# En este módulo, abordaremos los fundamentos de la programación, explorando el funcionamiento de las computadoras, la ejecución de programas y la definición y construcción de lenguajes de programación.

# Diferenciación entre lenguajes de programación compilados e interpretados.

# Exploración de Python y las herramientas relevantes para el desarrollo.

# Comprensión de qué es un programa y cómo opera.

# Los programas confieren utilidad a una computadora; sin ellos, incluso la más potente sería simplemente un pisapapeles. De manera análoga, un barco carece de propósito sin un timón. Aunque las computadoras pueden realizar tareas complejas, su capacidad intrínseca se limita a operaciones simples. Por ejemplo, una computadora no puede comprender el valor de una función matemática complicada por sí misma.

# Las computadoras solo pueden evaluar resultados de operaciones básicas, como sumar, restar y multiplicar, pero lo hacen a una velocidad que supera ampliamente la capacidad humana y pueden repetir estas acciones de manera prácticamente ilimitada.

# Consideremos el escenario en el que deseamos calcular el promedio de la cantidad de lluvia a lo largo de un periodo. La computadora puede realizar este cálculo, pero carece de conciencia sobre conceptos como la cantidad de lluvia o el tiempo. Por lo tanto, es imperativo indicar a la computadora que:

# Solicite un número representando la cantidad de lluvia.

# Solicite un número representando la cantidad de días.

# Divida la cantidad de lluvia entre la cantidad de días y almacene el resultado en memoria.

# Muestre el resultado en un formato comprensible.

# Estas cuatro acciones simples constituyen un programa. Aunque estos ejemplos son incomprensibles para una computadora, son traducibles a un lenguaje comprensible por la máquina; la palabra clave aquí es "lenguaje".

# Lenguajes Naturales y Lenguajes de Programación

# Según la Real Academia Española (RAE), el lenguaje es la facultad humana de expresarse y comunicarse mediante sonidos articulados u otros sistemas de signos. En este contexto, se convierte en un medio para expresar y plasmar pensamientos. Existen diversos tipos de lenguajes, algunos prescinden del habla o la escritura, como el lenguaje corporal.

# La lengua materna, utilizada para pensar y comunicarse, es un ejemplo de lenguaje natural. Las computadoras también tienen su propio lenguaje, primitivo y conocido como lenguaje de máquina. A pesar de la sofisticación de las computadoras, carecen de inteligencia y solo responden a comandos predeterminados.

# Estos comandos son simples, como "Agarra este número, súmale este otro y guarda el resultado". Este conjunto de comandos se conoce como "Instruction List" (IL). Aunque las IL varían en tamaño entre diferentes computadoras, todas se componen de instrucciones básicas.

# Componentes de un Lenguaje

# Cada lenguaje, ya sea natural o de máquina, consta de elementos clave:

# Alfabeto: Conjunto de símbolos que forman palabras del lenguaje.

# Léxico: Conjunto de palabras que conforman el vocabulario del lenguaje.

# Sintaxis: Conjunto de reglas que determinan si una secuencia de palabras es válida gramaticalmente.

# Semántica: Conjunto de reglas que determinan si una frase tiene sentido.

# La IL actúa como el alfabeto en un lenguaje de máquina, siendo el conjunto más simple y fundamental de símbolos para dar instrucciones a la computadora. Dada la brecha entre la IL y la lengua materna humana, surge la necesidad de un lenguaje común comprensible tanto por humanos como por computadoras.

# Lenguajes de Programación de Alto Nivel

# Para superar esta brecha, se introducen los lenguajes de programación de alto nivel. Estos lenguajes, similares a los naturales, utilizan símbolos, palabras y convenciones legibles para las personas. Permiten instrucciones más complejas que las ofrecidas por las IL, siendo el "código fuente" el programa escrito en estos lenguajes.

# Compilación vs. Interpretación

# La programación implica establecer una secuencia de instrucciones para lograr un efecto deseado. Estas instrucciones deben cumplir con condiciones como:

# Correcta escritura alfabética.

# Conformidad con el diccionario léxico del lenguaje.

# Adherencia a las reglas sintácticas del lenguaje.

# Coherencia semántica para tener sentido.

# Un programa correctamente escrito debe convertirse en lenguaje de máquina para ejecutarse en la computadora. Esto se logra mediante dos enfoques:

# Compilación: Traducción única del código fuente, generando un archivo ejecutable (.exe, por ejemplo) distribuido a los usuarios finales.

# Interpretación: Traducción cada vez que se ejecuta el programa, requiriendo un intérprete para la ejecución.

# Python

# Python, un lenguaje de programación destacado, es obra de Guido van Rossum. Aunque lleva el nombre de una serpiente, se inspiró en la serie de comedia "Monty Python's Flying Circus". Su rápida difusión se debe al esfuerzo colaborativo de numerosos programadores, testers y entusiastas.

# Estos ajustes buscan ofrecer un enfoque más claro y formal para la presentación del contenido. Si tienes preguntas adicionales o necesitas más ajustes, no dudes en comentar.

# Herramientas de Desarrollo

# Para el desarrollo de programas en Python, se requieren varias herramientas esenciales:

# Editor de Texto: Se utiliza para escribir el código. Se recomienda un editor dedicado con características especiales que no están disponibles en herramientas simples. Este editor va más allá de las capacidades estándar del sistema operativo.

# Consola: Permite la ejecución del programa y, en caso necesario, detenerlo manualmente si se vuelve incontrolable.

# Debugger: Una herramienta que ejecuta el código paso a paso, permitiendo la inspección en cada etapa de la ejecución.

# Afortunadamente, existen herramientas que integran todas estas funciones en una aplicación única, conocidas como Entornos Integrados de Desarrollo y Aprendizaje (IDLE, por sus siglas en inglés).

# Python2 y Python3:

# Python2: Una versión anterior del Python original. Su desarrollo ha sido intencionalmente estancado, aunque aún recibe actualizaciones.

# Python3: La versión actual del lenguaje, que será utilizada en este curso.

# Descarga e Instalación de Python3:

# Usuarios de Linux: Python podría ya estar instalado, verifíquelo ingresando "python3" en la terminal.

# Usuarios de Windows, MacOS y Linux (si no está instalado): Descargar desde https://www.python.org/downloads/. Seleccionar la opción predeterminada, pero asegurarse de marcar la casilla "Agregar Python 3.x al PATH".

# Verificación de la instalación: Ejecutar "python3" en la terminal. Si se muestra el intérprete de Python, la instalación es exitosa.

# Nuestro Primer Programa en Python:

# Crear un nuevo archivo fuente haciendo clic en "File" en el menú IDLE y seleccionando "New File".

# Guardar el archivo sin establecer una extensión específica (confiar en los valores predeterminados para ".py").

# Escribir un programa simple para imprimir "¡Hola, Mundo!":

|  |
| --- |
| print("¡Hola, Mundo!") |

# Guardar y ejecutar el programa con F5 o desde el menú "Run".

# Este programa consta de partes esenciales:

# print: Función que envía texto a la consola u realiza otras acciones.

# Paréntesis: Encierran los argumentos de la función (en este caso, el texto a imprimir).

# Comillas: Encierran el texto que se imprimirá.

# Cada elemento desempeña un papel crucial en el código. "print" es una función, que en Python puede causar un efecto (como imprimir en la consola) o evaluar un valor y devolverlo como resultado.

# Depuración de Nuestro Primer Programa:

# Al colocar el paréntesis de cierre, IDLE resalta la sección del texto entre paréntesis, recordando ponerlos en pares y evitando errores.

# En caso de errores, IDLE informa sobre SyntaxError (error de sintaxis) y muestra dónde ocurrió en la ventana del editor.

# Debugging implica corregir errores en la sintaxis o lógica del código. Al ejecutar un código con errores, IDLE proporciona información sobre el tipo de error, su ubicación y sugerencias de ayuda.

# ¡Felicidades por ejecutar y depurar tu primer programa en Python!

# Actividades Módulo 1

### Actividad 1

El inglés es un ejemplo de:

* Lenguaje de programación .
* Lenguaje natural.
* Lenguaje de máquina.

### Actividad 2

Un lenguaje de alto nivel es:

* Un tipo de lenguaje de programación.
* Una lengua hablada por personas de la alta sociedad.
* Una lengua hablada en la antigüedad.

### Actividad 3

Un Compilador es:

* Un programa de computadora diseñado para traducir programas escritos en un lenguaje de alto nivel a lenguaje de máquina.
* Un programa de computadora diseñado para traducir programas escritos en lenguaje de máquina a uno de alto nivel.
* Un sinónimo de procesador.

|  |
| --- |
| python  Copy code  # Módulo 1: Configuración e Introducción a Python  # Herramientas de Desarrollo  # ...  # Descarga e instalación de Python3  # ...  # Nuestro Primer Programa en Python  print("¡Hola, Mundo!")  python  Copy code  # Módulo 2: Tipos de datos, operadores, variables y comentarios  # Tipos de datos, operadores, variables y comentarios  # ...  # La función PRINT()  # ...  # Modificación del código usado en el Módulo 1  print("¡Hola, Mundo!\n")  python  Copy code  # Uso de escape para nueva línea  print("¡Hola, Mundo!\nSeguimos aprendiendo Python.")  # Uso de la barra invertida como carácter de escape  print("¡Hola, Mundo!\\nSeguimos aprendiendo Python.")  python  Copy code  # Uso de secuencias de escape en Python3  print("¡Hola, Mundo!\nSeguimos\taprendiendo\tPython.")  python  Copy code  # Uso de múltiples argumentos en la función print()  print("¡Hola,", "Mundo!", "Seguimos", "aprendiendo", "Python.")  python  Copy code  # Uso de argumentos de palabra clave en la función print()  print("¡Hola, Mundo!", end="-")  print("Seguimos aprendiendo Python.", sep="-")  python  Copy code  # Números Enteros  # ...  # Uso de guiones bajos en literales numéricos  number\_with\_underscores = 11\_111\_111  print(number\_with\_underscores)  # Uso de números octales y hexadecimales  octal\_number = 0o123  hexadecimal\_number = 0x123  print(octal\_number, hexadecimal\_number)  python  Copy code  # Números Flotantes (Float)  # ...  # Uso de notación científica en Python3  speed\_of\_light = 3E8  print(speed\_of\_light)  python  Copy code  # Texto (String)  # ...  # Uso de escape para incluir comillas dentro de un string  message\_with\_escape = "Hoy sigo aprendiendo \"Python\""  print(message\_with\_escape)  # Uso de apóstrofes para delimitar un string  message\_with\_apostrophes = 'Hoy sigo aprendiendo "Python"'  print(message\_with\_apostrophes) |

### Contenido Módulo 2: Tipos de datos, operadores, variables y comentarios

### Tipos de datos, operadores, variables y comentarios

### En este módulo, nos sumergiremos en:

### La función print() y sus componentes.

### Tipos de datos, literales, operadores y expresiones en Python.

### Variables y las reglas que las rigen.

### Realización de operaciones básicas de entrada y salida.

### La función print()

### La función print() consiste en el nombre de la función, paréntesis, y los argumentos, formando la invocación de la función. Python sigue un proceso específico al encontrar una invocación:

### Comprueba si el nombre especificado es legal, cancelando la ejecución si no lo encuentra.

### Verifica si los requisitos de la función para el número de argumentos le permiten ser invocada de esa manera.

### Entra en la función, toma los argumentos y los pasa a la función.

### La función ejecuta el código, provoca el efecto deseado, evalúa resultados y termina la tarea.

### Python regresa al código y reanuda su ejecución desde la invocación.

### En el caso de print(), toma los argumentos y los convierte en un formato legible para el humano. La barra invertida \ se utiliza como carácter de escape para introducir secuencias especiales, como \n para nueva línea.

### Vamos a modificar nuestro código del Módulo 1:

### python

### Copy code

### print("¡Hola, Mundo!\n")

### La barra invertida seguida de 'n' representa un salto de línea. La función print() interpreta y muestra el texto con un formato adecuado.

### Números Enteros

### Python maneja dos tipos principales de números: enteros (int) y de punto flotante (float). Los enteros no tienen parte fraccionaria. Se pueden representar de diversas maneras, permitiendo guiones bajos para mejorar la legibilidad: 11\_111\_111. También se pueden expresar como octales (0o123) o hexadecimales (0x123).

### python

### Copy code

### print(123) # Entero

### print(-123) # Número negativo

### print(0o123) # Octal

### print(0x123) # Hexadecimal

### Números de Punto Flotante (Float)

### Los números de punto flotante (float) representan valores con parte decimal. Python acepta la notación científica con la letra 'e' o 'E'. Por ejemplo, 2.5 o 3E8 (equivalente a 3 \* 10^8).

### python

### Copy code

### print(2.5) # Número de punto flotante

### print(4.0) # Python acepta notación abreviada sin cero decimal

### print(3E8) # Notación científica

### Texto (String)

### Las cadenas (strings) se utilizan para procesar texto. Se pueden definir con comillas simples o dobles. Para incluir comillas dentro de una cadena, se utilizan caracteres de escape (\" o \'). También se pueden usar apóstrofes para delimitar cadenas.

### python

### Copy code

### print("Yo soy una string")

### print("Hoy sigo aprendiendo \"Python\"")

### print('Otra forma de delimitar cadenas usando apóstrofes')

### Estas son las bases del manejo de tipos de datos en Python. Continuaremos explorando operadores y expresiones en la siguiente sección del módulo.

### Exponenciación:

### En matemáticas clásicas, usamos superíndices (por ejemplo,

### 2

### 3

### 2

### 3

### ), pero Python utiliza \*\* en lugar de esa notación. Es crucial recordar que si ambos argumentos de \*\* son enteros, el resultado también será un entero; sin embargo, si al menos un argumento es un flotante, el resultado será un flotante.

### Multiplicación:

### El operador de multiplicación utiliza el símbolo \* (asterisco). Python respeta las propiedades de los valores proporcionados, como se explicó anteriormente.

### División:

### El operador de división usa el símbolo / (diagonal). A diferencia de la multiplicación, el resultado siempre es un flotante, independientemente de si el resultado parece ser un número entero. En ocasiones, este comportamiento puede causar inconvenientes cuando se requiere un resultado como número entero. Para esto, se utiliza la División Entera.

### División Entera (floor division):

### El operador de división entera utiliza el símbolo // (doble diagonal). A diferencia de la división estándar (/), la división entera redondea siempre hacia abajo y produce enteros o flotantes dependiendo de los valores involucrados.

### Resto (módulo):

### El operador de módulo (% - porcentaje) da como resultado el resto de la división entera. Por ejemplo,

### 14

### /

### /

### 4

### 14//4 da como resultado 3,

### 3

### ∗

### 4

### 3∗4 da como resultado 12, y

### 14

### −

### 12

### 14−12 da como resultado 2, que es el resto.

### Suma y Resta:

### Los operadores de suma (+) y resta (-) son comunes y están alineados con los estándares matemáticos. Además, el operador de resta puede cambiar el signo de un número.

### Prioridades de los Operadores:

### En expresiones con varios operadores, se sigue un orden de prioridades. Por ejemplo, en la expresión

### 2

### +

### 3

### ∗

### 5

### 2+3∗5, la multiplicación se realiza antes de la suma, dando como resultado 17.

### Python define la jerarquía de los operadores, donde los de mayor jerarquía se evalúan primero. Es importante comprender estas prioridades para evitar ambigüedades en las expresiones.

### Estos conceptos sobre operadores y expresiones son fundamentales para desarrollar programas más complejos en Python. A medida que avances en tu aprendizaje, podrás aplicar estos conocimientos en situaciones más desafiantes y escribir código más sofisticado. ¡Sigue explorando y practicando!

|  |
| --- |
| ntroducción a los Fundamentos de la Programación en Python:  python  Copy code  print("En este módulo, exploraremos los fundamentos de la programación en Python, comprendiendo el funcionamiento de las computadoras, la ejecución de programas y la construcción de lenguajes de programación.")  print("Aprenderemos sobre la diferencia entre lenguajes compilados e interpretados, qué es Python, y las herramientas esenciales para trabajar.")  print("Exploraremos cómo funcionan los programas, destacando su importancia al hacer que una computadora sea útil.")  Lenguajes Naturales y de Programación:  python  Copy code  print("En el contexto de la programación, el lenguaje es una herramienta que nos permite expresar y comunicar ideas. Mientras que los humanos utilizamos lenguajes naturales, las computadoras tienen su propio lenguaje, el primitivo lenguaje de máquina.")  print("Las computadoras, a pesar de su sofisticación, carecen de inteligencia y responden a comandos simples, como instrucciones en un conjunto llamado 'Instruction List' (IL).")  print("Necesitamos un lenguaje común para humanos y computadoras, y es aquí donde entran los lenguajes de programación de alto nivel, como Python.")  Compilación vs. Interpretación:  python  Copy code  print("La programación implica establecer una secuencia de instrucciones para lograr un efecto deseado. Un programa debe cumplir con condiciones de corrección como la ortografía, la sintaxis y la semántica.")  print("Transformamos programas de alto nivel a lenguaje de máquina a través de compilación o interpretación. Los lenguajes de alto nivel están diseñados para una de estas categorías.")  ¿Qué es Python?:  python  Copy code  print("Python, desarrollado por Guido van Rossum, es un lenguaje de programación notable por su crecimiento global. Su nombre proviene de la serie de comedia 'Monty Python's Flying Circus'.")  print("En Python, cada valor tiene un tipo específico, y entre ellos, los valores booleanos (True y False) representan la verdad o falsedad en respuestas a preguntas.")  Herramientas de Desarrollo y Primer Programa en Python:  python  Copy code  print("Para el desarrollo en Python, necesitamos un editor de texto, una consola y un debugger. Los entornos integrados de desarrollo son herramientas útiles.")  print("Python3 es la versión actual del lenguaje. La instalación se verifica con 'python3' en la terminal. Herramientas como IDLE simplifican el proceso de desarrollo.")  print("Un simple programa en Python puede ser nuestro primer paso. Creemos un archivo fuente, escribamos el código para imprimir '¡Hola, Mundo!' y ejecutemos el programa.")  Valores Booleanos y Operadores Aritméticos:  python  Copy code  print("Python utiliza valores booleanos (True/False) para representar la veracidad. Estos valores son esenciales al hacer preguntas y tomar decisiones en programas.")  print("Ahora exploraremos operadores aritméticos en Python, como +, -, \*, /, //, %, y \*\*, que realizan diversas operaciones en valores numéricos.")  Exponenciación y Operadores Aritméticos Continuados:  python  Copy code  print("La exponenciación en Python se realiza con el operador \*\*. Otros operadores incluyen la multiplicación (\*), división (/), división entera (//), resto (%), suma (+), y resta (-).")  print("Cada operador tiene reglas específicas, como la prioridad de operaciones. Por ejemplo, la multiplicación tiene prioridad sobre la suma en una expresión.")  Estos ajustes buscan mejorar la claridad y formalidad del texto, utilizando un lenguaje más profesional y estructurado. |

# Desarrollo en Python 3.12

## Herramientas de Desarrollo

Para un desarrollo eficiente en Python 3.12, es esencial contar con las herramientas adecuadas:

1. **Editor de Texto Avanzado:** Utiliza un editor de texto con características avanzadas que faciliten la escritura de código. Puedes optar por IDEs populares como VSCode, PyCharm o Jupyter Notebooks.
2. **Consola Interactiva:** Utiliza una consola interactiva para ejecutar y probar fragmentos de código de forma rápida. Además, puedes considerar el uso de entornos virtuales para gestionar las dependencias de tu proyecto.
3. **Depurador (Debugger):** Familiarízate con un depurador que te permita ejecutar el código paso a paso y realizar un seguimiento detallado de la ejecución para facilitar la identificación y corrección de errores.
4. **Control de Versiones:** Utiliza sistemas de control de versiones como Git para gestionar tu código y colaborar de manera efectiva en proyectos.

## Python 3.12 - Descarga e Instalación

Si aún no tienes instalado Python 3.12, sigue estos pasos:

* **Usuarios de Linux:** Python podría estar preinstalado, verifica ingresando "python3" en la terminal.
* **Usuarios de Windows, MacOS y Linux (si no está instalado):** Descarga desde <https://www.python.org/downloads/>. Durante la instalación, asegúrate de marcar la casilla "Agregar Python 3.x al PATH".

## Nuestro Primer Programa en Python

Ahora, crearemos un programa simple para imprimir "¡Hola, Mundo!" en Python 3.12. Sigue estos pasos:

1. **Editor de Texto:** Utiliza un editor avanzado para crear un nuevo archivo fuente.
2. **Código Python:**

pythonCopy code

print("¡Hola, Mundo!")

1. **Ejecución:** Guarda el archivo con extensión ".py" y ejecútalo desde la consola o el editor.

Este programa básico consta de partes esenciales, como la función print que envía texto a la consola.

## Binding en Operadores

El Binding de un operador define el orden de evaluación en expresiones con operadores de la misma prioridad. La mayoría de los operadores en Python tienen un Binding hacia la izquierda, lo que significa que se evalúan de izquierda a derecha.

Es importante comprender cómo se evalúan expresiones, ya que puede afectar el resultado. Por ejemplo, el operador de exponenciación (\*\*) tiene Binding derecho, como se evidencia en casos como 2 \*\* 2 y 4 \*\* 3.

A continuación, se presenta una tabla de operadores y sus prioridades:

| Símbolos | Tipo de Expresión |
| --- | --- |
| + - | UNARIA |
| \*\* | BINARIA |
| \* / // % | BINARIA |
| + - | BINARIA |

## Operadores y Paréntesis

Se pueden utilizar paréntesis para cambiar el orden de las operaciones en una expresión. Las sub-expresiones dentro de paréntesis se evalúan primero, siguiendo las reglas aritméticas.

Ejemplo:

pythonCopy code

resultado = (3 + 2) \* 5

## Variables en Python 3.12

Las variables son contenedores para almacenar resultados intermedios y permiten reutilizar valores en operaciones posteriores. Algunas reglas para nombrar variables:

* Deben comenzar con una letra.
* Pueden contener letras, dígitos y guiones bajos.
* Distinguen mayúsculas y minúsculas.
* No pueden tener el mismo nombre que las palabras reservadas de Python.

Ejemplos de nombres de variables:

pythonCopy code

mi\_variable = 10

nombre\_variable = "Ejemplo"

## Función Input()

La función input() permite la interacción con el usuario al leer datos desde la consola. Su resultado es siempre un string, por lo que puede ser necesario convertirlo a otros tipos (int, float) según sea necesario.

Ejemplo:

pythonCopy code

nombre = input("Ingrese su nombre: ")

edad = int(input("Ingrese su edad: "))

Recuerda que los comentarios en Python comienzan con # y se extienden hasta el final de la línea. Pueden ser útiles para explicar el código o desactivar temporalmente ciertas partes durante pruebas.

Estas son algunas de las bases para desarrollar en Python 3.12 de manera efectiva. ¡Explora más funciones y características avanzadas para potenciar tus habilidades de programación!

# Actividades Módulo 2

### Actividad 1

Un caracter de escape le debe su nombre al hecho de que:

* No puede ser atrapado debido a su alta velocidad.
* Cambia el significado del carácter después de él.
* Escapa del archivo fuente hacia la memoria de la computadora.

### Actividad 2

La diferencia más importante entre números enteros y fraccionarios es que:

* No pueden ser usados simultáneamente.
* Se almacenan de manera diferente en la memoria de la computadora.
* Los int no pueden ser literales mientras que los float sí.

### Actividad 3

El operador // entre dos números:

* No puede ser utilizado .
* Realiza una división entera.
* Realiza una división regular.

### Actividad 4

El resultado de la la siguiente suma 123 + 0.0:

* Es 123.
* Es 123.0
* No puede ser evaluado.

### Actividad 5

Solo una de las siguientes afirmaciones es verdadera, ¿Cuál?

* La suma precede a la multiplicación .
* La multiplicación precede a la suma .
* Ninguna de las anteriores.

### Actividad 6

Una KeyWord es es una palabra que:

* Es la palabra más importante del programa.
* No puede ser usada como nombre de variable.
* No se puede usar como nombre de función .

### Actividad 7

Un valor retornado por la función input() es:

* Un entero.
* Un float.
* Una string.

### Actividad 8

Un literal es:

* Una persona que no habla ambigüedades.
* Un valor concreto.
* Un sinónimo de variable.

### Actividad 9

¿Cuáles son literales de string en Python?

* '95'
* "95"
* 95
* 'hola mundo'
* 95.0
* "hola mundo"

### Actividad 10

Toda función retorna o resulta en un valor que puede ser almacenado en una variable

* Verdadero
* Falso

# Actividades Módulo 3

### Actividad 1

Usando uno de los operadores de comparación, escribir un programa simple que solicite al usuario un valor N como entrada, que es un número entero, e imprima False si N es menor que 100, y True si N es mayor o igual que 100 sin usar un bloque if.

Datos de prueba:

* + Usuario ingresa 55; el programa responde False
  + Usuario ingresa 99; el programa responde False
  + Usuario ingresa 100; el programa responde True
  + Usuario ingresa 101; el programa responde True

### Actividad 2

Escribir un programa donde el usuario deba ingresar por consola un valor entre 0 y 255. El valor 255 equivale al color blanco, cualquier otro valor equivale a negro.

Informar por pantalla si se ingresó el valor correspondiente al blanco o a negro (cualquier otro valor es inválido).

### Actividad 3

Escribir un programa donde el usuario deba ingresar un valor entre 0 y 255. Se le informará cuál fue el color ingresado según los siguientes rangos:

* + 0 a 100: Rojo
  + 101 a 180: Verde
  + 181 a 255: Azul

Cualquier otro valor es inválido.

### Actividad 4

Imaginemos que nuestra computadora es fanática de las milanesas, cada vez que el usuario ingrese alguna forma de la palabra “Milanesa”, la computadora gritará “¡Me encanta la MILANESA!”.

Escribir un programa que use esta idea y solicite al usuario ingresar un texto y que genere que la computadora:

* + Escriba “¡Sí, me encanta la MILANESA, el mejor alimento del mundo!”, si el valor

ingresado es “MILANESA”.

* + Escriba “Yo quiero una MILANESA!!!!!!” si el valor ingresado es milanesa.
  + Escriba “No se que es {valor ingresado}, yo quiero MILANESA!!!!!” si el valor

ingresado es cualquier otra cosa.

### Actividad 5

Escribir un programa de manera tal que nos permita determinar si x es divisible por z,. Para esto, deberá pedirle al usuario que ingrese un valor para x, uno para z y luego imprimir por consola el mensaje "x es divisible por z" si, y solo si, esta condición se da, caso contrario debe imprimir el mensaje "x no es divisible por z".

### Actividad 6

Juan quiere retirar $X del cajero automático.

El cajero solo aceptará la transacción si X múltiplo de 5, y la cuenta de Juan tiene dinero suficiente para realizar el retiro (incluidas comisiones).

Por cada retiro el banco cobra $0.50. Escriba un programa que pida ingresar un monto a retirar y un balance de cuenta actual, calcule el balance resultante luego de la operación y lo exponga por consola.

Datos de prueba

* + Balance = 100
  + Retiro = 30
  + El programa dirá: 69.50
  + Balance = 0
  + Retiro = 10

El programa dirá: 0.

* + Balance = 100
  + Retiro = -30
  + El programa dirá: 100
  + Balance = 10
  + Retiro = 10

El programa dirá: 10.

### Actividad 7

Escribir un programa que:

* + Pida un usuario.
  + Pida una contraseña.
  + Verifique si el usuario es válido o inválido (se considera un usuario válido aquel

cuyo valor sea “istea”)

* + Verifique si la contraseña es válida o inválida (se considera una contraseña válida

aquella cuyo valor es “1234”.

* + Si el usuario no es válido deberá informar “Usuario inválido”.
  + Si el usuario es válido pero la contraseña no lo es, informar “Contraseña inválida”.
  + Si usuario y contraseña son válidos informar “Bienvenido” seguido del nombre del

usuario.

### Actividad 8

En una empresa se estipula un sueldo básico de $120 para quienes tienen categoría **D** o **E** y una antigüedad menor a 5 años y trabajaron por lo menos 160 horas en el mes; quienes

no alcancen las 160 horas mensuales tendrán un básico de $100.

* + Los empleados de categoría **C** con una antigüedad menor a 5 años que trabajaron por lo menos 160 horas en el mes recibirán $220, quienes no alcancen las 160 horas mensuales tendrán un básico de $200.
  + Los empleados de categorías **A** o **B** con una antigüedad menor a 5 años que trabajaron por lo menos 160 horas al mes recibirán $420, quienes no alcancen las 160 horas mensuales tendrán un básico de $400.
  + En todos los casos, si la antigüedad es mayor a 5 años, sumar un 1% por año extra.

Se pide crear un programa que permita calcular el salario según la situación laboral ingresada.

Los datos a ingresar serán: categoría, horas mensuales y antigüedad.

### Actividad 9

Hacer un programa que simule un cajero automático. Para esto, primero se pedirá al usuario que indique que tarea quiere realizar mediante el ingreso de un número:

1. Ingresar dinero en la cuenta
2. Extraer dinero de la cuenta
3. Ver saldo disponible
4. Salir

Considerar que el programa inicia con un saldo inicial en la cuenta de $1000.

### Actividad 10

En dungeons & dragons™ para determinar cuánto daño hará un personaje al atacar debemos considerar una serie de variables y condiciones.

Nuestro personaje tiene una lista de atributos, estos son:

* + Fuerza
  + Agilidad
  + Aguante
  + Fuerza del arma
  + Peso del arma

Además de los atributos inherentes al personaje tenemos que tener en cuenta algunos atributos del enemigo al que estamos atacando:

* + Aguante
  + Agilidad
  + Escudo
  + Vida

Hay un elemento más, un poco de azar, contamos con dos dados que debemos lanzar, el dado de la suerte y el dado de la puntería.

Ahora bien, para calcular el resultado de nuestro ataque debemos seguir los siguientes pasos:

1. Primero debemos calcular el daño de ataque de nuestro personaje:
   * Para esto, hay que calcular el daño base de nuestro personaje sumando su fuerza y agilidad y a esto le restamos el resto de la división del peso del arma por el aguante.
   * Si el peso del arma es divisible por su fuerza (la del arma) tendremos un bono equivalente al 3% del daño base. A esto lo llamamos “bonificación por arma equilibrada”.
   * Si nuestra fuerza (del personaje) es menor que el peso del arma tenemos penalidad del 2% de la fuerza del arma. A esto lo llamamos “penalidad por exceso de peso”
   * Tenemos que lanzar los dados (en nuestro caso el valor de estos lo damos nosotros), estos dados son los regulares (de 6 caras):
     + Si el dado de puntería es par, agregamos el valor de este a nuestro daño de ataque, si es impar lo restamos.
     + Si el dado de suerte es mayor o igual a 5, recibimos la bendición divina, esto significa que nuestro ataque atravesará el escudo enemigo si lo tiene (es decir, el escudo del enemigo valdrá 0).
2. Luego de haber calculado nuestro daño de ataque es momento de calcular los puntos de defensa del enemigo, esta se calcula multiplicando su aguante por su agilidad y al resultado de esto le sumamos lo que resulte de la suma de su escudo y el 50% de este.
3. Finalmente calculamos el daño a aplicar, para esto a nuestro daño de ataque le restamos su defensa.
4. Si el daño a aplicar es negativo, significa que nuestro ataque fue muy débil, en consecuencia no deberíamos aplicarlo. Es decir, el daño a aplicar debe ser 0.
5. Le restamos a la vida del enemigo el daño a aplicar.
6. Si su vida es 0 o menos entonces nuestro enemigo ha muerto.

Debemos escribir un programa que dados los atributos de nuestro personaje, los del enemigo y el valor de los dados nos indique mediante consola:

1. Cual es nuestro daño de ataque.
2. Cual es el puntaje de defensa del enemigo.
3. El valor de la vida del enemigo luego de aplicar el daño.
4. Si el enemigo murió (solo si así fue).

# **Contenido Módulo 3: Strings, Operadores de Comparación, Preguntas y Respuestas**

## Strings, Operadores de Comparación, Preguntas y Respuestas

En este módulo, profundizaremos en conceptos clave que abarcan:

* **Operadores de String**
* **Datos de tipo Booleano**
* **Toma de decisiones en Python**
* **Condiciones y ejecución condicional**

### Operadores de String

En el módulo anterior, examinamos operadores aritméticos y abreviados. Ahora, regresemos por un momento a los operadores + y \*, los cuales desempeñan funciones específicas cuando se aplican a strings.

#### Concatenación (Concat)

El operador + aplicado a dos strings se convierte en un operador de concatenación:

pythonCopy code

string1 + string2

Este operador une dos o más strings en uno. No es conmutativo, y los tipos de datos no pueden mezclarse.

#### Replicación (Replication)

El operador \* aplicado a un string y a un número se convierte en un operador de replicación:

pythonCopy code

string \* number

Este operador replica el string la cantidad de veces indicada por el número.

#### Conversión de datos: str()

La función str() convierte un número en un string:

pythonCopy code

str(number)

Esta función puede realizar más funciones que exploraremos posteriormente.

### Nuestro Primer Programa en Python

Ahora, veamos un ejemplo que calcula la longitud de la hipotenusa según el teorema de Pitágoras y muestra el resultado usando print() sin necesidad de comillas:

pythonCopy code

cateto1 = 3

cateto2 = 4

hipotenusa = (cateto1\*\*2 + cateto2\*\*2)\*\*0.5

print(str(hipotenusa))

### Preguntas y Respuestas

Los programas formulan preguntas y reciben respuestas. En Python, las respuestas son de dos tipos: True o False. Utilizamos operadores especiales para plantear preguntas:

#### Comparación: Operador de Igualdad (==)

Para preguntar si dos valores son iguales:

pythonCopy code

1 == 2 # False

2 == 2 # True

"abc" == "abc" # True

"ABC" == "abc" # False

#### Comparación: Operador No Igual a (!=)

Este operador compara los valores y devuelve True si son diferentes:

pythonCopy code

1 != 2 # True

2 != 2 # False

"abc" != "abc" # False

"ABC" != "abc" # True

#### Comparación: Mayor que (>)

Para preguntar si un valor es mayor que otro:

pythonCopy code

var\_manzanas > var\_peras

#### Comparación: Mayor o Igual que (>=)

Variante no estricta del operador mayor que:

pythonCopy code

centigrade\_outside >= 0.0

#### Comparación: Menor o Igual que (<=)

Operador de comparación para preguntar si un valor es menor o igual que otro:

pythonCopy code

current\_velocity\_mph <= 85

### Condiciones y Ejecución Condicional

Python permite tomar decisiones basadas en condiciones. La estructura básica es la sentencia if. Si queremos expresar un "Plan B", usamos else. También podemos anidar condiciones con elif:

pythonCopy code

if condition:

# Realizar acciones si la condición es verdadera

else:

# Realizar acciones si la condición es falsa

Si queremos verificar múltiples condiciones, utilizamos elif:

pythonCopy code

if condition1:

# Acciones si la condition1 es verdadera

elif condition2:

# Acciones si la condition1 es falsa y la condition2 es verdadera

else:

# Acciones si ambas condiciones son falsas

La estructura if-elif-else se organiza en una cascada, donde solo una rama se ejecuta. La sangría es crucial para la legibilidad del código.

¡Sigue practicando y explorando estas estructuras para mejorar tu comprensión!

# Actividades Módulo 4

### Actividad 1

Determinar el valor de cada variable al final de cada programa sin ejecutar el código.

|  |  |
| --- | --- |
| 1 | i = 1  while (i < 16): i \*= 2 |
| 2 | i = 1  j = -1  k = (i >= 0) or (j >= 00) and (i <= 0) or (j <= 0) |
| 3 | i = 0  j = 2  k = 1  v = not(j > k) or (i == j) |
| 4 | i = 7  j = i - 1  while (not(i != 0)):  i /= 2  j += 1 |

### Actividad 2

Escribir un programa que compute si un año es o no bisiesto. Un año es bisiesto si es:

* Divisible entre 4.
* No divisible entre 100.
* Divisible entre 400. (2000 y 2400 son bisiestos pues aún siendo divisibles entre 100 lo son también entre 400. Pero los años 1900, 2100, 2200 y 2300 no lo son porque sólo son divisibles entre 100).
* El programa debe escribir en pantalla la respuesta de con los mensajes: "El año ingresado es bisiesto" o "El año ingresado no es bisiesto" según corresponda.

### Actividad 3

Escribir un programa que pida un número entero positivo mayor a 0 muestre por consola todos los números entre 0 (el 0 incluido) y este. Tener en cuenta que el usuario está obligado a ingresar un número positivo mayor a 0, si ingresa un valor invalido, se le deberá continuar preguntando hasta que el valor sea válido.

### Actividad 4

Escribir un programa que pida un número entero positivo mayor a 0 muestre por consola todos los números entre 0 (el 0 incluido) y este pero en orden inverso. Tener en cuenta que el usuario está obligado a ingresar un número positivo mayor a 0, si ingresa un valor invalido, se le deberá continuar preguntando hasta que el valor sea válido.

### Actividad 5

Escribir un programa que le pida al usuario ingresar dos números enteros, el programa debe validar:

* Si los números ingresados son iguales o el segundo número es menor al primero, Si alguna de estas condiciones se diera, se debe continuar pidiendo el segundo valor hasta que este sea mayor que el primero ingresado.

El programa luego procederá a mostrar en consola cuales son todos los números pares que se encuentren dentro del rango, incluyendo los extremos.

*Ayudin: operador %*

### Actividad 6

Escribir un programa que le pida al usuario ingresar dos números enteros y muestre el resultado de la suma entre ellos. El programa debe continuar haciendo lo mismo hasta que el usuario ingrese 0 como alguno de los valores, mostrando de todas formas el resultado de la suma. Al final del programa debe saludarnos diciendo "Adiós"

Por ejemplo:

Ingrese un numero: 10 Ingrese otro numero: 5 Suma: 15

Ingrese un numero: 4 Ingrese otro numero: 0 Suma: 4

Adiós

### Actividad 7

Escribir un programa que solicite al usuario ingresar un el número de un mes para luego mostrar el nombre de dicho mes en pantalla. Cuando el usuario ingrese un número de mes incorrecto, el programa debe mostrar el mensaje "Error: Número de mes incorrecto".

### Actividad 8

Escribir un programa que solicite al usuario ingresar un el número de un mes para luego mostrar el nombre de dicho mes en pantalla. Cuando el usuario ingrese un número de mes incorrecto, el programa debe mostrar el mensaje "Error: Número de mes incorrecto".

# [Módulo 5: Manipulación Avanzada de Datos en Python 3.12](https://docs.python.org/3/py-modindex.html)

## Tema: Strings, Enums y Librerías

### Enums: Representación Elegante de Constantes

En desarrollos más complejos, representar constantes mediante variables string puede volverse ineficiente. Para abordar este problema, Python introduce las enumeraciones (Enums), que permiten agrupar valores relacionados de manera más estructurada.

pythonCopy code

from enum import Enum

class Color(Enum):

AMARILLO = "#FFFF00"

ROJO = "#FF0000"

AZUL = "#0000FF"

VERDE = "#00FF00"

VIOLETA = "#800080"

NARANJA = "#FF8000"

* La clase Color es una enumeración.
* Los atributos Color.ROJO, Color.VERDE, etc., son miembros de la enumeración y actúan como constantes.
* Los miembros tienen nombres y valores, facilitando su uso y comparación.

### Manipulación Avanzada de Strings

Las cadenas en Python admiten diversas operaciones:

* **Concatenación y Replicación:** Utilizamos + para concatenar y \* para replicar.

pythonCopy code

cadena\_concatenada = "Hola, " + "mundo!"

cadena\_replicada = "Python " \* 3

* **Operación** ord()**:** Obtenemos el valor ASCII/Unicode de un carácter.

pythonCopy code

valor\_ascii = ord('A')

* **Operación** len()**:** Obtener la longitud de una cadena.

pythonCopy code

longitud\_cadena = len("Python")

* **Operación** chr()**:** Obtenemos el carácter correspondiente a un valor ASCII/Unicode.

pythonCopy code

caracter\_a\_partir\_de\_ascii = chr(65)

* **Indexación e Iteración:** Accedemos a caracteres individualmente y recorremos la cadena.

pythonCopy code

primer\_caracter = cadena[0]

for char in cadena:

print(char)

* **Substring:** Extraemos un fragmento de una cadena.

pythonCopy code

substring = cadena[1:5]

* **Operadores** in **y** not in**:** Verificamos si una cadena está presente o ausente en otra.

pythonCopy code

presente = "Python" in cadena

ausente = "Java" not in cadena

* **Funciones** min() **y** max()**:** Obtenemos el valor mínimo y máximo de una cadena basado en su valor ASCII/Unicode.

pythonCopy code

valor\_minimo = min(cadena)

valor\_maximo = max(cadena)

* **Métodos** index() **y** find()**:** Buscamos la posición de un substring.

pythonCopy code

posicion\_index = cadena.index("substring")

posicion\_find = cadena.find("substring")

* **Métodos** upper() **y** lower()**:** Formateamos la cadena a mayúsculas o minúsculas.

pythonCopy code

mayusculas = cadena.upper()

minusculas = cadena.lower()

* **Métodos** lstrip() **y** rstrip()**:** Eliminamos espacios en blanco al inicio o al final de la cadena.

pythonCopy code

sin\_espacios\_inicio = cadena.lstrip()

sin\_espacios\_final = cadena.rstrip()

* **Método** replace()**:** Reemplazamos ocurrencias de un substring con otro.

pythonCopy code

cadena\_reemplazada = cadena.replace("viejo", "nuevo")

### Módulo Math, Random y DateTime

El módulo math proporciona funciones matemáticas avanzadas, mientras que random ofrece herramientas para generar números pseudoaleatorios. Por otro lado, datetime facilita el trabajo con fechas y horas.

Ejemplo de uso:

pythonCopy code

import math

import random

from datetime import datetime

# Funciones matemáticas

raiz\_cuadrada = math.sqrt(16)

potencia = math.pow(2, 3)

# Números pseudoaleatorios

numero\_aleatorio = random.random()

# Fecha y hora actual

fecha\_actual = datetime.now()

Este es un breve resumen de las capacidades avanzadas de Python 3.12 en el manejo de strings, enumeraciones y algunas librerías clave. Estas funcionalidades proporcionan una base sólida para el desarrollo de aplicaciones más complejas y profesionales. ¡Explora más en la documentación oficial para aprovechar al máximo estas herramientas!

## Actividades Módulo 5

#### Actividad 1

Escribir un programa que solicite al usuario ingresar un texto, la aplicación debe validar que dicho texto contenga entre 10 y 20 caracteres, de lo contrario debe indicar un mensaje de error y repetir el pedido. Finalmente debe mostrar el texto en pantalla completamente en mayúsculas.

#### Actividad 2

Escribir un programa que solicite al usuario ingresar dos textos, el primero de al menos 8 caracteres y el segundo de al menos 20 caracteres. En ambos casos se debe repetir el pedido si estas condiciones no se cumplen. El programa debe indicar si el primer texto se encuentra en alguna parte del segundo.

#### Actividad 3

Escribir un programa que solicite al usuario un texto de al menos 3 caracteres. Si esta condición no se cumple, debe repetir el pedido. El programa procederá a indicar si se trata o no de un palíndromo (un palíndromo es una palabra que se lee igual de izquierda a derecha que de derecha a izquierda, por ejemplo: ana, neuquén)

#### Actividad 4

Escribir un programa que traduzca el texto ingresado por el usuario a pirata, para esto el programa tiene que cumplir ciertas condiciones:

* El usuario tiene que escribir al menos 10 caracteres.
* Los piratas solo gritan! La respuesta siempre estará toda en mayúsculas.
* La palabra “hola”, se la reemplazará por “**AJOY**”
* En cada espacio entre palabras se agrega el texto “ **AAARR** “
* Al final agrega “ **IO JO JO Y UNA BOTELLA RON**”.

Por ejemplo:

* El usuario ingresa “hola como estas?”, la aplicación muestra en consola “**AJOY AAARR COMO AAARR ESTAS? IO JO JO Y UNA BOTELLA DE RON**”
* El usuario ingresa “Supercalifragilisticexpialidocious”, la aplicación muestra en consola “**SUPERCALIFRAGILISTICEXPIALIDOCIOUS IO JO JO Y UNA BOTELLA DE RON**”

#### Actividad 5

Escribir un programa que solicite al usuario dos textos de al menos 4 caracteres. El programa también debe validar que la cantidad de caracteres sea par, si no se cumple alguna de estas condiciones debe repetir el pedido. Luego, el programa debe unir la primera mitad del primer texto con la segunda mitad del segundo texto y mostrarla en consola. Por ejemplo:

Primer palabra: hola Segunda palabra: chau Resultado: hoau

#### Actividad 6

Escribir un programa que pida al usuario ingresar información de dos Vectores de dos dimensiones, calcule la magnitud de cada uno de los vectores, cree una versión normalizada y luego determine la distancia euclidiana entre los dos primeros.

El programa debe mostrar en pantalla la magnitud de los vectores, las versiones normalizadas y sus magnitudes y finalmente la distancia entre ambos.

A recordar:

* Los vectores tienen tantos componentes como dimensiones tienen, en este caso al ser de 2D tendrán X e Y, usaremos el tipo de dato float.
* Para calcular la magnitud de un vector usamos
* Para normalizar un vector dividimos sus componentes por la magnitud.
* Diagrama  Descripción generada automáticamenteLa distancia euclidiana entre dos vectores se determina usando la

fórmula

#### Actividad 7

Modificar el ejercicio de dungeons & dragons de la clase 3 para que el valor de los dados sea calculado aleatoriamente usando Random.

En dungeons & dragons™ para determinar cuánto daño hará un personaje al atacar debemos considerar una serie de variables y condiciones. Nuestro personaje tiene una lista de atributos, estos son:

* Fuerza
* Agilidad
* Aguante
* Fuerza del arma
* Peso del arma

Además de los atributos inherentes al personaje tenemos que tener en cuenta algunos atributos del enemigo al que estamos atacando:

* Aguante
* Agilidad
* Escudo
* Vida

Hay un elemento más, un poco de azar, contamos con dos dados que debemos lanzar, el dado de la suerte y el dado de la puntería.

Ahora bien, para calcular el resultado de nuestro ataque debemos seguir los siguientes pasos:

1. Primero debemos calcular el daño de ataque de nuestro personaje:
   * Para esto, hay que calcular el daño base de nuestro personaje sumando su fuerza y agilidad y a esto le restamos el resto de la división del peso del arma por el aguante.
   * Si el peso del arma es divisible por su fuerza (la del arma) tendremos un bono equivalente al 3% del daño base. A esto lo llamamos “bonificación por arma equilibrada”.
   * Si nuestra fuerza (del personaje) es menor que el peso del arma tenemos penalidad del 2% de la fuerza del arma. A esto lo llamamos “penalidad por exceso de peso”
   * Tenemos que lanzar los dados (en nuestro caso el valor de estos lo damos nosotros), estos dados son los regulares (de 6 caras):
   * Si el **dado de puntería** es par, agregamos el valor de este a nuestro daño de ataque, si es impar lo restamos.
   * Si el **dado de suerte** es mayor o igual a 5, recibimos la bendición divina, esto significa que nuestro ataque atravesará el escudo enemigo si lo tiene (es decir, el escudo del enemigo valdrá 0).
2. Luego de haber calculado nuestro daño de ataque es momento de calcular los puntos de defensa del enemigo, esta se calcula multiplicando su **aguante** por

su **agilidad** y al resultado de esto le sumamos lo que resulte de la suma de su **escudo** y el 50% de este.

1. Finalmente calculamos el daño a aplicar, para esto a nuestro daño de ataque le restamos su defensa.
2. Si el daño a aplicar es negativo, significa que nuestro ataque fue muy débil, en consecuencia no deberíamos aplicarlo. Es decir, el daño a aplicar debe ser 0.
3. Le restamos a la **vida** del enemigo el daño a aplicar.
4. Si su vida es 0 o menos entonces nuestro enemigo ha muerto.

Debemos escribir un programa que dados los atributos de nuestro personaje, los del enemigo y el valor de los dados nos indique mediante consola:

* + Cual es nuestro daño de ataque.
  + Cual es el puntaje de defensa del enemigo.
  + El valor de la vida del enemigo luego de aplicar el daño.
  + Si el enemigo murió (solo si así fue).

## Contenido Módulo 6. Otra forma de iterar

Otra forma de iterar

En este módulo aprenderemos sobre:

* + Estructura de repetición FOR
  + Sentencias Break y Continue
  + Rama Else

#### Empleando FOR en nuestros bucles

Otro tipo de **LOOP** disponible en Python proviene de la observación de que a veces es más importante contar los "giros o vueltas" del bucle que verificar las condiciones.

Imaginemos que el cuerpo de un bucle debe ejecutarse exactamente cien veces. Si quisieramos utilizar el bucle while para hacerlo, puede tener este aspecto:

i = 0

while i < 100:

# hacer\_algo()

i += 1

(ejemplos/clase6\_for\_1.py)

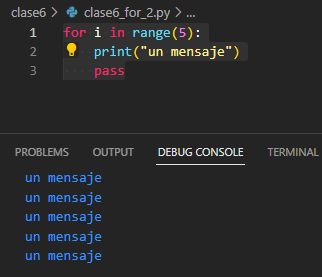
Hay un bucle especial para este tipo de tareas, y se llama **FOR:**

En realidad, el bucle **FOR** está diseñado para realizar tareas más complicadas, puede "explorar" grandes colecciones de datos elemento por elemento. Veremos cómo hacerlo pronto, pero ahora presentaremos una variante más sencilla de su aplicación. Observemos el siguiente ejemplo:

for i in range(5):

print("un mensaje")

pass



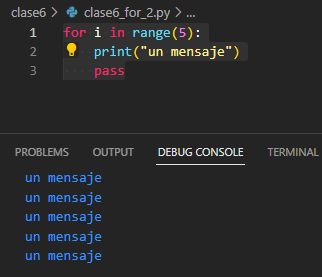
Examinemos uno a uno los nuevos elementos que este ejemplo nos trae:

* + La palabra reservada for abre el bucle **FOR. N**ota: no hay condición después de eso; no tienes que pensar en las condiciones, ya que se verifican internamente, sin ninguna intervención.
  + Cualquier variable después de la palabra reservada **FOR** es la **variable de control** del bucle; cuenta los giros del bucle y lo hace automáticamente.
  + La palabra reservada **IN** introduce un elemento de sintaxis que describe el rango de valores posibles que se asignan a la variable de control.
  + La función **range()** (esta es una función muy especial) es responsable de generar todos los valores deseados de la variable de control; en nuestro ejemplo, la función creará (incluso podemos decir que alimentará el bucle con) valores subsiguientes del siguiente conjunto: 0, 1, 2 .. 97, 98, 99. Nota: en este caso, la función **range()** comienza su trabajo desde 0 y lo finaliza un paso (un número entero) antes del valor de su argumento.
  + Nótese la palabra clave **pass** dentro del cuerpo del bucle no hace nada en absoluto; es una instrucción vacía, la colocamos aquí porque la sintaxis del bucle for exige al menos una instrucción dentro del cuerpo, y podemos dejar comentado la instrucción de PRINT sin ningún inconveniente.

La invocación de la función **range()** puede estar equipada con dos argumentos, no solo uno:

for i in range (3,8):

print("repeticion")



En este caso, el primer argumento determina el valor inicial (primero) de la variable de control.

El último argumento muestra el primer valor que **no se asignará** a la variable de control.

La función **range()** solo acepta enteros como argumentos y genera secuencias de enteros.

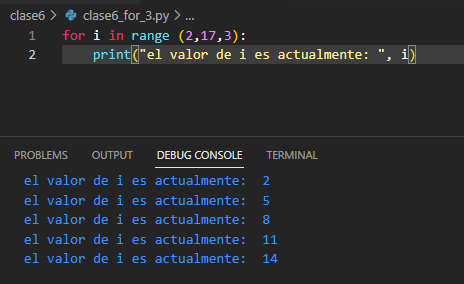
El primer valor mostrado es 3 (tomado del primer argumento de range()).

El último es 7, aunque el segundo argumento de range() es 8, haciendo que la cantidad total de repeticiones sea 5.

Por último, la función **range()** también puede aceptar un tercer argumento, el cual determina la cantidad que incrementa el valor de nuestra variable de control con cada iteración de nuestro **LOOP**

for i in range (2,17,3):

print("el valor de i es actualmente: ", i)



**Nota:** en caso de que el conjunto generado por la función **range()** esté vacío, el bucle no se ejecutara en lo absoluto, ejemplo:

for i in range(1, 1):

#### Las sentencias BREAK y CONTINUE

Hasta ahora hemos tratado el cuerpo del bucle como una secuencia indivisible e inseparable de instrucciones que se realizan completamente en cada bucle. Sin embargo, como desarrolladores, podríamos enfrentar las siguientes opciones:

* + Parece que no es necesario continuar el bucle en su totalidad; se debe abstener de seguir ejecutando el cuerpo del bucle e ir más allá.
  + Parece que necesitamos comenzar el siguiente giro del bucle sin completar la ejecución del turno actual.

Python proporciona dos instrucciones especiales para la implementación de estas dos tareas. Digamos, por razones de precisión, que su existencia en el lenguaje no es necesaria, es decir, un programador experimentado puede codificar cualquier algoritmo sin estas instrucciones. Tales adiciones, que no mejoran el poder expresivo del lenguaje, sino que sólo simplifican el trabajo del desarrollador, a veces se denominan “Syntactic sugar” o “azúcar sintáctica”.

Estas dos instrucciones son:

* + **break:** sale del bucle inmediatamente, e incondicionalmente termina la operación del bucle; el programa comienza a ejecutar la instrucción más cercana después del cuerpo del bucle.
  + **continue:** se comporta como si el programa hubiera llegado repentinamente al final del cuerpo; el siguiente turno se inicia y la expresión de condición se prueba de inmediato.

Ambas palabras son palabras clave reservadas.

Ahora veamos un ejemplo simple para ilustrar cómo funcionan las dos instrucciones:

# break - ejemplo

print("La instruccion break:")

for i in range (1,6):

if i == 3:

break

print("Dentro del bucle: ", i)

print("Fuera del bucle.")

# continue - ejemplo

print("\nLa instruccion continue:")

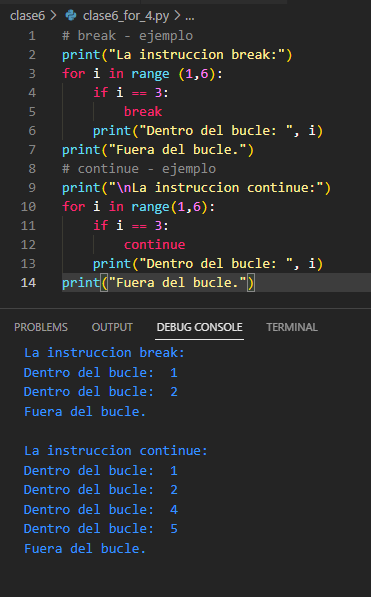
for i in range(1,6):

if i == 3:

continue

print("Dentro del bucle: ", i)

print("Fuera del bucle.")



Para ambos cuerpos, nuestras nuevas instrucciones se ejecutarán en el bucle número 3, por lo que para el primer cuerpo, el mensaje se mostrará en pantalla solo 2 veces hasta llegar al bucle 3. Y para el segundo cuerpo, la instrucción **CONTINUE** producirá que se ejecute el siguiente ciclo, salteando el print() del bucle 3.

#### El bucle FOR y la rama ELSE

Los bucles **for** se comportan de manera un poco diferente, veamos el siguiente ejemplo:

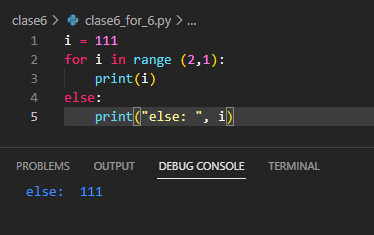
i = 111

for i in range (2,1):

print(i)

else:

print("else: ", i)



Cuando el cuerpo del bucle no se ejecuta, la variable de control conserva el valor que tenía antes del bucle.

**Nota:** si la variable de control no existe antes de que comience el bucle, no estará cuando la ejecución llegue a la rama else.

## Actividades Módulo 6

#### Actividad 1

Escribir un programa que pida un número entero positivo mayor a 0 muestre por consola todos los números entre 0 (el 0 incluido) y este. Tener en cuenta que el usuario está obligado a ingresar un número positivo mayor a 0, si ingresa un valor invalido, se le deberá continuar preguntando hasta que el valor sea válido. Debe usar el loop **for.**

#### Actividad 2

Escribir un programa que pida un número entero positivo mayor a 0 muestre por consola todos los números entre 0 (el 0 incluido) y este pero en orden inverso. Tener en cuenta que el usuario está obligado a ingresar un número positivo mayor a 0, si ingresa un valor invalido, se le deberá continuar preguntando hasta que el valor sea válido. Debe usar el loop **for.**

#### Actividad 3

Escribir un programa que le pida al usuario ingresar dos números enteros, el programa debe validar:

* Si los números ingresados son iguales o el segundo número es menor al primero, Si alguna de estas condiciones se diera, se debe continuar pidiendo el segundo valor hasta que este sea mayor que el primero ingresado.

El programa luego procederá a mostrar en consola cuales son todos los números pares que se encuentren dentro del rango, incluyendo los extremos. Debe usar el loop for.

*Ayudin: operador %*

#### Actividad 4

Escribir un programa donde se ingresan por consola 24 temperaturas entregadas por un sensor. Los datos que entrega el sensor son de la forma: 36.3938284030. La unidad para informar los valores calculados es °K

Se pide informar:

1. Temperatura promedio detectada por el sensor.
2. Mínima temperatura registrada por el sensor.

#### Actividad 5

Se necesita una aplicación para el servicio meteorológico. Para esto se pide que se carguen por teclado la cantidad de agua caída, en milímetros (x ej: 78.5mm, 101.2mm, etc) día a día durante un periodo de una semana.

* + Se pide determinar el día de la semana de mayor lluvia, el de menor y el promedio de milímetros de agua caída.

## Contenido Módulo 7. Listas

¿Por qué necesitamos listas?

Puede que en algún momento tengamos que leer, almacenar, procesar y, finalmente, imprimir docenas, quizás cientos, tal vez incluso miles de números, ¿entonces qué?

¿Necesitamos crear una variable separada para cada valor? ¿Tendremos que pasar largas horas escribiendo sentencias como la que se muestra a continuación?

var1 = int(input())

var2 = int(input())

var3 = int(input())

var4 = int(input())

var5 = int(input())

var6 = int(input())

:

:

varN = int(input())

Si aun así esto parece simple, pensemos cómo se vería escrito en papel un programa que lee cinco números del tipo int y los escribe en pantalla ordenados de menor a mayor (a este tipo de algoritmos los llamamos **sorting**). Lo más probable es que la hoja de papel no alcance.

Hasta ahora hemos aprendido cómo declarar variables que pueden almacenar exactamente un valor dado a la vez. Tales variables a veces se denominan **escalares** por analogía con las matemáticas. Todas las variables que has usado hasta el momento son realmente escalares.

Ahora pensemos en lo conveniente que sería declarar una variable que podría almacenar más de un valor. Por ejemplo, cien, o mil o incluso diez mil. Todavía sería una y la misma variable, pero muy amplia y espaciosa, ¿suena conveniente no?

Quizás pero, ¿cómo manejaríamos un contenedor con tantos valores diferentes?

¿Cómo elegiríamos únicamente el que necesitamos? ¿Los enumeramos? y decimos dame el segundo valor; asignarle tal valor al decimoquinto.

Veamos cómo declarar tales **variables de múltiples valores**. Haremos esto con el ejemplo que acabamos de sugerir. Escribiremos un programa que ordene una secuencia de números, para este caso en particular, asumiremos que son solo 5.

Creemos una variable llamada ***numbers***; se le asigna no sólo un número, sino que se llena con una lista que consta de cinco valores. La lista comienza y termina con corchetes, el espacio entre los corchetes es llenado con cinco números separados por comas.

numbers = [10, 5, 7, 2, 1]

Digamos lo mismo utilizando una terminología adecuada: **numbers es una lista que consta de cinco valores, todos ellos números**. También podemos decir que esta sentencia crea una lista de longitud igual a cinco (ya que contiene cinco elementos).

Los elementos dentro de una lista pueden tener diferentes tipos. Algunos de ellos pueden ser enteros, otros son flotantes y otros pueden ser listas.

**NOTA: r**ecordemos que Python es un lenguaje con base índice 0, por lo que en esta lista el primer elemento estaría en la posición 0, y el último en la posición 4.

Antes de continuar, debemos mencionar que nuestra lista es una **colección de elementos**, pero cada **elemento es un escalar**.

#### Indexando nuestra lista

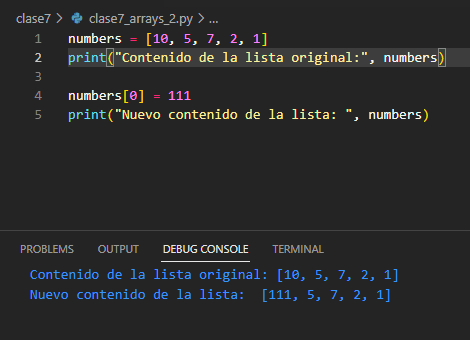
Supongamos que tenemos que cambiar el valor de uno de los elementos de nuestra lista y, ¿cómo lo hacemos? Veamos un ejemplo:

numbers = [10, 5, 7, 2, 1]

print("Contenido de la lista original:", numbers)

numbers[0] = 111

print("Nuevo contenido de la lista: ", numbers)



El valor dentro de los corchetes que selecciona un elemento de la lista se llama un **índice**, mientras que la operación de seleccionar un elemento se conoce como **indexación**.

Ahora supongamos que queremos copiar el valor del quinto elemento al segundo elemento:

numbers = [10, 5, 7, 2, 1]

print("Contenido de la lista original:", numbers)

numbers[1] = numbers[4]

print("Nuevo contenido de la lista:", numbers)

(ejemplos/clase7\_arrays\_3.py)

**Nota:** todos los índices utilizados hasta ahora son literales. Sus valores se fijan en el tiempo de ejecución, pero **cualquier expresión también puede ser un índice**, esto abre muchas posibilidades.

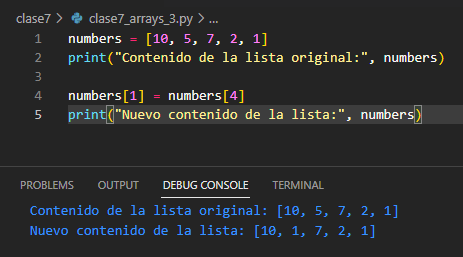
#### Accediendo a elementos de una lista

Se puede acceder a cada uno de los elementos de la lista por separado, por ejemplo, se puede imprimir el 3er elemento de nuestra lista:

numbers = [10, 5, 7, 2, 1]

print("Contenido de la lista original:", numbers)

print("3er elemento:", numbers[2])



También se puede apreciar como, en el punto anterior, pudimos imprimir en pantalla el conjunto entero de elementos de la lista como un todo. Python decora la salida de una manera que sugiere que todos los valores presentados formen una lista.

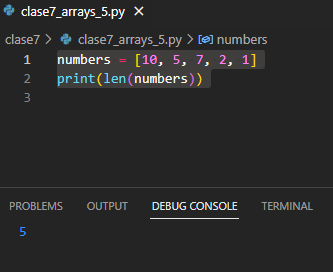
#### La función len()

La longitud de una lista **puede variar durante la ejecución**. Se pueden agregar nuevos elementos a la lista, mientras que otros pueden eliminarse de ella, esto significa que es una entidad muy dinámica.

Si queremos verificar la longitud actual de la lista, podemos usar una función llamada **len()** (su nombre proviene de length - longitud). La función toma el nombre de la lista como un argumento y devuelve el número de elementos almacenados actualmente dentro de ella (en otras palabras, la longitud de la lista).

numbers = [10, 5, 7, 2, 1]

print(len(numbers))



#### Eliminando elementos de una lista

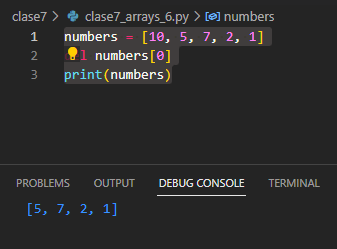
Así como podemos acceder a cualquier elemento de la lista, también podemos eliminar cualquier elemento en todo momento, esto se hace con una instrucción llamada **del** (de la palabra DELETE, eliminar en inglés). **Nota:** es una instrucción, no una función.

Al igual que cuando queremos acceder a un elemento en particular, debemos indicar con el índice al elemento que queremos eliminar, este desaparecerá de la lista y la longitud de ella se reducirá en uno.

numbers = [10, 5, 7, 2, 1]

del numbers[0]

print(numbers)

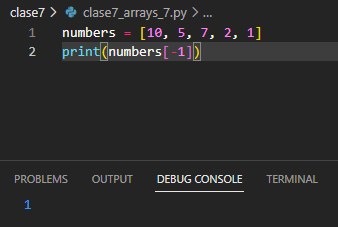


Hay que destacar que no podemos acceder a un elemento que no existe , no podemos obtener su valor ni asignarle un valor. Por lo que si quisiéramos acceder al elemento del índice [4], esto causaría el siguiente error en la ejecución del programa: **IndexError list index out of range**

Al igual que como hemos podido apreciar en el capítulo 5, al manipular strings también podiamos indicar índices con valores negativos, en listas funciona de la misma manera, indicar un índice negativo se interpretará como que nuestra lista debe ser leída de “atrás hacia adelante”. **Un elemento con un índice -1 es el último en una lista.**

numbers = [10, 5, 7, 2, 1]

print(numbers[-1])



#### Funciones VS Metodos

Antes de continuar con la manipulación de listas, y ver como agregar nuevos elementos a una, debemos aprender una diferencia esencial entre funciones y métodos.

Un método **es un tipo específico de función,** se comporta como una función y se parece a una función, pero difiere en la forma en que actúa y en su estilo de invocación.

Una función no pertenece a ningún dato, obtiene datos, puede crear nuevos datos y, generalmente, produce un resultado.

Un método hace todas estas cosas, pero también puede **cambiar el estado de una entidad seleccionada.** Es propiedad de los datos para los que trabaja, mientras que una función es propiedad de todo el código.

Esto también significa que invocar un método requiere alguna especificación de los datos.

En general, una invocación de función típica puede tener este aspecto:

result = function(arg)

La función toma un argumento, hace algo y devuelve un resultado. Una invocación de un método típico usualmente se ve así:

result = data.method(arg)

**Nota:** el nombre del método está precedido por el nombre de los datos que posee. A continuación, se agrega un punto, seguido del nombre del método y un par de paréntesis que encierran los argumentos.

El método se comportará como una función, pero puede hacer algo más: cambiar el estado interno de los datos a partir de los cuales se ha invocado.

#### Agregando elementos a la lista: append() e insert()

Podemos incluir un nuevo elemento al final de nuestra lista con el método **append().** Toma el valor de su argumento y lo coloca al final de la lista que posee el método. La longitud del mismo aumenta en uno.

El método **insert()** es un poco más inteligente: puede agregar un nuevo elemento en cualquier posición de la lista, no solo al final. Esta es su sintaxis:

list.insert(location, value)

* El primero muestra la **ubicación requerida** del elemento a insertar. **Nota:** todos los elementos existentes que ocupan ubicaciones a la derecha del nuevo elemento (incluido el que está en la posición indicada) se desplazan a la derecha, para hacer espacio para el nuevo elemento.
* El segundo es el **elemento a insertar.**

Veamos un ejemplo con ambos métodos en funcionamiento:

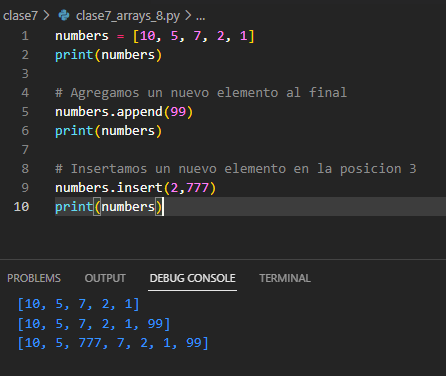
numbers = [10, 5, 7, 2, 1] print(numbers)

# Agregamos un nuevo elemento al final numbers.append(99)

print(numbers)

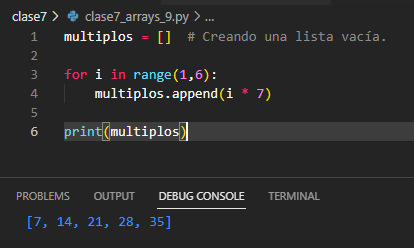
# Insertamos un nuevo elemento en la posicion 3 numbers.insert(2,777)

print(numbers)



Sabiendo esto, ahora también podríamos trabajar con listas vacías, las mismos se crean declarándolas con su lista de elementos con dos corchetes vacíos.

Veamos un ejemplo en el cual crearemos una lista que contenga los 5 primeros múltiplos de 7:



multiplos = [] # Creando una lista vacía.

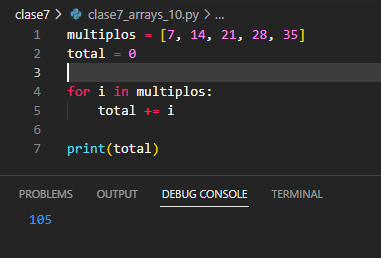
for i in range(1,6): multiplos.append(i \* 7)

print(multiplos)

#### Haciendo uso de los las listas

El bucle **for** tiene una variante muy especial que puede procesar las listas de manera muy efectiva. Supongamos que queremos calcular la suma de todos los valores almacenados en la lista de nuestro último ejemplo. Necesitaremos una variable cuya suma se almacenará y se le asignará inicialmente un valor de 0 - su nombre será **total**.

multiplos = [7, 14, 21, 28, 35]



total = 0

for i in multiplos: total += i

print(total)

* La instrucción **for** especifica la variable utilizada para navegar por la lista **(i)** seguida de la palabra clave in y el nombre de la lista siendo procesada (múltiplos).
* A la variable **(i)** se le asignan los valores de todos los elementos de la lista subsiguiente, y el proceso ocurre tantas veces como haya elementos en la lista.
* Esto significa que usa la variable **(i)** como una copia de los valores de los elementos, y no necesita emplear índices.
* La función **len()** tampoco es necesaria.

#### Listas en acción

Dejemos las listas de lado por un momento y planteemos un pequeño problema. Supongamos que necesitamos reordenar los elementos de una lista, es decir, revertir el orden de los mismos; el primero se intercambiara por el quinto; el segundo con el cuarto; y el tercero permanece en su posición inicial.

¿Cómo podríamos intercambiar los valores de dos variables entre sí?, veamos el siguiente fragmento:

variable\_1 = 1

variable\_2 = 2

variable\_2 = variable\_1 variable\_1 = variable\_2

Si hacemos algo como esto, perdemos el valor previamente almacenado en **variable\_2**. Cambiar el orden de las tareas tampoco nos va a ayudar. Es como intentar intercambiar el contenido líquido de 2 vasos llenos, sin la ayuda de un recipiente para que contenga por unos segundos el contenido del primer vaso sin

derramarlo.

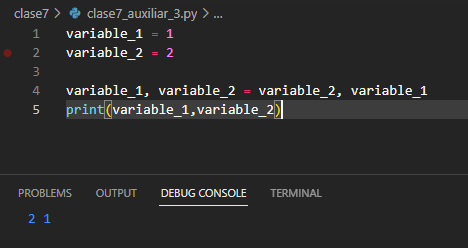
Para este caso necesitamos una tercera variable que sirva como **almacenamiento auxiliar.**

**variable\_1 = 1**

**variable\_2 = 2**

**auxiliar = variable\_1 variable\_1 = variable\_2 variable\_2 = auxiliar**

Afortunadamente, Python ofrece una forma más conveniente de hacer el intercambio, siguiendo la siguiente sintaxis:



variable\_1 = 1

variable\_2 = 2

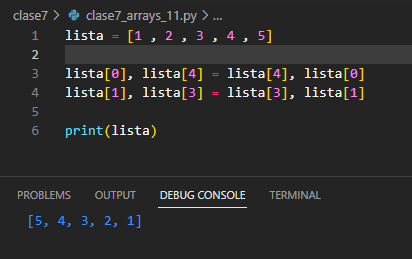
variable\_1, variable\_2 = variable\_2, variable\_1

Sabiendo esto, ahora podemos intercambiar fácilmente los elementos dentro de nuestra lista, siguiendo nuestro primer enunciado:

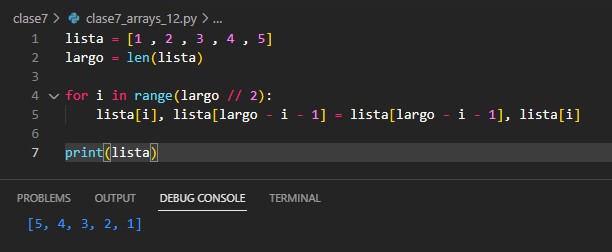
lista = [1 , 2 , 3 , 4 , 5]

lista[0], lista[4] = lista[4], lista[0] lista[1], lista[3] = lista[3], lista[1]

print(lista)



Se ve fácil con solo 5 elementos, ¿verdad?, ¿qué pasa cuando necesitamos hacerlo con 100, 500 o más de 1000 elementos? Podemos usar el **bucle for** para hacer lo mismo automáticamente, independientemente de la longitud de la lista.



lista = [1 , 2 , 3 , 4 , 5]

largo = len(lista)

for i in range(largo // 2):

lista[i], lista[largo - i - 1] = lista[largo - i - 1], lista[i]

print(lista)

* Asignamos la variable **largo** a la longitud de la lista actual (esto hace que nuestro código sea un poco más claro y más corto).
* Preparamos el bucle **for** para que se ejecute su cuerpo **length // 2** veces (esto funciona bien para listas con longitudes pares e impares, porque cuando la lista contiene un número impar de elementos, el del medio permanece intacto).
* Intercambiamos el elemento **i** (desde el principio de la lista) por el que tiene un índice igual a (**largo-i-1**), desde el final de la lista; en nuestro ejemplo, for **i** igual a 0 a la (**largo-i-1**) da 4; for i igual a 3, da 3, esto es exactamente lo que necesitábamos.

## Actividades Módulo 7

#### Actividad 1

Escribir una aplicación que dado una lista de strings, mostrará en consola **todos los elementos de la lista junto con su índice**. Por ejemplo:

**Lista:** {"a", "b", "c"} Muestra:

Elemento 0 tiene a Elemento 1 tiene b Elemento 2 tiene c

#### Actividad 2

Escribir una aplicación que, dado una lista de números, indicará “Todos pares” en consola si todos los elementos de la lista son números pares, de lo contrario, “Hay impares”.

#### Actividad 3

Escribir una aplicación que dada una variable entera que llamaremos **cant** cree una lista de strings con tantos elementos como **cant** indique y luego asignar cada elemento de forma secuencial con el valor "**ELEM\_{n}**" donde n es número entre 1 y **cant** incluido. Finalmente debe mostrar cada elemento de la lista en consola. Por ejemplo:

cant = 4

ListaResultado = {"ELEM\_1","ELEM\_2","ELEM\_3","ELEM\_4"}

#### Actividad 4

Escribir una aplicación que dado una lista de números, indique por consola la **cantidad de elementos impares**.

#### Actividad 5

Implementar una aplicación de manera tal que dada una lista y un valor a buscar, indique por consola el índice de la primera coincidencia del valor buscado o -1 si no existe dicho valor entre los elementos de la lista.

#### Actividad 6

Escribir un programa que dada una lista, este deberá intercambiar la posición de cada par de elementos. Es importante tener en cuenta que la lista debe tener una cantidad de elementos par.

Ejemplo de Lista:

{1, 2, 3, 4}

Luego de ejecutado el programa, la lista debe quedar así:

{2, 1, 4, 3}

#### Actividad 7

Escribir una aplicación que dado una lista de números, nos dirá si este se encuentra ordenada o no, esto quiere decir que cada elemento debe ser mayor o igual al previo, si hay

alguno que no cumpla con esta condición, la aplicación indicará que no es una lista ordenada. Si la lista está vacía, está ordenada.

Ejemplo de Lista:

{1,5,5,7,9,22,22,22,88,78474}

Respuesta: Ordenado.

Ejemplo de Lista:

{1,5,7,3,9,22,88,78474}

Retorno: Desordenado.

## Contenido Módulo 8. Ordenamiento de listas

#### Ordenamiento de Listas (bubblesort)

Ahora que ya aprendimos cómo intercambiar los elementos dentro de una lista, es hora de aprender a **ordenarlos.** Se han inventado muchos algoritmos de clasificación que difieren mucho en velocidad, así como en complejidad. Vamos a mostrar un algoritmo muy simple, fácil de entender, pero desafortunadamente, tampoco es muy eficiente. Se usa muy raramente, y ciertamente no para listas extensas.

Digamos que una lista se puede ordenar de dos maneras:

* Ascendente (o más precisamente, no descendente): si en cada par de elementos adyacentes, el primer elemento no es mayor que el segundo.
* Descendente (o más precisamente, no ascendente): si en cada par de elementos adyacentes, el primer elemento no es menor que el segundo.

En las siguientes secciones ordenaremos la lista en orden ascendente, de modo que los números se ordenen de menor a mayor. Aquí está la lista que usaremos:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **8** | **10** | **6** | **2** | **4** |

Pensemos de la siguiente manera: tomaremos el primer y el segundo elemento y los compararemos; si determinamos que están en el orden incorrecto (es decir, el primero es mayor que el segundo), los intercambiaremos; si su orden es válido, no haremos nada. Un vistazo a nuestra lista confirma esto último: l**os elementos 01 y 02 están en el orden correcto, así como 8<10.**

Ahora veamos el segundo y el tercer elemento. Están en las posiciones equivocadas,

¿cierto? Tenemos que intercambiarlos:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **8** | **6** | **10** | **2** | **4** |

Avanzamos al siguiente par, y nuevamente notamos que 10 no es menor que 2, y deben ser intercambiados

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **8** | **6** | **2** | **10** | **4** |

Avanzamos al siguiente par, la historia se repite una vez más y es necesario intercambiarlos ya que 10 no es menor que 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **8** | **6** | **2** | **4** | **10** |

El primer paso a través de la lista ya está terminado. Aunque todavía estamos lejos de terminar nuestro trabajo, pero algo curioso ha sucedido mientras tanto. El elemento más grande, **10, ya ha llegado al final de la lista.** Para este elemento, esta es la posición deseada. No se puede decir lo mismo de los demás elementos, al menos no por ahora.

Visualicemos por un momento nuestra lista de una forma vertical:

|  |
| --- |
| **10** |
| **4** |
| **2** |
| **6** |
| **8** |

El elemento que contiene el 10, al comenzar estaba en la parte inferior y ahora se encuentra en la parte superior. Podríamos decir que flotó hasta la superficie de la lista, al igual que las burbujas flotan en un vaso de cerveza. Este método de **ordenamiento (Sorting en inglés)** obtiene su nombre de esta observación, se lo denomina **Ordenamiento de burbuja(Bubblesort, en inglés).**

Retomemos nuestro ordenamiento. Miramos el primer y el segundo elemento, es necesario un intercambio:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **6** | **8** | **2** | **4** | **10** |

Repetimos el proceso para los pasos 2 y 3: intercambiamos el 8 con el 2; y luego el 8 con el 4.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **6** | **2** | **8** | **4** | **10** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **6** | **2** | **4** | **8** | **10** |

Queda completa la segunda pasada, ya que 8 ya está en su lugar. Comenzamos con la tercera vuelva y comparamos los 2 primeros elementos, nuevamente el 6 y el 2 necesitan ser intercambiados.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **6** | **4** | **8** | **10** |

Ahora 6 necesita ir a su lugar correspondiente. Cambiamos el segundo y el tercer elemento:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **4** | **6** | **8** | **10** |

¡La lista ya está ordenada! No tenemos nada más que hacer. Esto es exactamente lo que queremos.

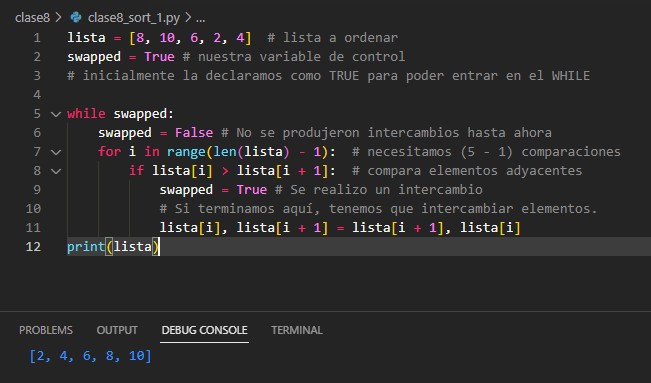
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **4** | **6** | **8** | **10** |

Como puedes ver, la esencia de este algoritmo es simple: **comparamos los elementos adyacentes y, al intercambiar algunos de ellos, logramos nuestro objetivo.**

Codifiquemos en Python todas las acciones realizadas durante un solo paso a través de la lista, y luego intentemos responder esta pregunta en el proceso: ¿Cuántos intercambios fueron necesarios para ordenar de manera ascendente nuestra lista?

Resolvamos este problema de la siguiente manera: creamos una nueva variable, su tarea es observar si se ha realizado algún intercambio durante el pase (recorrido de los pares de la lista) o no. **Si no hay intercambio, entonces la lista ya está ordenada**, y no hay que hacer nada más. Creamos una variable llamada **swapped (intercambio en inglés)**, y le asignamos un valor de **False** para indicar que no hay intercambios. De lo contrario, se le asignará **True**.

lista = [8, 10, 6, 2, 4] # lista a ordenar swapped = True # nuestra variable de control



# inicialmente la declaramos como TRUE para poder entrar en el WHILE

while swapped:

swapped = False # No se produjeron intercambios hasta ahora for i in range(len(lista) - 1): # necesitamos (5 - 1)

comparaciones

if lista[i] > lista[i + 1]: # compara elementos adyacentes

swapped = True # Se realizo un intercambio

# Si terminamos aquí, tenemos que intercambiar

elementos.

lista[i], lista[i + 1] = lista[i + 1], lista[i]

print(lista)

* Declaramos nuestra lista y variable de control
* Ingresamos al **WHILE**, el cual seguirá realizando iteración, siempre que se hagan intercambios en los pases (recorrida completa de la lista)
* **FOR** realizará una cantidad de comparaciones equivalente a **len(lista)-1** (en este caso, 4 veces)
* **IF** comparará los elementos de nuestra lista, y en caso de cumplirse la condición (en este caso, que el elemento de la izquierda sea mayor al de la derecha) hará el intercambio y lo registrará en nuestra variable de control **swapped**
* Finalizadas las iteraciones del **FOR**, el loop **WHILE** comprobará el estado de **swapped**, si se han hecho intercambios en el último pase por la lista, el bloque continúa iterando hasta que no se hagan intercambios
* Por último, nuestro programa imprimirá la lista ya ordenada en pantalla Python, sin embargo, **tiene sus propios mecanismos de clasificación**. Nadie necesita escribir sus propias clases, ya que hay un número suficiente de herramientas listas para usar.

Explicamos este sistema de clasificación porque es importante aprender cómo procesar los contenidos de una lista y mostrarte cómo puede funcionar la clasificación real.

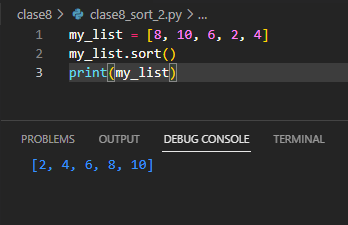
Si queremos que Python ordene nuestra lista, podemos hacerlo usando el método

sort():

my\_list = [8, 10, 6, 2, 4]

my\_list.sort()

print(my\_list)

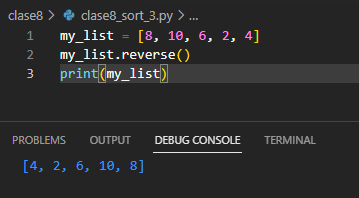


También hay un método de lista llamado **reverse()**, que puedes usar para invertir la lista, por ejemplo:

my\_list = [8, 10, 6, 2, 4]

my\_list.reverse()

print(my\_list)



#### Listas y su funcionamiento interno

Ahora veamos una característica importante y muy sorprendente de las listas, que las distingue de las variables ordinarias. Hay que hacer mucho énfasis en esto, ya que puede afectar nuestros programas y causar graves problemas si se olvida o se pasa por alto.

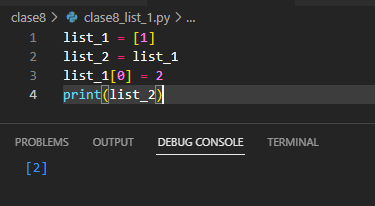
Veamos el siguiente fragmento de código:

list\_1 = [1]

list\_2 = list\_1

list\_1[0] = 2

print(list\_2)



* Crea una lista de un elemento llamada **list\_1**
* La asigna a una nueva lista llamada **list\_2**
* Cambia el único elemento de **list\_1**
* Imprime la list\_2

Es inesperado el hecho de que el programa mostrará como resultado: [2], no [1], que parece ser la solución obvia.

Las listas (y muchas otras entidades complejas de Python) se almacenan de diferentes maneras que las variables ordinarias (escalares). Se podría decir que:

* El nombre de una variable ordinaria es el **nombre de su contenido**.
* El nombre de una lista es el nombre de una ubicación de memoria **donde se almacena la lista.**

Esto significa que la asignación: **list\_2 = list\_1** copia el nombre de la lista, no su contenido. En efecto, los dos nombres (**list\_1 y list\_2**) identifican la misma ubicación en la memoria de la computadora. Modificar uno de ellos afecta al otro, y viceversa.,

¿cómo evitamos esto?

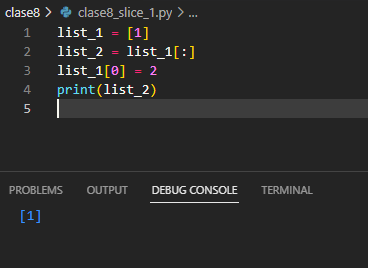
Afortunadamente Python nos provee de una herramienta para ello, **Slices(Rebanadas en inglés)**. Veamos un ejemplo simple:

list\_1 = [1]

list\_2 = list\_1[:]

list\_1[0] = 2

print(list\_2)



Una de las formas más generales de usar rebanadas es la siguiente:

my\_list[start:end]

* **start** es el índice del primer elemento incluido en la rebanada.
* **end** es el índice del primer elemento no incluido en la rebanada.

Como podemos ver, se asemeja a la indexación, pero los dos puntos en el interior hacen una gran diferencia.

Una rebanada de este tipo crea una nueva lista (de destino), tomando elementos de la lista de origen: los elementos de los índices desde el principio hasta el end

**- 1.**

**NOTA:** no hasta el **end**, sino hasta **end-1**. Un elemento con un índice igual a fin es el primer elemento que no participa en la segmentación.

Al igual que en la indexación, es posible utilizar valores negativos tanto para el inicio como para el fin. Si **start** especifica un elemento que se encuentra más allá del descrito por **end** (desde el punto de vista inicial de la lista), la rebanada estará vacía. Veamos un ejemplo:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[2:5]

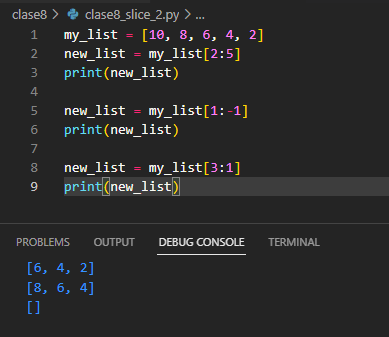
print(new\_list)

new\_list = my\_list[1:-1]

print(new\_list)

new\_list = my\_list[3:1]

print(new\_list)



Si omitimos el **start**, se asume que deseas obtener un segmento que comienza en el elemento con índice 0.

En otras palabras, la rebanada sería de esta forma:

my\_list[:end]

Es un equivalente más compacto de:

my\_list[0:end]

Del mismo modo, si omitimos el **end**, se supone que deseamos que el segmento termine en el elemento con el índice **len(my\_list).**

En otras palabras, la rebanada sería de esta forma:

my\_list[start:]

Es un equivalente más compacto de:

my\_list[start:len(my\_list)]

Veamos un ejemplo:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[:]

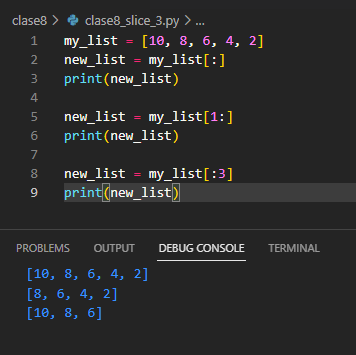
print(new\_list)

new\_list = my\_list[1:]

print(new\_list)

new\_list = my\_list[:3]

print(new\_list)



La instrucción **del,** descrita anteriormente en el capítulo 7, también puede eliminar más de un elemento de la lista a la vez, la única diferencia es que indicar solo el índice correspondiente a **start,** hará que solo se elimine el elemento correspondiente a esa posición. Al que en el caso anterior podemos borrar la lista entera usando como argumento **[:]**

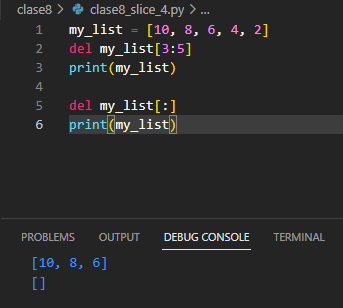
my\_list = [10, 8, 6, 4, 2]

del my\_list[3:5]

print(my\_list)

del my\_list[:]

print(my\_list)



**NOTA:** debemos tener cuidado al momento de utilizar la instrucción **del**, ya que utilizarla sin indicar el slice que deseamos borrar (osea, del my\_list) causará **que se borre la lista, y no su contenido.**

Python también nos permite trabajar con 2 operadores que ya hemos visto en la clase 5, estos son capaces de revisar la lista para verificar si un valor específico está almacenado dentro de ella o no. Estos operadores son **IN y NOT IN**

Al igual que con strings, estos operadores necesitan 2 argumentos, el primero será el elemento que buscaremos, y el segundo la lista donde se realizará la búsqueda, este último argumento también acepta slices de una lista:

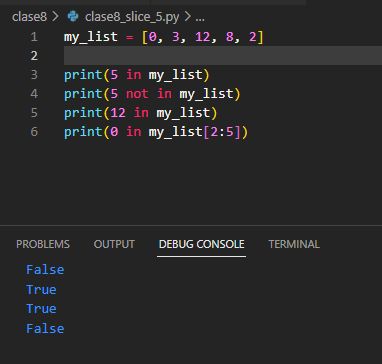
my\_list = [0, 3, 12, 8, 2]

print(5 in my\_list)

print(5 not in my\_list)

print(12 in my\_list)

print(0 in my\_list[2:5])

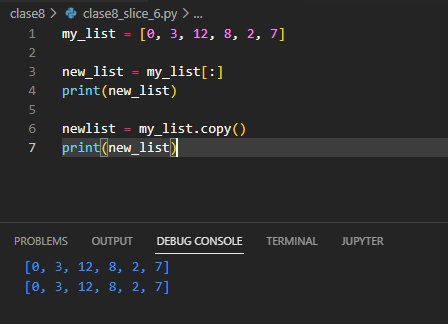


#### Metodo copy()

Por último, debemos mencionar un método de las librerías estándar de Python, el método **copy(),** el cual se encarga de crear una copia de una lista ya existente. Este método se agrega al final de un objeto de lista, y nos devolverá como resultado una copia de dicha lista. Este método no acepta parámetros. Veamos un ejemplo:

my\_list = [0, 3, 12, 8, 2, 7] new\_list = my\_list[:] print(new\_list)

newlist = my\_list.copy() print(new\_list)



Como podemos apreciar, este método tiene la misma funcionalidad que hacer un slice usando **[:]**. Con el fin de mantener los primeros ejemplos en una manera homogénea, hemos omitido este método, pero es bueno conocer las herramientas que Python nos ofrece.

#### La función Enumerate()

Otra función de la librería estándar de Python que nos puede ser útil, es **enumerate(),** toma como argumento un objeto iterable y retorna otro cuyos elementos son tuplas (en el siguiente capitulo veremos en detalle qué son y cómo funcionan) compuesta de 2 objetos, el primero de los cuales indica la posición de un elemento perteneciente a nuestro iterable y el segundo, el elemento mismo.

enumerate(iterable,start=0)

**Nota:** se puede ingresar un **int** como segundo parámetro, el cual indicará la posición desde donde comenzará a trabajar, su valor por defecto es 0.

La función **enumerate()** es especialmente útil cuando al emplear un bucle **for** se precisa tanto de los elementos de un iterable como de su posición.

Veamos un ejemplo:

ista = ["a", "b", "c", "d", "e"]

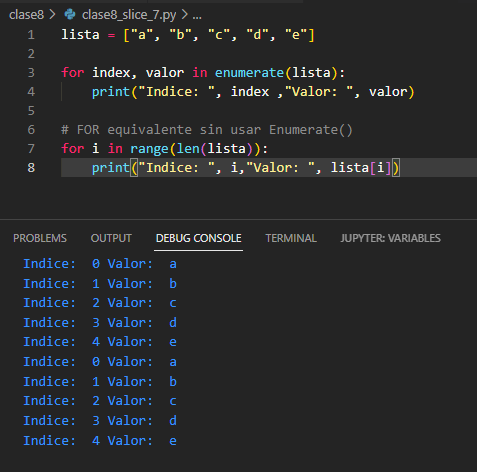
for index, valor in enumerate(lista):

print("Indice: ", index ,"Valor: ", valor)

# FOR equivalente sin usar Enumerate()

for i in range(len(lista)):

print("Indice: ", i,"Valor: ", lista[i])



## Actividades Módulo 8

#### Actividad 1

Escribir una aplicación que dado una lista de strings, mostrará en consola **solo la primera mitad de los elementos**. La primera mitad de una lista de cantidad de elementos impar es del elemento 0 hasta el resultado de la división entera excluida, esto significa que si la lista tiene 5 elementos; debe mostrar los elementos con index 0 1 ya que la división entera entre 5 y 2 es 2.

#### Actividad 2

Escribir una aplicación que dada una lista de strings, en consola **solo la segunda mitad de los elementos**. La segunda mitad de una lista de cantidad de elementos impar es desde el

resultado de la división entera excluida hasta el final, esto significa que si la lista tiene 5 elementos; debe mostrar los elementos con index 2 3 y 4 ya que la división entera entre 5 y 2 es 2

#### Actividad 3

Escribir una aplicación que, dados una lista, un entero que llamaremos inicio y un entero que llamaremos cant, mostrará en consola **cant** elementos de la lista a partir de **inicio**.

#### Actividad 4

Escribir una aplicación que, dada una lista, la app **debe crear una copia de dicha lista**, luego asignar un mismo valor arbitrario a todos los elementos del primer y finalmente, luego mostrar todos los elementos de ambas en consola. Por ejemplo:

* ListaOriginal: {1,2,3,4}
* ListaCopia va a tener: {1,2,3,4}

Asignamos el valor arbitrario 88 a todos los elementos de la listaOriginal: {88,88,88,88}

#### Actividad 5

Escribir una aplicación que dada una lista de cualquier tipo, creará una **nueva Lista** que tendrá los elementos del primero pero en orden inverso.

#### Actividad 6

Crear una aplicación que, dados dos listas indique “Están todos” por consola si todos los elementos del primer Lista se encuentran en alguna parte cualquiera del segundo, de lo contrario, “Falta X” donde x es el elemento presente en el primer Lista pero no en el segundo.

## Contenido Módulo 9. Listas dentro de listas

Listas dentro de listas: bidimensionales

Las listas pueden constar de escalares (es decir, números) y elementos de una estructura mucho más compleja (ya has visto ejemplos como cadenas, booleanos o

incluso otras listas en las lecciones del resumen de la sección anterior). Veamos más de cerca el caso en el que **los elementos de una lista son listas.**

Encontramos estos **arreglos** frecuentemente en nuestras vidas, probablemente el mejor ejemplo de esto sea un **tablero de ajedrez.**

Un tablero de ajedrez está compuesto de filas y columnas, hay ocho filas y ocho columnas, cada columna está marcada con las letras de la A a la H, y cada línea está marcada con un número del uno al ocho.

La ubicación de cada campo se identifica por pares de letras y dígitos. Por lo tanto, sabemos que la esquina inferior derecha del tablero (la que tiene la torre blanca) es A1, mientras que la esquina opuesta es H8.

Supongamos que podemos usar los números seleccionados para representar cualquier pieza de ajedrez, también podemos asumir que **cada fila en el tablero de ajedrez es una lista.**

Veamos el siguiente fragmento de código:

row = []

for i in range(8):

row.append(WHITE\_PAWN)

Crea una lista que contiene ocho elementos que representan la segunda fila del tablero de ajedrez (la que está llena de peones, White pawn, significa peón blanco en inglés).

El mismo efecto se puede lograr mediante una **comprensión de lista**, la sintaxis especial utilizada por Python para completar o llenar listas masivas.

Una comprensión de lista es en realidad una lista, pero **se creó sobre la marcha durante la ejecución del programa, y no se describe de forma estática.**

Ahora veamos:

row = [WHITE\_PAWN for i in range(8)]

La parte del código colocada dentro de los paréntesis específica:

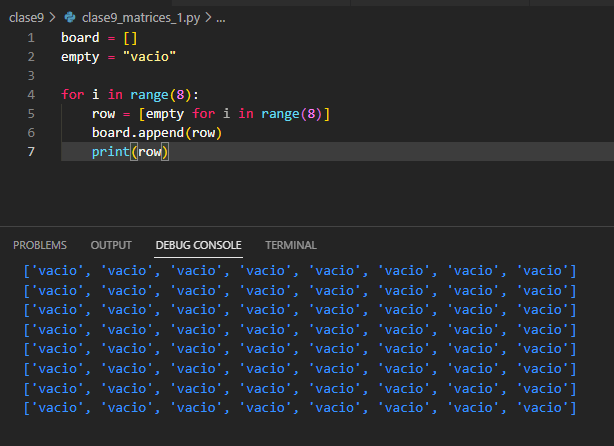
* Los datos que se utilizarán para completar la lista (**WHITE\_PAWN**)
* La cláusula que especifica cuántas veces se producen los datos dentro de la lista (**for i in range(8)**)

Supongamos también que un símbolo predefinido denominado **EMPTY** designa un campo vacío en el tablero de ajedrez.

Entonces, si queremos crear una lista de listas que representan todo el tablero de ajedrez, se puede hacer de la siguiente manera:

board = [] empty = "vacio" for i in range(8):

row = [empty for i in range(8)] board.append(row) print(row)



* La parte interior del bucle crea una fila que consta de ocho elementos (cada uno de ellos es igual a **EMPTY**) y lo agrega a la lista del **board**.
* La parte exterior se repite **ocho veces**.
* Imprimimos cada **row** generado, uno a uno
* En total, la lista **board** consta de 64 elementos (todos iguales a **EMPTY**). Este modelo imita perfectamente el tablero de ajedrez real, que en realidad es una lista de elementos de ocho elementos, todos ellos en filas individuales. Habiendo hecho esto, ahora nuestra variable **board** es un **Array bidimensional**, o en comparación con los términos algebraicos, también se la conoce como **matriz.**

Al ser también una lista de comprensión, su generación puede ser expresada como hemos visto en la clase anterior, el código seria este:

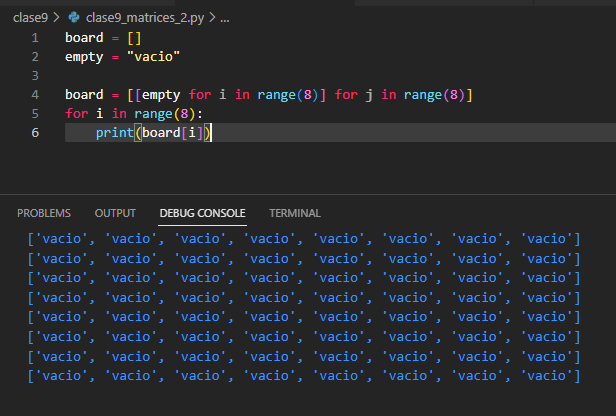
board = []

empty = "vacio"

board = [[empty for i in range(8)] for j in range(8)]

for i in range(8):

print(board[i])



#### Accediendo a nuestra matriz

El acceso al campo seleccionado del tablero requiere dos índices: el primero selecciona la fila; el segundo el número del campo dentro de la fila, el cual es un número de columna.

Echa un vistazo al tablero de ajedrez, cada campo contiene un par de índices que se deben dar para acceder al contenido del campo:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H |
| 8 | **[0] [0]** | **[0] [1]** | **[0] [2]** | **[0] [3]** | **[0] [4]** | **[0] [5]** | **[0] [6]** | **[0] [7]** |
| 7 | **[1] [0]** | **[1] [1]** | **[1] [2]** | **[1] [3]** | **[1] [4]** | **[1] [5]** | **[1] [6]** | **[1] [7]** |
| 6 | **[2] [0]** | **[2] [1]** | **[2] [2]** | **[2] [3]** | **[2] [4]** | **[2] [5]** | **[2] [6]** | **[2] [7]** |
| 5 | **[3] [0]** | **[3] [1]** | **[3] [2]** | **[3] [3]** | **[3] [4]** | **[3] [5]** | **[3] [6]** | **[3] [7]** |
| 4 | **[4] [0]** | **[4] [1]** | **[4] [2]** | **[4] [3]** | **[4] [4]** | **[4] [5]** | **[4] [6]** | **[4] [7]** |
| 3 | **[5] [0]** | **[5] [1]** | **[5] [2]** | **[5] [3]** | **[5] [4]** | **[5] [5]** | **[5] [6]** | **[5] [7]** |
| 2 | **[6] [0]** | **[6] [1]** | **[6] [2]** | **[6] [3]** | **[6] [4]** | **[6] [5]** | **[6] [6]** | **[6] [7]** |
| 1 | **[7] [0]** | **[7] [1]** | **[7] [2]** | **[7] [3]** | **[1] [4]** | **[7] [5]** | **[7] [6]** | **[7] [7]** |

Echando un vistazo a la figura que se muestra arriba, si queremos colocar algunas piezas de ajedrez en el tablero, deberíamos indicar para cada pieza sus índices correspondientes (pensémoslo como coordenadas), agreguemos todas las **torres (ROOK)** que van en cada una de las esquinas del tablero: A1, H1, A8 y H8.

board = []

empty = "vacio"

rook = "TORRE"

board = [[empty for i in range(8)] for j in range(8)]

board[0][0] = rook

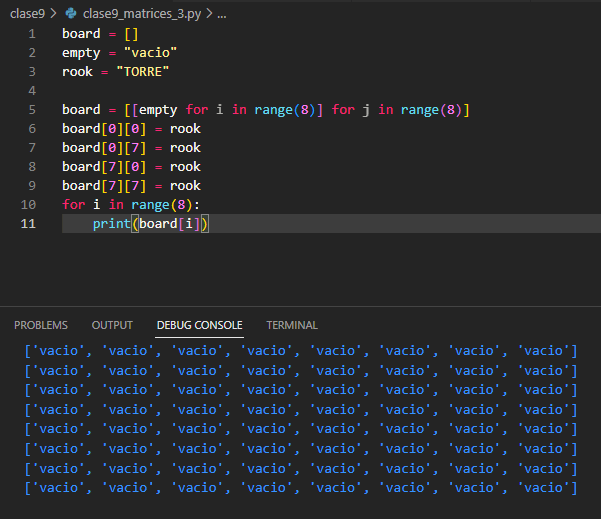
board[0][7] = rook

board[7][0] = rook

board[7][7] = rook

for i in range(8):

print(board[i])



#### Naturaleza multidimensional de las listas: aplicaciones avanzadas

Profundicemos en la naturaleza multidimensional de las listas. Para encontrar cualquier elemento de una lista bidimensional tenemos que usar dos coordenadas:

* Una vertical (número de fila).
* Una horizontal (número de columna).

Imaginemos que tenemos que armar un programa para una estación meteorológica automática, este aparato va a registrar la temperatura una vez por hora y deberá realizarlo y almacenarlo durante todo un mes, esto nos da un total de 24 x 31 = 744 elementos en nuestra lista. Empecemos armando una lista capaz de almacenar estos elementos de manera ordenada.

Primero, debemos decidir qué tipo de datos sería adecuado para esta aplicación, en este caso sería mejor un float ya que este termómetro puede medir la temperatura con una precisión de 0.1 ℃.

Luego tomaremos la decisión arbitraria de que las **filas** registrarán las lecturas cada hora exactamente, por lo que la fila tendrá 24 elementos y cada una de las **filas** se asignará a un día del mes (supongamos que cada mes tiene 31 días, por lo que necesita 31 filas). Aquí está el par apropiado de comprensiones (**h** es para las horas, **d** para el día):

temps = [[0.0 for h in range(24)] for d in range(31)]

¡Listo! Ya tenemos nuestra nueva matriz, aunque de momento está llena de ceros, pero tiene las medidas necesarias para que pueda ser leída fácilmente una vez que se haya recopilado toda la información de temperatura durante 1 mes.

Ahora es el momento de comenzar a usar los datos recolectados, pensemos que queremos determinar la temperatura promedio mensual del mediodía, tendríamos que sumar las 31 lecturas registradas al mediodía y dividir la suma por 31. Supongamos que la temperatura de medianoche se almacena primero. Aquí está el código:

for day in temps:

total += day[11]

average = total / 31

print("Temperatura promedio al mediodía:", average)

Ahora debemos encontrar cuál fue la mayor temperatura durante todo el mes, para ello debemos leer cada valor de fila como si fuese una lista, tal como hemos aprendido en módulos anteriores.

highest = -100.0

for day in temps:

for temp in day:

if temp > highest:

highest = temp

print("La temperatura más alta fue:", highest)

Nota:

* La variable **day** itera en todas las filas de la matriz temps.
* La variable **temp** itera a través de todas las mediciones tomadas en un día. Por ultimo, debemos contar los días en que la temperatura al mediodía fue de al menos 20 ℃:

hot\_days = 0

for day in temps:

if day[11] > 20.0:

hot\_days += 1

print(hot\_days, "fueron los días calurosos.")

#### ¿Por qué necesitamos funciones?

Hasta ahora hemos implementado varias veces el uso de **funciones**, pero solo se han visto algunas de sus ventajas. Solo se han invocado funciones para utilizarlas como herramientas, con el fin de hacer la vida más fácil, y para simplificar tareas tediosas y repetitivas.

Cuando se desea mostrar o imprimir algo en consola se usamos **print()**. Cuando se desea leer el valor de una variable se emplea **input()**, combinados posiblemente con **int() o float().**

También se ha hecho uso de algunos métodos, los cuales también son funciones, pero declarados de una manera muy específica. Ahora aprenderemos a escribir nuestras propias funciones, y cómo utilizarlas. Desde muy sencillas hasta algunas más complejas.

1. Es posible que un fragmento de código se repita varias veces en nuestros programas, puede ser que esté repetido literalmente o con leves modificaciones que consisten en usar el mismo algoritmo pero, por ejemplo, con diferentes variables.

Los programadores no podemos resistirnos a simplificar el trabajo y comenzamos a copiar fragmentos de código usando “copy-paste”. Ahora imaginemos que había un error en ese fragmento de código original que copiamos, vamos a tener que ir a todos los lugares donde se copió el código y corregirlo.

Existe también el riesgo de que las correcciones causen otros errores diferentes, la ley de Murphy está en todas partes.

Entonces podemos definir la primera condición que nos va a ayudar a decidir en qué momento deberíamos escribir nuestra propia función así: ***Si un fragmento de código en particular aparece en más de un lugar****, consideremos la posibilidad de escribirlo en forma de una función que será invocada en todos los lados donde antes estaba dicho fragment*o.

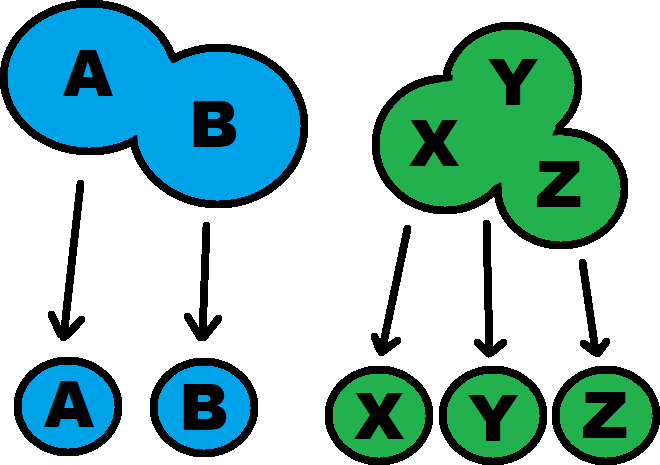
1. Puede suceder que el algoritmo que queremos implementar sea tan complejo que la función main empieza a crecer descontroladamente y de repente empezamos a tener problemas para trabajar con nuestro código.

Podríamos intentar mejorar la situación comentando detalladamente el código, pero en realidad nos vamos a dar cuenta que en vez de ayudar, empeora la situación (mucho comentario hace el código más largo y más difícil de leer). Hay quienes dicen que una función bien escrita debería poder verse entera de un vistazo.

Un programador atento divide el código (o más precisamente, el problema) en varias partes y codifica cada una dentro de una función. Esto simplifica enormemente el trabajo que nos puede llevar un programa dado que permite que cada parte de este

puede ser escrita y probada por separado. A este proceso se lo suele llamar

descomposición.



Definamos entonces la segunda condición: **Si un fragmento de código se vuelve tan grande que se dificulta leerlo y entenderlo**, consideremos la posibilidad de dividirlo en problemas más pequeños e implementar cada uno en forma de una función separada. La descomposición debería continuar hasta conseguir una serie de funciones más pequeñas, fáciles de entender y probar.

#### Nuestra primera función

Para crear una función es necesario **definirla.** Veamos como se define y como es la sintaxis:

def function\_name():

function\_body

* Comienza con la palabra reservada **def** (que significa definir).
* Sigue el **nombre de la función** (las reglas para darle nombre a las funciones son las mismas que para las variables).
* Después del nombre de la función, hay un espacio para un **par de paréntesis**

(de momento están vacíos, pero eso cambiará pronto).

* La línea debe de terminar con **dos puntos**.
* La línea inmediatamente después de def marca el comienzo del **cuerpo de la función,** donde al menos una **instrucción anidada** será ejecutada cada vez que la función sea invocada.

A continuación definamos una función simple, se llamará **message**, y lo que hará será imprimir un mensaje en pantalla cada vez que sea invocada:

def message():

print("hola soy una función!")

Simple, ¿verdad? Cada vez que invoquemos esta función imprimirá en pantalla el saludo. Parece algo sencillo, una función de este estilo es totalmente **utilizable.** Ahora veamos un ejemplo más práctico, en el cual imprimimos en varias ocasiones un mensaje cada vez que es necesario que el usuario ingrese un valor. En lugar de tener una línea para cada mensaje vamos a reemplazarlo por una función que haga dicha impresión.

def message():

print("Ingresa un valor: ")

message()

a = int(input())

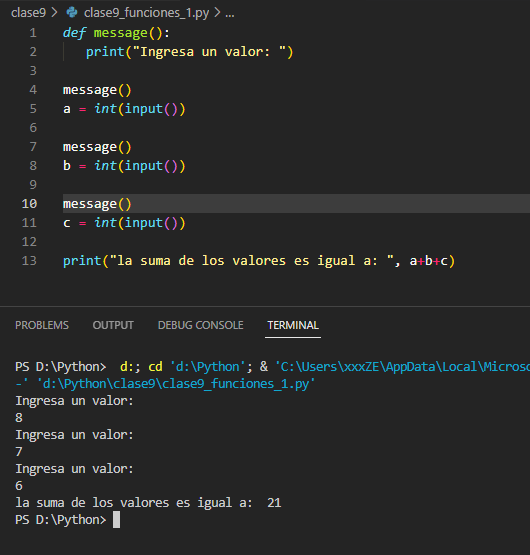
message()

b = int(input())

message()

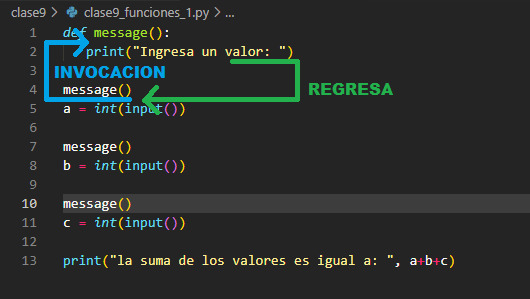
c = int(input())

print("la suma de los valores es igual a: ", a+b+c)



#### Funcionamiento y restricciones de una función

Veamos cómo es el flujo en nuestro último ejemplo:



* Cuando se **invoca** una función, Python recuerda el lugar donde esto ocurre y

**salta** hacia dentro de la función invocada.

* El **cuerpo de la función** es entonces **ejecutado**.
* Al llegar al final de la función, Python **regresa** al lugar inmediato después de donde ocurrió la invocación.
* Este proceso se repite todas las veces que nuestra función sea invocada Existen dos consideraciones muy importantes, la primera de ella es:

No se debe invocar una función antes de que se haya definido.

Recordemos que Python lee el código de arriba hacia abajo, no va a adelantarse en el código para determinar si la función invocada está definida más adelante. El lugar correcto para definirla es antes de ser invocada, en caso de que no sea así, se interrumpe la ejecución, y un mensaje de error nos indicará que la función no fue definida.

NameError: name 'message' is not defined

La segunda consideración es más sencilla:

Una función y una variable no pueden compartir el mismo nombre.

Veamos el siguiente fragmento de código:

def message():

print("Ingresa un valor: ")

message = 1

El asignar un valor al nombre "message" causa que Python olvide su rol anterior. La función con el nombre de message ya no estará disponible.

#### Funciones parametrizadas

El potencial completo de una función se revela cuando puede ser equipada con una interfaz que es capaz de aceptar datos provenientes de la invocación, dichos datos

pueden modificar el comportamiento de la función, haciéndola más flexible y adaptable a condiciones cambiantes.

Un parámetro es una variable, pero existen dos factores que hacen a un parámetro diferente:

* **Los parámetros sólo existen dentro de las funciones en donde han sido definidos**, y el único lugar donde un parámetro puede ser definido es entre los paréntesis después del nombre de la función, donde se encuentra la palabra reservada **def**.
* **La asignación de un valor a un parámetro de una función se hace en el momento en que la función es invocada**, especificando el argumento correspondiente.

def function(parameter):

Veamos un ejemplo:

def mostrarlista(paramostrar):

print(paramostrar)

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[:]

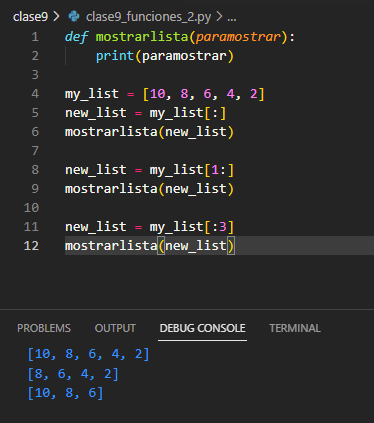
mostrarlista(new\_list)

new\_list = my\_list[1:]

mostrarlista(new\_list)

new\_list = my\_list[:3]

mostrarlista(new\_list)



En este ejemplo, similar al del módulo anterior donde vimos slices, en lugar de tener una línea que imprima cada nuevo slice que hacemos a modo de prueba, es reemplazado por una función que recibirá como parámetro una lista para mostrar en pantalla.

Recordemos: **especificar uno o más parámetros en la definición de la función** es un requerimiento, y se debe cumplir durante la invocación de la misma. **Se debe proveer el mismo número de argumentos como haya parámetros definidos.**

El no hacerlo provocará un error.

## Actividades Módulo 9

#### Actividad 1

*Nota: No olvide que puede usar el argumento end="" print("hola", end="") para no realizar el salto de línea*

Escribir un programa que dada una matriz de cualquier tipo de dato, el programa mostrará en consola el contenido de la matriz de la siguiente forma:

* A modo de ejemplo, una matriz de 3x4 se vería así:

**[a][x][n][b]**

**[v][w][y][z]**

**[h][a][q][e]**

#### Actividad 2

Escribir un programa que dada una matriz de cualquier tipo de dato, el programa mostrará en consola la matriz pero con **las columnas en orden inverso**, por ejemplo:

* **Parámetro:**

{{'a','x','n','b'},

{'v','w','y','z'},

{'h','a','q','e'}}

* **Resultado:**

[b][n][x][a]

[z][y][w][v]

[e][q][a][h]

#### Actividad 3

Escribir un programa que dada una matriz de cualquier tipo de dato, el programa mostrará en consola la matriz pero con las filas y columnas en orden inverso, por ejemplo:

* **Parámetro:**

{{'a','x','n','b'},

{'v','w','y','z'},

{'h','a','q','e'}}

* **Resultado:**

[e][q][a][h]

[z][y][w][v]

[b][n][x][a]

#### Actividad 4

Escribir un programa que dada matriz de cualquier tipo de dato, el programa mostrará en consola **solo las filas de índice par**, por ejemplo:

* **Parámetro:**

{{'a','x','n','b'},

{'v','w','y','z'},

{'h','a','q','e'}}

* **Resultado:**

[a][x][n][b]

[h][a][q][e]

#### Actividad 5

Escribir un programa que dada una matriz de cualquier tipo de dato, el programa mostrará en consola **solo las columnas de índice par**, por ejemplo:

* **Parámetro:**

{{'a','x','n','b'},

{'v','w','y','z'},

{'h','a','q','e'}}

* **Resultado:**

[a][n]

[v][y]

[h][q]

#### Actividad 6

Escribir un programa que dadas dos matrices del mismo tipo, el programa debe **mostrar true si las dos matrices son iguales**, de lo contrario, false. Dos matrices son iguales cuando el tamaño de sus dimensiones es igual y todas sus celdas son iguales.

#### Actividad 7

Escribir un programa que dada una matriz de cualquier tipo, el programa copiará todos los elementos de la matriz a **una nueva matriz** pero con las filas en orden inverso, luego la muestra en pantalla.

Ejemplo:

* **matA:**

{{'a','x','n','b'},

{'v','w','y','z'},

{'h','a','q','e'}}

* **Retorna:**

{{'h','a','q','e'},

{'v','w','y','z'},

{'a','x','n','b'}}

## Contenido Módulo 10. La instrucción Return

Efectos y resultados: la instrucción return

Todas las funciones presentadas anteriormente tienen algún tipo de efecto: producen un texto y lo envían a la consola.

Por supuesto, las funciones, al igual que las funciones matemáticas, pueden tener resultados. Para lograr que las funciones **devuelvan un valor** (aunque no solo para este propósito) se usa la instrucción **return** (regresar o retornar en inglés).

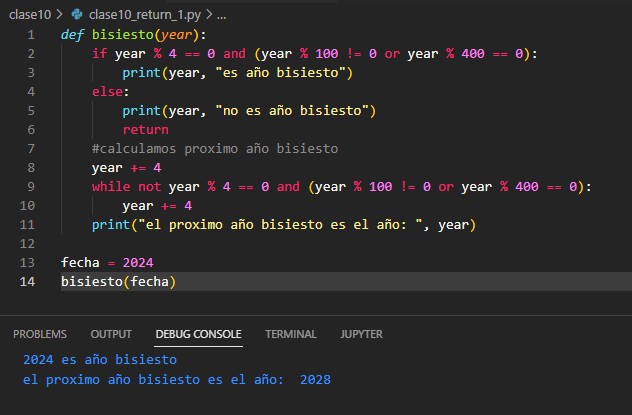
Nota: es una palabra clave reservada de Python.

La instrucción **return** tiene dos variantes diferentes: la primera consiste en la palabra reservada en sí, **sin nada que la siga.**

Cuando se emplea dentro de una función, provoca la **terminación inmediata de la ejecución de la función, y un retorno instantáneo (de ahí el nombre) al punto de invocación,** básicamente se utiliza para **terminar las actividades de la función** antes de que el control llegue al final del cuerpo de la misma**.** Si una función no está destinada a producir un resultado, emplear la instrucción **return** no es obligatorio, se ejecutará implícitamente al final.

Veamos este ejemplo con una función simple, la cual recibirá un valor entero correspondiente a un año, y calculará si corresponde a un año bisiesto o no. Matemáticamente podemos saber si un año es bisiesto si este es **múltiplo de 4**. Si además **es múltiplo de 100 no será bisiesto** (ten en cuenta que 100 es múltiplo de 4 y por tanto cualquier número que sea múltiplo de 100 también es múltiplo de 4, a no ser que sea **múltiplo de 400, que sí será bisiesto).**

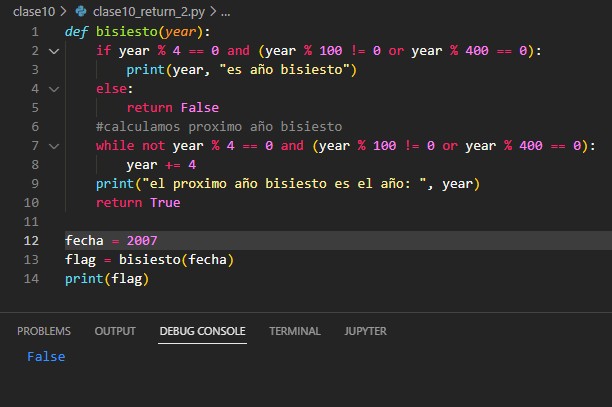
En caso de ser bisiesto, lo informará en pantalla y realizará el calculo para saber cuando es el próximo año bisiesto. Debido a que esta última actividad solo debería realizarse cuando el año es bisiesto, implementamos el return para interrumpir la función para los casos que correspondan:



La segunda variante de **return** está extendida **con una expresión.** Existen dos consecuencias de darle uso de esta manera:

* Provoca la **terminación inmediata de la ejecución de la función** (nada nuevo en comparación con la primer variante).
* Además, la función evaluará el valor de la expresión **y lo devolverá** (de ahí el nombre una vez más) como el resultado de la función.

Supongamos que en nuestro último ejemplo, **en caso de que el año no sea bisiesto**, ya no queremos que imprima mensajes en pantalla, sino que nos devuelva un valor (en este caso un FALSE) el cual guardaremos en una **Variable.**



Ahora la instrucción **return**, enriquecida con la expresión que le dimos, “transporta” dicho valor desde el cuerpo de la función hasta donde fue invocada, y en este caso es almacenado en una variable. Esto no es obligatorio, la función en este caso puede ser invocada, y el valor que retorna puede ser descartado o ignorado.

Entonces:

* Si una función intenta devolver un resultado útil, **es obligatorio** que contenga la segunda variante de la instrucción **return**
* **No es obligatorio** darle uso al valor de retorno de la función, aunque el mismo se pierde de manera definitiva

Antes de continuar, es necesario aclarar unos detalles sobre un valor bastante curioso, el valor llamado **None** (ninguno en inglés). **None** es una palabra reservada de Python.

Sus datos no representan valor razonable alguno; en realidad, no es un valor en lo absoluto; por lo tanto, **no debe participar en ninguna expresión.** Por ejemplo: print(None + 10)

Esta expresión causará un error:

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'Solo existen dos tipos de circunstancias en las que **None** se puede usar de manera segura:

* Cuando **se le asigna a una variable** o se devuelve como **el resultado de una función**
* Cuando **se compara con una variable** para diagnosticar su estado interno. Ya conociendo este nuevo valor, debemos saber que si una **función** no devuelve un cierto valor utilizando una cláusula de expresión **return**, se asume que devuelve implícitamente **None**. Veamos un ejemplo:

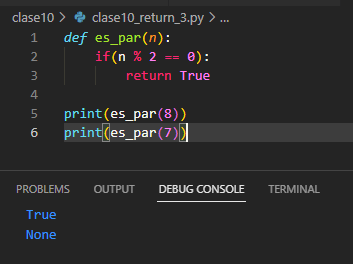
def es\_par(n):

if(n % 2 == 0):

return True

print(es\_par(8))

print(es\_par(7))



Nuestra función sólo devolverá un valor para los casos en los que el argumento con el que se la invoca es un número par. La próxima vez que veamos **None** como el resultado de una función, puede ser síntoma de un error sutil dentro de la función.

#### Funciones y Listas como argumentos o resultados

Existen dos preguntas adicionales que deben responderse; la primera es: ¿Se puede enviar una lista a una función como un argumento?

¡Por supuesto que se puede! Cualquier entidad reconocible por Python puede desempeñar el papel de un argumento de función, aunque debemos **asegurarnos de que la función sea capaz de darle uso.**

Entonces, si pasamos una lista a una función, esta tiene que manejarla como una lista. Vamos a un ejemplo:

def sumalista(milista):

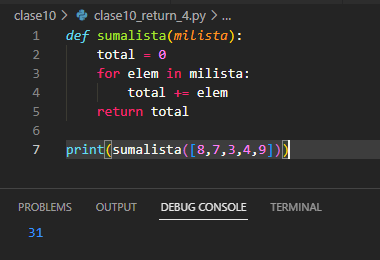
total = 0

for elem in milista:

total += elem

return total

print(sumalista([8,7,3,4,9]))



Nuestra nueva función recibirá como argumento una lista, la cual será procesada iterando entre todos sus elementos y sumándolos en una variable que devolverá como resultado. Invocar la función con un único valor entero, nos devolverá un error, ya que el bucle **for no puede iterar un solo valor entero**. Para este ejemplo, nos mostrará el siguiente error:

Exception has occurred: TypeError

'int' object is not iterable

La segunda pregunta es, ¿puede una lista ser el resultado de una función? ¡Sí, por supuesto! Cualquier entidad reconocible por Python puede ser un resultado de función.

Veamos otro ejemplo práctico:

def regresiva(n):

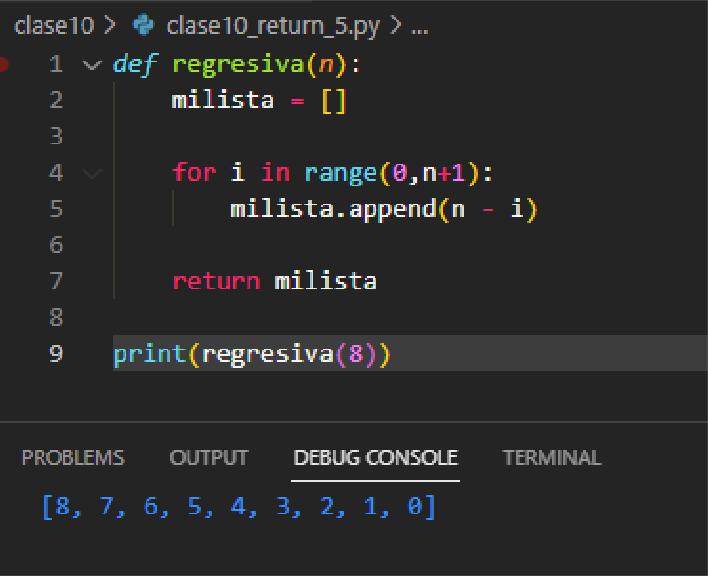
milista = []

for i in range(0,n+1):

milista.append(n - i)

return milista

print(regresiva(8))



Nuestra nueva función recibirá un valor entero y devolverá como resultado una lista compuesta por todos los números que van desde el valor ingresado hasta 0, inclusive.

#### Las funciones y sus alcances

Comencemos con una definición: El **SCOPE** (alcance en inglés) de un nombre (por ejemplo, el nombre de una variable) es la parte del código donde el nombre es reconocido correctamente.

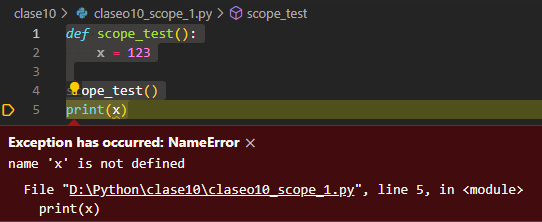
Por ejemplo, el **scope** del parámetro de una función es la función en sí. El parámetro es inaccesible fuera de la función. Veámoslo con un ejemplo:

def scope\_test():

x = 123

scope\_test()

print(x)



Nuestro programa no correrá, esto era de esperarse.

Ahora veamos cómo es que Python define los **scopes** y cómo podemos usarlos para nuestro beneficio:

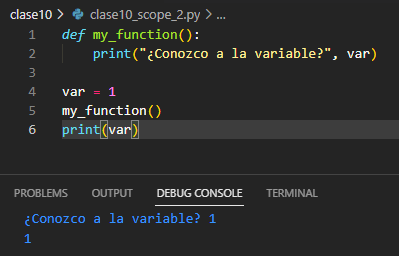
def my\_function():

print("¿Conozco a la variable?", var)

var = 1

my\_function()

print(var)



La respuesta es: una variable que existe fuera de una función tiene alcance dentro del cuerpo de la función.

Esta regla tiene una excepción muy importante, intentemos encontrarla modificando el código y realicemos otra prueba:

def my\_function():

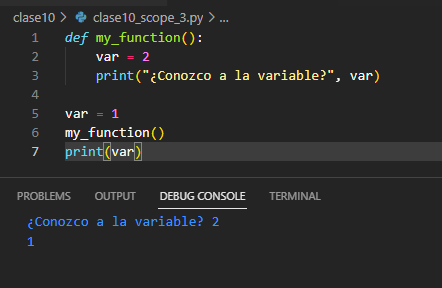
var = 2

print("¿Conozco a la variable?", var)

var = 1

my\_function()

print(var)



El resultado cambió, ¿qué es lo que ocurrió?

* La variable **var** creada dentro de la función no es la misma que la que se definió fuera de ella, parece ser que **hay dos variables diferentes con el mismo nombre**.
* La variable de la función es una sombra de la variable fuera de la función. La regla anterior se puede definir de una manera más precisa y adecuada:

Una variable que existe fuera de una función tiene un alcance dentro del cuerpo de la misma, excluyendo a aquellas que tienen el mismo nombre.

También significa que **el alcance de una variable existente fuera de una función solo se puede implementar dentro de una cuando su valor es leído**. El asignar un valor hace que la función cree su propia variable.

¿Una función es capaz de modificar una variable que fue definida fuera de ella?, esto sería muy incómodo y propenso a errores, afortunadamente, la respuesta es no.

Aun así existe un método especial en Python el cual puede extender el alcance de una variable incluyendo el cuerpo de las funciones para poder no solo leer los valores de las variables sino también modificarlos. Este efecto es causado por la palabra clave reservada llamada **global**:

Veamos su efecto implementándolo en nuestro último ejemplo:

def my\_function():

global var

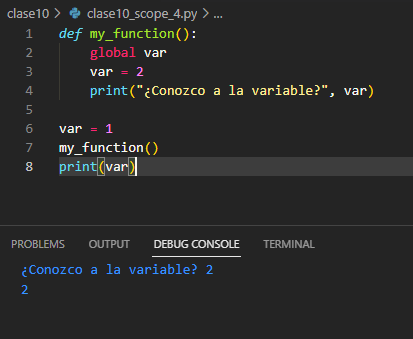
var = 2

print("¿Conozco a la variable?", var)

var = 1

my\_function()

print(var)



#### Interacción de funciones con sus argumentos

Ahora vamos a ver cómo interactúa una función con sus argumentos. En el siguiente ejemplo la función cambia el valor de su parámetro, ¿este cambio afecta el argumento?

def my\_function(n):

print("Yo recibí", n)

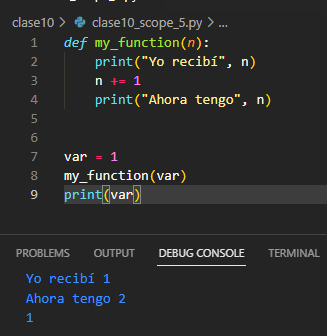
n += 1

print("Ahora tengo", n)

var = 1

my\_function(var)

print(var)



Al cambiar el valor del parámetro este **no se propaga fuera de la función** (más específicamente, no cuando la variable es un valor escalar, como en el ejemplo). Esto también significa que una función recibe el **valor del argumento,** no el argumento en sí. Esto es cierto para los valores escalares.

Vale la pena revisar cómo funciona esto con las listas

def my\_function(my\_list\_1): print("Print #1:", my\_list\_1)

print("Print #2:", my\_list\_2)

my\_list\_1 = [0,1]

print("Print #3:", my\_list\_1)

print("Print #4:", my\_list\_2)

my\_list\_2 = [2, 3] my\_function(my\_list\_2) print("Print afuera:", my\_list\_2)

def my\_function(my\_list\_1):

print("Print #1:", my\_list\_1)

print("Print #2:", my\_list\_2)

my\_list\_1 = [0,1]

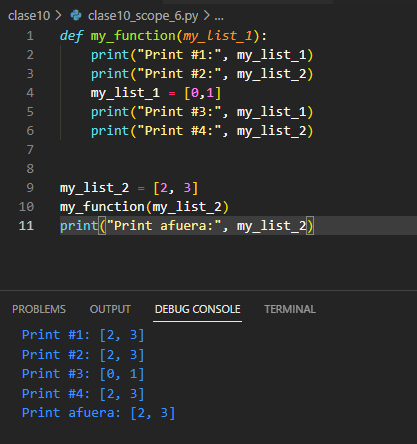
print("Print #3:", my\_list\_1)

print("Print #4:", my\_list\_2)

my\_list\_2 = [2, 3]

my\_function(my\_list\_2)

print("Print afuera:", my\_list\_2)



Parece ser que se sigue aplicando la misma regla, cambiar el valor del parámetro **no se propaga fuera de la función**. Finalmente, la diferencia se puede observar en el siguiente ejemplo:

def my\_function(my\_list\_1):

print("Print #1:", my\_list\_1)

print("Print #2:", my\_list\_2)

del my\_list\_1[0]

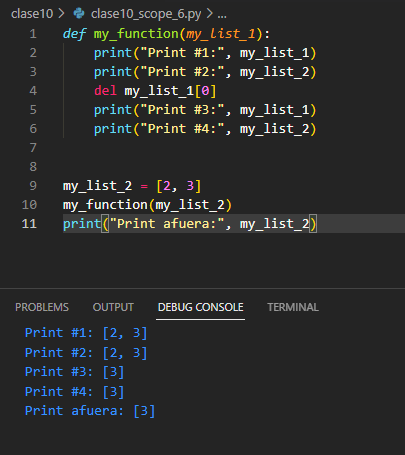
print("Print #3:", my\_list\_1)

print("Print #4:", my\_list\_2)

my\_list\_2 = [2, 3]

my\_function(my\_list\_2)

print("Print afuera:", my\_list\_2)



No se modifica el valor del parámetro, en lugar de eso se modifica la lista identificada por él. Con esto deducimos que:

* Si el argumento es una **lista**, el cambiar el valor del parámetro correspondiente no afecta la lista. **Recuerda:** las variables que contienen listas son almacenadas de manera diferente que las escalares.
* Pero si se modifica **la lista identificada por el parámetro**, esta reflejará el cambio.

Este es un caso muy particular, pero hay que recordar esta clase de excepciones.

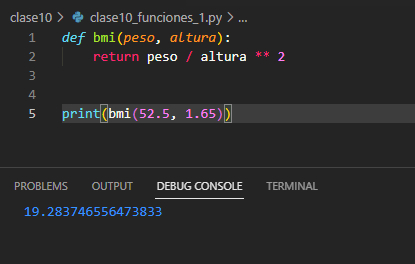
#### Funciones con múltiples parámetros

Para reforzar todos estos conceptos de funciones, comencemos creando una función que se encargue de calcular el índice de masa corporal (IMC). Dicho cálculo se realiza dividiendo el peso (expresado en kilogramos), por la altura al cuadrado (expresada en metros). Nuestra función tendrá **2 parámetros.**

def bmi(peso, altura):

return peso / altura \*\* 2

print(bmi(52.5, 1.65))



La función cumple con lo planteado, pero es un poco sencilla. Deberíamos comprobar que los parámetros son valores confiables. Agreguemos esta verificación y regresar **None** si cualquiera de los dos es incorrecto.

def bmi(peso, altura):

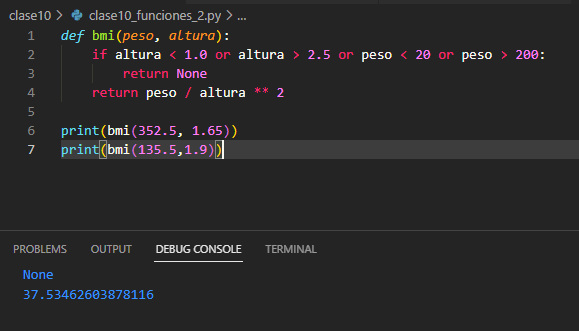
if altura < 1.0 or altura > 2.5 or peso < 20 or peso > 200:

return None

return peso / altura \*\* 2

print(bmi(352.5, 1.65))

print(bmi(135.5,1.9))



Perfecto, ahora verificará que los valores ingresados sean confiables (aunque los valores usados son totalmente arbitrarios, pero sirven como ejemplo), ¿qué más hemos omitido? Esta función devolverá valores incorrectos en caso de que alguien ingrese valores en otro sistema métrico como, por ejemplo, pulgadas para altura o libras para el peso. Agreguemos 2 funciones adicionales para realizar las conversiones necesarias:

def bmi(peso, altura):

if altura < 1.0 or altura > 2.5 or peso < 20 or peso > 200:

return None

return peso / altura \*\* 2

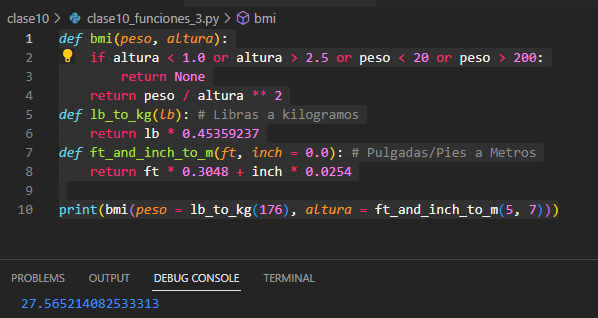
def lb\_to\_kg(lb): # Libras a kilogramos

return lb \* 0.45359237

def ft\_and\_inch\_to\_m(ft, inch = 0.0): # Pulgadas/Pies a Metros

return ft \* 0.3048 + inch \* 0.0254

print(bmi(peso = lb\_to\_kg(176), altura = ft\_and\_inch\_to\_m(5, 7)))



**NOTA:** hay un detalle adicional en la función que convierte los pies y pulgadas a metros, esto se agregó para aquellos casos en que el usuario ingrese un valor de pies+pulgadas, puramente en pies. Este cambio le agrega al segundo parámetro de esa función, un valor por defecto, en caso de que el parámetro no sea especificado. Finalmente nuestro código es capaz de calcular IMC, aun cuando especificamos la altura en pies y el peso en libras.

Ahora armemos un segundo ejemplo, esta vez trabajaremos con triángulos. Armemos una función de los 3 parámetros ingresados, correspondiente al largo cada uno de sus 3 lados, pueden formar un triángulo.

Por el principio de **DESIGUALDAD TRIANGULAR**, sabemos que la suma arbitraria de dos lados tiene que ser mayor que la longitud del tercer lado.

La función retornará **True** si todos los lados pueden formar un triángulo, y **False** de lo contrario.

def is\_a\_triangle(a, b, c):

if a + b <= c:

return False

if b + c <= a:

return False

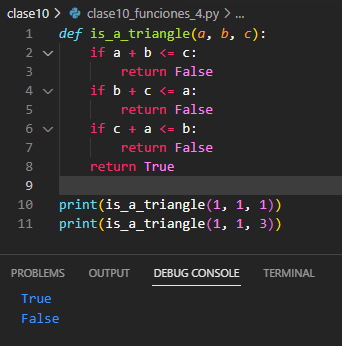
if c + a <= b:

return False

return True

print(is\_a\_triangle(1, 1, 1))

print(is\_a\_triangle(1, 1, 3))



Perfecto, nuestra nueva función cumple con lo planteado, pero podemos compactar la cascada de **IF,** reemplazando la misma por una única sentencia que devolverá el resultado de una triple comparación lógica.

def is\_a\_triangle(a, b, c):

return a + b > c and b + c > a and c + a > b

print(is\_a\_triangle(1, 1, 1))

print(is\_a\_triangle(1, 1, 3))

Se ha negado la condición, se invirtieron los operadores relacionales y se reemplazaron los **or** con **and**, obteniendo una **expresión universal para probar triángulos**.

## Actividades Módulo 10

#### Actividad 1

Escribir una **función** que reciba como **parámetro** dos números que llamaremos a y b, la función debe **retornar** la sumatoria de todos los números que se encuentren entre a y b, incluidos los extremos.

#### Actividad 2

Escribir una función que reciba como parámetro dos números que llamaremos a y b, la función debe **retornar el promedio** de todos los números que se encuentren entre a y b, incluidos los extremos.

#### Actividad 3

Escribir una **función** que reciba como **parámetro** una string que llamaremos separador y dos números que llamaremos a y b. Mostrará por consola todos los números entre a y b de menor a mayor separados por **separador**.

Por ejemplo para los parámetros:

separador: “hola”

a: 1

b: 10

* Debe mostrar:

1hola2hola3hola4hola5hola6hola7hola8hola9hola10

#### Actividad 4

Escribir una **función** de nombre EsPar que reciba como parámetro un número, la función debe retornar true si el parámetro recibido es par, false de lo contrario.

#### Actividad 5

Escribir una **función** que reciba por parámetro una string que llamaremos mes y un número entero que llamaremos **anio**, la función debe retornar la cantidad de días del mes recibido en el **año** recibido. El mes es válido si está correctamente escrito indistinto de las mayúsculas o minúsculas, si el mes es invalido la función retorna -1.

#### Actividad 6

Escribir una **función** que reciba como parámetro un número, que llamaremos nivel, debe estar entre 0 y 255. La función debe retornar el nombre del color según lo que el nivel representa.

* Los rangos son:

100 o menos: Rojo

101 a 180: Verde

181 a 255: Azul más de 255: Negro

#### Actividad 7

Escribir una función **mostrarMenu** que reciba como parámetro una lista que llamaremos opciones. La función le mostrará todas las opciones precedidas por su correspondiente índice + 1 y luego pedirá al usuario que indique la opción que desea, validando que sea una opción correcta, mientras el usuario indique una opción inválida, la función informará “opción inválida” y re-preguntará. Cuando la opción elegida sea válida, retornará el índice de la opción elegida, es decir, si el usuario indica 2, debe retornar 1.

Ejemplo:

Opciones = {“Primer opción”, “Segunda opción”, “Tercer opción”, “Salir”}

1 - Primer opción

2 - Segunda opción

3 - Tercer opción

4 - Salir

Su elección: 5 Opción inválida Su elección:

#### Actividad 8

Se desea implementar un programa que analice los datos de temperatura ambiente en una determinada ciudad para obtener algunas conclusiones climáticas.

Ingresar por teclado y almacenar los datos de temperatura ambiente correspondientes a una semana (7 días). (Formato x ej: 27.93, 28.09)

* Definir una función que retorne el número de día de la semana fue el de menor temperatura
* Definir una función que retorne true si se superó la menor temperatura anual o false de lo contrario (si la menor temperatura estuvo por debajo de la menor temperatura anual que es de -5.22º)
* Definir una función que retorne el promedio de temperatura semanal

Hacer el siguiente menú de opciones para probar el programa

* Cargar datos
* Informar día de menor temperatura
* Informar si se superó la menor temperatura anual
* Informar el promedio de temperatura semanal

## Contenido Módulo 11. Tuplas

Tuplas

Antes de comenzar a hablar acerca de **tuplas y diccionarios**, se deben introducir dos conceptos importantes: **tipos de secuencia** y **mutabilidad**.

Un tipo de secuencia es un tipo de dato en Python el cual es capaz de almacenar más de un valor (o ninguno si la secuencia está vacía), que pueden ser

secuencialmente examinados elemento por elemento.

Debido a que el bucle **for** es una herramienta especialmente diseñada para iterar a través de las secuencias, podemos definirlas de la siguiente manera: una secuencia es un tipo de dato que puede ser escaneado por el bucle **for**.

Hasta ahora hemos trabajado con uno de los tipos de secuencia, las **listas**. Las listas son un clásico ejemplo de secuencias, aunque existen otros tipos.

La segunda noción, **la mutabilidad**, es una propiedad de cualquier tipo de dato en Python que describe su disponibilidad para poder ser alterados, debido a esto, solo habrá dos tipos de datos en Python, los que pueden cambiar libremente durante la ejecución y los que no. A estos se los conoce como **mutables** e **inmutables**, respectivamente.

Los datos **mutables** pueden ser actualizados libremente en cualquier momento, a esta operación se le denomina **"in situ"**. In situ es una expresión en Latín que se traduce literalmente como en posición, en el lugar o momento. Hasta ahora hemos visto tipos de datos como strings o listas, que entran en esta categoría, una lista puede ser modificada con instrucción **append()**

Su contraparte, los datos **inmutables**, no pueden ser modificados de esta manera. Imaginemos que una lista solo puede ser asignada y leída, no podríamos adjuntar ni remover un elemento de la lista, si agregamos un elemento al final de la lista provocaría que la lista se cree desde cero, se tendría que crear una lista completamente nueva, la cual contenga los elementos ya existentes más el nuevo elemento.Un ejemplo de esto es el tipo de dato que veremos a continuación, las **tuplas, la cual** es una secuencia **inmutable que** se puede comportar como una lista pero no puede ser modificada en el momento.

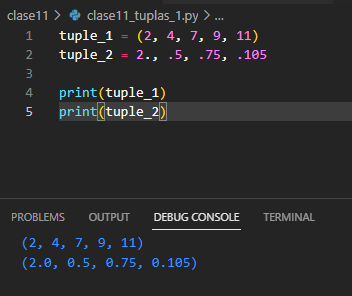
Lo primero que las distingue de las listas, es la sintaxis empleada para crearlas. Las tuplas utilizan paréntesis, mientras que las listas usan corchetes, aunque también es posible crear una tupla tan solo separando los valores por comas.

tuple\_1 = (2, 4, 7, 9, 11)

tuple\_2 = 2., .5, .75, .105

print(tuple\_1)

print(tuple\_2)



Nota: cada elemento de una tupla puede ser de distinto tipo (float,, int, string, o cualquier otro tipo de dato).

También es posible crear una tupla vacía, solo se necesitan unos paréntesis. Si se desea crear una tupla de **un solo elemento**, se debe considerar el hecho de que, debido a la sintaxis, una tupla debe poder distinguirse de un valor entero ordinario, se debe colocar una coma al final, de no hacerlo, será considerado una variable normal. empty\_tuple = ()

print(empty\_tuple)

single\_tuple\_1 = (1, )

single\_tuple\_2 = 1.,

print(single\_tuple\_1)

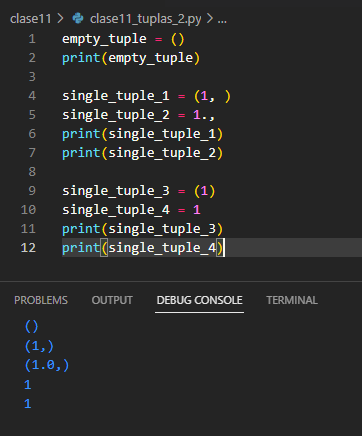
print(single\_tuple\_2)

single\_tuple\_3 = (1)

single\_tuple\_4 = 1

print(single\_tuple\_3)

print(single\_tuple\_4)



#### Interacciones con Tuplas

Si queremos leer los elementos de una tupla, funciona de la misma manera con las listas, pudiendo usar **SLICE** y **FOR** sin inconvenientes.

my\_tuple = (9, 99, 999, 9999, 99999)

print(my\_tuple[0])

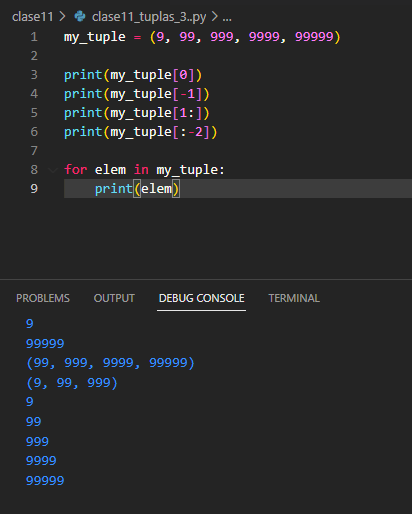
print(my\_tuple[-1])

print(my\_tuple[1:])

print(my\_tuple[:-2])

for elem in my\_tuple:

print(elem)



Nuevamente recordemos que pese a las similitudes con las listas, las tuplas no pueden ser modificadas, e intentar hacerlo causará un error de ejecución

AttributeError: 'tuple' object has no attribute 'append'

¿Qué más pueden hacer las tuplas?

* La función **len()** acepta tuplas, y un valor igual a la cantidad de elementos que contiene.
* El operador **+** puede unir tuplas
* El operador **\*** puede multiplicar las tuplas
* Los operadores **in** y **not in** funcionan de la misma manera que en las listas.

Veamos el siguiente ejemplo:

my\_tuple = (9, 99, 999)

t1 = my\_tuple + (555, 5555)

t2 = my\_tuple \* 3

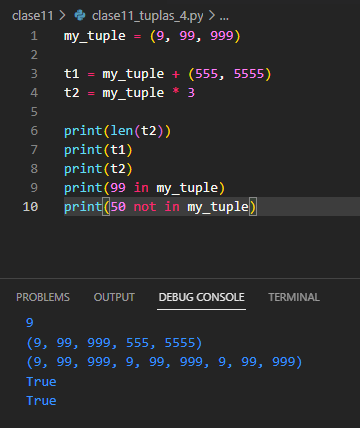
print(len(t2))

print(t1)

print(t2)

print(99 in my\_tuple)

print(50 not in my\_tuple)



Una de las propiedades de las tuplas más útiles es que **pueden aparecer en el lado izquierdo del operador de asignación**. Este fenómeno ya se vio con anterioridad, cuando fue necesario encontrar una manera de intercambiar los valores entre dos variables.

var = 123

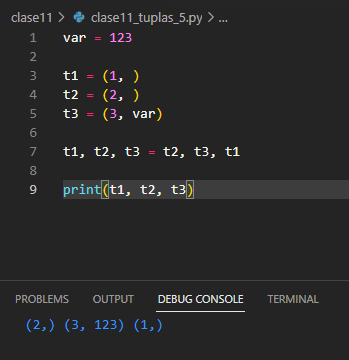
t1 = (1, )

t2 = (2, )

t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)



Este ejemplo nos muestra tres tuplas interactuando, en efecto, los valores almacenados en ellas "circulan" entre ellas. **t1** se convierte en **t2**, **t2** se convierte en **t3**, y **t3** se convierte en **t1**.

Nota: los elementos de una tupla pueden ser variables, no solo literales. Además, pueden ser expresiones si se encuentran en el lado derecho del operador de asignación.

#### Diccionarios

El **diccionario** es otro tipo de estructura de datos de Python, **no es una secuencia** pero puede adaptarse fácilmente a un procesamiento secuencial y además es **mutable**.

Un **diccionario** en Python funciona de la misma manera que funciona un diccionario bilingüe. Por ejemplo, se tiene la palabra en español "gato" y se necesita su equivalente en francés. Lo que se haría es buscar en el diccionario para encontrar la palabra "gato". Eventualmente la encontrarás, y sabrás que la palabra equivalente en francés es "chat".

En el mundo de Python, la palabra que se está buscando se denomina **Key**(key en inglés). La palabra que se obtiene del diccionario es denominada **valor**. Esto significa que un diccionario es un conjunto de pares de **keys y valores.**

Estas son algunas de las características de los **diccionarios:**

* Cada key debe de ser **única, n**o es posible tener una key duplicada.
* Una key **puede ser un tipo de dato de cualquier tipo,** puede ser un número (int o float), o incluso un string.
* Un diccionario no es una lista, porque una lista contiene un conjunto de valores numerados, mientras que **un diccionario almacena pares de valores.**
* La función **len()** aplica también para los diccionarios, regresa la **cantidad de pares** (key-valor) en el diccionario.
* Un diccionario es una **herramienta de un solo sentido**. Si fuese un diccionario español-francés, podríamos buscar en español para encontrar su contraparte en francés pero no de francés a español.

#### ¿Cómo crear un diccionario?

Utilicemos un ejemplo para ver la sintaxis de un diccionario:

dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

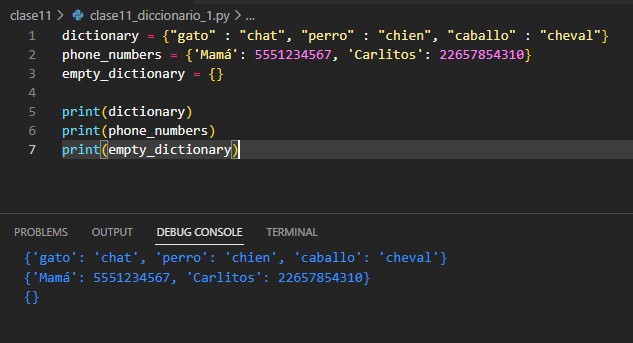
phone\_numbers = {'Mamá': 5551234567, 'Carlitos': 22657854310}

empty\_dictionary = {}

print(dictionary)

print(phone\_numbers)

print(empty\_dictionary)



El diccionario emplea **keys** y **valores** las cuales ambas son **strings**. En el segundo, las **keys** son **strings** pero los valores son **enteros**. El orden inverso también es posible así como la combinación **número** a **número**.

La lista de todos los pares es encerrada con **llaves**, mientras que los pares son separados por **comas**, y las keys y valores por **dos puntos**. Los diccionarios vacíos son construidos por un par vacío de llaves.

#### Interacciones con diccionarios

Una vez que tenemos nuestros diccionarios armados, ¿cómo accedemos a ellos? Para obtener cualquiera de los valores que contiene, debemos proporcionar una **key** válida.

El obtener el valor de un diccionario es semejante a la indexación, gracias a los corchetes alrededor del valor de la key. Siempre teniendo en cuenta que si la key es un string, se lo debe especificar como tal, y que las keys son sensibles a mayúsculas y minúsculas, por ejemplo, la key “carlitoS” no se encuentra en el diccionario que creamos a modo de agenda. Intentar obtener un valor para dicha key resultará en un error.

Para evitar este error, se puede comprobar si la key existe en el diccionario antes de acceder a ella, utilizando alguno de los operadores **IN o NOT IN**

**phone\_numbers = {**

**'Mamá': 5551234567,**

**'Carlitos': 22657854310,**

**'Martin': 5532486619**

**}**

**print("Numero de telefono de Carlitos:", phone\_numbers['Carlitos'])**

**print("Numero de telefono de Mamá:", phone\_numbers['Mamá'])**

**if 'Martin' in phone\_numbers: print("Numero de telefono de Martin:",**

**phone\_numbers['Martin'])**

**if 'Lucho' in phone\_numbers:**

**print("Numero de telefono de Martin:", phone\_numbers['Lucho'])**

phone\_numbers = {

'Mamá': 5551234567,

'Carlitos': 22657854310,

'Martin': 5532486619

}

print("Numero de telefono de Carlitos:", phone\_numbers['Carlitos'])

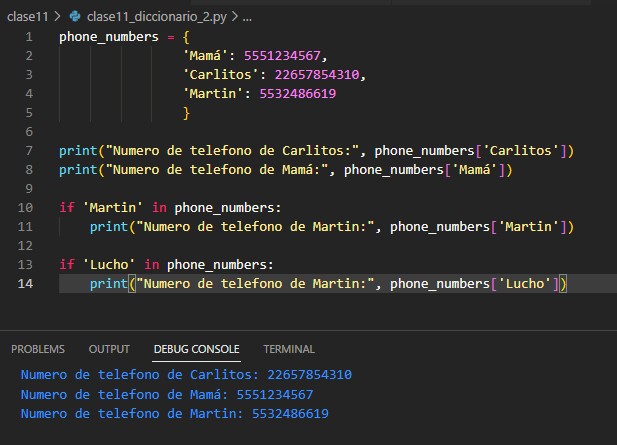
print("Numero de telefono de Mamá:", phone\_numbers['Mamá'])

if 'Martin' in phone\_numbers:

print("Numero de telefono de Martin:", phone\_numbers['Martin'])

if 'Lucho' in phone\_numbers:

print("Numero de telefono de Martin:", phone\_numbers['Lucho'])



Cuando escribimos una expresión grande o larga, es una buena práctica mantenerla alineada verticalmente, así es como hacemos que el código sea más legible y más amigable. Este tipo de formato se llama **sangría francesa.**

¿Qué pasa si queremos acceder a la totalidad del diccionario, podemos recorrerlo con un bucle **for**? No y sí.

No, porque un diccionario no es un tipo de dato secuencial, el bucle **for** no es útil en este caso.

Si, porque hay herramientas simples y muy efectivas que pueden adaptar cualquier diccionario a los requerimientos del bucle **for** (en otras palabras, se construye un enlace intermedio entre el diccionario y una entidad secuencial temporal). El primero de ellos es un método denominado **keys()**, el cual es parte de todo diccionario. El método retorna o regresa una lista de todas las keys dentro del diccionario, al tener una lista de keys se puede acceder a todo el diccionario de una manera fácil y útil.

phone\_numbers = {

'Mamá': 5551234567,

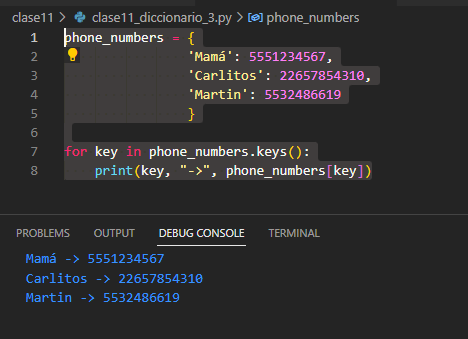
'Carlitos': 22657854310,

'Martin': 5532486619

}

for key in phone\_numbers.keys():

print(key, "->", phone\_numbers[key])



¿Deseas que la salida esté ordenada? Solo hay que agregar al bucle la función

sorted()

alfabeto = {

'Alpha': 40,

'Gamma': 20,

'Epsilon': 15,

'Beta': 60

}

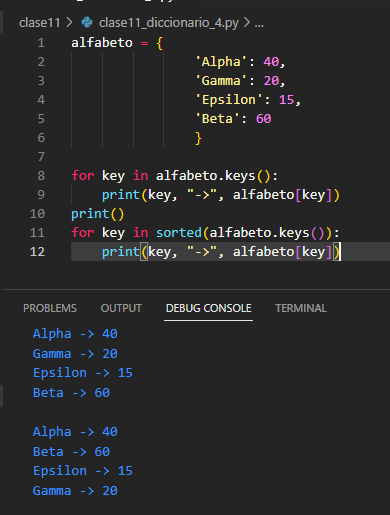
for key in alfabeto.keys():

print(key, "->", alfabeto[key])

print()

for key in sorted(alfabeto.keys()):

print(key, "->", alfabeto[key])



Otra manera de acceder es hacerlo usando el método **items(), e**ste método regresa una lista de tuplas (este es el primer ejemplo en el que las tuplas son más que un ejemplo de sí mismas) donde cada tupla es un par de cada key con su valor. dictionary = {

"gato" : "chat",

"perro" : "chien",

"caballo" : "cheval"

}

for spanish, french in dictionary.items():

print(spanish, "->", french)



Nota la manera en que la tupla ha sido utilizada como una **variable** del bucle **for**.

De igual manera también existe un método denominado **values()**, funciona muy similar al de keys(), pero regresa una lista de valores.

dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

for french in dictionary.values():

print(french)



Como el diccionario no es capaz de automáticamente encontrar la key de un valor dado (recuerda que un diccionario es una **herramienta de un solo sentido)**, el rol de este método es algo limitado.

#### Interacciones con diccionarios

El asignar un nuevo valor a una key existente es sencillo, debido a que los diccionarios son completamente mutables, no existen obstáculos para modificarlos, se va a reemplazar el valor "chat" por "minou", lo cual no es muy adecuado, pero funcionará con nuestro ejemplo.

Para agregar una nueva key con su valor a un diccionario es tan simple como cambiar un valor, solo se tiene que asignar a **una nueva key que no haya existido antes.**

Por último, con la instrucción **del()**, podemos eliminar una key del diccionario, tengamos en cuenta que hacer esto también elimina el valor asociado a dicha key.

Nota: el eliminar una key no existente provocará un error. Veamos un ejemplo con estas nuevas interacciones: dictionary = {

'gato': 'chat',

'perro': 'chien',

'caballo': 'cheval'

}

# reemplazamos el valor de Key

dictionary['gato'] = 'minou'

print(dictionary)

# añadimos un nuevo par al diccionario

dictionary['cisne'] = 'cygne'

print(dictionary)

# eliminamos una key

del dictionary['perro']

print(dictionary)

#### Las tuplas y los diccionarios pueden trabajar juntos

Ahora que ya conocemos las tuplas y diccionarios, es hora de darles un uso más práctico, y ver cómo interactúan entre sí. Imaginemos que hay que armar un programa que cumpla las siguientes condiciones:

* Necesitas un programa para calcular los promedios de tus alumnos.
* El programa pide el nombre del alumno seguido de su calificación.
* Los nombres son ingresados en cualquier orden.
* El ingresar un nombre vacío finaliza el ingreso de los datos. Nota 1: ingresar una puntuación vacía generará la excepción ValueError, pero no te preocupes por eso ahora, verás cómo manejar tales casos cuando hablemos de excepciones en la segunda parte de la serie del curso.
* Una lista con todos los nombres y el promedio de cada alumno debe ser mostrada al final.

Veamos el código y analicemos línea por línea:

school\_class = {}

while True:

name = input("Ingresa el nombre del estudiante: ") if name == '':

break

score = int(input("Ingresa la calificación del estudiante (0- 10): "))

if score not in range(0, 11): break

if name in school\_class: school\_class[name] += (score,)

else:

school\_class[name] = (score,)

for name in sorted(school\_class.keys()): adding = 0

counter = 0

for score in school\_class[name]: adding += score

counter += 1

print(name, ":", adding / counter)

* **Línea 1:** crea un diccionario vacío para ingresar los datos, el nombre del alumno es empleado como key, mientras que todas las calificaciones asociadas son almacenadas en una tupla (la tupla puede ser el valor de un diccionario, esto no es un problema).
* **Línea 3:** se ingresa a un bucle "infinito" (no te preocupes, saldremos de él en el momento indicado).
* **Línea 4:** se lee el nombre del alumno aquí.
* **Línea 5-6:** si el nombre es un string vacío (), salimos del bucle.
* **Línea 8:** se pide la calificación del estudiante (un valor entero en el rango del 1- 10).
* **Línea 9-10:** si la puntuación ingresada no está dentro del rango de 0 a 10, se abandona el bucle.
* **Línea 12-13:** si el nombre del estudiante ya se encuentra en el diccionario, se alarga la tupla asociada con la nueva calificación (observa el operador +=).
* **Línea 14-15:** si el estudiante es nuevo (desconocido para el diccionario), se crea una entrada nueva, su valor es una tupla de un solo elemento la cual contiene la calificación ingresada.
* **Línea 17:** se itera a través de los nombres ordenados de los estudiantes.
* **Línea 18-19:** inicializa los datos necesarios para calcular el promedio (**sum** y

**counter**).

* **Línea 20-22:** se itera a través de la tupla, tomando todas las calificaciones subsecuentes y actualizando la suma junto con el contador.
* **Línea 23:** se calcula e imprime el promedio del alumno junto con su nombre.

#### Errores: nuestro pan de cada día

Todos los programadores queremos escribir código libre de errores y hacemos todo lo posible para alcanzar este objetivo. Lamentablemente, nada es perfecto en este mundo y los programas no son una excepción. Hablando de “excepción” ya vamos a volver a esta palabra pero con un significado que no tiene nada que ver con el que estamos acostumbrados.

Errar es humano, es imposible no cometer errores y es imposible escribir código sin errores, esto no significa que escribir programas desordenados y defectuosos sea una virtud. Más bien podemos decir que incluso el programador más puntilloso no puede evitar defectos menores o mayores, solo quienes no hacen nada no cometen errores. Paradójicamente, aceptar esta difícil verdad nos puede convertir en mejores programadores y mejorar la calidad de nuestro código. Veamos cómo:

#### Errores de datos vs. errores de código

Lidiar con errores de programación tiene, al menos, dos partes, la primera es cuando salta un problema porque nuestro código, aparentemente correcto, se alimenta de datos incorrectos. Por ejemplo, esperamos que se ingrese al código un valor, pero el usuario, en un descuido, ingresa algunas letras al azar, ¿suena familiar?.

Puede suceder que nuestro código termine en ese momento y el usuario se quede con un mensaje de error conciso y a la vez ambiguo en la pantalla, el usuario no va a estar contento y tampoco deberíamos estarlo nosotros.

La segunda parte de lidiar con errores de programación se da cuando ocurre un comportamiento indeseado a raíz de errores que se cometieron cuando escribimos el código. Este tipo de errores son comúnmente llamados bugs (bichos en inglés). Por ejemplo, mientras escribimos un programa que nos pregunta cuántas mascotas tenemos no consideramos que las cantidades sólo pueden con números naturales (positivos y 0) y no lo validamos, haciendo que nuestro programa nos permita indicar que tenemos -10 mascotas.

Esta idea no es tan descabellada como puede parecer, incidentes de este tipo eran comunes en tiempos en que las computadoras ocupaban grandes pasillos, consumían kilovatios de electricidad y producían enormes cantidades de calor. Afortunadamente estos tiempos se han ido para siempre y los únicos errores que pueden estropear tu código son los que tú mismo sembraste en el código. Por lo tanto, intentaremos

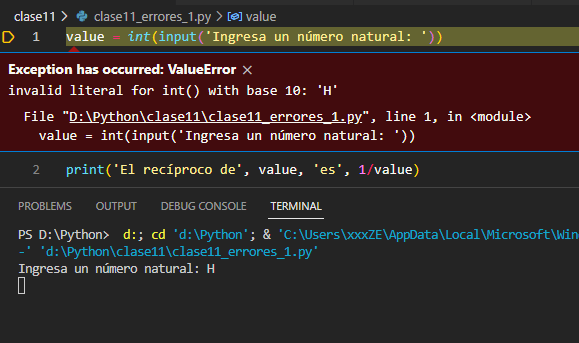
mostrarte cómo encontrar y eliminar tus errores, en otras palabras, cómo depurar tu código.

#### Cuando los datos no son lo que deberían ser

Escribamos un fragmento de código trivial: el programa debe pedir un número entero positivo y mostrará en consola su recíproco (el recíproco de un número es 1/numero, todos los números tienen recíproco excepto el 0), veamos:

value = int(input('Ingresa un número natural: ')) print('El recíproco de', value, 'es', 1/value)

¿Qué podría salir mal con un código tan compacto y simple? Ingresar datos que no sean un número entero positivo (que también podría ser ingresar nada o un cero) va a arruinar la ejecución del programa, esto es lo que veríamos:



Todas las líneas que muestra Python en el mensaje de error son significativas e importantes, pero la **primera** línea parece ser la más valiosa. La primera línea nos indica el nombre de la **excepción** la cual provoca que nuestro código se detenga. Su nombre aquí es **ValueError**. El resto de la línea es solo una breve explicación que especifica con mayor precisión la causa de la excepción ocurrida.

¿Cómo lo afrontamos? ¿Cómo protegemos nuestro código de la terminación abrupta? La primera idea que se nos puede ocurrir es verificar si los datos proporcionados por el usuario son válidos y negarnos a cooperar si son incorrectos, en este caso, la verificación puede basarse en el hecho de que esperamos que el string de entrada contenga solo dígitos.

Con los conocimientos que hemos adquirido hasta ahora no sería difícil escribir e implementar esta verificación, aunque también es posible comprobar si la variable value es de tipo **int** de otra manera. Python tiene un medio especial para este tipo de comprobaciones: es un operador llamado **is**. La revisión en sí puede verse de la siguiente manera:

type(value) is int

Su resultado es **verdadero** si el valor actual de la variable **value** es del tipo **int**.

De todas maneras, esto último es a modo informativo, ya que actualmente no queremos realizar ninguna validación preliminar de datos, ¿por qué? Porque esta no es la forma que Python recomienda.

#### Python y el manejo de errores

En el mundo de Python hay una regla que dice: **"EAFP, Easier to Ask Forgiveness than Permission”**, o por su traducción al español, más vale pedir perdón que pedir permiso. Aplicado a programación, se interpreta como "es mejor manejar un error cuando ocurre que tratar de evitarlo".

Este es uno de los principio fundamentales de Python, el cual sin duda nos alienta a dejar un poco de lado las validaciones para apoyarnos más de las excepciones, que al final del día, el mantra de Python nos indica que las excepciones no deben pasar desapercibidas (Errors should never pass silently)

Te preguntarás: "´Pero, ¿cómo debo pedir perdón cuando el programa finaliza y no queda nada más por hacer?" Aquí es dónde algo llamado excepción entra en escena.

Veamos este pequeño bloque de pseudocódigo:

try:

# Es un lugar donde

# tu puedes hacer algo

# sin pedir permiso.

except:

# Es un espacio dedicado

# exclusivamente para pedir perdón.

* El primero comienza con la palabra clave reservada **try,** este es el lugar donde se coloca el código que se sospecha que es riesgoso y puede terminar en caso de un error, este tipo de error lleva por nombre **excepción**, mientras que la ocurrencia de la excepción se le denomina **raising** (levantamiento en inglés, pero el término **generar** es más acorde en este contexto); podemos decir que se genera (o **raise**, nuevamente, contexto) una excepción.
* El segundo: la parte del código que comienza con la palabra clave reservada **except,** esta parte fue diseñada para manejar la excepción; depende de nosotros lo que queramos hacer aquí, puedes limpiar el desorden o simplemente puede barrer el problema debajo de la alfombra e ignorar el error (aunque es preferible la primera opción).

Entonces podríamos decir que estos dos bloques funcionan así:

* La palabra clave reservada **try** marca el lugar donde intentas hacer algo sin permiso.
* La palabra clave reservada **except** comienza un lugar donde pedimos perdón. Ahora implementemos esto a nuestro pequeño codigo armado para calcular el recíproco de un número entero:

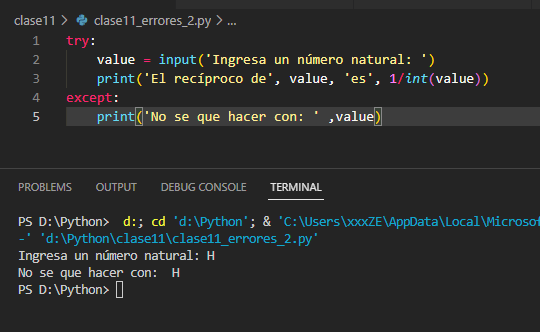
try:

value = input('Ingresa un número natural: ')

print('El recíproco de', value, 'es', 1/int(value))

except:

print('No se que hacer con: ' ,value)



* Cualquier fragmento de código colocado entre **try** y **except** se ejecuta de una manera muy especial. Cualquier error que ocurra aquí dentro **no terminará la ejecución** del programa, en cambio el control **saltará inmediatamente a la primera línea situada después de la palabra clave reservada except**, y no se ejecutará ninguna otra línea del bloque **try**.
* El código en el bloque **except** se activa solo cuando se ha encontrado una excepción dentro del bloque **try**. No hay forma de llegar por ningún otro medio.
* Cuando el bloque **try** o **except** se ejecutan con éxito, el control vuelve al proceso normal de ejecución y cualquier código ubicado más allá en el archivo

fuente se ejecuta normalmente.

¿Es **ValueError** la única forma en que el control podría caer dentro del bloque **except**? Obviamente no, aun hay mas formas en las que que el programa puede generar un error. Imaginemos que el usuario ingresa un 0 como valor de entrada, cualquier división por 0 nos devuelve un error matemático, y en Python este error viene acompañado del siguiente mensaje:

Exception has occurred: ZeroDivisionError

division by zero

Como es de esperarse, el comportamiento del código será el mismo que en el caso anterior: el usuario verá el mensaje **"No se que hacer con: "**, lo que parece bastante razonable en este contexto, pero también es posible que desees manejar este tipo de problema de una manera un poco diferente. Existen dos maneras de hacerlo, la primera de ellas es simple y complicada al mismo tiempo, puedes agregar dos bloques try por separado, uno que incluya la invocación de la función **input()** donde se puede generar la excepción **ValueError**, y el segundo dedicado a manejar posibles problemas inducidos por la división. Ambos bloques **try** tendrían su propio **except** y, de esa manera, tendrías un control total sobre dos errores diferentes. Pero como ya hemos dicho antes, es mejor pedir perdón que pedir permiso. Afortunadamente Python nos ofrece una segunda forma más sencilla de afrontar este tipo de desafíos.

try:

value = input('Ingresa un número natural: ')

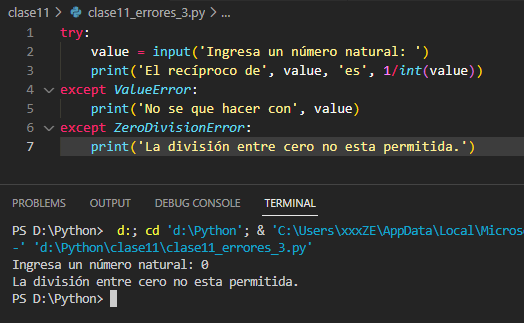
print('El recíproco de', value, 'es', 1/int(value))

except ValueError:

print('No se que hacer con', value)

except ZeroDivisionError:

print('La división entre cero no esta permitida.')



Como puedes ver, agregamos un segundo **except, e**sta no es la única diferencia, toma en cuenta que ambos **except** tienen nombres de **excepción específicos**. En esta variante cada una de las excepciones esperadas tiene su propia forma de manejar el error, pero se debe enfatizar en que solo una de todas puede interceptar el control; **si se ejecuta una, todas las demás permanecen inactivas**. Además, la cantidad de excepciones no está limitada, puedes especificar tantas o tan pocas como necesites, pero no se te olvide que **ninguna de las excepciones se puede especificar más de una vez.**

De momento no podemos imaginar algún otro error posible que pueda generarse, aun así intentar crear una excepción por defecto que maneje los errores que aún no logramos identificar y especificar.

Agregamos un tercer **except**, pero esta vez no tiene un nombre de **excepción específico**; podemos decir que es **anónimo** o (lo que está más cerca de su función real) es por defecto. Cuando se genere una excepción y no haya un **except** dedicado a esa excepción, esta será manejada por la excepción por defecto. Recuerda que el **except por defecto** debe ser el último **except**, siempre.

try:

value = input('Ingresa un número natural: ') print('El recíproco de', value, 'es', 1/int(value))

except ValueError:

print('No se que hacer con', value) except ZeroDivisionError:

print('La división entre cero no está permitida en nuestro Universo.')

except:

print('Ha sucedido algo extraño, ¡lo siento!')

#### Otras excepciones útiles

Analicemos brevemente algunas excepciones útiles (o más bien, las más comunes) que podemos llegar a experimentar:

ZeroDivisionError

Esta aparece cuando intentamos forzar a Python a realizar cualquier operación que provoque una división en la que el divisor es cero o no se puede distinguir de cero. Hay que tener en cuenta que hay más de un operador de Python que puede hacer que se genere esta excepción. Estos son: **/ , //** y %

ValueError

Espera esta excepción cuando estás manejando valores que pueden usarse de manera inapropiada en algún contexto. En general, esta excepción se genera cuando una función (como **int()** o **float()**) recibe un argumento de un tipo adecuado, pero su valor es inaceptable.

TypeError

Esta excepción aparece cuando intentas aplicar un dato cuyo tipo no se puede aceptar en el contexto actual. Por ejemplo, intentar usar un valor **float** como índice de una lista (la misma regla también se aplica a las tuplas). **TypeError** es un nombre adecuado para describir el problema y una excepción adecuada a generar.

AttributeError

Esta excepción llega, entre otras ocasiones, cuando intentas activar un método que no existe en un elemento con el que se está tratando. Por ejemplo intentar usar el metodo **append()** sobre una tupla, que como hemos aprendido, es **inmutable** y no posee este atributo.

SyntaxError

Esta excepción se genera cuando el control llega a una línea de código que viola la gramática de Python. Puede sonar extraño, pero algunos errores de este tipo no se pueden identificar sin ejecutar primero el código, este tipo de comportamiento es típico de los lenguajes interpretados, el intérprete siempre trabaja con prisa y no tiene tiempo para revisar todo el código fuente, y se conforma con comprobar el código que se está ejecutando en el momento.

Como dijimos antes, estas son solo algunas de las excepciones más comunes, para ver la lista completa de excepciones de las librerías estándar de Python, lo puedes hacer con la documentación oficial de Python ingresando al siguiente link: <https://docs.python.org/3/library/exceptions.html>

## Actividades Módulo 11

#### Actividad 1

Escribir una función llamada **realizarTodasLasOp** que reciba como parámetro dos números y retorna una tupla con el resultado de todas las operaciones matemáticas entre estos dos números, el orden esperado sería: suma, resta, multiplicación y división. Ante cualquier error, retorna una tupla vacía.

#### Actividad 2

Escribir una función que reciba como parámetro una string, cuente la cantidad de veces que aparece cada caracter en este texto y retorne un diccionario donde las letras sean la clave y la cantidad de repeticiones sea el valor. No debe distinguir mayúsculas y minúsculas.

#### Actividad 3

Escribir un programa que haga las veces de diccionario, para esto mostraremos un menú al usuario con las opciones:

1. Agregar término
2. Listar
3. Buscar
4. Salir

Luego de realizada la opción requerida, se debe volver a este menú. Se detalla la funcionalidad pretendida para cada opción:

* + **Agregar término:** Solicita al usuario que indique un término y su significado, si dicho término no existe, se lo almacena; de lo contrario, se indica que el término ya existe en nuestro diccionario.
  + **Listar:** Muestra todos los términos junto con sus significados de esta forma:

terminoA: definición terminoB: definición

…

terminoN: definición

* + **Buscar:** Solicita al usuario que indique un término y muestra su significado en

pantalla, si el término no existe, muestra el mensaje “término no encontrado”

#### Actividad 4

Escribir una aplicación de registro de stock, la aplicación debe exhibir un menú con las siguientes opciones:

1. Agregar producto
2. Listar
3. Valorizar stock
4. Aumentar precios
5. Salir

Luego de realizada la opción requerida, se debe volver a este menú. Cada producto debe tener como información:

* + Código interno (un número entero): No puede ser negativo ni 0
  + Marca
  + Nombre
  + Precio unitario: No puede ser negativo ni 0
  + Disponibles: Es la cantidad de productos de este tipo disponibles, solo puede estar entre 0 y 999999

Se detalla la funcionalidad pretendida para cada opción:

1. **Agregar producto**: Debe pedir la información del nuevo producto a agregar al stock y luego agregarlo (validar todos los datos).
2. **Listar:** Muestra todos los productos en stock.
3. **Valorizar Stock:** Debe mostrar cuantos productos tenemos en total y la sumatoria del precio de todos ellos. Esperar a que el usuario presione enter.
4. **Aumentar Precios:** Esto es Argentina, la opción no puede faltar. Pide la inflación mensual en porcentaje y aumentará el precio de todos los productos en stock en ese porcentaje.

Utilice una lista cuyos elementos serán diccionarios, cada diccionario tendrá como claves los nombres de los datos requeridos para cada producto.

## Contenido Módulo 12. Acceso a archivos

Accediendo archivos desde código en Python

Una de las tareas más comunes de un programador es procesar información almacenada en archivos que están normalmente almacenados en dispositivos de almacenamiento como hdd, ssd, ópticos, red, etc.

No es difícil imaginar un programa que ordena 20 números, así como no es para nada difícil imaginar que el usuario de dicho programa ingresa estos 20 números directamente usando el teclado.

Es más difícil imaginar un programa que realice la misma tarea pero con 20000 números a ordenar, y que el usuario no va a cometer ningún error al ingresarlos.

Mucho más fácil de imaginar sería que estos números están almacenados en un archivo que va a ser leído por nuestro programa, el programa ordena los números y no los muestra en pantalla, sino que los escribe en otro archivo de forma ordenada.

Si queremos implementar una base de datos simple, la única forma de almacenar información entre ejecuciones del programa es guardarla en un archivo (o archivos si nuestra base de datos es más compleja).

#### Nombres de archivos

Los diferentes sistemas operativos pueden tratar diferente a los archivos, por ejemplo, Windows usa una convención de nombramiento diferente a la que utilizan los sistemas operativos basados en Unix/Linux.

Si usamos la noción de nombre de archivo canónico (un nombre que define unívocamente la ubicación de un archivo sin importar su nivel en la estructura de carpetas o directorios) podemos ver que estos nombres se ven diferentes en Windows y en Linux:

|  |  |
| --- | --- |
| **Windows** | **Unix/Linux** |
| C:\directorio\archivo | /directorio/archivo |

Como podemos ver, los sistemas basados en Unix/Linux no usan una letra para identificar el disco y todos los directorios parten de un directorio raíz llamado /, mientras windows reconoce el directorio raíz como \.

Adicionalmente, los nombres de archivos en los sistemas Unix/Linux son sensibles a las mayúsculas. En Windows los archivos mantienen el nombre que se les dio pero no se hace distinción entre mayúsculas y minúsculas, con lo cual no podríamos tener un archivo “HOLA.txt” y “hola.txt” en el mismo directorio en Windows pero sí en Unix/Linux.

La diferencia más importante es que **usan dos separadores diferentes para los nombres de los directorios**, Windows usa \ y Unix/Linux usa /. Hoy en día Windows reconoce ambas formas de separador de directorio con lo cual podemos utilizarlos de manera intercambiable, hay que tener en cuenta que no ocurre lo mismo con Unix/Linux.

Por suerte Python es lo suficientemente inteligente como para poder convertir barras en contrabarras cada vez que descubre que el sistema operativo lo requiere.

Cualquier programa escrito en Python (y no solo en Python, porque esa convención se aplica a prácticamente todos los lenguajes de programación) no se comunica con los archivos directamente, sino a través de algunas entidades abstractas que se nombran de manera diferente en los distintos lenguajes o entornos, los términos más utilizados son **handles** (un tipo de puntero inteligente) o **Streams** (un tipo de puntero inteligente). Los usaremos como sinónimos aquí.

El programador que tiene un conjunto de funciones y métodos puede realizar ciertas operaciones en el stream que afectan los archivos reales utilizando mecanismos contenidos en el núcleo del sistema operativo.

De esta forma podemos implementar el proceso de acceso a cualquier archivo, incluso cuando el nombre del archivo es desconocido al momento de escribir el programa.Las operaciones realizadas con el stream abstracto reflejan las actividades relacionadas con el archivo físico.

Para vincular el stream con el archivo es necesario realizar una operación explícita, la operación de conectar un stream con un archivo es llamada **abrir el archivo**, mientras que desconectar el enlace se denomina **cerrar el archivo.**

Por lo tanto, la conclusión es que la primera operación realizada en el stream es siempre **open (abrir)** y la última es **close (cerrar)**. El programa, en efecto, es libre de manipular el stream entre estos dos eventos y manipular el archivo asociado.

Esta libertad está limitada por las características físicas del archivo y la forma en que se abrió.

Digamos nuevamente que la apertura del stream puede fallar, y puede ocurrir debido a varias razones, la más común es la falta de un archivo con un nombre específico, también puede suceder que el archivo físico exista, pero el programa no puede abrirlo, o puede darse el riesgo de que el programa haya abierto demasiados streams y el sistema operativo específico no permita la apertura simultánea de más de 200 archivos, por dar un ejemplo. Un programa bien escrito debe detectar estas aperturas fallidas y reaccionar en consecuencia.

#### Streams

La apertura del stream no solo está asociada con el archivo, sino que también se debe declarar la manera en que se procesará el stream. Esta declaración se llama **open mode** (modo de apertura), si la apertura es exitosa, el programa solo podrá realizar las operaciones que sean consistentes con el modo abierto declarado.

Hay dos operaciones básicas a realizar con el stream:

* **Lectura del stream:** las porciones de los datos se recuperan del archivo y se colocan en un área de memoria administrada por el programa (por ejemplo, una variable).
* **Escritura del stream: l**as porciones de los datos de la memoria (por ejemplo, una variable) se transfieren al archivo.

Hay tres modos básicos utilizados para abrir un stream:

* **Modo Lectura:** un stream abierto en este modo permite solo **operaciones de lectura**; intentar escribir en la transmisión provocará una excepción (la excepción se llama **UnsupportedOperation**).
* **Modo Escritura:** un stream abierto en este modo permite solo operaciones de escritura; intentar leer el stream provocará la excepción mencionada anteriormente.
* **Modo Actualizar:** un stream abierto en este modo permite tanto lectura como escritura.

**El stream se comporta casi como una grabadora**. Cuando leemos algo de un stream un cabezal virtual se mueve sobre la transmisión de acuerdo con el número de bytes transferidos desde el stream.

Cuando escribimos algo en el stream el mismo cabezal se mueve a lo largo del stream registrando los datos de la memoria.



Siempre que hablemos de leer y escribir en el stream, trata de imaginar esta analogía. Los libros de programación se refieren a este mecanismo como la **posición actual del archivo**, aquí también usaremos este término.

#### Streams: cómo maneja los datos

Debido al tipo de contenido de los streams, **se dividen en tipo texto y binario.**

Los streams de texto están estructurados en líneas; es decir, contienen caracteres tipográficos (letras, dígitos, signos de puntuación, etc.) dispuestos en filas (líneas), como se ve a simple vista cuando se mira el contenido del archivo en el editor.

Este tipo de archivo es escrito (o leído) principalmente **carácter por carácter, o línea por línea.**

Los streams binarios no contienen texto, sino una secuencia de bytes de cualquier valor, esta secuencia puede ser, por ejemplo, un programa ejecutable, una imagen, un audio o un videoclip, un archivo de base de datos, etc.

Debido a que estos archivos no contienen líneas, las lecturas y escrituras se relacionan con porciones de datos de cualquier tamaño, por lo tanto, los datos se leen y escriben **byte a byte**, **o bloque a bloque**, donde el tamaño del bloque generalmente varía de uno a un valor elegido arbitrariamente.

En esta oportunidad solo nos abocaremos a trabajar únicamente con operaciones simples sobre archivos de texto plano.

#### Como abrir Streams

Si queremos abrir un stream esto se realiza mediante la función open() que tiene la siguiente sintaxis:

stream = open(file, mode = 'r', encoding = None)

* El nombre de la función **open()**: si la apertura es exitosa, la función devuelve un objeto stream; de lo contrario, se genera una excepción (por ejemplo, FileNotFoundError si el archivo que vas a leer no existe).
* El primer parámetro de la función **(file)** especifica el nombre del archivo que se asociará al stream.
* El segundo parámetro **(mode)** especifica el modo de apertura utilizado para el stream; es una string llena de una secuencia de caracteres, y cada uno de ellos tiene su propio significado especial, en este caso **‘r’** indica que es un stream de lectura. **Nota:** en caso de no ser especificado, el valor por defecto es **‘r’**
* El tercer parámetro (**encoding**) especifica el tipo de codificación (por ejemplo,

**UTF-8** cuando se trabaja con archivos de texto).

#### Modos de apertura de Streams

Lectura: ‘r’

* El stream será abierto en **modo lectura.**
* El archivo asociado con el stream debe existir y tiene que ser legible, de lo contrario la función open() generará una excepción.

Escritura: ‘w’

* El stream será abierto en **modo escritura.**
* El archivo asociado con el stream no necesita existir, si no existe, se creará; si existe, se truncará a la longitud de cero (se borra); si la creación no es posible (por ejemplo, debido a permisos del sistema) la función open() generará una excepción.

Adjuntar / Append: ‘a’

* El stream será abierto en **modo adjuntar.**
* El archivo asociado con el stream no necesita existir; si no existe, se creará; si existe, el cabezal de grabación virtual se establecerá al final del archivo (el contenido anterior del archivo permanece intacto).

Lectura y Actualizacion: ‘r+’

* El stream será abierto en **modo lectura y actualización.**
* El archivo asociado con el stream debe existir y tiene que permitir escritura, de lo contrario la función open() generará una excepción.
* Se permiten operaciones de lectura y escritura en el stream.

Escritura y Actualizacion: ‘w+’

* El stream será abierto en **modo escritura y actualización.**
* El archivo asociado con el stream no necesita existir; si no existe, se creará; el contenido anterior del archivo permanece intacto.
* Se permiten operaciones de lectura y escritura en el stream.

#### Especificando al stream para manejar el archivo

Como hemos mencionado antes, según el contenido, los steams **se dividen tanto en tipo binario y tipo texto.** Al momento de abrir un nuevo streams, también podemos forzar el modo en el que queremos que se abra el stream.

Si hay una letra **‘b’** al final de la string del modo significa que el stream se debe abrir en el modo binario.

Si la string del modo termina con una letra **‘t’** el stream es abierto en modo texto. En caso de que no se especifique, este será el modo por defecto.

Finalmente, la apertura exitosa del archivo establecerá la posición actual del archivo (el cabezal virtual de lectura/escritura) antes del primer byte del archivo **si el modo no es ‘a’** y después del último byte del archivo **si el modo es ‘a’.**

|  |  |
| --- | --- |
| **Modo texto** | **Modo** |
| rt | rb |
| wt | wb |
| at | ab |
| r+t | r+b |
| w+t | w+b |

Nota: También podemos abrir un archivo para su creación exclusiva, hacemos esto usando

el modo de apertura ‘x’, si el archivo ya existe, la función open() generará una excepción.

#### Abriendo nuestro primer stream

Supongamos que tenemos que desarrollar un programa que lea el contenido del archivo de texto llamado:

D:\Python\Archivos\textfile.txt

Nuestro código se vería así:

try:

stream = open("D:/Python/Archivos/textfile.txt", "rt")

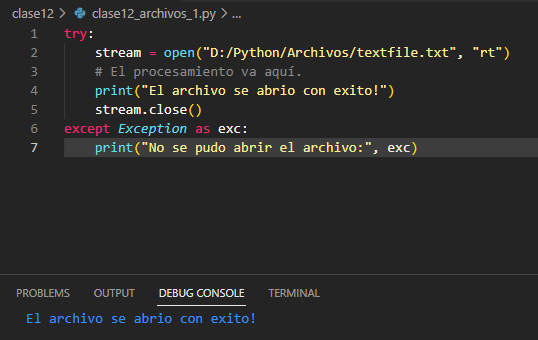
# El procesamiento va aquí.

print("El archivo se abrio con exito!")

stream.close()

except Exception as exc:

print("No se pudo abrir el archivo:", exc)



* Hemos abierto el bloque **try-except** ya que queremos manejar los errores de tiempo de ejecución.
* Se emplea la función **open()** para intentar abrir el archivo especificado (ten en cuenta la forma en que hemos especificado el nombre del archivo).
* El modo de apertura se define como **lectura de texto** (como **texto** es la configuración predeterminada, no es necesario la **‘t’** en el parámetro de modo).
* En caso de éxito obtenemos un objeto de la función **open()** y lo asignamos a la variable del stream, agregamos un print para confirmar que el código se ejecutó.
* Si **open()** falla, manejamos la excepción imprimiendo la información completa del error (es bueno saber qué sucedió exactamente).

#### Cerrando nuestro primer stream

La última operación realizada en un stream debe ser cerrarlo, esa acción se realiza mediante un método invocado desde dentro del objeto del stream: **stream.close().**

* El nombre de la función es fácil de entender **close()**, es decir cerrar.
* La función no espera argumentos; el stream no necesita estar abierto.
* La función no devuelve nada pero genera una excepción IOError en caso de un error.
* La mayoría de los desarrolladores creen que la función close() siempre tiene éxito y, por lo tanto, no hay necesidad de verificar si ha realizado su tarea correctamente, esta creencia está sólo parcialmente justificada. Si el stream se abrió para escribir y luego se realizó una serie de operaciones de escritura, puede ocurrir que los datos enviados al stream aún no se hayan transferido al dispositivo físico (debido a los mecanismos de caché o buffer), dado que el cierre del stream obliga a los buffers a descargarse, es posible que dichas descargas fallen y, por lo tanto, close() falle también.

#### Diagnosticando problemas con los streams

El objeto IOError está equipado con una propiedad llamada errno (el nombre viene de la frase error number, número de error) y puedes accederla de la siguiente manera:

try:

# Algunas operaciones con streams.

except IOError as exc:

print(exc.errno)

Veamos alguna de **las constantes más útiles para detectar errores en los streams:**

errno.EACCES → Permiso denegado

El error se produce cuando, por ejemplo, intentas abrir un archivo en modo escritura, pero este archivo posee atributos de solo lectura.

errno.EBADF → Número de archivo incorrecto

El error se produce cuando intentas, por ejemplo, operar un stream sin abrirlo.

errno.EEXIST → Archivo existente

El error se produce cuando intentas, por ejemplo, cambiar el nombre de un archivo con su nombre anterior.

errno.EFBIG → Archivo demasiado grande

El error ocurre cuando intentas crear un archivo que es más grande que el máximo permitido por el sistema operativo.

errno.EISDIR → Es un directorio

El error se produce cuando intentas tratar un nombre de directorio como el nombre de un archivo ordinario.

errno.EMFILE → Demasiados archivos abiertos

El error se produce cuando intentas abrir simultáneamente más streams de los aceptables para el sistema operativo.

errno.ENOENT → El archivo o directorio no existe

El error se produce cuando intentas acceder a un archivo o directorio inexistente.

errno.ENOSPC → No queda espacio en el dispositivo

El error ocurre cuando no hay espacio libre en el dispositivo.

Nuevamente, estos son solo algunos casos seleccionados para este curso, para poder ver la lista completa de constantes, podemos acceder a la documentación oficial de Python:

<https://docs.python.org/3/library/errno.html>

Ahora supongamos que queremos ser más cuidadosos y claros a la hora de manejar errores en streams, seguramente estemos pensando en implementar algo como esto:

import errno

try:

stream = open("D:/Python/Archivos/textfile.txt", "rt")

# El procesamiento va aquí.

print("El archivo se abrio con exito!")

stream.close()

except Exception as exc:

if exc.errno == errno.ENOENT:

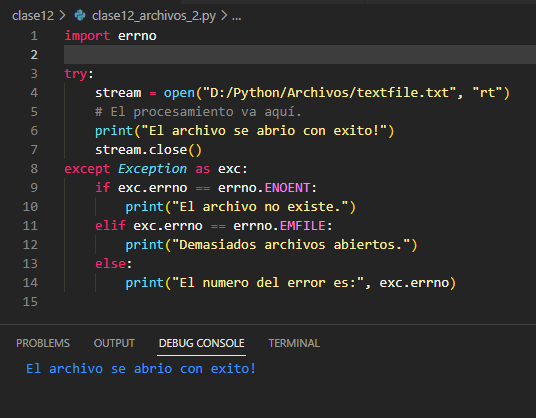
print("El archivo no existe.")

elif exc.errno == errno.EMFILE:

print("Demasiados archivos abiertos.")

else:

print("El numero del error es:", exc.errno)



Afortunadamente Python cuenta con una función que **simplificará el código de manejo de errores.** Su nombre es **strerror()**, y proviene del módulo **os** y espera solo un argumento: un número de error.

Su función es simple: proporciona un número de error y una string que describe el significado del error.

**Nota:** Si pasamos un código de error inexistente (un número que no está vinculado a ningún error real), la función generará una excepción ValueError.

Veamos cómo funciona modificando el nombre del archivo a abrir, para forzar un error:

import errno

from os import strerror

try:

stream = open("D:/AAAAAAAAAAAAAAAAAAAAAAAA.txt", "rt")

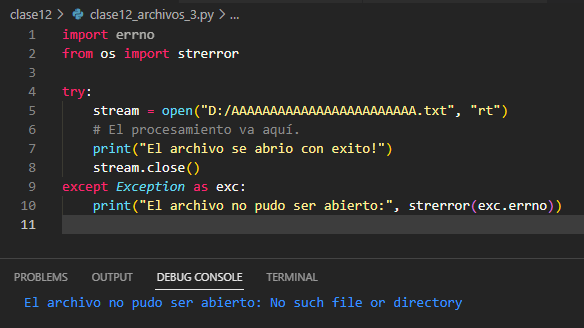
# El procesamiento va aquí.

print("El archivo se abrio con exito!")

stream.close()

except Exception as exc:

print("El archivo no pudo ser abierto:", strerror(exc.errno))



#### Procesamiento de archivos de texto

Ahora veremos algunas técnicas básicas que podemos utilizar para leer el contenido del archivo y procesarlo.

El procesamiento será muy simple: vamos a copiar el contenido del archivo a la consola y contaremos todos los caracteres que el programa ha leído.

Pero, recordemos, nuestra comprensión de un archivo de texto es muy estricta, es un archivo de texto sin formato, puede contener solo texto, sin decoraciones adicionales, como formato, diferentes fuentes, alineaciones, etc., es por eso que debemos evitar crear el archivo utilizando un procesador de texto avanzado como MS Word, LibreOffice Writer o algo así. Utilizaremos los conceptos básicos que ofrece nuestro sistema operativo (Bloc de notas, vim, gedit, etc.).

Si nuestros archivos de texto contienen algunos caracteres nacionales no cubiertos por el charset ASCII estándar, es posible que necesitemos un paso adicional. La invocación de **open()** va requerir un argumento que denote una codificación específica del texto.

stream = open('file.txt', 'rt', encoding='utf-8')

La lectura del contenido de un archivo de texto se puede realizar utilizando diferentes métodos. Veamos un ejemplo:

from os import strerror

try:

counter = 0

stream = open("D:/Python/Archivos/textfile.txt", "rt", encoding="utf-8")

char = stream.read(1)

while char != '':

print(char, end='')

counter += 1

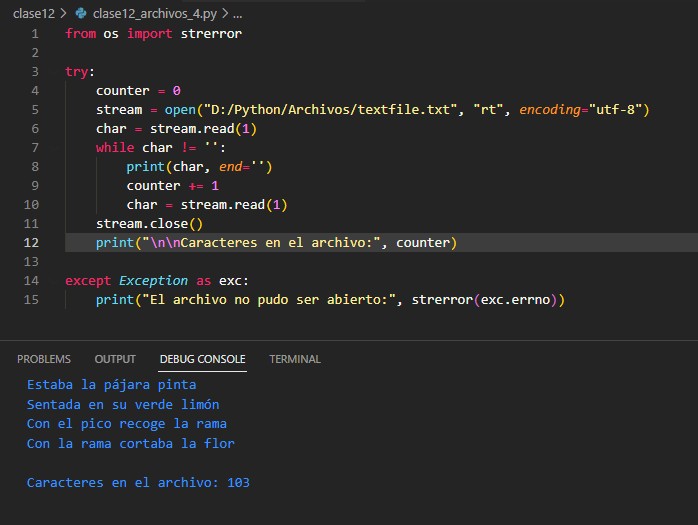
char = stream.read(1)

stream.close()

print("\n\nCaracteres en el archivo:", counter)

except Exception as exc:

print("El archivo no pudo ser abierto:", strerror(exc.errno))



Analicemos el código:

* Se usa el mecanismo **try-except** y se abre el archivo con el nombre (text.txt en este caso).
* Nota como la función **open(),** tiene especificado el tipo de codificación utf-8, esto se debe a que nuestro archivo contiene caracteres con tildes
* Intenta leer el primer carácter del archivo (**char = stream.read(1)**).
* Si tiene éxito (esto se demuestra por el resultado positivo de la condición while), se muestra el carácter. Nota: el argumento end=, ¡es importante! ¡No querrás saltar a una nueva línea después de cada carácter!.
* También se actualiza el contador (counter),
* Intenta leer el siguiente carácter y el proceso se repite.

Si quisiéramos manejar el contenido del archivo como un **conjunto de líneas**, no como un montón de caracteres, el método readline() es el que tenemos que usar.

El método intenta **leer una línea completa de texto del archivo** y la devuelve como una string en caso de éxito. De lo contrario, devuelve una string vacía.

from os import strerror

try:

character\_counter = line\_counter = 0

stream = open("D:/Python/Archivos/textfile.txt", "rt", encoding="utf-8")

line = stream.readline()

while line != '':

line\_counter += 1

for char in line:

print(char, end='')

character\_counter += 1

line = stream.readline()

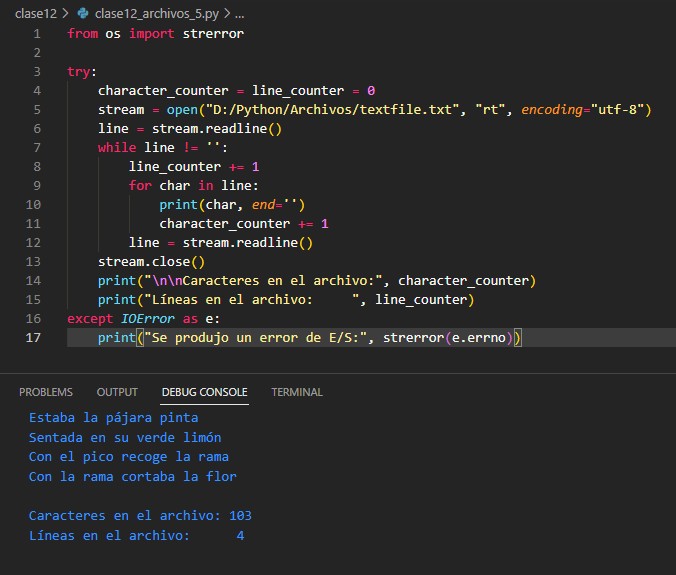
stream.close()

print("\n\nCaracteres en el archivo:", character\_counter)

print("Líneas en el archivo: ", line\_counter)

except IOError as e:

print("Se produjo un error de E/S:", strerror(e.errno))



Escribir archivos es más simple, de hecho, hay un solo método que podemos usar para esta tarea: el nombre del método es write() y espera un único argumento, una string que será transferida al archivo (no olvidemos que hay que abrir el archivo en modo escritura, de lo contrario el uso de write() va a fallar).

Debemos tener en cuenta que no se agregarán caracteres de fin de línea al final, con lo cual tendremos que hacerlo nosotros mismos si es que lo necesitamos. Escribamos un archivo en un breve ejemplo:

from os import strerror

try:

stream = open("D:/Python/Archivos/textfile.txt", "wt") for i in range(10):

s = "linea nro " + str(i+1) + "\n" stream.write(s)

stream.close() print("Archivo escrito")

except IOError as e:

print("Se produjo un error de E/S:", strerror(e.errno))

De esta manera, logramos escribir un archivo de texto que contiene:

linea nro 1

linea nro 2

linea nro 3

linea nro 4

linea nro 5

linea nro 6

linea nro 7

linea nro 8

linea nro 9

linea nro 10

#### El módulo OS

En esta sección aprenderemos sobre un módulo llamado **os**, que nos permite interactuar con el sistema operativo usando Python.

Proporciona funciones que están disponibles en sistemas **Unix y/o Windows**. Si estamos familiarizados con la consola de comandos, veremos que algunas funciones dan los mismos resultados que los comandos disponibles en los sistemas operativos. Un buen ejemplo de esto es la función **mkdir**, que te permite crear un directorio como el comando **mkdir** en **Unix y Windows**.

Aprenderemos las funciones del módulo **os**, para realizar operaciones en **archivos** y

**directorios** junto con los comandos correspondientes.

Además de las operaciones de archivos y directorios, el módulo **os** permite:

* Obtener información sobre el sistema operativo.
* Manejar procesos.
* Operar en streams de Entrada/Salida usando descriptores de archivos.

La administración de procesos y el trabajo con file descriptors no será tratado, porque estos son temas más avanzados que requieren conocimiento de los mecanismos del sistema operativo.

#### Obtener información sobre el sistema operativo

Antes de crear nuestra primera estructura de directorios, veamos cómo obtener información sobre el sistema operativo actual. El módulo os proporciona una función llamada uname, que devuelve un objeto que contiene los siguientes atributos:

* **systemname:** almacena el nombre del sistema operativo.
* **nodename:** almacena el nombre de la máquina en la red.
* **release:** almacena el release (actualización) del sistema operativo.
* **version:** almacena la versión del sistema operativo.
* **machine:** almacena el identificador de hardware, por ejemplo, x86\_64.

Llevado a la practica se vería así

import os

print(os.uname())Resultado:

posix.uname\_result(sysname='Linux', nodename='192d19f04766', release='4.4.0-164-generic', version='#192-Ubuntu SMP Fri Sep 13 12:02:50 UTC 2019', machine='x86\_64')

La función **uname** devuelve un objeto que contiene información sobre el sistema operativo. El código anterior se ejecutó en **Ubuntu 16.04.6 LTS**. Desafortunadamente, la función uname solo funciona en algunos sistemas Unix. Si usamos **Windows**, podemos usar la función **uname** en el módulo **platform**, que devuelve un resultado similar.

El módulo **os** permite distinguir rápidamente el sistema operativo mediante el atributo

**name**, que soporta uno de los siguientes nombres:

* **posix:** obtendremos este nombre si usamos Unix.
* **nt:** obtendremos este nombre si usamos Windows.
* **java:** obtendremos este nombre si nuestro código está escrito en Jython.

#### Creando directorios en Python

El módulo **os** proporciona una función llamada **mkdir**, la cual, como el comando **mkdir** en **Unix** y **Windows**, nos permite crear un directorio. La función **mkdir** requiere una ruta que puede ser relativa o absoluta. Recordemos cómo se ven ambas rutas en la práctica:

* **my\_first\_directory:** esta es una ruta relativa que creará el directorio my\_first\_directory en el directorio de trabajo actual.
* **./my\_first\_directory:** esta es una ruta relativa que apunta explícitamente al directorio de trabajo actual, tiene el mismo efecto que la ruta anterior.
* **../my\_first\_directory:** esta es una ruta relativa que creará el directorio my\_first\_directory en el directorio superior del directorio de trabajo actual.
* **/python/my\_first\_directory:** esta es una ruta absoluta que creará el directorio my\_first\_directory, que a su vez está en el directorio raíz de python.

La función **mkdir** puede tomar opcionalmente el argumento **mode**, que especifica los permisos del directorio, sin embargo, en algunos sistemas, el argumento **mode** se ignora.

También contamos con unafunción llamada **listdir**. La función **listdir** devuelve una lista que contiene los nombres de los archivos y directorios que se encuentran en la ruta pasada como argumento, si no se le pasa ningún argumento, se utilizará el directorio de trabajo actual.

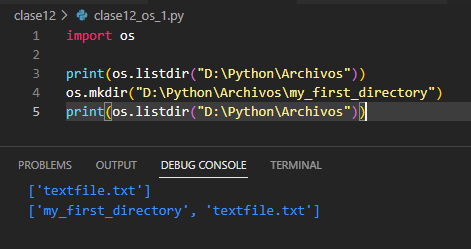
Veamos ambas funciones en un ejemplo:

import os

print(os.listdir("D:\Python\Archivos"))

os.mkdir("D:\Python\Archivos\my\_first\_directory")

print(os.listdir("D:\Python\Archivos"))



Nota: Tengamos en cuenta que ejecutar el programa ya que no es posible crear un directorio que ya existe

dos veces generará un FileExistsError,

¿Qué sucede si necesitamos crear otro **directorio dentro del directorio que acabamos de crear**? Por supuesto, podemos ir al directorio creado y crear otro directorio dentro de él, pero afortunadamente el módulo os proporciona una función llamada **makedirs**, que facilita esta tarea. La función makedirs permite la creación recursiva de directorios, lo que significa que se crearán todos los directorios de la ruta.

Para movernos entre directorios podemos usar una función llamada **chdir**, que cambia el directorio de trabajo actual a la ruta especificada. Como argumento toma cualquier ruta relativa o absoluta. Para nuestro siguiente ejemplo lo usaremos para mostrar la creación de los nuevos directorios. Por otra parte, si en lugar de necesitar movernos entre directorios necesitamos saber nuestro directorio actual de trabajo, podemos usar la función **getcwd,** la cual no llega argumentos, y nos devolverá como resultado el directorio de trabajo actual.

Por último, ¿qué sucede cuando queremos **borrar** un directorio? Python nos provee de otras 2 funciones para esto: **rmdir** y **removedirs.** La primera función **rmdir**, tomará la ruta como argumento y eliminará el directorio indicado, la segunda función **removedirs** , funciona de manera similar, tomará la ruta como argumento y eliminará el directorio y todos sus subdirectorios que estén vacíos.

Nota: para las dos funciones, en caso de que alguno de los directorios no exista o no esté vacío, generará una excepción.

Veamos estas nuevas funciones en acción:

import os

print("path actual:",os.getcwd())

print("antes de crear:", os.listdir("D:\Python\Archivos"))

# creamos una los nuevos directorios

os.makedirs("D:\Python\Archivos\my\_first\_directory\carpeta1")

os.mkdir("D:\Python\Archivos\my\_first\_directory\carpeta2")

print("luego de crear:",os.listdir("D:\Python\Archivos"))

# cambiamos el directorio de trabajo al primero que se creó

os.chdir("D:\Python\Archivos\my\_first\_directory")

print("subcarpetas",os.listdir())

# borramos lo que se creo para el ejemplo

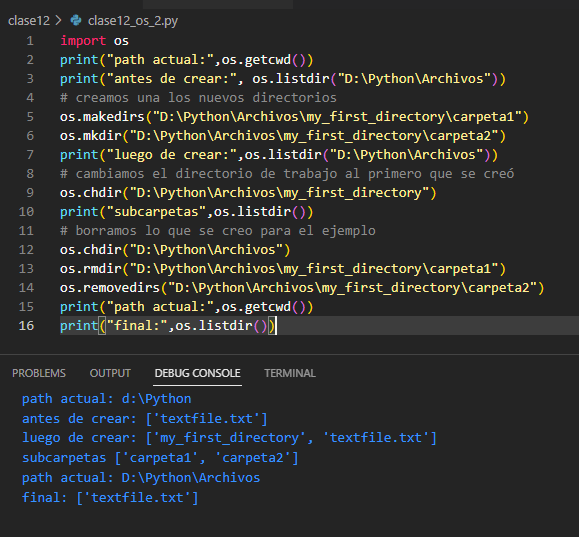
os.chdir("D:\Python\Archivos")

os.rmdir("D:\Python\Archivos\my\_first\_directory\carpeta1")

os.removedirs("D:\Python\Archivos\my\_first\_directory\carpeta2")

print("path actual:",os.getcwd())

print("final:",os.listdir())



#### Modulo datetime

Para finalizar, aprenderemos sobre el módulo de python llamado **datetime.**

Este módulo nos proporciona clases para trabajar con la **fecha y hora**. La fecha y la hora tienen innumerables usos y probablemente sea difícil encontrar una aplicación de producción que no los utilice. A continuación, se muestran algunos ejemplos:

* **Registro de eventos:** gracias al conocimiento de la fecha y la hora, podemos determinar cuándo ocurre exactamente un error crítico en nuestra aplicación. Al crear registros, podemos especificar el formato de fecha y hora.
* **Seguimiento de cambios en la base de datos:** a veces es necesario almacenar información sobre cuándo se creó o modificó un registro, el módulo datetime será perfecto para este caso.
* **Validación de datos:** podemos leer la fecha y hora actuales en Python. Conociendo la fecha y hora actuales, podremos validar varios tipos de datos, por ejemplo, si un cupón de descuento ingresado por un usuario en nuestra aplicación sigue siendo válido.
* **Almacenamiento de información importante: l**a fecha y la hora de ciertas acciones deben conservarse y debemos ocuparnos de ello.

Ahora que conocemos la importancia de este módulo, veamos qué herramientas nos brinda.

#### Obtener la fecha local actual y crear objetos del tipo fecha

Una de las clases proporcionadas por el módulo **datetime** es una clase llamada **date**. Los objetos de esta clase representan una fecha que consta de **año**, **mes** y **día**. Por defecto se muestra en ese orden, más adelante veremos cómo darle un formato que nos sea más útil o apropiado.

El método **today** devuelve un objeto del tipo **date** que representa la fecha local actual. Toma en cuenta que el objeto date tiene tres atributos: año, mes y día.

Ten cuidado, porque estos atributos son de solo lectura. Para crear un objeto date, debes pasar los parámetros año, mes y día de la siguiente manera:

from datetime import date my\_date = date(2019, 11, 4)

print(my\_date)

Al crear un objeto **date**, toma en cuenta las siguientes restricciones:

* **Año:** El parámetro **año** debe ser mayor o igual a 1 (constante **MINYEAR**) y menor o igual a 9999 (constante **MAXYEAR**).
* **Mes:** El parámetro **mes** debe ser mayor o igual a 1 y menor o igual a 12.
* **Dia:** El parámetro **día** debe ser mayor o igual a 1 y menor o igual que el último día del mes y año indicados.

No respetar estas restricciones hará que la creación de nuestro nuevo objeto produzca una excepción.

Veamos esto en un ejemplo:

from datetime import date

today = date.today()

micumple = date(1990,2,14)

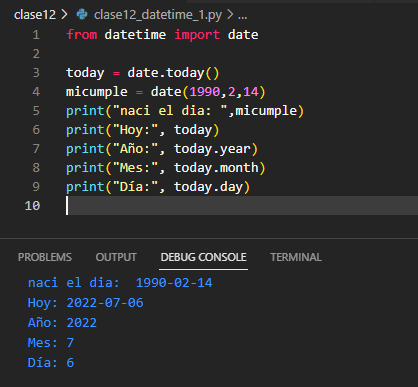
print("naci el dia: ",micumple)

print("Hoy:", today)

print("Año:", today.year)

print("Mes:", today.month)

print("Día:", today.day)



#### Creando un objeto de fecha usando el formato ISO

El módulo **datetime** proporciona varios métodos para crear un objeto **date, u**no de ellos es el método **fromisoformat**, que toma una fecha en el formato **YYYY-MM-DD** (correspondientes a Year, month y day), compatible con el estándar ISO 8601.

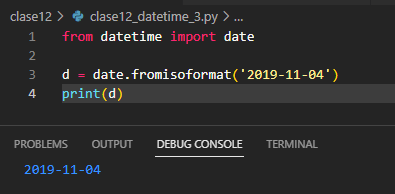
El estándar ISO 8601 define cómo se representan la fecha y la hora, se usa a menudo, por lo que vale la pena tomarse un momento para familiarizarse con él. Echa un vistazo a la imagen que describe los valores requeridos por el formato:

Veamos un ejemplo:

from datetime import date

d = date.fromisoformat('2019-11-04')

print(d)



#### Método replace()

A veces es posible que debas reemplazar el año, el mes o el día con un valor diferente, no puedes hacer esto con los atributos de año, mes y día porque son de solo lectura, en este caso, puedes utilizar el método llamado **replace**.

Este método lleva la siguiente sintaxis:

d = d.replace(year=YYYY, month=MM, day=DD)

Solo es necesario ingresar un parámetro, los otros 2 son opcionales, permitiéndonos invocar este método al indicar los valores que deseamos modificar.

Veamos un ejemplo:

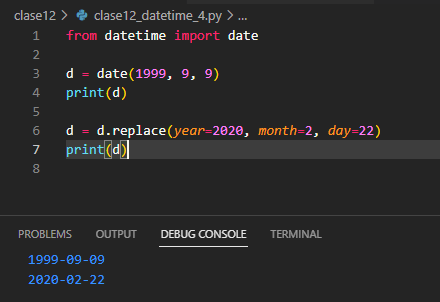
from datetime import date

d = date(1999, 9, 9)

print(d)

d = d.replace(year=2020, month=2, day=22)

print(d)



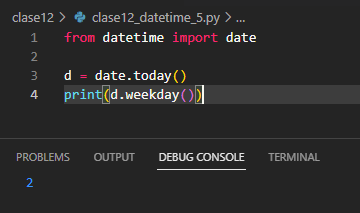
#### Metodo weekday()

Uno de los métodos más útiles que facilita el trabajo con fechas es el método llamado **weekday**. Devuelve el día de la semana como un número entero, donde 0 es el Lunes y 6 es el Domingo.

from datetime import date

d = date.today()

print(d.weekday())



#### Creando objetos time

Ya vimos cómo presentar una **fecha** utilizando el objeto **date**. El módulo **datetime** también tiene una clase que te permite presentar la hora, llamada **time**, esta es su sintaxis:

time(hour, minute, second, microsecond, tzinfo, fold)

El constructor de la clase **time** acepta los siguientes parámetros opcionales:

* **Hour:** El parámetro hour debe ser mayor o igual que 0 y menor que 23.
* **Minute:** El parámetro minute debe ser mayor o igual que 0 y menor que 59.
* **Second:** El parámetro second debe ser mayor o igual que 0 y menor que 59.
* **Microsecond:** El parámetro microsecond debe ser mayor o igual que 0 y menor que 1000000.
* **Tzinfo:** El parámetro tzinfo debe ser un objeto de la subclase tzinfo o None (por defecto).
* **Fold:** El parámetro fold debe ser 0 o 1, su valor por defecto es 0

El parámetro **tzinfo** está asociado con las zonas horarias, mientras que **fold** está asociado con el tiempo de pared, no los usaremos durante este curso, pero es bueno saber que existen.

Veamos cómo crear un objeto de tiempo con un ejemplo:

from datetime import time

t = time(14, 53, 20, 1)

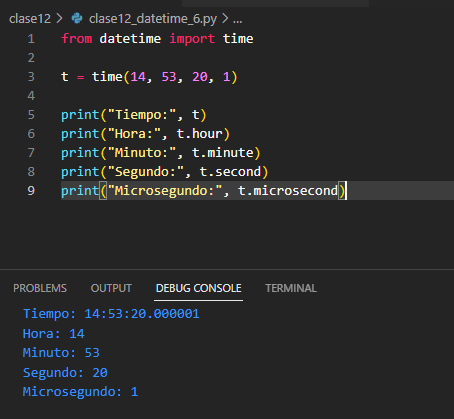
print("Tiempo:", t)

print("Hora:", t.hour)

print("Minuto:", t.minute)

print("Segundo:", t.second)

print("Microsegundo:", t.microsecond)



#### El módulo time

Además de la clase **time**, la biblioteca estándar de Python ofrece un módulo llamado **time**, que proporciona una función relacionada con el tiempo. Ya se tuvo la oportunidad de aprender la función llamada **time** cuando se habló de la clase **date, a**hora veamos otra función interesante de este módulo, la función **sleep,** la cual suspende la ejecución del programa por la cantidad de segundos que se le haya indicado como parámetro.

Veamos un ejemplo:

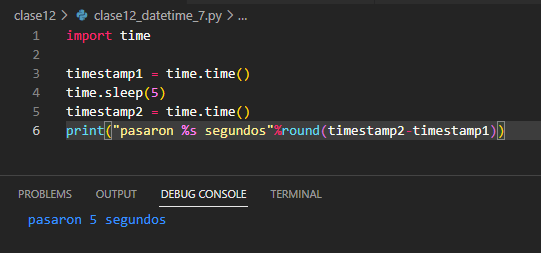
import time

timestamp1 = time.time()

time.sleep(5)

timestamp2 = time.time()

print("pasaron %s segundos"%round(timestamp2-timestamp1))



#### Creación de objetos datetime

En el módulo **datetime** la fecha y la hora se pueden representar como objetos separados o como un solo objeto. Para el módulo **time, l**a clase que combina fecha y hora se llama **datetime**.

datetime(year, month, day, hour, minute, second, microsecond, tzinfo, fold)

El constructor de la clase **datetime** acepta los siguientes parámetros:

* **Year:** El parámetro year debe ser mayor o igual a 1 (constante MINYEAR) y menor o igual a 9999 (constante MAXYEAR).
* **Month:** El parámetro month debe ser mayor o igual a 1 y menor o igual a 12.
* **Day:** El parámetro day debe ser mayor o igual a 1 y menor o igual al último día del mes y año indicados.
* **Hour:** El parámetro hour debe ser mayor o igual que 0 y menor que 23.
* **Minute:** El parámetro minute debe ser mayor o igual que 0 y menor que 59.
* **Second:** El parámetro second debe ser mayor o igual que 0 y menor que 59.
* **Microsecond:** El parámetro microsecond debe ser mayor o igual que 0 y menor que 1000000.
* **Tzinfo:** El parámetro tzinfo debe ser un objeto de la subclase tzinfo o None (por defecto).
* **Fold:** El parámetro fold debe ser 0 o 1, su valor por defecto es 0

Veamos un ejemplo:

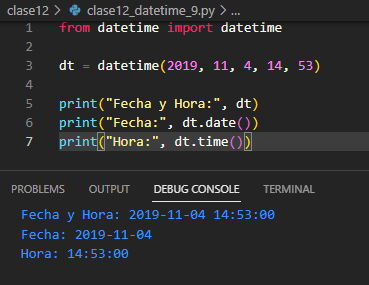
from datetime import datetime

dt = datetime(2019, 11, 4, 14, 53)

print("Fecha y Hora:", dt)

print("Fecha:", dt.date())

print("Hora:", dt.time())



El ejemplo crea un objeto **datetime** que representa el 4 de noviembre de 2019 a las 14:53:00. Todos los parámetros pasados al constructor van a atributos de clase de solo lectura. Son: year, month, day, hour, minute, second, microsecond, tzinfo, y fold.

El ejemplo muestra dos métodos que devuelven dos objetos diferentes. El método llamado **date** devuelve el objeto date con el año, mes y día dados, mientras que el método llamado **time** devuelve el objeto time con la hora y minuto dados.

#### ¿Qué hora es?

La clase **datetime** tiene varios métodos que devuelven la fecha y hora actuales. Estos métodos son:

* **today():** devuelve la fecha y hora local actual con el atributo tzinfo establecido a None.
* **now()**: devuelve la fecha y hora local actual igual que el método today, a menos que le pasemos el argumento opcional tz. El argumento de este método debe ser un objeto de la subclase tzinfo.
* **utcnow():** devuelve la fecha y hora UTC actual con el atributo tzinfo establecido a None.

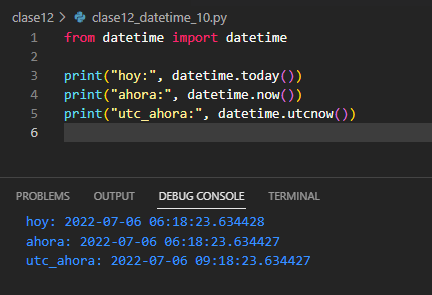
Veamos un ejemplo con los 3 en funcionamiento:

from datetime import datetime

print("hoy:", datetime.today())

print("ahora:", datetime.now())

print("utc\_ahora:", datetime.utcnow())



Como podemos ver, el resultado de los tres métodos es el mismo, las pequeñas diferencias se deben al tiempo transcurrido entre llamadas posteriores.

Nota: Para más información sobre los objetos tzinfo remitirse a la documentación.

#### Formato de fecha y hora

Todas las clases del módulo **datetime** presentadas hasta ahora tienen un método llamado **strftime**. Este es un método muy importante, porque nos permite devolver la fecha y la hora en el formato que especifiquemos.

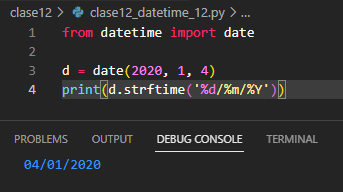
El método **strftime** toma solo un argumento en forma de string que especifica un formato que puede constar de directivas.

Una directiva es una string que consta del carácter % (porcentaje) y una letra minúscula o mayúscula, por ejemplo, la directiva %Y significa el año con el siglo como número decimal. Veamos en un ejemplo:

from datetime import date

d = date(2020, 1, 4)

print(d.strftime('%d/%m/%Y'))



Hemos pasado un formato que consta de tres **directivas** separadas por **/** (diagonal) al método **strftime, p**or supuesto, el carácter separador se puede reemplazar por otro carácter, o incluso por un string.

Podemos poner cualquier carácter en el formato, pero solo las directivas reconocibles se reemplazarán con los valores apropiados, para este ejemplo usamos las siguientes directivas:

* **%Y:** devuelve el año con el siglo como número decimal. En nuestro ejemplo, esto es 2020.
* **%m:** devuelve el mes como un número decimal con relleno de ceros. En nuestro ejemplo, es 01.
* **%d:** devuelve el día como un número decimal con relleno de ceros. En nuestro ejemplo, es 04.

El formato de hora funciona de la misma forma que el formato de fecha, pero requiere el uso de directivas adecuadas. Veamos el siguiente ejemplo:

from datetime import time

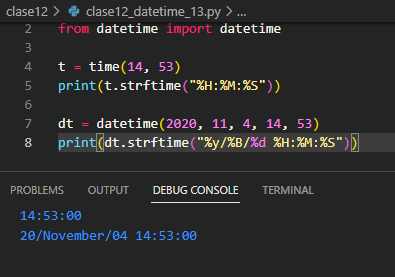
from datetime import datetime

t = time(14, 53)

print(t.strftime("%H:%M:%S"))

dt = datetime(2020, 11, 4, 14, 53)

print(dt.strftime("%y/%B/%d %H:%M:%S"))



El primero de los formatos utilizados se refiere solo al tiempo. Como puedes adivinar,

**%H** devuelve la **hora** como un número decimal con relleno de ceros, **%M** devuelve el **minuto** como un número decimal con relleno de ceros, mientras que **%S** devuelve el **segundo** como un número decimal con relleno de ceros.

El segundo formato utilizado combina directivas de **fecha** y **hora**. Hay dos nuevas directivas, **%y** y **%B**. La directiva **%y** devuelve el **año sin siglo como un número decimal con relleno de ceros,** mientras que la directiva **%B** devuelve el mes como el nombre completo.

La lista completa de directivas se puede encontrar en: [https://docs.python.org/es/3/library/datetime.html#strftime-and-strptime-format-codes](https://docs.python.org/es/3/library/datetime.html" \l "strftime-and-strptime-format-codes)

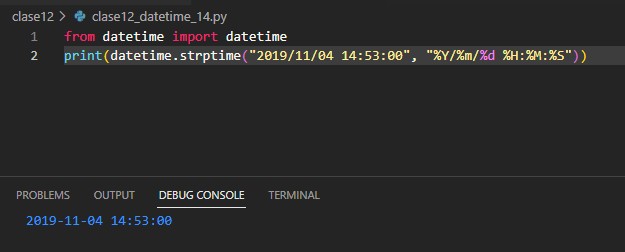
#### Metodo strptime()

Saber cómo crear un formato puede ser útil cuando se usa un método llamado strptime en la clase datetime. A diferencia del método strftime, crea un objeto datetime a partir de una string que representa una fecha y una hora.

El método strptime requiere especificar el formato en el que guardamos la fecha y la hora. Ejemplo:

from datetime import datetime

print(datetime.strptime("2019/11/04 14:53:00", "%Y/%m/%d %H:%M:%S"))



En el ejemplo hemos especificado dos argumentos obligatorios: el primero es una fecha y hora como una string: "**2019/11/04 14:53:00**", mientras que el segundo es un formato que facilita el análisis a un objeto datetime. Si el formato que se especifica no coincide con la fecha y la hora en la string, generará una excepción **ValueError**.

#### Operaciones con fecha y hora: timedelta

Tarde o temprano tendremos que realizar algunos cálculos sobre la fecha y la hora. Afortunadamente eso se trabaja en el tema **timedelta** del módulo datetime que se creó con tal propósito.

Para crear un objeto **timedelta**, simplemente realizamos una resta en los objetos date o datetime:

from datetime import date

from datetime import datetime

d1 = date(2020, 11, 4)

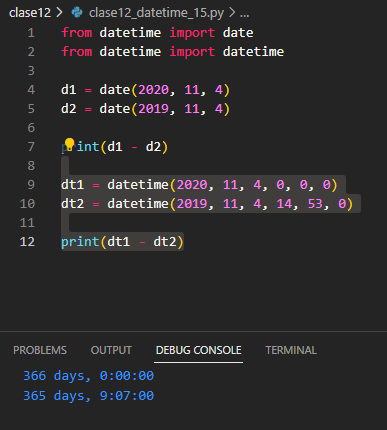
d2 = date(2019, 11, 4)

print(d1 - d2)

dt1 = datetime(2020, 11, 4, 0, 0, 0)

dt2 = datetime(2019, 11, 4, 14, 53, 0)

print(dt1 - dt2)



El ejemplo anterior muestra la resta entre dos objetos date y datetime. En el primer caso recibimos la diferencia en días que es 366, en el segundo caso obtuvimos un resultado diferente debido a que especificamos también el horario, por eso recibimos 365 días, 9 horas y 7 minutos.

#### Creando objetos timedelta

Ya sabemos que el resultado de la resta entre dos date o datetime producen un timedelta.

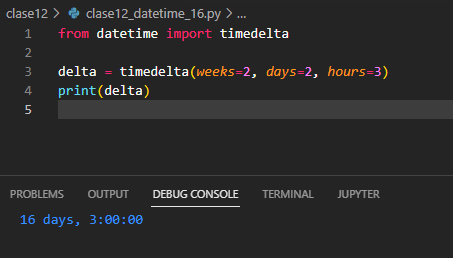
Por supuesto, también podemos crear nuestro propio **timedelta, c**on este fin debemos proveer alguno de los argumentos que son days, seconds, microseconds, minutes, hours y weeks. Cada uno de estos argumentos es opcional y vale 0 por defecto.

Veamos:

from datetime import timedelta

delta = timedelta(weeks=2, days=2, hours=3)

print(delta)



El resultado obtenido de 16 días se dio convirtiendo el argumento weeks a días (2 semanas son 14 días) y agregando los dos del argumento days, este comportamiento es normal ya que **timedelta** solo almacena días, segundos y microsegundos internamente.

#### Más operaciones con timedelta

Ahora que sabemos más sobre **timedelta**, veamos que otras operaciones podemos hacer con **date** y **datetime**:

from datetime import timedelta

from datetime import date

from datetime import datetime

delta = timedelta(weeks=2, days=2, hours=2)

print(delta)

delta2 = delta \* 2

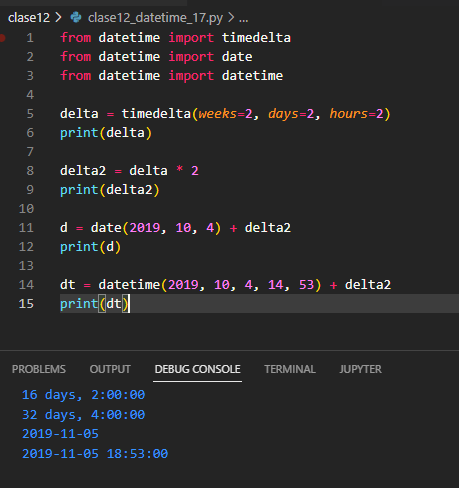
print(delta2)

d = date(2019, 10, 4) + delta2

print(d)

dt = datetime(2019, 10, 4, 14, 53) + delta2

print(dt)



Como podemos ver, los timedelta pueden ser multiplicados por números, también podemos sumar y restar timedeltas tanto a dates como datetimes.

## Actividades Módulo 12

#### Actividad 1

Escribir una aplicación que solicite al usuario la fecha y hora de su nacimiento, para eso se debe solicitar, el año, el numero de mes, el numero de dia, la hora y minuto únicamente (asumimos que el segundo fue 0) a continuación le indicará cuántos días han pasado entre su nacimiento y el momento de la ejecución del programa.

#### Actividad 2

Escribir un programa que implemente una calculadora de fechas, de manera tal que nos permita sumarles o restarles un lapso de tiempo, para eso el usuario debe indicar cuál de las operaciones quiere realizar mediante un menú y luego proveer año mes y día de la fecha y cantidad de días, horas, minutos y segundos con las cual se va a operar para luego mostrar el resultado en pantalla.

#### Actividad 3

Escribir un programa que haga las veces de diccionario, para esto mostraremos un menú al usuario con las opciones:

1. Agregar término
2. Listar
3. Buscar
4. Guardar
5. Salir

Luego de realizada la opción requerida, se debe volver a este menú. Se detalla la funcionalidad pretendida para cada opción:

* + **Agregar término**
  + **Listar:**: Solicita al usuario que indique un término y su significado, si dicho término no existe, se lo almacena; de lo contrario, se indica que el término ya existe en nuestro diccionario.

terminoA: definición terminoB: definición

…

terminoN: definición

* + **Buscar:** Solicita al usuario que indique un término y muestra su significado en

pantalla, si el término no existe, muestra el mensaje “término no encontrado”.

* + **Guardar:** Escribe cada término junto a su definición, uno por línea, con el formato

“término:definición” en un archivo de texto plano de nombre “diccionario.txt”.