() | Forzar la salida del intérprete |
| sys.getdefaultencoding() | Retorna la codificación de caracteres por defecto |
| sys.getfilesystemencoding() | Retorna la codificación de caracteres que se utiliza para convertir los nombres de archivos unicode en nombres de archivos del sistema |
| sys.getsizeof(object[, default]) | Retorna el tamaño del objeto pasado como parámetro. El segundo argumento (opcional) es retornado cuando el objeto no devuelve nada. |

# Objetos file ( Manejo de archivos )

Un archivo es información identificada con un nombre que puede ser almacenada de manera permanente en el directorio de un dispositivo.

### Abrir archivo

Antes de poder realizar cualquier operación de lectura/escritura hay que abrir el archivo con `open()` indicando su ubicación y nombre, seguido, opcionalmente, por el modo o tipo de operación a realizar y la codificación que tendrá el archivo. Si no se indica el tipo de operación el archivo se abrirá en modo de lectura y si se omite la codificación se utilizará la codificación actual del sistema. Si no existe la ruta del archivo o se intenta abrir para lectura un archivo inexistente se producirá una excepción del tipo IOerror.

> objetoArchivo = open(nombreArchivo, modoApertura)

```
ObjArchivo = open('/home/archivo.txt')
ObjArchivo = open('/home/archivo.txt', 'r')
ObjArchivo = open('/home/archivo.txt', mode='r', encoding='utf-8')
```

¿Y qué codificación utiliza nuestro sistema? Podemos averiguarlo ejecutando las siguientes líneas de código:

```
import locale
print(locale.getpreferredencoding())
```

#### Modos de Apertura

| Modo | Significado |
|---|---|
| r | Read - Lectura, el archivo ya debe existir |
| r+ | Lectura/Escritura |
| w | Write - Escritura, si el archivo no existe lo crea, y si existe lo sobreescribe |

| Modo | Significado |
|---|---|
| a | Append - Añadir, agrega datos al final del archivo |
| b | Binario |
| + | Permite lectura/escritura simultánea |
| U | Salto de línea universal: win cr+lf, linux lf y mac cr |
| rb | Lectura binaria |
| wb | Sobreescritura binaria |
| r+b | Lectura/Escritura binaria |

## Cerrar archivo

Después de terminar de trabajar con un archivo lo cerraremos con el método `close()`.

```
ObjArchivo.close()
```

## Leer archivo: read, readline, readlines

- Con el método `read()` es posible leer un número de bytes determinados (caracteres). Si no se indica número se leerá todo lo que reste o si se alcanzó el final de fichero devolverá una cadena vacía. Ejemplo:

```
# Abre archivo en modo lectura
archivo = open('archivo.txt','r')

# Lee los 9 primeros bytes
cadena1 = archivo.read(9)

# Lee la información restaste
cadena2 = archivo.read()

# Muestra la primera lectura
print(cadena1)

# Muestra la segunda lectura
print(cadena2)

# Cierra el archivo
archivo.close
```

- El método `readline()` lee de un archivo una línea completa Ejemplo:

```
# Abre archivo en modo lectura
archivo = open('archivo.txt','r')

# inicia bucle infinito para leer línea a línea
while True:
    linea = archivo.readline()  # lee línea
    if not linea:
        break  # Si no hay más se rompe bucle
    print(linea)  # Muestra la línea leída
archivo.close  # Cierra archivo
```

- El método `readlines()` lee todas las líneas de un archivo como una lista. Si se indica el parámetro de tamaño leerá esa cantidad de bytes del archivo y lo necesario hasta completar la última linea. Ejemplo:

```
# Abre archivo en modo lectura
archivo = open('archivo.txt','r')

# Lee todas la líneas y asigna a lista
lista = archivo.readlines()

# Inicializa un contador
numlin = 0

# Recorre todas los elementos de la lista
for linea in lista:
    # incrementa en 1 el contador
    numlin += 1
    # muestra contador y elemento (línea)
    print(numlin, linea)

archivo.close  # cierra archivo
```

## Escribir en archivo: write, writelines

- El método `write()` escribe una cadena y el método `writelines()` escribe una lista a un archivo. Si en el momento de escribir el archivo no existe se creará uno nuevo. Ejemplo:

```
cadena1 = 'Datos'  # declara cadena1
cadena2 = 'Secretos'  # declara cadena2

# Abre archivo para escribir
archivo = open('datos1.txt','w')

# Escribe cadena1 añadiendo salto de línea
archivo.write(cadena1 + '\n')

# Escribe cadena2 en archivo
archivo.write(cadena2)

# cierra archivo
archivo.close

# Declara lista
lista = ['lunes', 'martes', 'miercoles', 'jueves', 'viernes']

# Abre archivo en modo escritura
archivo = open('datos2.txt','w')

# Escribe toda la lista en el archivo
archivo.writelines(lista)

# Cierra archivo
archivo.close
```

## Mover el puntero: seek(), tell()

- El método `seek()` desplaza el puntero a una posición del archivo y el método `tell()` devuelve la posición del puntero en un momento dado (en bytes). Ejemplo:

```
# Abre archivo en modo lectura
archivo = open('datos2.txt','r')

# Mueve puntero al quinto byte
archivo.seek(5)

# lee los siguientes 5 bytes
```

```
cadena1 = archivo.read(5)

# Muestra cadena
print(cadena1)

# Muestra posición del puntero
print(archivo.tell())

# Cierra archivo
archivo.close
```

## Manejadores de contexto

La sentencia `with` se utiliza con objetos que soportan el protocolo de manejador de contexto y garantiza que una o varias sentencias serán ejecutadas automáticamente. Esto nos ahorra muchas líneas de código, a la vez que nos garantiza que ciertas operaciones serán realizadas sin que lo indiquemos explícitamente. Uno de los ejemplos más claros es cuando leemos un archivo de texto. Al terminar esta operación siempre es recomendable cerrar el archivo. Gracias a with esto ocurrirá automáticamente, sin necesidad de llamar al método close().

Un ejemplo simple de como sería sin la sentencia with cuando queremos abrir un archivo:

```
try:
    f = open("test_with.txt", 'w')
    f.write("prueba")
except:
    print "error al abrir fichero"
finally:
    if f:
        f.close()
```

El código es bastante autoexplicativo. `f` es el nombre al que asignamos un objeto archivo. Esta operación, la apertura de un archivo y escritura en él, puede ocasionar varios errores de naturaleza múltiple (no esta el archivo, no tenemos permisos…). Por ello capturamos la excepción y gestionamos. Si todo sale bien pasaremos a la clausula 'finally' y allí gestionaremos la liberación del recurso, en este caso simplemente cerrar el archivo.

En este caso, somos nosotros los que estamos gestionando el recurso al cerrarlo ocurra o no una excepción. Mientras que somos responsables de responder ante las excepciones que este objeto nos quiera comunicar, gestionar el cierre del recurso es algo que el mismo recurso podría gestionar mejor que nosotros.

¿No estaría mejor abrir el archivo, operar sobre el y listo?

```
try:
    with open("test22.txt",'r') as f:
        f.write("prueba")
except:
    print "no esta el fichero"
```

En este fragmento ocurre eso precisamente, gracias a 'with'. Si observamos notaremos la ausencia del cierre del recurso, la comprobación de que si está asignado a la variable 'f' y por supuesto, la clausula 'finally', ya que es el propio recurso quien se gestiona su propio cierre.

En otras palabras, el protocolo de gestor de contextos definidos en el PEP 343 permite la extracción de la parte aburrida de la estructura try..except..finally en una clase separada manteniendo solo el bloque de interés.

¿Como ocurre esto?

Cualquier tipo puede ser gestionado por 'with' a través de un "Manejador de contexto" (Context manager). Un manejador de contexto son dos funciones que van a ser llamadas por Python durante la ejecución. Los nombres son '**enter**' y '**exit**'. Cuando se opera sobre el objeto se llama a '**enter**' y cuando se sale, por cualquier motivo, se llama a '**exit**'.

Un ejemplo:

```
class With_Test(object):
    def __init__(self, n):
        print "en init..."
        self.n = n
    def __enter__(self):
        print "entrando..."
        return self
    def __exit__(self, exc_type, exec_value, traceback):
        print "saliendo..."
    def metodo(self):
        print "n vale: %s" % self.n

with With_Test(8) as wt:
    wt.metodo()
```

Esto imprimirá por pantalla:

```
en init...
entrando...
n vale: 8
saliendo...
```

1. Primero se llama al método `__enter__` . Puede devolver un valor que será asignado a una variable.
2. El bloque de código bajo `with` se ejecutará. Como si fueran condiciones `try` . Se podrá ejecutar con éxito hasta el final o, si algo falla, puede `break, continue o return o lanzar una excepción` . Pase lo que pase, una vez que el bloque ha finalizado, se producirá una llamada al método `__exit__` . Si se lanzó una excepción, la información se manda a `__exit__` .. En el caso normal, las excepciones podrían ser ignoradas como si fueran una condición finally y serían relanzadas después de que finalice **exit**.

Pensemos que nos queremos asegurar de que un fichero se ha cerrado inmediatamente después de que hayamos escrito en él:

```
class closing(object):
        def __init__(self, obj):
                self.obj = obj
        def __enter__(self):
                return self.obj
        def __exit__(self, *args):
                self.obj.close()

with closing(open('file', 'w')) as f:
        f.write('the contents\n')
```

Aquí hemos hecho que la llamada a `f.close()` se haga después de que se salga del bloque `with` . Ya que el cerrar ficheros es una operación tan común, el soporte para esto ya está presente en la clase `file` . Dispone de un método **exit** que llama a close y que podría ser usado como un gestor de contexto:

```
with open('archivo.txt', 'a') as f:
        f.write('more contents\n')
```

# Serialización

El proceso de serialización consiste en transformar un objeto determinado en un texto en base a un lenguaje específico, para ser almacenado o bien transferido y, por último, restablecido al objeto original. Por ejemplo, guardar una lista de Python en un archivo de texto o base de datos, y luego cargarlo cuando sea necesario. Formatos comunes entre los distintos lenguajes de programación incluyen XML y JSON.

En Python existen distintos tipos de serialización, accesibles a través de la librería estándar: marshal, shelve y pickle. Todos permiten serializar objetos independientemente de la plataforma, pero con algunas diferencias.

## Pickle

Pickle forma parte de la librería estándar de Python, por lo que siempre está disponible. Es rápido, la mayor parte está escrito en C, como el propio intérprete de Python. Puede almacenar estructuras de datos de Python todo lo complejas que se necesite.

¿Qué puede almacenar el módulo pickle?

- Todos los tipos de datos nativos que Python soporta: booleanos, enteros, números de coma flotante, números complejos, cadenas, objetos bytes, arrays de byte y None.
- Listas, tuplas, diccionarios y conjuntos que contengan cualquier combinación de tipos de dato nativos.
- Listas, tuplas, diccionarios y conjuntos de datos que contengan cualquier combinación de listas, tuplas, diccionarios y conjuntos conteniendo cualquier combinación de tipos de datos nativos (y así sucesivamente, hasta alcanzar un máximo nivel de anidamiento8.2).
- Funciones, clases e instancias de clases (con ciertas limitaciones).

Para ver un ejemplo haremos lo siguiente:

Primero declararemos un diccionario:

```
dic = {}
dic['cadena'] = 'texto'
dic['tupla'] = (1,2,3)
dic['lista'] = [4,5,6]
dic['booleano'] = True
dic['numero'] = 56
```

Posteriormente importamos el módulo `pickle` y abrimos el archivo. Con el método `dump()` vamos a serializar el contenido en un archivo.

```
import pickle
with open('datos.pickle','wb') as f:
        pickle.dump(dic,f)
```

Esa última sentencia era muy importante.

El módulo pickle toma una estructura de datos Python y la guarda en un archivo.

Para hacer esto, serializa la estructura de datos utilizando un formato de datos denominado `el protocolo pickle` .

Este protocolo es específico de Python; no existe ninguna garantía de compatibilidad entre lenguajes de programación. Probablemente no puedas abrir el archivo datos.pickle con Perl, PHP, Java u otro lenguaje.

No todas las estructuras de datos de Python se pueden serializar con el módulo pickle. El protocolo pickle ha cambiado varias veces para acomodar nuevos tipos de datos que se han ido añadiendo a Python, pero aún tiene limitaciones.

Como resultado de estos cambios, no existe garantía de compatibilidad entre diferentes versiones de Python. Las versiones nuevas de Python soportan los formatos antiguos de serialización, pero las versiones viejas de Python no soportan los formatos nuevos (puesto

que no soportan los tipos de datos nuevos).

A menos que especifiques otra cosa, las funciones del módulo pickle utilizarán la última versión del protocolo pickle. Esto asegura que dispones de la máxima flexibilidad en los tipos de datos que puedes serializar, pero también significa que el archivo resultante no podrá leerse en versiones de Python más antiguas que no soporten la última versión del protocolo.

La última versión del protocolo pickle es un formato binario. Asegúrate de abrir el fichero en modo binario o los datos se corromperán durante la escritura.

Ahora lo siguiente es hacer otro programa donde obtengamos los valores haciendo lo contrario.

```
import pickle
with open("datos.pickle","rb") as f:
        dic2 = pickle.load(f)

print(dic2)
```

La función `pickle.load()` toma un objeto stream, lee los datos serializados del stream, crea un nuevo objeto Python, recrea los datos serializados en el nuevo objeto Python y devuelve el objeto.

### Serialización con "pickle" sin pasar por un fichero

Los ejemplos de la sección anterior te mostraron cómo serializar un objeto Python directamente a un fichero en disco. Pero ¿qué sucede si no necesitas un fichero? Puedes serializar a un objeto bytes en memoria.

```
>>> dic = {'numero': 56, 'cadena': 'texto', 'booleano': True, 'lista': [4, 5, 6], 'tupla': (1, 2, 3)}
>>> b = pickle.dumps(dic)
>>> type(b)
<class 'bytes'>
>>> dic2 = pickle.loads(b)
>>> print(dic2)
{'numero': 56, 'cadena': 'texto', 'lista': [4, 5, 6], 'booleano': True, 'tupla': (1, 2, 3)}
```

La función `pickle.dumps()` (observa la 's' al final del nombre de la función) realiza la misma serialización que la función `pickle.dump()`. Pero en lugar de tomar como parámetro un objeto stream y serializar sobre él, simplemente retorna los datos serializados.

Puesto que el protocolo pickle utiliza un formato de datos binario, la función `pickle.dumps()` retorna un objeto `bytes`.

La función `pickle.loads()` (de nuevo, observa la 's' al final del nombre de la función) realiza la misma operación que la función `pickle.load()`. Pero en lugar de tomar como parámetro un objeto stream y leer de él los datos, toma un objeto bytes que contenga datos serializados.

# Expresiones regulares

## Regex

Seguramente en muchos lados te has encontrado con páginas o programas que te mandan un mensaje como "*Ingresa un correo válido*" o "*Ingresa una dirección válida*". Y la pregunta es, *¿Cómo detecta tal programa o página que estamos ingresando un correo o dirección válida?* Esa tarea podemos lograrla con la ayuda de las Expresiones Regulares (también conocidas como Regex).

Las expresiones regulares son patrones que nosotros podemos utilizar para encontrar cierta combinación de caracteres. Para poder hacer uso de esta herramienta en Python, basta con que nosotros importemos un módulo llamado *re* (refiriéndose a Regular Expression).

```
import re
```

### Métodos dentro del módulo re

Al igual que en muchos otros módulos, nosotros podemos encontrar diferentes métodos que nos ayudarán a ejecutar diferentes acciones con las cadenas. Por ejemplo, nosotros podemos utilizar el método *search()* para verificar si un patrón se encuentra en alguna posición dentro de la cadena. Observa el siguiente código:

```
import re

texto = "En esta cadena se encuentra una palabra mágica"

#Devuelve un objeto de tipo match
print(re.search("mágica", texto))

#Devuelve un None
print(re.search("hola", texto))
```

Esto es lo que nos muestra al ejecutar el programa:

```
<_sre.SRE_Match object; span=(40, 46), match='mágica'>
None
```

La primera línea nos indica que el programa devuelve un objeto de tipo Match, nos muestra la posición donde encontró la coincidencia (de la posición 40 a 46 en la cadena) y también la palabra con la que coincidió. En la segunda línea se nos indica que el programa devuelve un None, ya que no encontró ninguna coincidencia dentro de la cadena.

Con esto ya tenemos una pequeña muestra de lo que se puede lograr con el módulo *re*, pero éste nos ofrece aún más métodos con los que podemos trabajar. Trabajemos con el mismo código mostrado anteriormente, y ahora veremos 3 métodos más. Si sabemos que el método *search()* nos devuelve un objeto de tipo match, nosotros podemos trabajar con ese objeto para obtener información de él, como pueden ser la posición de la coincidencia dentro de nuestro texto. Eso lo logramos con los métodos *start()*, *end()* y *span()*.

```
import re

texto = "En esta cadena se encuentra una palabra mágica"
encontrado = re.search("mágica", texto)
print(encontrado.start()) #Devuelve un 40, que es donde inicia la equivalencia
print(encontrado.end()) #Devuelve un 46 que es donde termina la equivalencia
print(encontrado.span()) #Nos devuelve el inicio y fin de la equivalencia, en forma de tupla
```

Al ejecutar el programa se nos muestra lo siguiente:

```
40
46
(40,46)
```

Los números que nos devuelve el programa es la posición en donde inicia y termina la coincidencia dentro del texto. La palabra que se buscó ('mágica') inició en la posición 40 dentro de *texto* y termina en la posición 46.

Hasta el momento sólo hemos utilizado el método *search()* para realizar búsquedas dentro de una cadena, pero no es el único que existe. Revisemos entonces un nuevo método llamado *match()*. El método *match()* **busca un patrón al inicio de una cadena**. Lo anterior es muy importante de recordar, ya que suele haber mucha confusión al iniciarse con este tema; se debe tener muy encuentra que **no** se busca la *palabra inicial*, sino el *inicio de cadena*.

Para poder explicarlo mejor, revisemos el siguiente código:

```
import re
texto = "Hola mundo"
print(re.match("Hola", texto))
```

Con el código anterior nosotros obtendríamos un objeto de tipo match, indicando que hubo una coincidencia.

```
<_sre.SRE_Match object; span(0,4), match='Hola'>
```

Todo muy bien hasta aquí, sin embargo... *¿Qué pasa si yo en mi texto pongo un 'espacio' al inicio?*. Podríamos pensar que también obtendríamos un match, pues al fin y al cabo, el texto está empezando con la palabra 'Hola', ¿No?... Pues NO. Como se había mencionado antes, el método *match()* busca el patrón al inicio de la cadena, y aunque se tenga un espacio, el propio espacio también se considera parte de la cadena; en estos casos se le suele llamar 'cadena vacía', ya que nosotros no vemos nada pero ahí está presente.

```
import re
texto = " Hola mundo" #Se agrega un espacio al inicio de la cadena
print(re.match("Hola", texto))
```

Para este caso, lo único que obtendremos en un None.

```
None
```

También se cuenta con un método capaz de cambiar una cadena por otra en caso de que se encuentre alguna coincidencia especificada. Lo anterior es posible con el método *sub()*, el cual recibe tres parámetros: la palabra a buscar, la palabra por la cuál se quiere intercambiar y el texto en el cuál buscar. Veamos el ejemplo:

```
import re

texto = "Hola amigo"
print(re.sub("amigo", "amiga", texto)) #Cambia la palabra 'amigo' por 'amiga'
```

Este método no devuelve ningún objeto de tipo match, sino que devuelve la cadena con los cambios especificados:

```
Hola amiga
```

Si nos fijamos bien en los ejemplos anteriores, sólo hemos revisado el caso en donde sólo se encuentra una coincidencia, pero... *¿Qué pasa si dentro de mi texto se encuentra la palabra buscada más de una vez?*. Los métodos anteriores funcionarán de la misma manera, sólo que encontrarán siempre la primera coincidencia. Si lo que se quiere es encontrar todas las coincidencias dentro de una cadena, podemos utilizar el método *findall()*. Observemos el siguiente ejemplo:

```
import re
texto = "hola python hola hola"
print(re.findall("hola",texto))
```

Lo que nos devuelve el método *findall()* será una lista con todas las coincidencias encontradas.

```
['hola','hola','hola']
```

Si nosotros quisiéramos saber el número de veces que se encontró una palabra, basta con utilizar el método *len()* que se estudió en la sección de tipos de datos, ya que se está trabajando con una lista.

```
import re
texto = "hola python hola hola"
print(len(re.findall("hola",texto"))) #Usando el método len() con una lista
```

Lo que se obtiene es la longitud de la lista:

```
3
```

> En este momento, el método *findall()* puede parecer de muy poca utilidad, sin embargo, su importancia se verá más adelante con el uso de los metacaracteres.

# Metacaracteres

Los metacaracteres son caracteres no alfabéticos que poseen un significado especial en las expresiones regulares. En la siguiente tabla se mostrarán varios de los metacaracteres existentes y para qué nos puede servir cada uno dentro de una expresión regular.

| Metacarácter | Descripción |
|---|---|
| . | Concuerda con cualquier caracter individual |
| * | Los caracteres anteriores o rango de valores pueden concordar cero o más veces |
| + | Los caracteres anteriores o rango de valores pueden concordar una o más veces (Siempre al menos una vez) |
| ? | Los caracteres anteriores son una parte opcional de la expresión; pueden concordar cero o una vez. |
| $ | Coincide con el fin de una serie |
| ^ | Coincide con el principio de una serie. Si se usa dentro de corchetes "[]" indica que ningún caracter dentro de ellos debe existir en la cadena de búsqueda. |
| () | Indica que los caracteres entre paréntesis se deben tratar como un patrón de caracteres. Puede contener un grupo de valores |
\| | Coincide con alguno de los patrones en alguno de los extremos de la barra vertical (uno u otro) |
| \ | Indica que el metacarácter siguiente se debe tratar como un caracter normal |
| [ ] | Contiene caracteres individuales y rangos de valor que deben concordar |
| {m, n} | Coincide de *m* a *n* instancias del patrón precedente |
| {m,} | Coincide con *m* o más instancias del patrón precedente |
| {m} | Coincide exactamente con m instancias del patrón precedente |

Ya que se conocen los metacaracteres y para qué sirven, veamos algunos ejemplos dentro de las expresiones regulares. Se utilizará el método *match()* visto anteriormente para cada uno de los ejemplos siguientes.

El primer ejemplo que veremos, será sin uso de ningún metacaracter.

- Sin uso de metacaracter

```
import re

if re.match("hola","hola"):
    print("Hizo match")
```

En este pequeñísimo programa, estamos diciendo que si la palabra *hola* se encuentra dentro de la cadena *hola* nos imprima la cadena "*Hizo match*". Efectivamente este será el caso, por lo que el programa imprimirá lo siguiente:

```
Hizo match
```

Debemos recordar que el método match nos servirá para poder encontrar la coincidencia al inicio de nuestra cadena de búsqueda. Teniendo lo anterior en mente, ahora pasemos al uso de los metacaracteres.

- Uso del metacaracter "."

```
import re

if re.match(".ola","hola"):
    print("Hizo match")
```

Haciendo uso de nuestra tabla, podemos ver que el metacaracter punto ".", nos servirá para reemplazarlo por cualquier caracter que nosotros queramos (puede tomar cualquier valor). Para esta caso estamos diciendo que la cadena debe empezar con cualquier caracter seguido de la cadena "ola", y vemos que efectivamente la cadena *hola* cumple con esa condición, por lo que el programa devuelve lo siguiente:

```
Hizo match
```

- Uso del metacaracter "\"

```
import re

if re.match("\.ola",".ola"):
    print("Hizo match")
```

Haciendo un programa muy similar al anterior, veamos el uso del metacaracter "". Podemos ver en la tabla que el metacaracter siguiente se debe de tratar como un caracter normal, es decir, el metacaracter "." ahora sí deberá tomarse como un punto dentro de la cadena de búsqueda; ya no cumplirá su función que era ser reemplazado por cualquier valor, sino que ahora sí tendremos que poner forzosamente al inicio de la cadena de búsqueda un punto. El programa devuelve lo siguiente:

```
Hizo match
```

- Uso del metacaracter "|"

```
import re

if re.match("python|jython|cython","cython"):
    print("Hizo match")
```

En el anterior programa, estamos haciendo uso del metacaracter "|", el cual nos indica que se tendrá una coincidencia si se tiene alguno de sus patrones que se encuentran en sus extremos. En otras palabras, habrá una coincidencia si nuestra palabra dice "python", "jython", o bien, "cython"; puede ser cualquiera de las tres y ninguna más. En el programa de muestra, se imprime lo siguiente:

```
Hizo match
```

- Uso del metacaracter "[]"

```
import re

if re.match("[pcj]ython","jython"):
    print("Hizo match")
```

Cuando nosotros colocamos caracteres dentro de los corchetes, estamos indicando que **un solo caracter** puede tomar cualquiera de esos valores, en este caso, ese caracter puede tomar el valor de p, j, o bien, una "c"; seguido de ese caracter se debe encontrar la cadena "ython". Por lo tanto, con la cadena de búsqueda "jython", el programa devuelve lo siguiente:

```
Hizo match
```

- Uso del metacaracter "()"

```
import re

if re.match("niñ(o|a)s","niños"):
    print("Hizo match")
```

De la tabla sabemos que los caracteres entre paréntesis se deben tratar como un patrón de caracteres, es decir, nosotros estamos agrupando un valor que puede ser, una "a" o una "o", dentro de las cadenas "niñ" y "s". Puede elegirse alguno de los dos caracteres especificados, e incluso dentro de los paréntesis podrían ponerse cadenas completas como podría ser (python|jython|cython) y cumpliría la misma función. Este programa devuelve lo siguiente:

```
Hizo match
```

## Rangos

Aunque nosotros ya vimos el uso del metacaracter "[]", lo vimos únicamente para poder hacer uso de los caracteres que se encontraban dentro de ellos, sin embargo, nosotros podemos usar el mismo metacaracter para indicar rangos de valores, como se muestra a continuación:

| Rango | Descripción |
|---|---|
| [0-9] | Indica un caracter que puede tomar un valor del 0 al 9 |
| [a-z] | Indica que un caracter puede tomar cualquier valor que sea una letra minúscula (a, b, c, d, ..., z) |
| [A-Z] | Indica que un caracter puede tomar cualquier valor que sea una letra mayúscula (A, B, C, D, ..., Z) |
| [0-9a-zA-Z] | Indica que un caracter puede tomar cualquier valor alfanumérico |
| [a-f5-8] | Indica que un caracter puede tomar culquier valor de la "a" a la "f" minúscula, o bien, algún valor numérico del 5 al 8. |

- Uso del metacaracter ^

```
import re

if re.match("h[^o]la","hula"):
    print("Hizo match")
```

Con la descripción de la tabla de metacaracteres, sabemos que el "^" nos puede servir para la negación de caracteres. En este ejemplo nos indica que todo lo que está dentro de los corchetes, no debe de ir en la cadena de búsqueda entre las cadenas "h" y "la". En este programa en específico nos indica que entra "h" y "la" puede existir cualquier caracter **a excepción de la letra "o"**. El programa devuelve:

```
Hizo match
```

## Cuantificadores

Dentro de los caracteres existen los llamados cuantificadores, los cuales nos indican la cantidad de veces que puede existir algún caracter dentro de una cadena de búsqueda. Entre los cuantificadores encontramos a los siguientes metacaracteres: +, *, ?, {}. Veamos un ejemplo con cada uno de ellos

```
import re

if re.match("python+", "python"): #Sin la n no hace match
        print("Hizo match 1")

if re.match("python*", "pytho"):
        print("Hizo match 2")

if re.match("python?", "pytho"):
        print("Hizo match 3")

if re.search("python{3}s", "pythonnns"):
        print("Hizo match 4")
```

El primer *if* nos indica que el caracter anterior a "+" (en este caso la letra "n") puede ir una o más veces. Es decir puede tener la palabra "python" o bien "pythonnnnnn", pero sin la letra "n" No haría match.

El segundo *if* muestra el uso del metacaracter "*", el cual nos dice que el caracter anterior (la letra "n") puede ir cero o más veces dentro de nuestra cadena de búsqueda. Con esto podemos tener la palabra "pytho" o "pythonnnn" y aún así imprimiría el match.

El tercer *if* muestra el uso del metacaracter "?", el cual nos indica que el caracter anterior (la letra "n") puede ir cero o una vez, es decir, puede o no existir. En la cadena de búsqueda, entonces, podríamos tener "pytho" o "python", pero nunca "pythonnnn". En el ejemplo imprimiría el match.

El cuarto y último *if* muestra el uso del metacaracter "{}", el cual indica que el caracter anterior a ellos (la letra "n") debe de ir exactamente 3 veces entre las cadenas "pytho" y "s", es decir, habrá coincidencia con la palabra "pythonnns" pero nunca con las cadenas "python" o "pythonnnns". Recordemos que con el uso de las llaves, podemos indicarle las veces exactas que queremos que aparezca algún caracter. Si nosotros utilizaramos el patrón "python{3-5}s", le estaríamos indicando que la letra "n" puede aparecer de 3 a 5 veces ("pythonnns", "pythonnnns", "pythonnnnns"), pero ni una más ni una menos.

El programa anterior imprime lo siguiente:

```
Hizo match 1
Hizo match 2
Hizo match 3
Hizo match 4
```

- Uso de los metacaracteres $ y ^

Ahora veamos el uso de los metacaracteres $ y ^ (en su otra forma).

De la tabla sabemos que $ coincide con el fin de una serie, es decir, si lo ponemos delante de cualquier caracter, estamos indicando que la cadena de búsqueda debe de terminar con dicho caracter. Con esto se puede intuir que el metacaracter $ debe de ir al final de la expresión regular. Para poder demostrar su utilizad, usaremos el método *search()* ya que con *match()* puede dificultar su comprensión.

```
import re

if re.search("Hola$","Adios Hola"):
        print("Hizo match")
```

En el programa estamos indicando que la cadena de busqueda debe de encontrar una cadena que diga "Hola" y además, que dicha cadena termine con el caracter "a". En este caso el programa imprime lo siguiente:

```
Hizo match
```

Pasando ahora con el metacaracter ^, podemos ver en la tabla que este nos indica el inicio de una cadena. Observa el siguiente ejemplo:

```
import re

if re.search("(^http)", "http://google.com"):
        print("Hizo match")
```

Con el metacaracter ^ y con ayuda de los paréntesis, le estamos indicando que nuestra cadena de búsqueda debe iniciar con la cadena "http", si esta no se encontrara al inicio, no marcaría ninguna coincidencia. En este caso, el programa devuelve:

```
Hizo match
```

Con estos ejemplos se ha visto el uso que tienen los metacaracteres dentro de las expresiones regulares, sin embargo, a veces pueden llegar a ser un poco complicados de escribir o de interpretar dentro de una expresión regular, es por eso que también se tienen algunas simplificaciones para algunos de los rangos. Aquí se tienen algunas de estas simplificaciones:

| Equivalencia | Descripción |
|---|---|
| \d | equivale [0-9] |
| \D | equivale [^0-9] |
| \A | iniciar con el match |
| \w | equivale [a-zA-Z_] alfanumérico |
| \W | equivale [^a-zA-Z_] |
| \s | equivale a cualquier caracter en blanco [ \n\t] |
| \S | equivale a cualquier caracter no en blanco |

A veces todos estas expresiones regulares, con metacaracteres en conjunto, pueden llegar a ser un poco complicadas en programas más elaborados. En el siguiente ejemplo, podemos ver una forma en la cual podemos validar correos con ayuda de las expresiones regulares.

```
import re
correo = input("Ingresa un correo: ")
expresion = "^[\w\d]+[\w\d.-]*@{1}\w+\.[a-z]{2,5}\.?[a-z]{2,5}?$"
if re.match(expresion, correo):
        print("Correo valido")
else:
        print("Correo invalido")
```

Pueden llegar a encontrarse expresiones regulares monstruosas, y puede que no solo exista una forma de identificar un patrón, pero con toda esta teoría ya es posible identificar y crear expresiones regulares simples o complejas cada vez que lo necesitemos.