



Curso Intensivo de Python - Tema 7

Agustín Arenas



Curso Intensivo de Python - Tema 7

- 1) Archivos
- 2) Serialización
- 3) Try y except
- 4) Cadenas



Archivos

- Cuando un programa necesita guardar información para un uso posterior, o porque cumple su objetivo, escribe dicha información en un archivo
- Así mismo, la información puede leerse desde un archivo para llevar a cabo operaciones de un programa
- Python tiene dos tipos de archivos: texto y binario. Un archivo de texto contiene información codificada en texto plano como ASCII o Unicode mientras que un binario los datos no son convertidos a texto (Solo un programa puede leerlo)

Archivos I

```
def leerArchivo(ubicacion_nombre):
    archivo = open(ubicacion_nombre, "r")
    #open(path,modo) permite abrir el archivo en diversos modos
    #r -> Lectura          w-> Escritura          a-> Añadir
    #rb -> Lectura binario  wb-> Escritura binario
    #El path en Windows debe hacerse con r'LocacionArchivo'
    todo_contenido = archivo.read()
    print('Todo el contenido:')
    print(todo_contenido)
    print('Se imprime linea por linea:')
    archivo.seek(0) #Se resetea el valor de la posicion de lectura
    for linea in archivo:
        print(linea, end='')
    archivo.seek(0)
    print('\nLectura mediante while y readline():')
    linea = archivo.readline() #Se lee hasta un '\n'
    while (linea != '***'):
        print(linea, end='')
        linea = archivo.readline()
    archivo.close() #FUNDAMENTAL ;NO OLVIDARSE DE CERRAR EL ARCHIVO!

def main():
    archivo = "C:/Users/agusk/OneDrive/Escritorio/Curso
Python/CursoIntensivoPython/Tercera clase/Código fuente y archivos/archivo.txt"
    leerArchivo(archivo)
    print('\nFin programa')
main()
```



Archivos II

```
import math as m

def leerArchivoContextManager(ubicacion_nombre):
    with open(ubicacion_nombre, "r") as archivo: #El
context manager automaticamte
        #cierra el archivo
        for linea in archivo:
            print(linea, end='')

def
escribirArchivo(ubicacion_archivo, nuevo_append, lis
ta_datos):
    modo = "w" if nuevo_append == True else "a"
    archivo = open(ubicacion_archivo, modo)
    for dato in lista_datos:
        archivo.write(dato)
    archivo.close()
```



Archivos II

```
def main():
    archivo = "C:/Users/agusk/OneDrive/Escritorio/Curso
Python/CursoIntensivoPython/Tercera clase/Código fuente y archivos/archivo.txt"
    leerArchivoContextManager(archivo)
    print('\nAhora se añaden datos...')
    datos = [
        '\n***\n',
        'Esto es la nueva linea y debe añadirse\n',
        'Siempre debe ser una cadena\n',
        '¡Ojo!\n',
        'Para Python pi='+str(m.pi)+'\n'
    ]
    escribirArchivo(archivo,False,datos)
    print('Se lee con nueva funcion:')
    leerArchivoContextManager(archivo)
    print('\nFin programa')
main()
```



Serialización

- Cuando se escribe un archivo que ya existe, su contenido se borra completamente
- Al leer un archivo, siempre se toma el `\n` por lo que se pueden usar funciones de cadenas como `strip()`
- Los archivos binarios son útiles para la serialización de objetos...

Serialización




- Si pensamos al objeto (lista, conjunto, objeto definido por el desarrollador, etc) como un flotador, el mismo es útil cuando está inflado
- Cuando se desinfla, sigue siendo en este caso un pato, pero ocupa menos espacio



Serialización

- En Python esto se llama **pickling**
- Con **`pickle.load(archivo)`** se carga el objeto
- Con **`pickle.dump(archivo, obj)`** se guarda el objeto



```
import pickle #Librería para serializar los objetos
SUELDO_BASE = 80000
```

```
"""
```

```
    Definiremos un diccionario
    Empleado que tiene los siguientes
    atributos:
```

```
+-----+
```

```
| Empleado |
```

```
+-----+
```

```
| IDEmpleado |
```

```
+-----+
```

```
| Nombre |
```

```
+-----+
```

```
| Apellido |
```

```
+-----+
```

```
| Antigüedad |
```

```
+-----+
```

```
| Sueldo* | (Base = 80.000, 15% por aca año de antigüedad)
```

```
+-----+
```

```
"""
```

```
def serializar(lista_empleados,nombre_archivo,escribir_agregar):
    modo = 'wb' if escribir_agregar else 'a' #SE DEBE ESCRIBIR EN BINARIO
    with open(nombre_archivo,modo) as archivo:
        for empleado in lista_empleados:
            pickle.dump(empleado,archivo) #Serializa el objeto
```


Serialización



Serialización

```
def deserializar(nombre_archivo, lista_empleados):
    with open(nombre_archivo, 'rb') as archivo:
        fin_archivo = False
        while (not fin_archivo):
            try:
                empleado = pickle.load(archivo)
                lista_empleados.append(empleado)
            except EOFError:
                fin_archivo = True

def imprimir_datos(empleado):
    print('Empleado ID: {0}'.format(empleado.get('IDEmpleado', 'Error en key')))
    print('Nombre: {0} \t Apellido: {1}'.format(
        empleado.get('Nombre', 'Error en key'),
        empleado.get('Apellido', 'Error en key')
    ))
    print('=====')
    print('Antigüedad: {0} años \t Sueldo Neto: {1:.2f}'.format(
        empleado.get('Antigüedad', 'Error en key'),
        SUELDO_BASE*0.15*empleado.get('Antigüedad', 'Error en key')
    ))
    print('\t\t\t---o---')
```



```
def main():
    empleado = {}
    lista_empleados = []
    empleado['IDEmpleado'] = '00000001'
    empleado['Nombre'] = 'Homero'
    empleado['Apellido'] = 'Simpson'
    empleado['Antiguedad'] = 25
    lista_empleados.append(empleado)
    empleado = {}
    empleado['IDEmpleado'] = '00000010'
    empleado['Nombre'] = 'Peter'
    empleado['Apellido'] = 'Griffin'
    empleado['Antiguedad'] = 10
    lista_empleados.append(empleado)
    empleado = {}
    empleado['IDEmpleado'] = '00000011'
    empleado['Nombre'] = 'Stan'
    empleado['Apellido'] = 'Smith'
    empleado['Antiguedad'] = 30
    lista_empleados.append(empleado)
    print('Datos:')
    for empleado in lista_empleados:
        imprimir_datos(empleado)
    archivo = 'empleado.data'
    serializar(lista_empleados, archivo, True)
    print('Serializacion finalizada')
    lista_deserializacion = []
    deserializar(archivo, lista_deserializacion)
    print('Objetos leidos... Imprimiendo datos')
    for empleado in lista_deserializacion:
        imprimir_datos(empleado)

main()
```

Serialización



Try y except

- En el código anterior hemos usado dos palabras reservadas **try/except**, pero... ¿Para qué sirven?
- Una excepción es un error que ocurre mientras un programa se ejecuta, provocando que el programa termine de forma abrupta. Pero podemos tratar estos errores mediante el manejo de excepciones
- El bloque try/except se compone principalmente de estos dos bloques:
 - El bloque try donde se ponen las sentencias que **pueden** generar una excepción
 - Uno o más bloques except que **manejan** las excepciones

Try y except

- La listas de excepciones puede encontrarse en la documentación oficial

Python » English » 3.11.2 » 3.11.2 Documentation » The Python Standard Library » Built-in Exceptions

Quick search | previous | next | modules | index

Built-in Exceptions

In Python, all exceptions must be instances of a class that derives from `BaseException`. In a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class's constructor.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, and not from `BaseException`. More information on defining exceptions is available in the Python Tutorial under [User-defined Exceptions](#).

Exception context

When raising a new exception while another exception is already being handled, the new exception's `__context__` attribute is automatically set to the handled exception. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used.

This implicit exception context can be supplemented with an explicit cause by using `from with raise`:

```
raise new_exc from original_exc
```

The expression following `from` must be an exception or `None`. It will be set as `__cause__` on the raised exception. Setting `__cause__` also implicitly sets the `__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g. converting `KeyError` to `AttributeError`), while leaving the old exception available in `__context__` for introspection when debugging.

The default traceback display code shows these chained exceptions in addition to the traceback for the exception itself. An explicitly chained exception in `__cause__` is always shown when present. An implicitly chained exception in `__context__` is shown only if `__cause__` is `None` and `__suppress_context__` is false.

Table of Contents

- Built-in Exceptions
 - Exception context
 - Inheriting from built-in exceptions
 - Base classes
 - Concrete exceptions
 - OS exceptions
 - Warnings
 - Exception groups
 - Exception hierarchy

Previous topic
Built-in Types

Next topic
Text Processing Services

This Page
[Report a Bug](#)
[Show Source](#)

Link de documentación



Try, except y finally

- Pueden añadirse dos bloque adiciones:
- El bloque **else** que se ejecuta **siempre** que no se alcance una excepción
- El bloque **finally** que se ejecuta **siempre**, independiente si se alcanza o no una excepción

Cadenas

- Las cadenas tienen una serie de funciones predefinidas muy útiles que podemos encontrar en la documentación oficial

Table of Contents

- Built-in Types
 - Truth Value Testing
 - Boolean Operations — `and`, `or`, `not`
 - Comparisons
 - Numeric Types — `int`, `float`, `complex`
 - Bitwise Operations on Integer Types
 - Additional Methods on Integer Types
 - Additional Methods on Float
 - Hashing of numeric types
- Iterator Types
 - Generator Types
- Sequence Types — `list`, `tuple`, `range`
 - Common Sequence Operations
 - Immutable Sequence Types
 - Mutable Sequence Types
 - Lists
 - Tuples
 - Ranges
- Text Sequence Type — `str`
 - String Methods
 - `printf`-style String Formatting
- Binary Sequence Types — `bytes`, `bytearray`, `memoryview`
 - Bytes Objects
 - `Bytearray` Objects
 - Bytes and `Bytearray` Operations
 - `printf`-style Bytes Formatting
 - Memory Views
- Set Types — `set`, `frozenset`
- Mapping Types — `dict`

String Methods

Strings implement all of the [common](#) sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, [Format String Syntax and Custom String Formatting](#)) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle ([printf-style String Formatting](#)).

The [Text Processing Services](#) section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.capitalize()`

Return a copy of the string with its first character capitalized and the rest lowercased.

Changed in version 3.8: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

`str.casefold()`

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter 'ß' is equivalent to "ss". Since it is already lowercase, `lower()` would do nothing to 'ß', `casefold()` converts it to "ss".

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

`str.center(width[, fillchar])`

Return centered in a string of length `width`. Padding is done using the specified `fillchar` (default is an ASCII space). The original string is returned if `width` is less than or equal to `len(s)`.

`str.count(sub[, start[, end]])`

Return the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

If `sub` is empty, returns the number of empty strings between characters which is the length of the string plus one.

`str.encode(encoding='utf-8', errors='strict')`

Return the string encoded to `bytes`.

encoding defaults to 'utf-8'; see [Standard Encodings](#) for possible values.

errors controls how encoding errors are handled. If `strict` (the default), a `UnicodeEncodeError` is raised if

[Link de documentación](#)



Cadenas

```
cadena = 'una cadena cualquiera'
print(cadena.capitalize())#Capitalizacion
print(cadena.split())#Separa por espacios, devuelve una lista
print(cadena.upper())#MAYUSCULAS!
print(cadena.lower())#minusculas!
cadena = '          una cadena          '
print(cadena.strip()+'.')#Remeuve espacios o caracteres
caracter = 'a'
print(caracter.isalpha())
print(caracter.isdigit())
digito = '10'
print(digito.isdigit())
```



¿Preguntas?