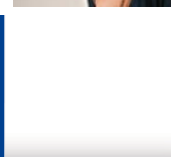
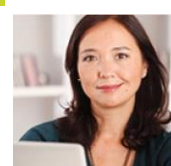
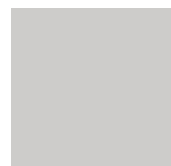


MÓDULO

3

Área: **NEGOCIOS**
Curso: **PROGRAMACIÓN BÁSICA (PYTHON)**
Módulo: **Revisión de programas computacionales**



IPP

SUEÑA • APRENDE • CRECE



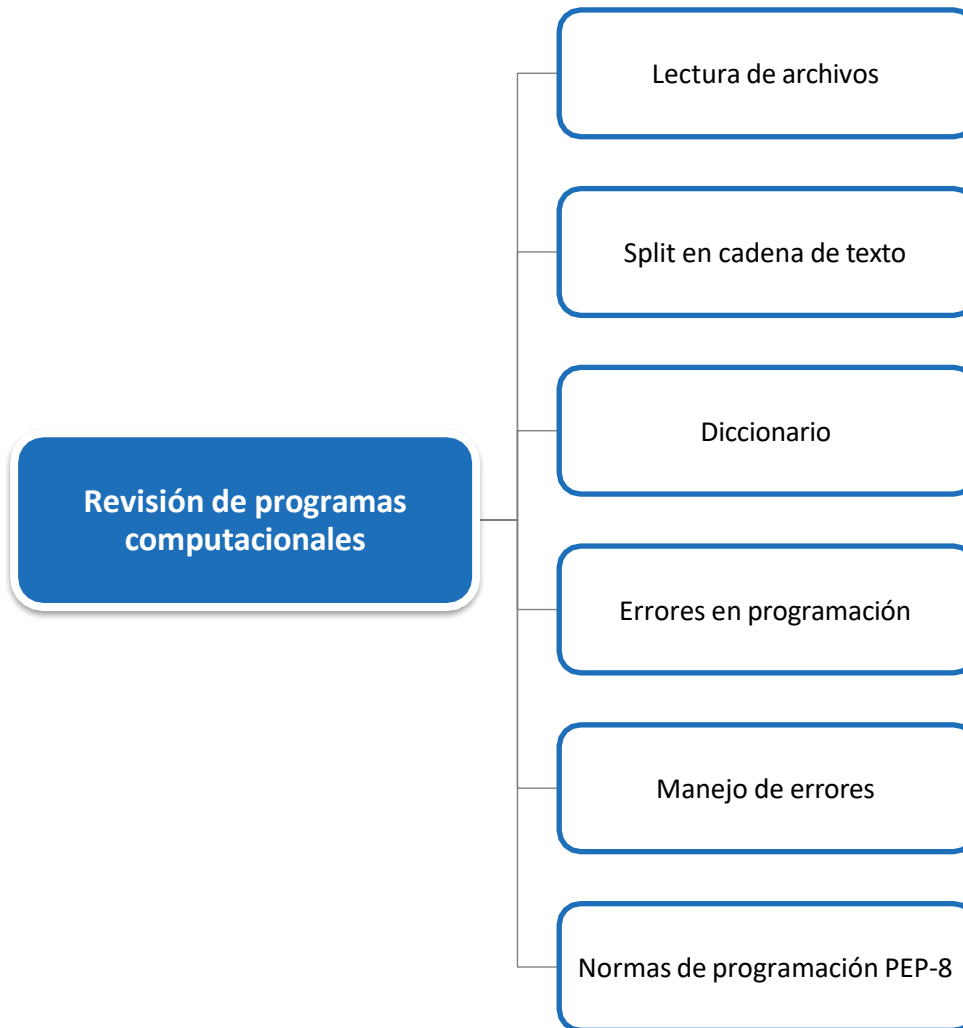
Índice

Contenido

Introducción	1
1. Lectura de archivos	2
1.1 Método: readline().....	7
2. Split en cadena de texto.....	9
3. Diccionario	10
3.1 Ejercicio de aplicación.....	11
4. Errores más comunes en la programación.....	15
4.1 El uso de variables inexistentes en el código	15
4.2 Tipografía.....	15
4.3 El uso de mayúsculas y minúsculas	15
4.4 Tabulación o espacios	15
4.5 Dos puntos (:) al final de cada ciclo o validación	16
4.6 Variables que empiecen con _, -, numero o carácter.....	17
4.7 Guardar archivo sin extensión .py.....	17
4.8 Ocupar tipo de dato equivocado	17
5. Manejo de errores	18
6. Normas de programación PEP-8	19
6.1 ¿Qué es PEP 8?.....	19
6.2 Propuestas de Mejora de PEP 8	19
a. Indentación	19
b. Máximo de 79 caracteres por línea	19
c. Argumentos de funciones ordenados.....	19
d. Validación de un Booleano	21
e. Espacios.....	21
f. Comentarios	22
g. Importación de librerías	22
7. Cierre.....	23



Mapa de Contenido



**RESULTADO DE
APRENDIZAJE
DEL MÓDULO**

Verifica la correcta aplicación de pasos lógicos en la construcción de un programa computacional, para determinar posibles errores de programación.

Introducción

Una situación común en programación es la búsqueda de errores que pueden causar fallas graves en la ejecución del programa desarrollado. A veces, un error tan simple como la falta de un punto y coma (;) puede ser la causa. En Python no se utilizan puntos y coma, pero es importante prestar atención a la indentación. Si un programador o su IDE usan cuatro espacios en lugar de un tab, y otro programador utiliza solo tab, puede haber problemas al momento de juntar el código, lo que puede provocar que el programa no funcione correctamente.

En el pasado, había casos en los que los programadores poco éticos creaban códigos que solo ellos podían entender. Esto creaba problemas para otros programadores que intentaban trabajar con ese código, lo que provocaba que los programas se estancaran y que su actualización dependiera de un solo programador. Para evitar esto, se inició la estandarización de la programación, también conocida como "buenas prácticas", tal como se describe en el PEP 8.

En el ámbito de la informática, se dice que

"un programa que no se actualiza es un programa que no se utiliza".

Esto es una realidad, ya que las necesidades de los clientes cambian constantemente. Es importante actualizar los programas para garantizar la seguridad, agregar nuevas funciones, integrarse con nuevos dispositivos o corregir errores.

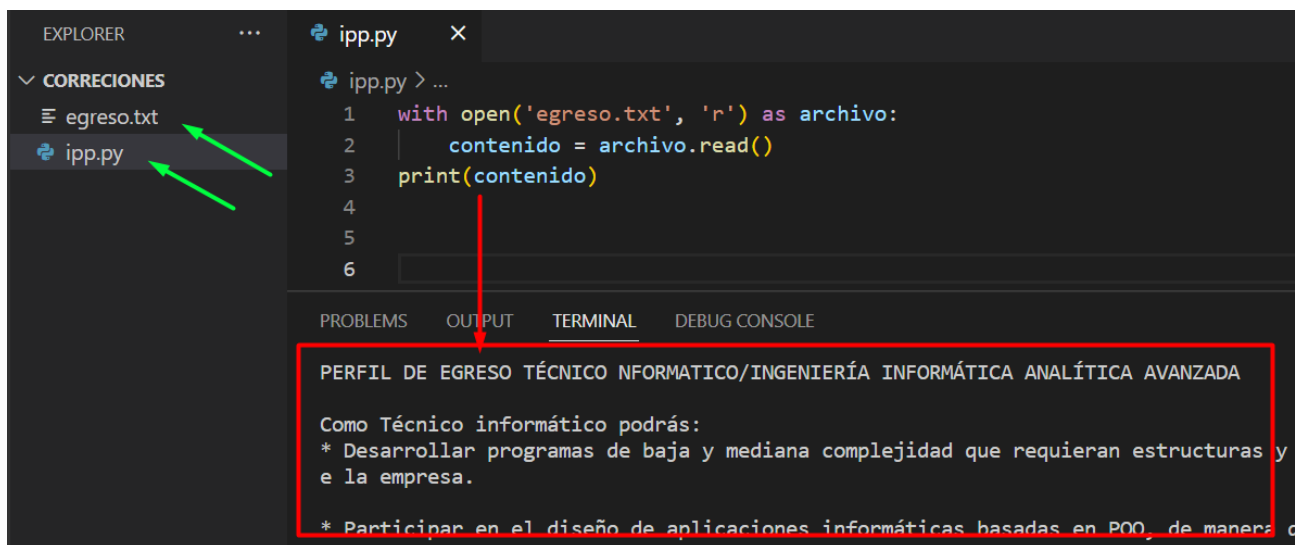
1. Lectura de archivos

En Python 3 es muy sencillo leer el contenido de un archivo, gracias a la función `open` integrada que funciona muy bien. ¿Cómo funciona?

Para leer un archivo en Python 3, sigue estos pasos:

- Crea un archivo con información en formato de texto (**.txt**) y lo guardas en la misma carpeta que tienes guardado tu archivo **".py"**.
- Abre un archivo en Python usando la función `open` con el nombre del archivo y el modo de apertura.
- El contenido del archivo se almacenará en la variable **"contenido"** y lo muestras con `print()`.

Por ejemplo, para abrir un archivo llamado "egreso.txt" en modo de solo lectura, se utilizaría la siguiente línea de código:



The screenshot shows a Python IDE with a file explorer on the left containing 'egreso.txt' and 'ipp.py'. The main editor shows a Python script in 'ipp.py' that opens 'egreso.txt' in read mode, reads its content into a variable named 'contenido', and prints it. The terminal at the bottom displays the output of the script, which is the text from 'egreso.txt'. A red box highlights the terminal output, and a red arrow points from the `print(contenido)` line in the script to the terminal.

```
1 with open('egreso.txt', 'r') as archivo:
2     contenido = archivo.read()
3     print(contenido)
4
5
6
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

PERFIL DE EGRESO TÉCNICO NFORMATICO/INGENIERÍA INFORMÁTICA ANALÍTICA AVANZADA

Como Técnico informático podrás:

- * Desarrollar programas de baja y mediana complejidad que requieran estructuras y e la empresa.
- * Participar en el diseño de aplicaciones informáticas basadas en POO, de manera c

- La línea `with open('egreso.txt', 'r') as archivo:` abre el archivo "egreso.txt" en modo de solo lectura.
- La palabra clave **with** se utiliza para que el archivo se cierre automáticamente una vez que se ha terminado de utilizar.
- La letra "r" en el segundo parámetro de `open` significa "read" (lectura), lo que indica que solo se permitirá leer el contenido del archivo.
- **as archivo** crea un objeto "archivo" que se utiliza para leer el contenido.
- La línea con la variable `contenido = archivo.read()` lee el contenido del archivo y lo almacena en la variable **contenido**.

También existen diferentes modos para leer un archivo, estos son:

modo	Significado	Definición y modo operación
r	Read – leer	Se ocupa solo para leer la información de un archivo
rb	Read bit – leer en binario	Se ocupa solo para leer la información de un archivo en binario
r+	Read plus – leer y escribir	Se ocupa para leer y después escribir en el archivo
rb+	Read bit plus – leer y escribir en binario	Se ocupa para leer y después escribir en el archivo en binario
w	Write – escribir	Escribir en el archivo, si este existe lo reemplaza completamente, si no existe lo crea
wb	Write bit – escribir en binario	Escribir en el archivo en binario, si este existe lo reemplaza completamente, si no existe lo crea
w+	Write plus – escribir y leer	Escribir y leer en el archivo, si este existe lo reemplaza completamente, si no existe lo crea
wb+	Write plus bit– escribir y leer en binario	Escribir y leer en el archivo en binario, si este existe lo reemplaza completamente, si no existe lo crea
a	Add – Añadir	Añade la información después de la última línea del archivo si este no existe lo crea
ab	Add bit- Añadir en binario	Añade la información en binario después de la última línea del archivo si este no existe lo crea
a+	Add plus - Añadir y lectura	Añade y lee la información en después de la última línea del archivo si este no existe lo crea
ab+	Add plus bit - Añadir y lectura binaria	Añade y lee la información en binario después de la última línea del archivo si este no existe lo crea

En Python, **si intentamos abrir un archivo en modo de lectura (Read) y este no existe, se generará un error.** Esto se debe a que el modo de lectura solo busca un archivo existente para leer su contenido. Por lo tanto, es importante verificar la existencia del archivo antes de intentar abrirlo en modo de lectura para evitar errores

Formas de recorrer un archivo en Python

La manera más básica de recorrer un archivo es mediante un bucle for. Sin embargo, esta opción no es muy recomendada, ya que puede generar problemas al procesar archivos grandes.

- Al recorrer cada punto del archivo con "for i in archivo", se corre el riesgo de consumir mucha memoria y tiempo, lo que puede afectar el rendimiento del programa.
- Por lo tanto, es recomendable utilizar otras formas de recorrer archivos en Python, como el uso de métodos como read(), readline() o readlines(), que permiten leer el archivo de manera más eficiente y controlada.

Supongamos que tenemos dos archivos llamados "archivo.txt" con el siguiente contenido:

línea 1	línea1
línea 2	línea2
línea 3	línea3

En el segundo archivo hay una línea vacía entre la línea 2 y la línea 3, a diferencia del primer archivo que no existen líneas vacías

Para recorrer estos archivo con un bucle for en Python, lo haremos por separado para que se pueda entender:

Tenemos el código del primer archivo.txt

The screenshot shows a Python IDE with a file named 'ipp.py'. The code in the editor is as follows:

```

1  with open("archivo.txt", "r") as archivo:
2      for linea in archivo:
3          print(linea)
4
5
6
7
8
    
```

A green arrow points from the `print(linea)` statement to the 'Print Output' panel on the right. The output panel shows the following text:

```

Print Output:
línea 1
línea 2
línea 3
    
```

Red boxes highlight the output for 'línea 1' and 'línea 2', and red arrows point from the corresponding lines in the code to these boxes. The output for 'línea 3' is not boxed.

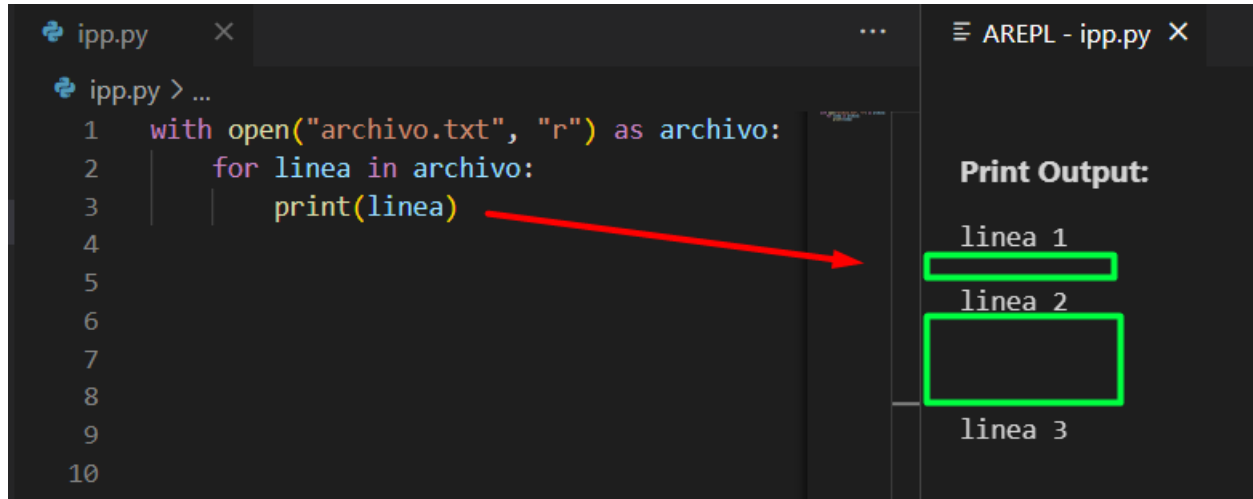
En este ejemplo, utilizamos la función "open" para abrir el archivo en modo lectura ("r") y luego lo asignamos a una variable llamada "archivo". Luego, utilizamos un bucle for para recorrer cada línea del archivo, que se almacena temporalmente en una variable llamada "linea". Finalmente, imprimimos cada línea del archivo en la consola utilizando la función "print"

La salida del programa muestra cada línea del archivo con una línea en blanco adicional entre cada una de ellas. Esto se debe a que cada línea del archivo contiene un carácter de salto de línea ("`\n`") al final.

Cuando leemos las líneas del archivo usando el bucle for, estos caracteres de salto de línea también se incluyen en la variable "linea". Por lo tanto, al imprimir cada línea en la consola utilizando la función "print", se agrega una línea en blanco adicional después de cada línea del archivo debido a los caracteres de salto de línea.

Sé que suena un poco complejo entenderlo, pero te invito a escribir el código y ver la salida que entrega, entenderás de mejor manera el efecto en la salida practicando en tu editor de código.

Ahora utilicemos el siguiente archivo, que contiene una línea en blanco entre la línea 2 y 3



```
ipp.py > ...
1  with open("archivo.txt", "r") as archivo:
2      for linea in archivo:
3          print(linea)
4
5
6
7
8
9
10
```

Print Output:

linea 1

linea 2

linea 3

Como puedes ver al tener un espacio Python agrega espacios vacíos adicionales , esto ocurre porque el ciclo for en este ejemplo se lee de esta manera:

"por cada elemento en el archivo, imprime el elemento y si encuentras espacios vacíos, imprime un espacio vacío"

El bucle for recorre el archivo línea por línea, y para cada línea del archivo, imprime la línea en la consola. Si el archivo contiene líneas vacías, el for también las recorrerá y, por lo tanto, se imprimirán líneas vacías en la consola.

¿Por qué tantos espacios?

En algunos casos, el final de línea en un archivo puede contener un salto de línea programado que se suma al salto de línea que se agrega automáticamente en la función `print()`. Por lo tanto, para evitar una doble línea en blanco, se recomienda borrar el salto de línea adicional en el archivo. Este salto de línea puede ser problemático en la programación, ya que puede causar errores y dificultades adicionales.

Para eliminar el salto de línea adicional en un archivo, una opción recomendada es utilizar el método **`replace()`**. Simplemente se puede cambiar el salto de línea por una cadena vacía en la variable, de esta manera la información se mostrará sin el salto de línea adicional. De esta forma, se puede mejorar la legibilidad y evitar errores en la programación.

Entonces el código quedaría de la siguiente manera:

```

1  # Abre el archivo en modo lectura ("r") y lo asocia a la variable "archivo".
2  # El bloque "with" se encarga de cerrar automáticamente el archivo al finalizar.
3  with open("archivo.txt", "r") as archivo:
4
5      # Recorre el archivo línea por línea
6      for linea in archivo:
7
8          # Reemplaza el carácter de salto de línea por una cadena vacía utilizando el método "replace()".
9          linea_sin_salto = linea.replace("\n", "") # Esto elimina el salto de línea al final de cada línea.
10         print(linea_sin_salto)                  # Imprime la línea sin el carácter de salto de línea.
11
12
    
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```

User@DESKTOP-DVUM85S MINGW64 ~/Desktop/correcciones
$ C:\Users\User\AppData\Local\Programs\Python\Python39\python.exe c:/Users/User/Desktop/correcciones/ipp.py
linea 1
linea 2
linea 3
    
```

Con el tiempo dejaremos de usar de manera excesiva los comentarios, por ahora se utilizarán para que vayan entendiendo de mejor manera el código

La manipulación de archivos es una tarea común en el desarrollo de software y, para trabajar con ellos, es necesario saber cómo leer su contenido. En Python, existen distintos métodos para realizar esta tarea, lo que permite a los programadores elegir el que mejor se adapte a sus necesidades. En este artículo, nos enfocaremos en presentar las diferentes formas de recorrer un archivo en Python, para que puedas elegir la opción más adecuada en tus proyectos.

1.1 Método: readline()

El método **readline()** permite leer una línea completa de un archivo. En cada llamada a este método, se lee una sola línea y se devuelve como una cadena. Si se llama a **readline()** nuevamente, se leerá la siguiente línea y así sucesivamente hasta que se llegue al final del archivo.

Aquí te dejo un ejemplo de cómo se usa **readline()**:

```

ipp.py  X
ipp.py > ...
1  with open("egreso.txt", "r") as archivo:
2      linea = archivo.readline() # lee la primera línea
3      while linea: # mientras haya una línea que leer
4          print(linea) # imprime la línea
5          linea = archivo.readline() # lee la siguiente línea
6
7
    
```

En este ejemplo, se abre el archivo "egreso.txt" en modo de lectura y se lee la primera línea del archivo con **readline()**. Luego, se entra en un ciclo **while** que sigue leyendo líneas con **readline()** hasta que se llegue al final del archivo, es decir, hasta que se obtenga una cadena vacía. En cada iteración del ciclo, se imprime la línea leída con **print()**.

```

AREPL - ipp.py  X  ...

Print Output:

PERFIL DE EGRESO TÉCNICO INFORMATICO/INGENIERÍA INFORMÁTICA
CIBERSEGURIDAD

Como Técnico informático podrás:

* Desarrollar programas de baja y mediana complejidad que requieran
estructuras y bases de datos de acuerdo con los requerimientos de la
empresa.

* Participar en el diseño de aplicaciones informáticas basadas en P00,
de manera de cumplir con las exigencias técnicas de los usuarios y de
la organización.

* Desarrollar aplicaciones web acordes a los requerimientos de la
organización y/o usuarios utilizando lenguajes como JavaScript, HTML y
CSS.
    
```

También se puede utilizar el método **readlines()** para leer todo el archivo y almacenarlo en una lista, en la que cada elemento corresponde a una línea del archivo. A continuación, se muestra el código anterior modificado para utilizar **readlines()**:

```
ipp.py
ipp.py > ...
1  with open("egreso.txt", "r") as archivo:
2      lineas = archivo.readlines()
3      for linea in lineas:
4          linea_sin_salto = linea.replace("\n", "")
5          print(linea_sin_salto)
6
7
```

En este caso, el método **readlines()** se utiliza para leer todas las líneas del archivo y almacenarlas en la variable **lineas**, que es una lista. Luego, se recorre la lista utilizando un ciclo **for** y se aplica la misma lógica que en el ejemplo anterior para eliminar el salto de línea y mostrar cada línea del archivo.

En cuanto a si es mejor utilizar un ciclo **while** o **readlines()**, depende del caso particular y de las necesidades del programa. En general, **readlines()** es útil cuando se necesita acceder a todas las líneas del archivo de una sola vez, por ejemplo, para procesar o filtrar los datos. En cambio, un ciclo **while** puede ser más útil si se requiere un control más fino del proceso de lectura del archivo, por ejemplo, si se necesita detener la lectura en función de cierta condición

En la lectura de archivos, es importante tener en cuenta el punto de lectura. El puntero es como una flecha que se mueve a través del documento y nos indica la posición actual. Saber la posición del puntero es fundamental para determinar qué línea mostrar o dónde escribir nueva información. Si no se utiliza de manera adecuada, y estamos trabajando en modo **"r+"** (lectura y escritura), podríamos sobrescribir información sin darnos cuenta

Por suerte, existen comandos que nos permiten saber la posición actual del cursor y también moverlo a otra posición. Algunos de estos comandos importantes son: **tell()**, que nos indica la posición actual del cursor, y **seek()**, que nos permite mover el cursor a una posición determinada.

seek(byte):	read(long):	tell():	write(string):
<ul style="list-style-type: none">• Mueve el cursor a la dirección de byte indicada.	<ul style="list-style-type: none">• lee todo el archivo. Si se le entrega un parámetro 'long', el programa leerá solo hasta el número de bytes que se especifique en 'long'.	<ul style="list-style-type: none">• Retorna la posición actual del cursor.	<ul style="list-style-type: none">• Escribe en el archivo en la posición actual del cursor, en este caso. Se le entrega por parámetro un string.

Ejercicio:

Realicemos un código para escribir al final en un archivo sin borrar su información anterior, sin utilizar el modo `r+` ni funciones de agregado

Para escribir en un archivo sin borrar su información anterior al final, podemos abrir el archivo en modo `"a"` (append) en lugar de en modo `"r+"` (read and write), que nos permitirá escribir al final del archivo sin afectar su contenido anterior

```

1 # Abrimos el archivo en modo "a" (append)
2 with open("mi_archivo.txt", "a") as archivo:
3     # Pedimos al usuario que ingrese la información que desea agregar
4     nueva_informacion = input("Ingrese la información que desea agregar: ")
5     # Escribimos la información al final del archivo
6     archivo.write(nueva_informacion + "\n")
7
8
User@DESKTOP-DVUM55 MINGW64 ~/Desktop/correcciones
$ C:/Users/User/AppData/Local/Programs/Python/Python39/python.exe c:/Users/User/D
Ingrese la información que desea agregar: "Esta es una línea agregada"

```

mi_archivo: Bloc de notas

Archivo Edición Formato Ver Ayuda

"Esta es una línea agregada"

En este ejemplo, el archivo `"mi_archivo.txt"` se abre en modo `"a"`, lo que nos permite agregar información al final del archivo sin borrar su contenido anterior. Luego, se le pide al usuario que ingrese la información que desea agregar y se escribe al final del archivo utilizando el método `write()`. El `"\n"` al final de la cadena es para agregar una nueva línea después de la información que se está agregando.

2. Split en cadena de texto

En programación, **'split'** es una función que se utiliza para separar un texto en un arreglo, utilizando un carácter o patrón específico como delimitador. Por ejemplo, se puede usar para separar las líneas de un texto en un arreglo, o para separar los términos separados por comas en una cadena de texto y guardarlos en un arreglo.

Es una función común en muchos lenguajes de programación, como Python, Java, C++, entre otros, y puede ser muy útil en el procesamiento de texto y la manipulación de datos.

```

1 texto = "huevo, leche, pan, mantequilla"
2
3 # separa la cadena por comas y guarda los términos en un arreglo
4
5 arreglo = texto.split(",")
6 print(arreglo)
7
8

```

AREPL - ipp.py

Print Output:

['huevo', 'leche', 'pan', 'mantequilla']

Variables:

-{ arreglo = ['huevo', 'leche', 'pan', 'mantequilla']

También existen otros comandos con los diccionarios, los cuales son:

Keys() :	Values() :	items():
• Este retorna todas las llaves del diccionario.	• Este retorna todos los valores del diccionario.	• Retorna una tupla de cada valor que existe en el diccionario.

3.1 Ejercicio de aplicación

Realicemos un programa con todo lo aprendido. Creemos una base de datos en un archivo de texto y después leamos este archivo y obtengamos información de él de manera útil.

Somos un Instituto, en donde guardamos toda la información de tres alumnos en un archivo de texto.

Este archivo contendrá la siguiente información para cada alumno: rut, nombre, apellido, fecha de nacimiento, carrera y materias (las materias estarán entre corchetes [] y se separan con dos puntos : cada una).

El programa debe realizar las siguientes acciones:

1. Guardar la información de cada persona en un diccionario con llave que sea su rut.
2. Mostrar las materias y el número de veces que cada una se repite.
3. Mostrar las carreras y el número de alumnos en cada una.

PASOS

- Creemos una base de datos en un archivo de texto **alumnos.txt**

```
ipp.py x
ipp.py > ...
1  # Creamos la base de datos de alumnos en un archivo de texto
2
3  with open("alumnos.txt", "w") as archivo:
4      archivo.write("12345678-9,Juan,Pérez,2000-05-12,Ingeniería Civil,[Cálculo I:Física I:Álgebra I]\n")
5      archivo.write("23456789-0,Maria,Gómez,2001-07-18,Ingeniería en Computación,[Programación I:Estructuras de Datos:Base de Datos]\n")
6      archivo.write("34567890-1,Diego,López,1999-12-28,Ingeniería Mecánica,[Mecánica I:Mecánica II:Termodinámica]\n")
7
8
```

Este bloque de código crea la base de datos de alumnos en un archivo de texto llamado alumnos.txt. Utiliza la función open para abrir el archivo en modo escritura ("w") y guarda la instancia de archivo en la variable archivo. Luego, se escribe la información de cada alumno en una línea del archivo utilizando la función write. Cada línea contiene el RUT, nombre, apellido, fecha de nacimiento, carrera y materias del alumno, separados por comas. Las materias están dentro de corchetes y separadas por dos puntos.

- Leemos la base de datos de alumnos desde el archivo de texto.

```
# Leemos la base de datos de alumnos desde el archivo de texto

with open("alumnos.txt", "r") as archivo:
    alumnos = {}
    for linea in archivo:
        rut, nombre, apellido, fecha_nacimiento, carrera, materias_str = linea.strip().split(",")
        materias = [materia.strip() for materia in materias_str[1:-1].split(":")]
        alumno = {"nombre": nombre, "apellido": apellido, "fecha_nacimiento": fecha_nacimiento, "carrera": carrera,
                  "materias": materias}
        alumnos[rut] = alumno
```

Este bloque de código lee la base de datos de alumnos desde el archivo de texto creado anteriormente. Utiliza la función open para abrir el archivo en modo lectura ("r") y guarda la instancia de archivo en la variable archivo. Luego, se crea un diccionario vacío alumnos que será utilizado para guardar la información de cada alumno. El ciclo for recorre cada línea del archivo utilizando la variable linea. Luego, se separa la información de la línea utilizando la función split y se guarda cada campo en una variable correspondiente. La información de las materias se guarda en una variable materias_str, se remueven los corchetes al inicio y al final de la cadena y se separan las materias utilizando split y los dos puntos como separador. Se crea un diccionario alumno con los datos del alumno y se agrega al diccionario alumnos con el RUT como llave.

1. Guardar la información de cada persona en un diccionario con llave que sea su rut.

```
# Guardamos la información de cada persona en un string con llave que sea su rut

info_alumnos = {}
for rut, alumno in alumnos.items():
    nombre = alumno["nombre"]
    apellido = alumno["apellido"]
    fecha_nacimiento = alumno["fecha_nacimiento"]
    carrera = alumno["carrera"]
    materias = alumno["materias"]
    info_alumnos[rut] = f"{nombre} {apellido}, nacido el {fecha_nacimiento}, carrera {carrera}, materias {', '.join(materias)}"
```

Este bloque de código crea un diccionario llamado info_alumnos con la información de cada alumno en un string. En cada iteración del bucle for, se extrae la información del diccionario alumno y se crea un string con formato que se guarda en el diccionario info_alumnos con el rut como llave.

2. Mostrar las materias y el número de veces que cada una se repite.

```
# Mostramos las materias y el número que esta se repite

materias = [materia for alumno in alumnos.values() for materia in alumno["materias"]]
conteo_materias = {}
for materia in materias:
    if materia in conteo_materias:
        conteo_materias[materia] += 1
    else:
        conteo_materias[materia] = 1
print("Materias:")
for materia, cantidad in conteo_materias.items():
    print(f"{materia}: {cantidad}")
```

Este bloque de código cuenta cuántas veces aparece cada materia en la base de datos de alumnos. Primero, se crea una lista de todas las materias que se encuentran en la base de datos, utilizando una lista por comprensión. Luego, se utiliza un diccionario llamado `conteo_materias` para contar cuántas veces aparece cada materia. Por último, se muestra en la consola la lista de materias y la cantidad de veces que aparece cada una.

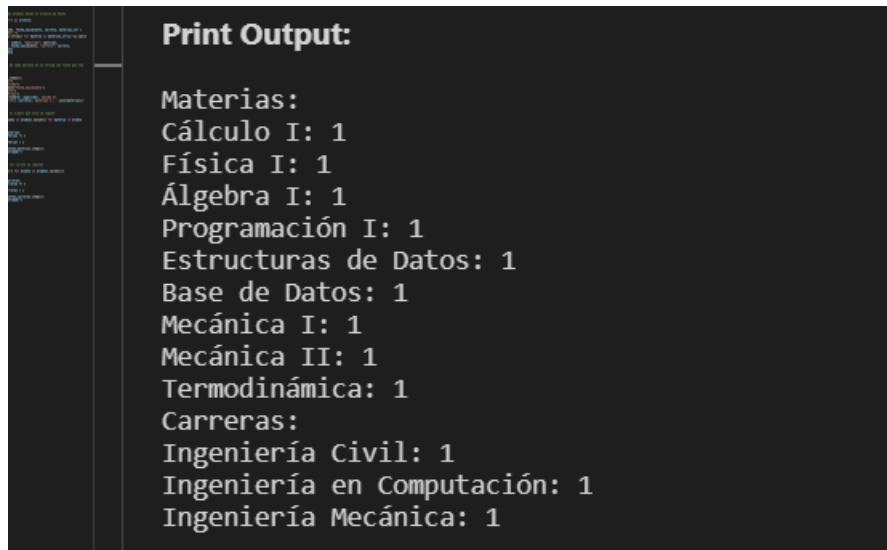
3. Mostrar las carreras y el número de alumnos en cada una.

```
# Mostramos las carreras y ver cuánto se repiten

carreras = [alumno["carrera"] for alumno in alumnos.values()]
conteo_carreras = {}
for carrera in carreras:
    if carrera in conteo_carreras:
        conteo_carreras[carrera] += 1
    else:
        conteo_carreras[carrera] = 1
print("Carreras:")
for carrera, cantidad in conteo_carreras.items():
    print(f"{carrera}: {cantidad}")
```

Este último bloque de código cuenta cuántas veces aparece cada carrera en la base de datos de alumnos. Primero, se crea una lista de todas las carreras que se encuentran en la base de datos, utilizando una lista por comprensión. Luego, se utiliza un diccionario llamado `conteo_carreras` para contar cuántas veces aparece cada carrera. Por último, se muestra en la consola la lista de carreras y la cantidad de veces que aparece cada una.

La salida del código anterior será:



The image shows a screenshot of a Python IDE. On the left, a portion of the code is visible, including a list of subjects and a loop that prints each subject followed by a value. On the right, the 'Print Output' window displays the result of the code execution.

```
Print Output:  
  
Materias:  
Cálculo I: 1  
Física I: 1  
Álgebra I: 1  
Programación I: 1  
Estructuras de Datos: 1  
Base de Datos: 1  
Mecánica I: 1  
Mecánica II: 1  
Termodinámica: 1  
Carreras:  
Ingeniería Civil: 1  
Ingeniería en Computación: 1  
Ingeniería Mecánica: 1
```

4. Errores más comunes en la programación

4.1 El uso de variables inexistentes en el código

En muchas ocasiones, al programar, podemos suponer que hemos declarado una variable que usamos regularmente en nuestro código. Sin embargo, al momento de ejecutarlo, nos encontramos con un error debido a que la variable en cuestión no ha sido declarada, como se muestra en el siguiente ejemplo:

4.2 Tipografía

Muchas veces ocurre que al momento escribir rápido se invierte el nombre de una variable o esta queda mal escrita.

4.3 El uso de mayúsculas y minúsculas

A veces utilizamos variables con diferentes combinaciones de mayúsculas y minúsculas. Sin embargo, es importante tener en cuenta que Python distingue entre las mayúsculas y las minúsculas, por lo que si la variable se escribe de manera diferente a como se definió, Python mostrará un error indicando que la variable no existe. Por lo tanto, es recomendable mantener una convención constante en la capitalización de nuestras variables para evitar errores

4.4 Tabulación o espacios

Es fundamental tener en cuenta que, en Python, la presencia o ausencia de espacios puede tener un impacto significativo en la ejecución del código. Por ejemplo, en una función, una validación o un ciclo, los espacios son importantes ya que Python los utiliza para determinar si una acción pertenece o no a un bloque de código. Es común que los programadores copien y peguen código de diferentes fuentes, y en ocasiones, este código puede contener espacios en lugar de tabulaciones. Si no se corrige esto, puede generar errores en la ejecución del código. Por lo tanto, es recomendable revisar cuidadosamente el código para asegurarse de que se esté utilizando la convención de espaciado correcta en cada bloque de código.

Ejemplo:

The screenshot shows a Python IDE with a file named 'ipp.py'. The code is as follows:

```
1 palabra = "hola"
2 for i in palabra:
3     print("la letra es: ") #esta
4     print(i) #esta con espacios
```

Line 4 has a red squiggly line under the variable 'i', indicating an error. A red box highlights the error message in the console:

```
line 4
print(i) #esta con espacios
IndentationError: unindent does not match any outer indentation level
```

Below the error message, the 'Print Output' section shows the result of the code execution:

```
la letra es:
la letra es:
la letra es:
la letra es:
a
```

Al utilizar una mezcla de espacios y tabuladores en el código Python, puede ocurrir un error de indentación, el cual se presenta cuando se mezclan diferentes estilos de indentación (espacios y tabuladores) dentro del mismo bloque de código

4.5 Dos puntos (:) al final de cada ciclo o validación

Cuando escribimos un ciclo while, for o un condicional if en Python, es importante incluir los dos puntos (:) al final de la línea donde se define la estructura. Los dos puntos indican que el bloque de código siguiente será ejecutado si se cumple la condición o se realiza una iteración del ciclo. Aunque es posible omitir los dos puntos, es una buena práctica incluirlos para que el código sea más claro y fácil de leer.

Es importante destacar que los dos puntos son ampliamente utilizados en Python, y se consideran una parte esencial de la sintaxis del lenguaje. Por lo tanto, se recomienda su uso en todas las estructuras de control de flujo.

```

ipp.py
1 # Ejemplo con dos puntos
2 x = 5
3 if x > 3:
4     print("x es mayor que 3")
5
AREPL - ipp.py
Print Output:
x es mayor que 3
  
```

The screenshot shows a Python IDE with a file named 'ipp.py'. The code contains a comment '# Ejemplo con dos puntos', a variable assignment 'x = 5', and an if statement 'if x > 3:' followed by an indented print statement 'print("x es mayor que 3")'. A green arrow points to the colon at the end of line 3. The output pane on the right shows 'Print Output:' followed by 'x es mayor que 3' in a pink box.

En este ejemplo, se utiliza el condicional if para evaluar si el valor de la variable x es mayor que 3. La línea que contiene la condición **termina con dos puntos**, y después de los dos puntos viene el bloque de código que se ejecutará si se cumple la condición. En este caso, el bloque de código contiene una instrucción print que imprime en la consola el mensaje "x es mayor que 3".

```

ipp.py 1
1 # Ejemplo sin dos puntos
2 x = 5
3 if x > 3
4     print("x es mayor que 3")
5
AREPL - ipp.py
line 3
if x > 3
^
SyntaxError: invalid syntax
  
```

The screenshot shows the same Python IDE with a file named 'ipp.py'. The code is similar to the previous example but the if statement on line 3 is 'if x > 3' without a colon. A green arrow points to the end of line 3. The output pane on the right shows an error: 'line 3', 'if x > 3', '^', and 'SyntaxError: invalid syntax' in a red box.

En este ejemplo, se utiliza la misma estructura if y la misma condición, pero no se incluyen los dos puntos al final de la línea que contiene la condición. Esto provoca un error de sintaxis en Python, ya que el intérprete espera encontrar los dos puntos para indicar que viene un bloque de código que se ejecutará si se cumple la condición. La salida del programa en este caso será un mensaje de error que indica que la sintaxis es inválida.

4.6 Variables que empiecen con `_`, `-`, número o carácter

En Python, no se pueden utilizar variables que comiencen con `_` (guión bajo), `-` (guión), números o caracteres especiales (`%$*#`, etc.).

```
_abc = 5           # inválido  
-xyz = 10          # inválido  
2variable = "hola" # inválido  
%valor = 20        # inválido
```

4.7 Guardar archivo sin extensión `.py`

Aunque en Linux o OSX no existe diferencia entre un documento sin extensión o con extensión `.py`, en Windows si el archivo no tiene la extensión nos entregará error de documento incompatible

4.8 Ocupar tipo de dato equivocado

En Python, si utilizas un tipo de dato equivocado para una operación o una función, puede ocurrir uno de los siguientes comportamientos:

- **Error de tipo (TypeError):** Python arrojará un error de tipo (TypeError) indicando que no se puede realizar la operación con el tipo de dato proporcionado. Por ejemplo, si intentas sumar un número con una cadena de texto, Python mostrará un error de tipo indicando que no se pueden sumar tipos diferentes.
- **Resultado inesperado:** Si usas un tipo de dato incorrecto, es posible que la operación se realice, pero el resultado puede ser inesperado o incorrecto. Por ejemplo, si intentas dividir dos números enteros utilizando el operador de división entera `//` en lugar del operador de división normal `/`, Python realizará la operación, pero el resultado será un número entero truncado y no una fracción.

5. Manejo de errores

Existe una forma sencilla de manejar errores en Python mediante la estructura try-except. Esta estructura funciona de manera similar a un if-else. Si todo funciona correctamente, el código se ejecuta dentro del bloque try. Si ocurre un error durante la ejecución del bloque try, el control del programa se transfiere a un bloque except, donde se puede manejar el error.

Es importante tener en cuenta que el bloque except se ejecutará solo si se produce un error dentro del bloque try. En caso contrario, el programa continuará su ejecución normalmente después del bloque try.

El manejo de excepciones mediante la estructura try-except es una técnica útil para detectar y gestionar errores en el código. Al manejar los errores de esta manera, se puede hacer que el programa sea más robusto y menos propenso a fallos inesperados.

Ejemplo:

Hagamos un ciclo infinito que reciba constantemente un número. Dentro de un bloque try, convertiremos el número ingresado a entero. Si el usuario ingresa una palabra o letra, no podremos convertirlo a un entero y se generará un error. En ese momento, el programa saltará al bloque except para manejar el error.

```
ipp.py
ipp.py > ...
1 while True:
2     try: # Indicamos que el siguiente bloque de código puede generar una excepción.
3         # Solicitamos al usuario que ingrese un número y lo asignamos a la variable 'num'.
4         num = int(input("Ingresa un número: "))
5         # Mostramos el número ingresado por el usuario.
6         print("El número ingresado es:", num)
7     except ValueError: # Si se genera una excepción de tipo ValueError, se ejecutará este bloque de código.
8         # Mostramos un mensaje de error indicando que el usuario debe ingresar un número entero.
9         print("Error: Debes ingresar un número entero.")
10
11
```

6. Normas de programación PEP-8

En el mundo de la programación, es poco común que un código complejo sea desarrollado por una sola persona; en cambio, usualmente es un esfuerzo colaborativo que involucra a varias personas. Si cada una de ellas programara de la manera que quisiera, esto podría generar dificultades al momento de revisar o compartir el código con otros, especialmente si el programador original no está disponible para aclarar sus decisiones de programación.

En el pasado, el mundo de la programación era menos estructurado y cada programador tenía su propio estilo, lo que a menudo resultaba en códigos poco organizados y difíciles de entender para otros desarrolladores. Con el tiempo, se han establecido buenas prácticas de programación que determinan la manera adecuada de declarar una variable, escribir una función, e incluso incluir comentarios en el código. En Python, se ha creado un estándar de programación conocido como PEP-8, que establece convenciones claras para la escritura de código, ayudando a que los programadores puedan colaborar de manera más efectiva en proyectos de programación.

6.1 ¿Qué es PEP 8?

PEP, es el acrónimo de las palabras en inglés “Python Enhancement Proposals”, que en español es “**Propuestas de Mejora Python**”. Consiste en una pequeña guía realizada para mejorar las prácticas en la programación de Python, fueron creadas el 2001 por Guido van Rossum, Barry Warsaw y Nick Coghlan. Están disponibles en su propia página web: <https://www.python.org/dev/peps/pep-0008/>

La guía más conocida y utilizada es PEP-8, que establece las convenciones para el estilo de escritura de código en Python, incluyendo el formato de indentación, la longitud de las líneas de código, la forma en que se deben nombrar las variables y funciones, y la forma en que se deben incluir los comentarios.

6.2 Propuestas de Mejora de PEP 8

a. Indentación

Nunca mezclar tabulaciones con espacios y si se ocupa espacios que estos siempre sean 4. A excepción que sea un código antiguo que no queramos arruinar, ahí se pueden utilizar 8 espacios por tabulación.

b. Máximo de 79 caracteres por línea

Si estás haciendo un código y este es muy largo se puede agregar un \ al final para continuar en la siguiente línea, aunque también se recomienda encerrar la operación en paréntesis (), y en la siguiente línea cerrar este.

c. Argumentos de funciones ordenados

Al momento de declarar funciones excedes de los 79 caracteres, se deben colocar los argumentos abajo de los suyos o de manera alineada.

Ejemplo:

```
Correcto:
#opcion1
variable = funcionx(variable_1, variable2
                    variable_3)

#opcion2
variable = funcionx(
    variable_1, variable2
    variable_3)

Incorrecto:
variable = funcionx(variable_1, variable2 variable_3)
```

Este fragmento de código es un ejemplo de cómo utilizar correctamente la indentación y los saltos de línea al llamar a una función en Python, según las convenciones establecidas en el PEP 8.

En el código "Correcto", se presentan dos opciones para llamar a la función `funcionx()` con tres argumentos: `variable_1`, `variable2`, y `variable_3`. Ambas opciones son correctas según el PEP 8.

- La "opcion1" muestra la llamada a la función en una sola línea, pero se divide en dos líneas utilizando un salto de línea después del segundo argumento y antes del tercer argumento. Además, la segunda línea está indentada en la misma cantidad de espacios que la primera línea, para indicar que es una continuación de la llamada a la función.
- La "opcion2" muestra la llamada a la función dividida en tres líneas, con cada argumento en su propia línea, y la primera línea indentada en la misma cantidad de espacios que la segunda y la tercera línea.

Ambas opciones son correctas según el PEP 8, y la elección entre ellas depende de las preferencias personales del programador y de la legibilidad del código.

Por otro lado, el código "Incorrecto" muestra una llamada a la función en una sola línea, pero los argumentos no están separados por comas. En cambio, los argumentos están escritos uno tras otro, sin separadores entre ellos. Esta forma de llamar a la función es incorrecta y no sigue las convenciones de PEP 8, lo que dificulta la lectura del código.

d. Validación de un Booleano

El PEP 8 recomienda no validar un booleano con un operador de igualdad (==). En su lugar, se debe utilizar el propio booleano para evaluar una expresión booleana:

Por ejemplo, en lugar de escribir:

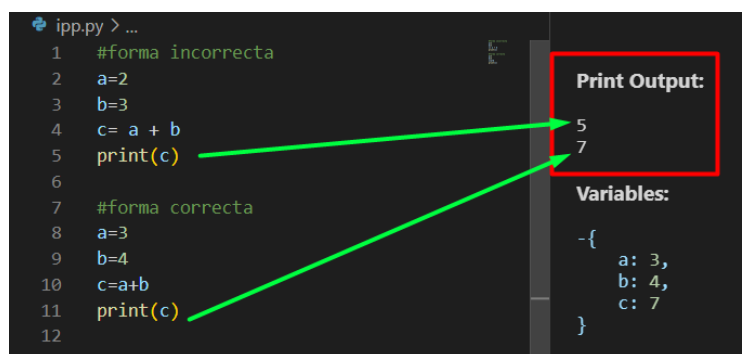
```
if resultado == True:  
    # hacer algo
```

Se debería escribir:

```
if resultado:  
    # hacer algo
```

e. Espacios

Nunca usar espacios innecesarios en el código. La razón detrás de esta recomendación es que los espacios innecesarios pueden dificultar la lectura del código y hacerlo menos legible. Además, puede agregar una carga innecesaria al tamaño del archivo del código fuente.



```
ipp.py > ...  
1  #forma incorrecta  
2  a=2  
3  b=3  
4  c= a + b  
5  print(c)  
6  
7  #forma correcta  
8  a=3  
9  b=4  
10 c=a+b  
11 print(c)  
12
```

Print Output:

```
5  
7
```

Variables:

```
-{  
  a: 3,  
  b: 4,  
  c: 7  
}
```

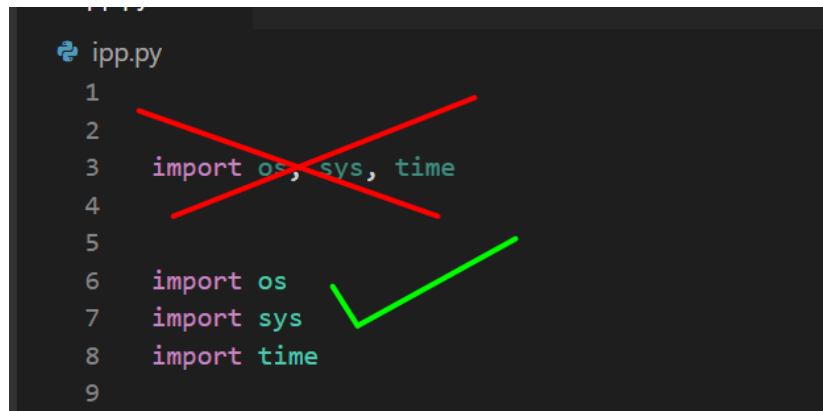
Seguir las recomendaciones de estilo de código, como las del PEP 8, hace que el código sea legible y fácil de entender para otros programadores, aunque el uso de espacios innecesarios en operaciones aritméticas no afecte la capacidad de compilación o ejecución del código en Python.

f. Comentarios

No se deben realizar comentarios obvios en el código. Los comentarios no deben contradecir el código.

g. Importación de librerías

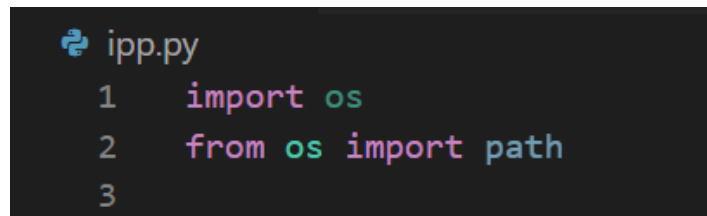
- Al momento de importar librerías, éstas no deben ir juntas si no separadas:



```
1  
2  
3 import os, sys, time  
4  
5  
6 import os  
7 import sys  
8 import time  
9
```

- Se permitirá el uso de importación consecutivo cuando vengan solo de la misma librería. En este caso, ambas importaciones pertenecen a la librería "os". Sin embargo, se debe tener cuidado de no importar módulos específicos de una librería en una sola línea, ya que puede hacer que el código sea menos legible. Es mejor separar cada importación en una línea separada.

Ejemplo:



```
1 import os  
2 from os import path  
3
```

7. Cierre

Los contenidos vistos en este módulo son fundamentales para la formación de un programador, ya que aprendieron a manejar archivos con Python, conocieron sus modos y comprendieron la importancia de cada uno. Con esta información, los estudiantes pueden comenzar a crear programas que involucren el manejo de archivos, como los que usan diccionarios y la función "split". Además, esta información les será de gran ayuda durante la programación, ya que estas son algunas de las herramientas más utilizadas en Python.

Posteriormente, se revisaron algunos de los errores más comunes que ocurren durante la programación, lo que les permitió comprender la conducta de los programadores. Finalmente, el conocimiento de la norma de estilo PEP-8 permitirá a los estudiantes mejorar su forma de programar, realizando programas más universales. Esto les será útil en su desarrollo profesional, ya que sus códigos podrán ser entendibles y perdurables en el tiempo, ya sea por ellos mismos o por otros programadores.