Programación Avanzada
IIC2233 2023-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattan

Anuncios

Jueves 26 de octubre 2023

1. Hoy tendremos la quinta experiencia.

2. Recuerden ir avanzando en su tarea 2.

3. Encuesta de Carga Académica ¡Respóndanla!

I/O

Forma de interactuar con un programa.

En el contexto de archivos, todo archivo se guarda en un computador como *bytes*.

Un programa es capaz de leer al nivel más mínimo de un archivo y manipularlo para generar más información.

Abriendo archivos con Python

Sabemos que, a su nivel más básico, el contenido de un archivo se compone de ***bytes***.

Sin embargo, dependiendo del contexto que le demos, podemos interpretar dichos *bytes* de **distintas formas** y, por lo tanto, llegar a **distintos resultados** al momento de abrir un archivo.

Una primera pista de cómo abordar un archivo se da por la extensión que este posea. Sin embargo, es importante recordar que en realidad, **cualquier archivo puede ser abierto por cualquier programa**. Lo que importa es si el programa sabe cómo extraer la información del archivo o no de forma correcta.

Abriendo archivos con Python

En Python, la forma más común de abrir archivos es con la función:

```
open(RUTA_ARCHIVO, MODO, encoding=ENCODING, errors=ERRORES)
```

Donde:

- `MODO`: modo en que queremos abrir el archivo (“x” para solo crear el archivo, “r” para solo lectura, “w” para sobreescribir la información original con información nueva, “a” para añadir información nueva después de la original, entre otros).

  También define la forma en que queremos ver la información (“t” para texto decodificado o “b” para *bytes*).

Abriendo archivos con Python

En Python, la forma más común de abrir archivos es con la función:

```
open(RUTA_ARCHIVO, MODO, encoding=ENCODING, errors=ERRORES)
```

Donde:

- `errors`: a veces no sabemos exactamente qué encoding se usó al escribir un archivo, y probamos con uno que no necesariamente podrá representar los contenidos del archivo en su totalidad. Este argumento permite indicar qué hacer en casos donde aparezcan *bytes* que no pudieron ser decodificados correctamente.

  Algunas opciones son '*strict*', '*ignore*', '*replace*', etc.

Acceder al contenido del archivo con Python

Podemos hacer uso de los contenidos del archivo de dos formas:

```
file = open("data/archivo_ejemplo", "r")

print(file.read())

file.close()
```

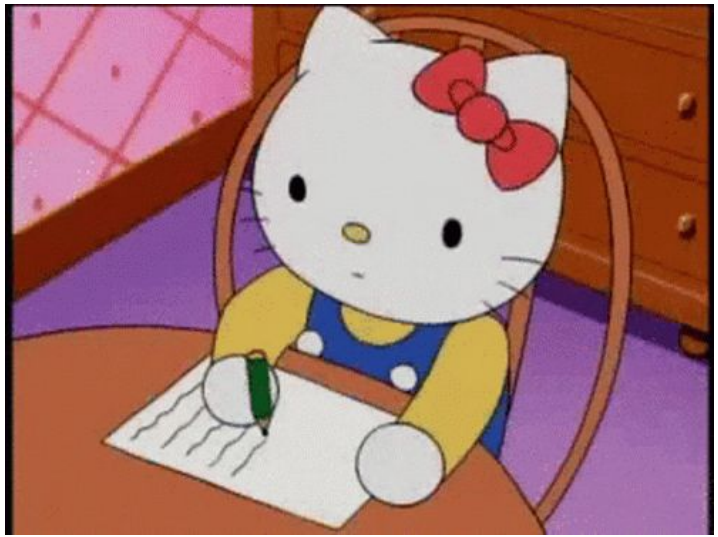
Almacenando el resultado de `open()` en una variable y usando sus métodos.
(¡Debemos recordar cerrar el archivo! 👀)

```
with open("data/archivo_ejemplo", "r") as file:

    contenido = file.read()
```

Con un *context manager* nos olvidamos de cerrar el archivo, se hará de forma automática al salir del bloque interno 😎

Strings



- Objetos que representan cadenas de caracteres
- Nos permiten trabajar con texto
- Secuencias inmutables
- Poseen varias funcionalidades de gran utilidad para su manejo

Strings como secuencias de caracteres

Conceptualizar *strings* de esta forma nos aplicar operaciones de secuencias de datos, como lo son las listas, a *strings*:

```
mi_string = "hola!"

mi_string[0]          # "h"

mi_string[-1]         # "!"

mi_string[1:3]        # "ol"

"Sopai" + "pillas"    # "Sopaipillas"

mi_string[0] = "H"    # Nop 👮
```

Podemos hacer *slicing* de elementos del string, y concatenar strings.

Sin embargo, recordar que los strings **no son mutables**.

Caracteres con secuencias de escape (/)

Al momento de escribir un *string*, hay caracteres que son precedidos por un **/ (*Backslash*).** Estos caracteres al momento de imprimirse presentan comportamientos especiales:

```
mi_string = "hola!\n"          # Incluimos un salto de línea al final

otro_string = "\"\'"           # Incluimos comillas sin cerrar el string

otro_mas = "hola \bmundo"      # "holamundo", borra el caracter anterior
```

Métodos útiles para *strings*

A veces necesitamos verificar alguna condición sobre nuestro *string* o modificarle algo*:*

Método	Descripción
`.isalpha() / .isdigit()`	Revisa si todos los caracteres son letras de algún alfabeto/dígitos, respectivamente.
`.isUpper()`	Revisa que todos los caracteres estén en mayúscula.
`.startswith() / .endswith()`	Revisa si el string empieza o comienza con un *substring* en específico.
`.find()`	Devuelve el índice donde comienza alguna secuencia a buscar.
`.replace()`	Reemplaza algún carácter o *substring* por otro.
`.split()`	Separa el string según un carácter.

Métodos útiles para *strings*

A veces necesitamos verificar alguna condición sobre nuestro *string* o modificarle algo*:*

Método	Descripción
`.isalpha()` / `.isdigit()`	Revisa si todos los caracteres son letras de algún alfabeto/dígitos.
`.replace()`	Reemplaza algún carácter o *substring* por otro.
`.split()`	Separa el string según un carácter.

Y muchos más…
https://docs.python.org/3/library/stdtypes.html#string-methods

Variables dentro de *strings* (*f-strings*)

A veces necesitamos que un *string* cambie dependiendo de ciertas variables en el código.

Aunque podríamos usar cosas como concatenación de *strings* para resolverlo, el usar *f-strings* nos permite lograr el mismo resultado pero con un código más legible.

Variables dentro de *strings* (*f-strings*)

```
# Las variables necesarias para crear nuestro string
from_email = "cruz@ing.puc.cl"
to_email = "alumnos@iic2233.com"
message = ("\nEste es un mail de prueba.\n"
           "\nEspero que el mensaje te sea de mucha utilidad!")
subject = "IIC2233 - Este correo es urgente"

# Nuestro f-string, notar como no tuvimos que hacer múltiples
# concatenaciones y tanto las variables como el string son legibles.
print(f"""
From: <{from_email}>
To: <{to_email}>
Subject: {subject}
{message}
""")
```

Mejoremos nuestros *strings*

Además, los *f-strings* nos permiten incluir algunas opciones para formatear nuestros textos de mejor forma. Por ejemplo, manipulandolos para alcanzar algún largo que queramos.

El formato a seguir es:

```
variable = ...

string_formateado = f"{variable:caracteralineamientotamañotipo}"
```

Mejoremos nuestros *strings*

```
f"{variable:caracteralineamientotamañotipo}"
```

```
variable = "Salchipapas"

# Alargamos el string hasta alcanzar el largo pedido (15), llenando con
# espacios, con alineamiendo a la izquierda.

mi_string = f"{variable: <15s}"         # 'Salchipapas    '

# El string debe tener exactamente 5 caracteres, si se debe llenar se llena con
# '#' y se alinea al medio en caso de necesitar relleno.

otro_string = f"{variable:#^5.5s}"      # 'Salch'
```

Mejoremos nuestros *strings*

```
f"{variable:caracteralineamientotamañotipo}"
```

```
vari

# Pa
# po
# ll

nume
```

¡Las posibilidades son infinitas!
https://docs.python.org/3/library/string.html#format-specification-mini-language

Expresiones Regulares (RegEx)

A veces necesitamos encontrar patrones en nuestros strings bastante específicos.

Para estos casos, podríamos tener funciones de varias líneas haciendo uso de métodos de string y procesando el string de forma progresiva…

…O usar una Expresión Regular, también llamadas RegEx 😎

Qué es una RegEx

Patrones de búsqueda, compuestos de secuencias de caracteres especializados, aplicables a *strings*. El objetivo es revisar si el patrón se encuentra una o más veces dentro del *string* a analizar.

Esto permite que la validación de un *string* para que siga un formato definido sea más concisa.

En **Python**, usaremos la librería **re** para trabajar con RegEx.

Caracteres básicos de una RegEx

Sintaxis	Descripción
`[ ]`	Clases de caracteres
`+`	Puede estar 1 o más veces
`*`	Puede estar 0 o más veces
`?`	Puede estar a lo más 1 vez
`{m, n}`	Puede estar entre m y n veces
`.`	Comodín (cualquier carácter)

Sintaxis	Descripción
`^`	Inicio del *string*
`$`	Final del *string*
`( )`	Agrupar
`|`	OR
`\`	Escapar caracteres especiales para usarlos

Caracteres básicos de una RegEx

Sintaxis	Descripción
`\s`	Cualquier espacio en blanco
`[a-z]`	Cualquier letra minúscula entre a y z
`[a-zA-Z]`	Cualquier letra entre A y Z, sin importar mayúscula o minúscula
`[0-9]`	Cualquier dígito entre 0 y 9
`[^arn]`	Cualquiera EXCEPTO a, r, n

Y muchos más…

Algunos ejemplos

Validar un rut:

[0-9]{1,2}\.[0-9]{3}\.[0-9]{3}-([0-9kK])

- Un número de 1 o 2 dígitos, que pueden ir entre 0 y 9 cada uno.
- Seguido de un punto ‘.’
- Numero de 3 digitos, que pueden ir entre 0 y 9.
- Seguido de otro punto
- Numero de 3 digitos, que pueden ir entre 0 y 9.
- Seguido de un guión ‘-’
- Un digito entre 0 y 9, o una K (mayúscula o minúscula)

Algunos ejemplos

Validar un correo con formato específico:

[a-zA-Z0-9_.]+@((seccion1|seccion2)\.)?(mimail|mail)\.cl

- Tiene que haber una letra, digito, guion bajo o punto por lo menos una vez.
- Luego un arroba
- Luego puede haber o no haber un “seccion1.” o “seccion2.” (no ambos).
- Luego debe haber un “mimail” o un “mail” (no ambos)
- Finalizar con .cl

Algunos ejemplos

Validar un correo con formato específico:

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(
?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0
b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[
a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\
.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?
:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\
x0e-\x7f])+)\])
```

Fuente: https://stackoverflow.com/questions/201323/how-can-i-validate-an-email-address-using-a-regular-expression

Algunos

Validar un corre

```
(?:[a-z0-9!#
?:[\x01-\x0
b\x0c\x0e-\
a-z0-9-]*[a
.){3}(?:(2(5
:[\x01-\x08
x0e-\x7f])+
```

```
|}~-]+)*|"(
1-\x09\x0
[a-z0-9](?:[
-9]?[0-9]))\
[a-z0-9]:(?
9\x0b\x0c\
```

Fuente: https://stackov a-regular-expression

RegEx no es trivial

Por eso hay muchas páginas que ayudan a entender qué diablos significa una expresión o cómo diablos escribo una:

- https://regexr.com/
- https://regex101.com/
- https://www.regextester.com/

Usar RegEx en Python

El módulo **re** nos permite aplicar RegEx para analizar *strings* en nuestro código:

```
re.match(patron, string) # Encontrar la primera ocurrencia desde el inicio

re.fullmatch(patron, string) # Todo el string cumple el patrón

re.search(patron, string) # Encontrar en cualquier lado el patrón

re.sub(patron, reemplazar_por, string) # Reemplazar un patrón

re.split(patron,string) # Separar según el patrón
```

Programación Avanzada
IIC2233 2023-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattan