Plan de Formación Profesional y Continua

cursos gratuitos
de formación profesional

[Programación en Python]

Fundamentos de Python

Introducción

En este manual, abordaremos los fundamentos de Python, uno de los lenguajes de programación más populares y versátiles en la actualidad. Esta lectura complementaria te será de utilidad para profundizar los temas vistos en clase y en los demás recursos asincrónicos.

Durante el curso, exploraremos los conceptos esenciales que te permitirán comprender y utilizar Python de manera efectiva. Conoceremos las mejores prácticas de codificación y aprenderemos a escribir código limpio, legible y eficiente. Además, nos sumergiremos en la resolución de problemas mediante la lógica de programación y la implementación de algoritmos simples.

Realizaremos ejercicios prácticos para aplicar los conceptos aprendidos, lo que te permitirá afianzar tus conocimientos y ganar confianza en el uso de Python como herramienta de programación.

Una vez finalizado este tema estarás en condiciones de abordar los distintos elementos que componen la sintaxis básica del lenguaje y sus principales funcionalidades cuando se trata de manipular información accediendo a distintos orígenes.

¡Prepárate para descubrir las bases sólidas que te permitirán construir aplicaciones, automatizar tareas y resolver problemas de manera eficiente utilizando Python!

Objetivos de aprendizaje

- Comprender la sintaxis básica del lenguaje Python
- Seleccionar y emplear librerías para la manipulación de datos en Python
- Aplicar conexiones efectivas con diversas fuentes de datos, permitiéndote acceder y gestionar información proveniente de diferentes orígenes de manera eficiente.
- Idear estrategias y técnicas avanzadas para acceder y manipular datos provenientes de diversas fuentes

Fundamentos de programación en Python

Python es conocido por su sintaxis simple y legible, lo que lo convierte en un lenguaje amigable y fácil de aprender. Si no tienes experiencia previa en programación, no te preocupes, porque en esta clase nos adentraremos en los conceptos básicos y te guiaremos en cada paso del camino.
Python es ampliamente utilizado en diversos campos, desde el desarrollo web y el análisis de datos hasta la inteligencia artificial y la automatización de tareas. Al aprender Python, estarás adquiriendo habilidades versátiles y altamente demandadas en el mundo laboral actual.

Versiones de Python 🖨️

Python es un lenguaje de programación en constante evolución, y a lo largo del tiempo, se han lanzado varias versiones de Python, cada una con mejoras y características adicionales. Las versiones principales de Python incluyen Python 1.x, Python 2.x y Python 3.x. A continuación, se detallan algunas de las versiones más significativas:

- **Python 1.0**: Lanzada en 1994, esta fue la primera versión oficial de Python. Estableció las bases del lenguaje y se centró en la simplicidad y la facilidad de uso.
- **Python 2.x:** Esta serie incluyó varias versiones, como Python 2.7, que se mantuvieron durante muchos años y se utilizaron ampliamente en la industria. Python 2.x introdujo mejoras en rendimiento y características adicionales.
- **Python 3.x**: La serie Python 3.x trajo cambios significativos en el lenguaje para abordar problemas de diseño y mejorar la consistencia. Aunque se esforzó por ser retrocompatible con Python 2.x, hubo cambios que rompieron la compatibilidad, lo que llevó a una transición gradual.
- **Python 3.0:** Lanzada en 2008, esta versión introdujo cambios importantes en la sintaxis y las características del lenguaje. Aunque la transición de Python 2.x a Python 3.x llevó tiempo, Python 3.x finalmente se estableció como la versión principal.

- **Python 3.8:** Una de las versiones más recientes en la serie Python 3, Python 3.8 incluye características como las asignaciones de expresiones (walrus operator) y mejoras en rendimiento.
- **Python 3.9**: Lanzada en 2020, Python 3.9 introdujo mejoras en la eficiencia de rendimiento y características como las uniones de tipo (type hints).
- **Python 3.10**: Aunque aún no se ha lanzado, Python 3.10 está en desarrollo y se espera que continúe mejorando el lenguaje.

La elección de la versión de Python a utilizar depende en gran medida de los proyectos específicos y las bibliotecas que se necesiten. En general, se recomienda utilizar la versión más reciente de la serie Python 3 para nuevos proyectos, ya que ofrece las últimas características y correcciones de errores.

El entorno de trabajo: Anaconda; El editor Spyder; Jupyter Notebooks. 💼

¿Qué es un portafolio de producto 🗃️?

El entorno de trabajo en Python es un aspecto fundamental para el desarrollo y ejecución de proyectos. Existen varias opciones disponibles, pero mencionaré tres de las más comunes: Anaconda, el editor Spyder y Jupyter Notebooks.

Anaconda: Anaconda es una plataforma de ciencia de datos y distribución de Python que simplifica la gestión de paquetes, entornos virtuales y bibliotecas. Viene con una amplia variedad de bibliotecas preinstaladas y herramientas como Conda, un gestor de paquetes que facilita la instalación y actualización de bibliotecas. Para instalar Anaconda, podes descargar el instalador adecuado para tu sistema operativo desde el sitio web oficial y seguir las instrucciones de instalación.

Editor Spyder: Spyder es un entorno de desarrollo interactivo (IDE) diseñado para la programación en Python, especialmente para tareas de análisis de datos y científicas. Ofrece un editor de código, una consola interactiva y ventanas de exploración para ayudar en el desarrollo y la depuración. Spyder se instala típicamente como parte de la distribución Anaconda, pero también podes instalarlo por separado si lo deseas.

Jupyter Notebooks: Jupyter Notebooks es una herramienta muy utilizada en la ciencia de datos y la investigación. Permite crear documentos interactivos que combinan código, texto, visualizaciones y explicaciones. Los notebooks son ideales para explorar datos, presentar resultados y compartir análisis. Podes instalar Jupyter Notebooks como parte de la distribución de Anaconda o a través de la instalación de paquetes con pip.

Instalación de Anaconda:

1. Ve al sitio web oficial de Anaconda: https://www.anaconda.com/products/distribution
2. Descarga el instalador adecuado para tu sistema operativo (Windows, macOS o Linux).
3. Ejecuta el archivo de instalación descargado y sigue las instrucciones del instalador. Asegúrate de seleccionar la opción "Add Anaconda to my PATH environment variable" durante la instalación en Windows. Esto facilitará el acceso a Python y otras herramientas desde la línea de comandos.
4. Una vez completada la instalación, podes abrir el "Anaconda Navigator," que es una interfaz gráfica que te permite gestionar entornos y paquetes de manera más amigable.

Instalación de Spyder (incluido en Anaconda):

Como parte de la instalación de Anaconda, Spyder se instalará automáticamente. Puedes acceder a Spyder desde Anaconda Navigator o ejecutarlo desde la línea de comandos utilizando el comando spyder.

Instalación de Jupyter Notebooks (incluido en Anaconda):

Jupyter Notebooks también se instalará automáticamente junto con Anaconda. Para abrir un cuaderno de Jupyter, podes usar Anaconda Navigator o ejecutar el siguiente comando en tu terminal:

```
jupyter notebook
```

Instalación de Python

El primer paso para comenzar a usar Python es instalarlo en tu computadora. Puedes descargar la última versión de Python desde el sitio web oficial de Python (https://www.python.org). Sigue las instrucciones de instalación adecuadas para tu sistema operativo.

Entorno de desarrollo integrado (IDE)

Un entorno de desarrollo integrado (IDE) es una herramienta que te ayuda a escribir, depurar y ejecutar código Python de manera más eficiente.

Instalación de Visual Studio Code

Visual Studio Code es un editor de texto que nos permitirá crear nuestros programas y ejecutarlos ya que cuenta con una terminal preintegrada.

1. Nos dirigimos a la página oficial https://code.visualstudio.com/
2. Reconocerá el sistema operativo en el que estamos y solo procedemos a realizar la descarga
3. Una vez descargado ejecutamos
4. Es una instalación sencilla así que no tendremos problemas

Fundamentos del lenguaje Python

Tipos de Datos 📖

Qué son los tipos de datos en Python

Un tipo de dato se refiere a la clasificación de los valores que pueden ser almacenados en una variable. Cada tipo de dato tiene características y propiedades específicas que lo hacen adecuado para ciertos usos y operaciones.

Los tipos de datos también se utilizan para determinar cómo se almacenan los valores en la memoria de la computadora y cómo se pueden manipular y operar los valores. Por ejemplo, los números enteros se almacenan como secuencias de bits en la memoria, y se pueden realizar operaciones aritméticas como suma, resta, multiplicación y división.

Tipos de datos en Python

- Numéricos
- De texto
- Booleanos
- De secuencias

- De mapeo
- Conjuntos
- De bytes

Python admite varios tipos de datos, como enteros, flotantes, cadenas, listas y diccionarios. Por ejemplo:

```
entero = 10
flotante = 3.14
cadena = "Hola, mundo!"
lista = [1, 2, 3, 4, 5]
diccionario = {"nombre": "John", "edad": 25}
```

Tipos de datos numéricos

int

Los datos tipo int (abreviatura de «entero»), en Python, son aquellos que representan números enteros. Estos son un tipo de número que no tiene parte decimal, es decir, son completos sin fracciones. En Python, los datos tipo int se representan simplemente escribiendo un número sin parte decimal. Por ejemplo:

1, 2, 3, 4, 5

Los datos tipo int se utilizan para realizar operaciones matemáticas, como sumas, restas, multiplicaciones y divisiones, así como para realizar comparaciones entre números enteros. Al trabajar con datos tipo int en Python, es importante tener en cuenta que el resultado de algunas operaciones puede ser de tipo flotante si se utiliza la división. Por ejemplo, si se divide 7 entre 2 en Python, el resultado es 3.5, que es un número tipo float. Si se quiere obtener solo la parte entera del resultado, se puede utilizar la función «división entera» (//), que devuelve el resultado entero de la división. Por ejemplo, 7 // 2 devuelve 3 (que es el resultado entero de la división).

float

Los datos tipo float (abreviatura de «coma flotante»), en Python, son aquellos que representan números con decimales. En otras palabras, los datos tipo float son números reales que pueden tener una parte entera y una parte decimal. En Python, los datos tipo float se representan escribiendo un número con un punto decimal. Por ejemplo:

1.23, 3.14, 45.5, -12.34

Los datos tipo float se utilizan para realizar operaciones matemáticas que involucran números con decimales. Al trabajar con estos, es importante tener en cuenta que algunos cálculos pueden arrojar una precisión limitada, debido a la forma en que se almacenan los números en la memoria de la computadora. En general, los números tipo float tienen una precisión de alrededor de 15-17 dígitos decimales. Además, al realizar cálculos con números tipo float, es posible que se produzcan errores de redondeo o de exactitud que pueden afectar los resultados de la operación.

complex

Los datos tipo complex (abreviatura de «números complejos»), en Python, son aquellos que representan números que tienen una parte real y una parte imaginaria. En matemáticas, los números complejos se definen como la suma de una parte real y una parte imaginaria multiplicada por la unidad imaginaria (que se denota como «i» o «j»).

Los datos tipo complex se representan escribiendo un número real seguido por una «j» para indicar la parte imaginaria.

Tipos de texto

str

Los datos tipo str (abreviatura de «cadena de caracteres» o «string»), en Python, son aquellos que representan una secuencia de caracteres. En otras palabras, los datos tipo string son texto que se puede representar mediante una serie de caracteres (letras, números, símbolos, etcétera). Los datos tipo string se representan entre comillas simples (') o dobles ("").

Los siguientes son ejemplos de datos tipo str en Python:

"Hola mundo", 'Python es genial', "1234".

Los datos tipo string se utilizan para almacenar texto y manipular cadenas de caracteres. Python tiene muchas funciones incorporadas a fin de trabajar con cadenas de caracteres, como la función len() para obtener la longitud de una cadena y la función split() para dividir una cadena en una lista de subcadenas separadas por un carácter específico.

Es importante tener en cuenta que en Python, las cadenas de caracteres son inmutables; es decir, no se pueden modificar una vez creadas. Si se necesita

manipular una cadena de caracteres, se hace una nueva con los cambios deseados.

Tipos booleanos

bool

Los datos tipo bool (abreviatura de «booleanos»), en Python, son aquellos que representan valores de verdad. En otras palabras, son valores verdaderos o falsos. En Python, los datos tipo bool se representan como True o False (con la primera letra en mayúscula).

Por ejemplo:

True, False

Se utilizan principalmente en expresiones lógicas y de control de flujo, como en declaraciones if y while. Las expresiones lógicas en Python pueden evaluar a un valor verdadero o falso, y se utilizan para tomar decisiones en el código.

Además de los valores True y False, Python tiene algunas funciones incorporadas que devuelven valores booleanos, como la función «bool()», que devuelve True si el argumento que se le pasa es verdadero, y False si el argumento es falaz.

Tipos de secuencias

list

Los datos tipo list (abreviatura de «listas»), en Python, son aquellos que representan una colección ordenada de elementos. En otras palabras, las listas son una estructura de datos que permite almacenar múltiples valores en un solo objeto y acceder a ellos por su posición en la lista.

En Python, los datos tipo lista se representan mediante corchetes «[ ]» que contienen los elementos de la lista separados por comas. Los elementos pueden ser de cualquier tipo de datos, incluyendo otros objetos, como otras listas.

Por ejemplo:

[1, 2, 3, 4, 5], ['a', 'b', 'c'], [1, 'a', True, [2, 3, 4]]

Los datos tipo lista se utilizan para almacenar y manipular colecciones de datos. Python tiene muchas funciones y métodos incorporados a fin de trabajar con listas, como la función len() para obtener la longitud de una lista y los métodos append() y extend() para agregar elementos a una lista.

Es importante tener en cuenta que las listas en Python son mutables, lo que significa que se pueden modificar una vez que se han creado. Esto permite agregar, eliminar o modificar elementos en una lista.

tuple

Los datos tipo tuple (abreviatura de «tuplas»), en Python, son similares a las listas, pero a diferencia de las listas, son inmutables. En otras palabras, las tuplas son una estructura de datos que permite almacenar múltiples valores en un solo objeto y acceder a ellos por su posición en la tupla, pero una vez creada la tupla, los elementos no pueden modificarse.

Los datos tipo tupla se representan mediante paréntesis «( )» que contienen los elementos de la tupla separados por comas. Los elementos pueden ser de cualquier tipo de datos, incluyendo otros objetos, como otras tuplas.

Por ejemplo:

(1, 2, 3, 4, 5), ('a', 'b', 'c'), (1, 'a', True, (2, 3, 4))

Los datos tipo tupla se utilizan principalmente para almacenar datos que no deben ser modificados una vez creados, como una fecha o coordenadas geográficas. Python tiene algunas funciones incorporadas a fin de trabajar con tuplas, como la función len() para obtener la longitud de una tupla y la función tuple() para crear una tupla a partir de una lista u otro iterable.

Es importante tener en cuenta que las tuplas en Python son inmutables; es decir, que no se pueden modificar una vez que se han creado. Si se necesita manipular una colección de datos, se debe usar una lista en su lugar.

range

Los datos tipo range, en Python, representan una secuencia inmutable de números enteros. Un objeto de tipo range se utiliza comúnmente para generar una secuencia de números enteros para su uso en un bucle for.

La función range() devuelve un objeto de tipo range. El objeto range toma tres argumentos: el valor inicial, el valor final (no incluido en la secuencia) y el tamaño del paso. Por defecto, el valor inicial es 0 y el tamaño del paso es 1.

Por ejemplo:

range(0, 10, 1), range(0, 10), range(1, 11, 2)

El primer ejemplo crea un objeto range que representa la secuencia de números enteros del 0 al 9 (inclusivo) con un tamaño de paso de 1. El segundo ejemplo es equivalente al primer ejemplo, ya que el valor inicial y el tamaño del paso se asumen como 0 y 1, respectivamente, por defecto. El tercer ejemplo crea un objeto range que representa la secuencia de números enteros del 1 al 9 (inclusivo) con un tamaño de paso de 2.

Tipos de mapeo

dict

Los datos tipo dict (abreviatura de «diccionario»), en Python, son una estructura de datos que permite almacenar un conjunto de datos como pares clave-valor, donde cada valor es accesible a través de una clave única. En otras palabras, los diccionarios permiten asociar valores con claves.

Los datos tipo dict se representan mediante llaves «{ }» que contienen una serie de pares clave-valor separados por comas y cada par se separa por dos puntos «:». Las claves en un diccionario son únicas y pueden ser de cualquier tipo inmutable, como una cadena, un número entero o una tupla. Los valores también pueden ser de cualquier tipo de datos.

Por ejemplo:

{'nombre': 'Juan', 'edad': 25, 'ciudad': 'Madrid'}, {1: 'uno', 2: 'dos', 3: 'tres'}, {(1, 2): 'valor de la tupla', 'clave': [1, 2, 3]}

En Python, los diccionarios son útiles para almacenar y acceder a datos de manera eficiente. Los datos pueden ser accedidos en un diccionario utilizando su clave, como en el siguiente ejemplo:

```
D = {'NOMBRE': 'JUAN', 'EDAD': 25, 'CIUDAD': 'MADRID'}

PRINT(D['NOMBRE']) # SALIDA: 'JUAN'
```

También es posible modificar los valores en un diccionario, agregar nuevos pares clave-valor y eliminar pares clave-valor existentes. Los diccionarios son muy versátiles y se utilizan ampliamente en programación.

Tipos de conjuntos

set

Los datos tipo set, en Python, son una colección de elementos únicos e inmutables; es decir, no debe haber duplicados en un conjunto y no se pueden cambiar después de ser creados. Un conjunto se crea utilizando llaves «{ }» o la función set() y los elementos se separan por comas.

Por ejemplo:

```
{1, 2, 3, 4, 5}{

'HOLA', 'MUNDO', 'PYTHON'}

SET([1, 2, 3, 4])
```

En el primer ejemplo se crea un conjunto con los elementos 1, 2, 3, 4 y 5; en el segundo, un conjunto con las cadenas ‘hola’, ‘mundo’ y ‘python’; y en el tercero, un conjunto a partir de una lista con los elementos 1, 2, 3 y 4.

Los conjuntos en Python son útiles para realizar operaciones de conjuntos como la unión, la intersección, la diferencia y la comprobación de pertenencia. También se pueden usar para eliminar duplicados de una lista o para encontrar elementos únicos en una secuencia.

Por ejemplo, la siguiente secuencia de código utiliza conjuntos para eliminar duplicados de una lista:

```
LISTA = [1, 2, 3, 4, 4, 3, 5, 6, 6]

CONJUNTO = SET(LISTA)

LISTA_SIN_DUPLICADOS = LIST(CONJUNTO)

PRINT(LISTA_SIN_DUPLICADOS) # SALIDA: [1, 2, 3, 4, 5, 6]
```

frozenset

Los datos tipo frozenset son similares a los conjuntos (sets), pero son inmutables; es decir, una vez creados no se pueden modificar. Se crean

utilizando la función frozenset() y se usan para almacenar elementos únicos, como los conjuntos.

A diferencia de los conjuntos, los datos tipo frozenset se pueden utilizar como elementos de diccionarios, ya que son inmutables y, por lo tanto, hashables. Además, los datos tipo frozenset es factible usarlos como elementos de otros conjuntos, ya que también son inmutables.

Acá hay algunos ejemplos de datos tipo frozenset en Python:

```
FSET = FROZENSET([1, 2, 3, 4])

PRINT(FSET) # SALIDA: FROZENSET({1, 2, 3, 4})

FSET2 = FROZENSET(('HOLA', 'MUNDO', 'PYTHON'))

PRINT(FSET2) # SALIDA: FROZENSET({'HOLA', 'MUNDO',
'PYTHON'})
```

En el primer ejemplo, se crea un frozenset con los elementos 1, 2, 3 y 4; y en el segundo, un frozenset con las cadenas 'hola', 'mundo' y 'python'. Al ser inmutables, no se pueden agregar o quitar elementos de un frozenset después de que se ha creado.

Tipos de bytes

bytes

Los datos tipo bytes, en Python, son una secuencia inmutable de bytes. Representan una cadena de bytes en bruto y se utilizan principalmente para trabajar con datos binarios como archivos, sockets de red, criptografía, entre otros.

Los datos tipo bytes se pueden crear utilizando una sintaxis de prefijo b antes de la cadena de bytes o utilizando la función bytes(). Los elementos de los datos tipo bytes son enteros en el rango de 0 a 255.

Aquí hay algunos ejemplos de datos tipo bytes en Python:

```
B = B'HOLA'

PRINT(B) # SALIDA: B'HOLA'

B2 = BYTES([0X68, 0X6F, 0X6C, 0X61])
```

```
PRINT(B2) # SALIDA: B'HOLA'
```

En el primer ejemplo se crea un objeto bytes utilizando la sintaxis de prefijo b; y en el segundo, un objeto bytes utilizando la función bytes() y una lista de enteros que representan los códigos ASCII de los caracteres 'h', 'o', 'l' y 'a'.

Los datos tipo bytes se pueden utilizar para codificar y decodificar datos binarios, y también a fin de realizar operaciones binarias, como XOR y desplazamiento de bits. Además, los datos tipo bytes es factible convertirlos a objetos bytearray, que son mutables.

bytearray

Los datos tipo bytearray, en Python, son similares a los datos tipo bytes, pero son mutables; es decir, una vez creados se pueden modificar. Los datos tipo bytearray se utilizan para representar secuencias de bytes modificables, como archivos binarios, buffers de red, entre otros.

Los datos tipo bytearray se pueden crear utilizando la función bytearray(). Al igual que los datos tipo bytes, los elementos de los datos tipo bytearray son enteros en el rango de 0 a 255.

Aquí hay algunos ejemplos de datos tipo bytearray en Python:

```
B_ARR = BYTEARRAY(B'HOLA')

PRINT(B_ARR) # SALIDA: BYTEARRAY(B'HOLA')

B_ARR[1] = 111 # CAMBIA EL SEGUNDO ELEMENTO A 'O'

PRINT(B_ARR) # SALIDA: BYTEARRAY(B'OOLA')

B_ARR2 = BYTEARRAY([104, 111, 108, 97])

PRINT(B_ARR2) # SALIDA: BYTEARRAY(B'HOLA')
```

En el primer ejemplo se crea un objeto bytearray utilizando la función bytearray(). En el segundo, se modifica el siguiente elemento del objeto bytearray a 'o'. En el tercero, se crea un objeto bytearray utilizando una lista de enteros que representan los códigos ASCII de los caracteres 'h', 'o', 'l' y 'a'.

Los datos tipo bytearray se pueden utilizar para realizar operaciones binarias, como XOR y desplazamiento de bits, y también convertir a objetos bytes, que son inmutables.

Los diferentes tipos de datos en Python tienen sus propias propiedades y métodos únicos, lo que permite a los programadores manipular y trabajar con ellos de diversas maneras. Entonces, ahora que conoces los tipos de datos disponibles, podrás elegir el más adecuado para el trabajo que deseas realizar.

En Python, podes iniciar una variable simplemente asignándole un valor. No es necesario declarar el tipo de variable antes de usarla.

```
mi_variable = 42  # Inicializamos la variable con el valor 42
```

En este ejemplo, hemos creado una variable llamada mi_variable y le hemos asignado el valor 42. Python determina automáticamente el tipo de variable basado en el valor asignado.

Puedes asignar cualquier tipo de valor a una variable en Python, ya sea un número, una cadena de texto, una lista, un objeto, etc. Python es un lenguaje de programación de tipado dinámico, lo que significa que no necesitas especificar el tipo de variable de antemano; el tipo se infiere en tiempo de ejecución.

```
# Mi primer programa en Python

nombre = "John"
edad = 25
print("Hola, mi nombre es", nombre, "y tengo", edad, "años.")
```

Sintaxis básica

La sintaxis de Python es simple y legible, lo que facilita su aprendizaje. Aquí hay algunos conceptos básicos de sintaxis que debes conocer:

- **Variables**: en Python, las variables se crean y asignan valores utilizando el operador de asignación (=). Por ejemplo:

```
nombre = "John"
edad = 25
```

Strings 📖

En Python, un string (o cadena) es una secuencia de caracteres, como letras, números y símbolos, que se utiliza para representar texto. Los strings son un tipo de dato fundamental en Python y se utilizan para una amplia gama de

tareas, desde mostrar mensajes en pantalla hasta procesar y manipular texto.

En Python, los strings se pueden definir utilizando comillas simples ('), comillas dobles (") o comillas triples (''' o """). Por ejemplo:

```
mi_string = "Hola, mundo!"  # Utilizando comillas dobles
otro_string = 'Python es genial' #Utilizando comillas simples
multilinea = '''Este es un string
con múltiples líneas'''  # Utilizando comillas triples
```

Los strings en Python son inmutables, lo que significa que no puedes cambiar caracteres individuales dentro de un string después de crearlo. Sin embargo, puedes realizar muchas operaciones y manipulaciones en strings, como concatenación, slicing (rebanado), búsqueda de subcadenas y más.

Acá hay algunos ejemplos de operaciones comunes en strings:

```
# Concatenación de strings
saludo = "Hola, "
nombre = "Juan"
mensaje = saludo + nombre  # El resultado es "Hola, Juan"

# Longitud de un string
longitud = len(mensaje)  # El resultado es 10

# Slicing (rebanado) de un string
subcadena = mensaje[0:5]
# Obtiene los primeros 5 caracteres: "Hola,"

# Búsqueda de subcadena
busqueda = "Juan" in mensaje  # El resultado es True

# Mayúsculas y minúsculas
mayusculas = mensaje.upper()
# Convierte a mayúsculas: "HOLA, JUAN"
```

```
minusculas = mensaje.lower()
# Convierte a minúsculas: "hola, juan"
```

Los strings son fundamentales para trabajar con texto en Python y se utilizan en numerosas aplicaciones, desde procesamiento de texto hasta manipulación de datos.

Métodos de String en Python 🛠️

Python proporciona una variedad de métodos incorporados que puedes utilizar para manipular y trabajar con strings de manera efectiva. Los métodos de string son funciones que se aplican a objetos de tipo string y permiten realizar diversas operaciones, como búsqueda, modificación, validación y formateo de cadenas de texto.

A continuación, se presentan algunos métodos de string comunes en Python:

len(string): Este método devuelve la longitud (cantidad de caracteres) de un string.

```
texto = "¡Hola, mundo!"
longitud = len(texto)  # Resultado: 13
```

upper(): Convierte todos los caracteres del string a mayúsculas.

```
texto = "Hola, mundo"
mayusculas = texto.upper()  # Resultado: "HOLA, MUNDO"
```

lower(): Convierte todos los caracteres del string a minúsculas.

```
texto = "Hola, Mundo"
minusculas = texto.lower()  # Resultado: "hola, mundo"
```

strip(): Elimina los espacios en blanco y caracteres de salto de línea al principio y al final del string.

```
texto = "   ¡Hola, mundo!   "
limpio = texto.strip()  # Resultado: "¡Hola, mundo!"
```

replace(subcadena, nueva_subcadena): Reemplaza todas las ocurrencias de una subcadena en el string por una nueva subcadena.

```
texto = "Hola, Python"
nuevo_texto = texto.replace("Python", "Mundo")
# Resultado: "Hola, Mundo"
```

split(separador): Divide el string en una lista de subcadenas utilizando el separador especificado.

```
texto = "manzana,banana,cereza"
frutas = texto.split(",")
# Resultado: ["manzana", "banana", "cereza"]
```

find(subcadena): Busca la primera aparición de una subcadena y devuelve la posición en la que se encuentra. Si no se encuentra, devuelve -1.

```
texto = "Hola, mundo"
posicion = texto.find("mundo")  # Resultado: 6
```

👉**Ejemplo práctico**: Supongamos que tenemos un string y deseamos limpiarlo y reemplazar una subcadena:

```
texto = "    ¡Hola, Python!    "
limpio = texto.strip()  # Elimina espacios en blanco
reemplazado = limpio.replace("Python", "Mundo")
# Reemplaza "Python" por "Mundo"
print(reemplazado)
```

El resultado será: "¡Hola, Mundo!".

Estos son solo algunos ejemplos de los muchos métodos de string disponibles en Python. Puedes explorar y utilizar estos métodos según tus necesidades específicas al trabajar con strings.

Conversiones de “tipo” 🖥️

En Python, puedes realizar conversiones de tipo para cambiar una variable de un tipo de dato a otro. Python proporciona funciones y operadores para realizar estas conversiones. Algunos ejemplos comunes de conversiones de tipo incluyen la conversión de un número entero a un número de punto flotante, la conversión de una cadena de texto a un número, etc.

Conversión de un número entero a un número de punto flotante:

```
entero = 42

flotante = float(entero) # Convierte el entero a un número de punto flotante
```

Conversión de un número de punto flotante a un número entero (truncamiento):

```
flotante = 3.14

entero = inf(flotante) # Convierte el flotante a un entero, truncando los decimales
```

Conversión de una cadena de texto a un número:

```
cadena = "123"

numero = int(cadena)  # Convierte la cadena a un número entero
```

Conversión de un número a una cadena de texto:

```
numero = 42

cadena = str(numero)  # Convierte el número a una cadena de texto
```

👉**Ejemplo Práctico:**

```
entero = 5
flotante = float(entero)
cadena_numero = "42"
numero = int(cadena_numero)
cadena = str(numero)
```

```
print("Entero:", entero)
print("Flotante:", flotante)
print("Cadena número:", cadena_numero)
print("Número:", numero)
print("Cadena:", cadena)
```

Este ejemplo muestra cómo convertir entre diferentes tipos de datos en Python, incluyendo enteros, flotantes y cadenas.

Estas son algunas de las conversiones de tipo más comunes en Python. Puedes realizar conversiones entre otros tipos de datos según tus necesidades.

Es importante entender las conversiones de tipo en Python para evitar errores y realizar operaciones adecuadas en tus programas.

- **Conversión segura:** No todas las conversiones son seguras, especialmente cuando conviertes entre tipos incompatibles. Asegúrate de que la conversión sea apropiada para el contexto y verifica los valores antes de realizar conversiones.
- **Pérdida de información:** Algunas conversiones pueden resultar en la pérdida de información. Por ejemplo, cuando conviertes un número de punto flotante a un número entero, los decimales se truncarán, lo que podría afectar el resultado de tus cálculos.
- **Errores de conversión:** Las conversiones pueden generar errores si los datos no son compatibles con el tipo de destino. Por ejemplo, tratar de convertir una cadena no numérica en un número generará un error.
- **Conversiones implícitas:** Python realiza conversiones implícitas en algunas situaciones. Por ejemplo, cuando sumas un número entero y un número de punto flotante, Python convertirá automáticamente el entero a un flotante antes de realizar la suma.
- **Funciones de conversión:** Python proporciona funciones incorporadas para realizar conversiones, como int(), float(), str(), bool(), entre otras. Puedes usar estas funciones según sea necesario en tus programas.

- ↪ **Evita confusiones:** Nombrar tus variables de manera descriptiva puede ayudarte a evitar confusiones al tratar con diferentes tipos de datos. Además, es importante documentar las conversiones de tipo en tu código para que otros desarrolladores puedan entenderlo fácilmente.
- ↪ **Conversiones personalizadas:** Podes definir tus propias funciones de conversión personalizadas si necesitas realizar conversiones específicas para tus aplicaciones.

Las conversiones de tipo son una parte fundamental de la programación en Python, y es esencial comprender cuándo y cómo realizarlas de manera segura y eficiente en tus programas.

Variables y asignación 🖨️

En Python, **las variables** son contenedores que se utilizan para almacenar datos.

Declaración de variables: En Python, **no es necesario declarar explícitamente el tipo de una variable antes de usarla**. Podes simplemente asignar un valor a una variable y Python determinará el tipo de variable automáticamente.

Reglas de nombres de variables:

Los nombres de variables en Python deben seguir algunas reglas:

- Deben comenzar con una letra (mayúscula o minúscula) o un guion bajo (_).
- Pueden contener letras, números y guiones bajos.
- No pueden contener espacios en blanco u otros caracteres especiales.
- Convenciones de nombres: Se recomienda seguir las convenciones de nombres de variables para que tu código sea más legible. Por ejemplo, usa letras minúsculas para los nombres de variables y usa guiones bajos para separar palabras en nombres compuestos (por ejemplo, mi_variable).

Asignación de valores: Puedes asignar un valor a una variable utilizando el operador de asignación (=). Por ejemplo, mi_variable = 42 asignará el valor 42 a la variable mi_variable.

Reasignación de valores: Puedes cambiar el valor de una variable simplemente asignándole un nuevo valor. Por ejemplo, mi_variable = 100 cambiará el valor de mi_variable a 100.

Variables mutables e inmutables: Algunos tipos de datos en Python, como las listas y los diccionarios, son mutables, lo que significa que puedes cambiar su contenido después de crearlos. Otros tipos, como las cadenas y las tuplas, son inmutables, lo que significa que no puedes cambiar su contenido una vez que se crean.

Variables como etiquetas: En Python, las variables actúan como etiquetas para objetos en lugar de contenedores de datos. Esto significa que cuando asignas una variable a otra, ambas apuntan al mismo objeto en memoria.

Eliminación de variables: Puedes eliminar una variable y liberar su memoria utilizando la instrucción del. Por ejemplo, del mi_variable eliminará la variable mi_variable.

Uso de variables en expresiones: Puedes usar variables en expresiones matemáticas, lógicas y de cadena. Por ejemplo, puedes realizar operaciones como resultado = a + b donde a y b son variables.

Tipos de datos de las variables: Las variables pueden contener una variedad de tipos de datos, como enteros, números de punto flotante, cadenas, listas, diccionarios, objetos personalizados, entre otros.

Las variables y la asignación son conceptos fundamentales en la programación en Python y son esenciales para el desarrollo de aplicaciones. Es importante comprender cómo trabajar con variables de manera eficiente y seguir buenas prácticas de nomenclatura para que tu código sea legible y mantenible.

```
# Asignación de un entero a una variable
numero = 42

# Asignación de una cadena a una variable
nombre = "Juan"
```

```
# Reasignación de valor a una variable
numero = 100

# Uso de variables en una expresión
resultado = numero * 2

# Asignación de una lista a una variable
mi_lista = [1, 2, 3, 4, 5]

# Cambio de valor en una lista
mi_lista[2] = 99

# Asignación de un diccionario a una variable
mi_diccionario = {"clave1": "valor1", "clave2": "valor2"}

# Acceso a un valor en el diccionario
valor = mi_diccionario["clave1"]

# Asignación de una tupla a una variable
mi_tupla = (1, 2, 3)

# Intentar cambiar un valor en una tupla (esto generará un error)
# mi_tupla[1] = 99
```

```
# Eliminación de una variable
del nombre


# Intentar usar una variable que ya ha sido eliminada (esto generará un error)
# print(nombre)
```

Formateo de strings 💼

En Python, puedes interactuar con los usuarios a través de la entrada y salida de datos.

En Python, input y print son funciones que se utilizan para interactuar con la entrada y salida de datos en programas.

input(): La función input se utiliza para obtener información ingresada por el usuario desde el teclado. Cuando se llama a input, espera a que el usuario ingrese datos y presione la tecla "Enter".

Entrada de datos:

```
nombre = input("Por favor, ingresa tu nombre: ")
print("Hola,", nombre)


# Para recibir entrada de texto desde el usuario
nombre = input("Por favor, ingresa tu nombre: ")
print("Hola,", nombre)


# Para recibir entrada numérica desde el usuario
edad = int(input("Por favor, ingresa tu edad: "))
print("Tienes", edad, "años.")
```

```
# Ten en cuenta que input() devuelve una cadena, por lo que convertimos la entrada a entero con int() en el ejemplo de edad.
```

En este ejemplo, input muestra un mensaje al usuario y luego espera a que ingrese su nombre. El nombre ingresado se almacena en la variable nombre.

print(): La función print se utiliza para mostrar información en la pantalla, como resultados, mensajes, variables, etc. Podes pasar uno o más argumentos a print, y se imprimirán en la pantalla.

Salida de datos:

```
mensaje = "¡Hola, mundo!"
print(mensaje)

# Para mostrar información al usuario
print("Este es un mensaje de salida.")

# Para mostrar valores de variables
numero = 42
print("El número es:", numero)

# Para formatear la salida
nombre = "Juan"
edad = 30
print("Nombre: {}, Edad: {}".format(nombre, edad))
```

En este caso, la cadena "¡Hola, mundo!" se imprimirá en la pantalla.

Estas funciones son esenciales para la interacción con el usuario y la presentación de resultados en programas Python.

Expresiones regulares 💼

Las expresiones regulares son patrones de búsqueda que te permiten realizar operaciones de búsqueda y manipulación de cadenas de texto de manera avanzada. Python ofrece el módulo **re** para trabajar con expresiones regulares.

1. Creación de un patrón:

Para utilizar expresiones regulares, primero define un patrón que describe el conjunto de cadenas que estás buscando.

```
import re

patron = r'\d+'  # Encuentra uno o más dígitos
```

2**. Búsqueda de coincidencias:**

Podes utilizar funciones como search() para buscar el patrón en una cadena.

```
texto = "La temperatura actual es 25 grados."
coincidencia = re.search(patron, texto)

if coincidencia:
    print("Se encontró el patrón:", coincidencia.group())
```

3. Otros métodos:

El objeto de coincidencia (match) proporciona varios métodos para obtener información sobre la coincidencia, como start(), end(), y group().

```
print("Inicio de la coincidencia:", coincidencia.start())
print("Fin de la coincidencia:", coincidencia.end())
```

👉**Ejemplo Práctico:**

```
import re
```

```
patron = r'\d+'
texto = "La temperatura actual es 25 grados."

coincidencia = re.search(patron, texto)

if coincidencia:
  print("Número encontrado:", coincidencia.group())
  print("Inicio de la coincidencia:", coincidencia.start())
  print("Fin de la coincidencia:", coincidencia.end())
```

Este ejemplo muestra cómo utilizar expresiones regulares para buscar un número en una cadena de texto y obtener información sobre la coincidencia.

Nota: Las expresiones regulares pueden ser complejas, pero proporcionan una poderosa herramienta para el procesamiento de cadenas de texto.

Instrucciones básicas de Python

En Python, las instrucciones básicas constituyen el cimiento esencial para cualquier programa. Estas incluyen la manipulación de variables, el trabajo con cadenas de texto, operaciones aritméticas, el uso de booleanos, la reasignación de variables, la implementación de comentarios, condicionales (IF, ELSE, ELIF), y la definición y utilización de funciones.

Estas estructuras fundamentales ofrecen la capacidad de **almacenar y manipular datos, tomar decisiones lógicas, y estructurar el código de manera eficiente**. Conocer y dominar dichas instrucciones provee a los programadores las herramientas necesarias para construir programas básicos y sentar las bases sólidas para comprender conceptos más avanzados en el desarrollo con Python.

Operadores y expresiones 🛠️

En Python, los operadores son símbolos especiales que se utilizan para realizar operaciones en variables y valores. Las expresiones son combinaciones de variables y operadores que producen un nuevo valor.

Las expresiones aritméticas se utilizan para realizar operaciones matemáticas y calcular resultados numéricos. Las expresiones aritméticas pueden involucrar operadores matemáticos como suma (+), resta (-), multiplicación (*), división (/), resto (%) y exponente (**).

```
# Definición de variables
a = 10
b = 5
# Operaciones aritméticas
suma = a + b
resta = a - b
multiplicacion = a * b
division = a / b
division_entera = a // b
resto = a % b
potencia = a ** b
# Impresión de resultados
print("Suma:", suma)
print("Resta:", resta)
print("Multiplicación:", multiplicacion)
print("División:", division)
print("División entera:", division_entera)
print("Resto:", resto)
print("Potencia:", potencia)
```

En el ejemplo anterior, se utilizan diferentes operadores aritméticos para realizar operaciones matemáticas básicas. Es importante tener en cuenta

que las operaciones se realizan de acuerdo con las reglas de precedencia matemática estándar.

Además de los operadores aritméticos básicos, Python también proporciona operadores de asignación compuestos que permiten combinar una operación aritmética con una asignación en una sola expresión.

Vamos a usar las 4 operaciones matemáticas básicas, suma, resta, multiplicación y división desde la consola. La sintaxis de Python para el uso de estas operaciones es igual a la que usamos en cualquier calculadora.

A medida que vayas desarrollando habilidades de programación te vas a dar cuenta de que podemos aplicarlas en muchas cosas más.

Suma (+)

La operación suma se realiza concatenando los números que queremos sumar con el signo de la suma (+).

Resta (-)

De manera análoga, la operación resta se realiza concatenando los números que queremos restar con el signo de la resta (-).

Multiplicación (*)

La operación multiplicación se realiza concatenando los números que queremos multiplicar con el signo de la multiplicación, el asterisco (*).

División (*)

La operación división se realiza ubicando signo de la división, la barra inclinada hacia la derecha (/), entre los números que queremos dividir.

% (resto): Calcula el resto de la división entre x e y. Ejemplo, x % y da como resultado 2, que es el resto de la división de 5 entre 3.

**** (potencia):** Eleva x a la potencia y. Ejemplo, x ** y da como resultado 125, que es 5 elevado a la potencia 3.

Operadores de Asignación

Además de los operadores aritméticos básicos, Python también proporciona operadores de asignación compuestos, que permiten combinar una operación aritmética con una asignación en una sola expresión.

- **Operador de Asignación Simple (=):** Se utiliza para asignar un valor a una variable.
- **Operador de Suma y Asignación (+=):** Se utiliza para sumar el valor de la variable con otro valor y asignar el resultado nuevamente a la variable.
- **Operador de Resta y Asignación (-=):** Se utiliza para restar el valor de la variable con otro valor y asignar el resultado nuevamente a la variable.
- **Operador de Multiplicación y Asignación (*=):** Se utiliza para multiplicar el valor de la variable por otro valor y asignar el resultado nuevamente a la variable.
- **Operador de División y Asignación (/=):** Se utiliza para dividir el valor de la variable por otro valor y asignar el resultado nuevamente a la variable.
- **Operador de Módulo y Asignación (%=):** Se utiliza para calcular el módulo del valor de la variable con otro valor y asignar el resultado nuevamente a la variable.

```
x = 10  # Asigna el valor 10 a la variable x


y = 5
y += 3  # Esto es equivalente a y = y + 3


z = 7
z -= 2  # Esto es equivalente a z = z - 2


w = 4
w *= 6  # Esto es equivalente a w = w * 6


v = 12
```

```
v /= 3  # Esto es equivalente a v = v / 3


u = 20

u %= 7  # Esto es equivalente a u = u % 7
```

En este caso, se utilizan operadores de asignación compuestos junto con los operadores aritméticos básicos para actualizar el valor de la variable realizando la operación aritmética correspondiente y asignando el resultado a la misma variable.

Operadores de comparación

Los operadores de comparación en Python **se utilizan para comparar dos valores y devolver un resultado booleano (true o false) que indica si la comparación es verdadera o falsa.**

Igual que (==): Compara si dos valores son iguales. Devuelve true si son iguales y false si no lo son.

```
x = 5

y = 5

resultado = x == y  # El resultado será True
```

En el ejemplo, la comparación x == y devuelve true porque ambos valores son considerados iguales, a pesar de que uno es un número y el otro es una cadena de texto.

Diferente de (!=): Compara si dos valores no son iguales. Devuelve true si son diferentes y false si son iguales.

```
x = 5

y = 10


if x != y:
```

```
    print("x no es igual a y")
else:
    print("x es igual a y")
```

Otros operadores de comparación

Además de la igualdad y desigualdad, existen otros operadores de comparación como >, <, >= y <=, que comparan si un valor es mayor, menor, mayor o igual, o menor o igual, respectivamente.

```
let x = 5;
let y = 10;


print(x > y); // Devuelve false
print(x < y); // Devuelve true
print(x >= y); // Devuelve false
print(x <= y); // Devuelve true
```

En el ejemplo, la comparación x < y devuelve true porque el valor de x es menor que el valor de y.

Operadores Lógicos

Los operadores lógicos en Python se utilizan para combinar o invertir valores booleanos y realizar operaciones lógicas.

Operador and: Este operador devuelve True si ambas expresiones que combina son verdaderas.

```
a = True
b = False
resultado = a and b  # El resultado será False
```

Operador or: Este operador devuelve True si al menos una de las expresiones que combina es verdadera.

```
x = True

y = False

resultado = x or y  # El resultado será True
```

Operador NOT: Este operador invierte el valor de la expresión. Si la expresión es verdadera, not la hace falsa, y viceversa.

```
z = True

resultado = not z  # El resultado será False
```

Estos operadores lógicos te permiten combinar expresiones y realizar operaciones lógicas en Python. puedes utilizarlos para tomar decisiones basadas en múltiples condiciones o para invertir el valor de una expresión booleana.

Control de Flujo 🗃️

El control de flujo en Python se refiere a las estructuras y declaraciones que permiten dirigir el flujo de ejecución de un programa. Estas estructuras determinan qué secciones del código se ejecutarán y cuándo, en función de ciertas condiciones y reglas. Las principales estructuras de control de flujo en Python son:

Instrucciones Condicionales (If - Else - Elif):

- **if:** Permite ejecutar un bloque de código si una condición es verdadera.
- **else**: Permite ejecutar un bloque de código cuando la condición en un if no es verdadera.
- **elif** (abreviatura de "else if"): Permite verificar múltiples condiciones en secuencia.

El control de flujo es fundamental en la programación, ya que te permite tomar decisiones y controlar la ejecución de tu programa en función de condiciones específicas. Con las instrucciones condicionales y cíclicas, puedes crear programas más dinámicos y versátiles.

Además de las estructuras básicas de control de flujo que mencioné, acá hay algunas consideraciones y consejos adicionales:

Anidamiento: Puedes anidar estructuras de control de flujo, es decir, colocar una dentro de otra. Esto es útil para manejar situaciones más complejas que involucran múltiples condiciones.

Instrucciones Break y Continue: Python proporciona las instrucciones break y continue. break se utiliza para salir de un bucle antes de que se cumpla la condición de finalización, y continue se usa para saltar la iteración actual en un bucle y pasar a la siguiente.

Uso de Comentarios: Los comentarios son útiles para documentar tu código y explicar la lógica detrás de tus estructuras de control. Puedes utilizar el símbolo # para crear comentarios en Python.

Operadores Lógicos: Puedes combinar condiciones utilizando operadores lógicos como and, or y not. Estos operadores son útiles para crear condiciones más complejas en tus instrucciones condicionales.

Evitar Anidamiento Excesivo: Aunque es posible anidar muchas estructuras de control, un anidamiento excesivo puede dificultar la comprensión del código. Trata de mantener tus estructuras de control lo más simples y legibles posible.

Instrucciones Condicionales (If – Else –Elif)🗂️

Las instrucciones condicionales en Python, como en muchos otros lenguajes de programación, se utilizan para tomar decisiones en función de ciertas condiciones. En Python, las estructuras condicionales más comunes son if, else, y elif (abreviatura de "else if").

Instrucción if: La instrucción **if** se usa para ejecutar un bloque de código si una condición es verdadera. La sintaxis básica es la siguiente:

```
if condicion:
    # Bloque de código si la condición es verdadera
```

Ejemplo:

```
edad = 18

if edad >= 18:

   print("Eres mayor de edad.")
```

Instrucción else: La instrucción else se usa en combinación con if para ejecutar un bloque de código cuando la condición del if es falsa. La sintaxis es:

```
if condicion:

   # Bloque de código si la condición es verdadera

else:

   # Bloque de código si la condición es falsa
```

Ejemplo:

```
edad = 15

if edad >= 18:

   print("Eres mayor de edad.")

else:

   print("Eres menor de edad.")
```

Instrucción elif: La instrucción elif se utiliza cuando tienes múltiples condiciones que deseas comprobar en orden. Puedes tener varios bloques elif después del if y, opcionalmente, un bloque else al final. La sintaxis es:

```
if condicion1:

   # Bloque de código si condicion1 es verdadera

elif condicion2:

   # Bloque de código si condicion2 es verdadera
```

```
else:
    # Bloque de código si ninguna de las condiciones anteriores es verdadera
```

Ejemplo:

```
puntuacion = 85
if puntuacion >= 90:
   print("Aprobado con A.")
elif puntuacion >= 80:
   print("Aprobado con B.")
elif puntuacion >= 70:
   print("Aprobado con C.")
else:
   print("Reprobado.")
```

Las estructuras condicionales son esenciales para tomar decisiones en la programación y permiten que un programa responda de manera dinámica a diferentes situaciones. Podes anidar estructuras condicionales para manejar casos más complejos.

Instrucciones Cíclicas (While –For) 📑

Las instrucciones cíclicas (bucles) son una parte fundamental de la programación y permiten ejecutar un bloque de código repetidamente hasta que se cumpla una condición o se haya recorrido una colección de elementos. En Python, los bucles más comunes son while y for. Aquí tienes una descripción de ambos:

Instrucciones Cíclicas (While - For):

- **while:** Permite ejecutar un bloque de código repetidamente mientras una condición sea verdadera.
- **for:** Se utiliza para iterar sobre una secuencia (como una lista, tupla o rango) y ejecutar un bloque de código para cada elemento.

Bucle While:

La instrucción while permite ejecutar un bloque de código mientras se cumpla una condición. La condición se verifica antes de cada ejecución del bucle.

Sintaxis:

```
while condicion:
    # Código a ejecutar mientras la condición sea verdadera
```

Ejemplo:

```
contador = 0
while contador < 5:
    print("Iteración", contador)
    contador += 1
```

En este ejemplo, el bucle while se ejecuta mientras el valor de contador sea menor que 5.

La instrucción while se utiliza para ejecutar un bloque de código mientras se cumple una condición. La condición se verifica antes de cada iteración.

Bucle For:

La instrucción for se utiliza para recorrer elementos de una secuencia (como una lista, tupla, cadena de caracteres, etc.). Se ejecuta una vez para cada elemento en la secuencia.

Sintaxis:

```
for elemento in secuencia:
    # Código a ejecutar para cada elemento
```

Ejemplo:

```
colores = ["rojo", "verde", "azul"]
for color in colores:
    print("Color:", color)
```

En este caso, el bucle for recorre la lista de colores y ejecuta el bloque de código una vez para cada color.

La instrucción for se utiliza para recorrer elementos en una secuencia, como una lista, tupla, cadena de caracteres o cualquier objeto iterable.

Funciones range() y enumerate():

La función range() genera una secuencia de números y es comúnmente utilizada en bucles for.

La función enumerate() se usa para obtener tanto el índice como el valor de los elementos en un bucle for.

Control de Bucles:

Es posible controlar los bucles con las instrucciones break y continue. break se utiliza para salir prematuramente de un bucle, mientras que continue permite omitir la iteración actual y continuar con la siguiente.

Puedes utilizar las instrucciones break y continuar para controlar el flujo de los bucles.

break: Permite salir del bucle prematuramente, incluso si la condición no se ha cumplido.

Ejemplo:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Este bucle for imprimirá números del 0 al 4 y se detendrá cuando ***i*** sea igual a 5.

Continue: Permite saltar la iteración actual y continuar con la siguiente.

Ejemplo:

```
for i in range(10):
    if i == 5:
        continue
    print(i)
```

Este bucle for imprimirá números del 0 al 9, omitiendo el número 5.

Los bucles while y for son esenciales en la programación y se utilizan para automatizar tareas repetitivas. La elección de cuál usar depende de la situación y el contexto del problema que estás resolviendo.

Listas 🗂️

En Python, una lista es una estructura de datos que permite almacenar múltiples elementos. Es mutable, lo que significa que podes modificar su contenido. **Las listas se definen mediante corchetes []** y pueden contener elementos de diferentes tipos.

1. Creación de Listas:

```
mi_lista = [1, 2, 3, "cuatro", 5.0]
```

2. Acceso a Elementos:

Puedes acceder a los elementos de una lista mediante su índice, comenzando desde 0.

```
primer_elemento = mi_lista[0]
```

Listas y Strings

Las listas y las cadenas de texto son estructuralmente similares. Podes convertir una cadena en una lista y viceversa.

```
cadena = "Hola"
lista = list(cadena)
```

Listas como Pilas

Puedes utilizar el método append() para agregar elementos al final de la lista y pop() para eliminar el último elemento, convirtiendo la lista en una pila.

```
pila = [1, 2, 3]
pila.append(4)
elemento_elim = pila.pop()
```

Listas como Colas

Puedes usar el módulo collections para implementar colas eficientes.

```
from collections import deque
```

```
cola = deque([1, 2, 3])
cola.append(4)
elemento_elim = cola.popleft()
```

Listas por Comprensión

Es una forma concisa de crear listas.

```
cuadrados = [x**2 for x in range(5)]
```

👉**Ejemplo Práctico:**

```
mi_lista = [1, 2, 3, "cuatro", 5.0]
primer_elemento = mi_lista[0]

cadena = "Hola"
lista = list(cadena)

pila = [1, 2, 3]
pila.append(4)
elemento_elim_pila = pila.pop()

cola = deque([1, 2, 3])
cola.append(4)
elemento_elim_cola = cola.popleft()

cuadrados = [x**2 for x in range(5)]
```

Este ejemplo abarca la creación de listas, acceso a elementos, transformación entre listas y cadenas, y el uso de listas como pilas y colas. También muestra cómo utilizar listas por comprensión para generar una lista de cuadrados.

Las listas en Python son colecciones ordenadas y modificables. Son versátiles y pueden contener elementos de diferentes tipos.

Métodos Básicos

- Creación: lista = [1, 2, 3]
- Añadir Elementos: lista.append(4)
- Eliminar Elementos: lista.remove(2)
- Acceso y Slicing: primer_elemento = lista[0], sublista = lista[1:3]

Métodos Avanzados

- Ordenamiento: lista.sort()
- Inversión: lista.reverse()
- Comprensión de Listas: [x*2 for x in lista]

Tablas Hash (Diccionarios)

Los diccionarios en Python son una implementación de tablas hash. Son colecciones sin orden que tienen una clave y un valor.

Métodos Básicos

- Creación: diccionario = {'clave1': 'valor1', 'clave2': 'valor2'}
- Añadir Elementos: diccionario['clave3'] = 'valor3'
- Eliminar Elementos: del diccionario['clave2']
- Acceso a Elementos: valor = diccionario['clave1']

Métodos Avanzados

- Iteración: [clave for clave, valor in diccionario.items()]
- Obtener Todas las Claves/Valores: diccionario.keys(), diccionario.values()

Colas

Las colas son estructuras de datos tipo FIFO (First In, First Out).

Uso de queue.Queue

- Creación: from queue import Queue, cola = Queue()
- Enqueue (Añadir): cola.put(item)
- Dequeue (Eliminar y Retornar): item = cola.get()

Uso de Deque para Colas

- Creación: from collections import deque, cola = deque()
- Añadir/Remover Elementos: cola.append(item), cola.popleft()

Pilas

Las pilas son estructuras de datos tipo LIFO (Last In, First Out).

Implementación con Listas

- Creación: pila = []
- Push (Añadir): pila.append(item)
- Pop (Eliminar y Retornar): item = pila.pop()

Uso de Deque para Pilas

- Creación: pila = deque()
- Añadir/Remover Elementos: pila.append(item), pila.pop()

Ejemplos Prácticos

Ejemplo de Lista

- Creación y manipulación de una lista:

```
lista = [3, 1, 4, 1, 5]
lista.append(9)
print(lista)  # [3, 1, 4, 1, 5, 9]
lista.sort()
print(lista)  # [1, 1, 3, 4, 5, 9]
```

- Creación y uso de un diccionario:

```
diccionario = {'manzana': 10, 'banana': 5}
diccionario['naranja'] = 7
print(diccionario)   # {'manzana': 10, 'banana': 5, 'naranja':
7}
```

- Uso de queue.Queue:

```
from queue import Queue
cola = Queue()
cola.put('a')
cola.put('b')
print(cola.get())  # 'a'
```

- Implementación de una pila con una lista:

```
pila = []
pila.append('a')
pila.append('b')
print(pila.pop())  # 'b'
```

Búsqueda de elementos 📑

La búsqueda de elementos en una lista es una operación común en la programación. En Python, puedes utilizar diferentes métodos y técnicas para buscar elementos en una lista.

1. Utilizando el Operador in:

Puedes utilizar el operador in para verificar si un elemento está presente en una lista.

```
cuadrados = [x**2 for x in range(5)]
```

2. Utilizando el Método index():

El método index() devuelve el índice de la primera aparición de un elemento.

```
cuadrados = [x**2 for x in range(5)]
```

3. Utilizando la Función enumerate():

La función enumerate() permite obtener tanto el índice como el valor en cada iteración.

```
cuadrados = [x**2 for x in range(5)]
```

Ejemplo Práctico:

```
cuadrados = [x**2 for x in range(5)]
```

Este ejemplo muestra diferentes enfoques para buscar elementos en una lista, utilizando el operador in, el método index() y la función enumerate().

Ordenamiento 📑

El ordenamiento es un proceso esencial en programación que organiza los elementos de una lista de acuerdo con ciertos criterios. Python ofrece métodos y funciones para realizar esta tarea de manera eficiente.

1. Método sort():

El método sort() ordena los elementos de una lista directamente. Puedes especificar el orden ascendente (reverse=False) o descendente (reverse=True).

```
mi_lista = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
mi_lista.sort()
print("Lista ordenada en orden ascendente:", mi_lista)

mi_lista.sort(reverse=True)
print("Lista ordenada en orden descendente:", mi_lista)
```

2. Función sorted():

La función sorted() devuelve una nueva lista ordenada sin modificar la lista original. Al igual que sort(), podes especificar el orden.

```
mi_lista = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
lista_ordenada_ascendente = sorted(mi_lista)
print("Lista ordenada en orden ascendente:", lista_ordenada_ascendente)

lista_ordenada_descendente = sorted(mi_lista, reverse=True)
print("Lista ordenada en orden descendente:",
lista_ordenada_descendente)
```

Ejemplo Práctico:

```
mi_lista = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

# Utilizando el método sort()
mi_lista.sort()
print("Lista ordenada en orden ascendente:", mi_lista)

mi_lista.sort(reverse=True)
print("Lista ordenada en orden descendente:", mi_lista)

# Utilizando la función sorted()
lista_ordenada_ascendente = sorted(mi_lista)
print("Lista ordenada en orden ascendente:", lista_ordenada_ascendente)

lista_ordenada_descendente = sorted(mi_lista, reverse=True)
print("Lista ordenada en orden descendente:",
lista_ordenada_descendente)
```

Este ejemplo muestra cómo ordenar una lista utilizando tanto el método sort() como la función sorted(), con opciones para orden ascendente y descendente.

Matrices 🗃️

Una matriz es una estructura bidimensional que contiene elementos organizados en filas y columnas. En Python, puedes representar matrices utilizando listas anidadas.

1. Creación de una Matriz:

Puedes crear una matriz utilizando listas anidadas. Cada lista interna representa una fila de la matriz.

```
lista_ordenada_descendente)
```

2. Acceso a Elementos:

Acceder a los elementos de la matriz se realiza mediante índices. Por ejemplo, mi_matriz[1][2] accede al elemento en la segunda fila y tercera columna.

3. Operaciones con Matrices:

Realizar operaciones como suma, resta y multiplicación de matrices puede hacerse con bucles y listas.

Ejemplo Práctico:

```
lista_ordenada_descendente)
```

Este ejemplo ilustra la creación de una matriz, el acceso a elementos y la suma de dos matrices.

Tuplas 

Las tuplas en Python son un tipo o estructura de datos que permite almacenar datos de una manera muy parecida a las listas, con la salvedad de que son inmutables.

Crear tupla Python

Las tuplas en Python o tuples son muy similares a las listas, pero con dos diferencias. Son inmutables, lo que significa que no pueden ser modificadas una vez declaradas, y en vez de inicializarse con corchetes se hace con (). Dependiendo de lo que queramos hacer, las tuplas pueden ser más rápidas.

```
tupla=(1,2,3)
print(tupla)#(1, 2, 3)
```

También pueden declararse sin (), separando por, todos sus elementos.

```
tupla=1,2,3
print(type(tupla))#<class 'tuple'>
print(tupla)#(1, 2, 3)
```

Trabajando con tuplas 🗳️

Como hemos comentado, las tuplas son tipos inmutables, lo que significa que una vez asignado su valor, no puede ser modificado. Si se intenta, tendremos un TypeError.

```
tupla=(1,2,3)
#tupla[0] = 5 # Error! TypeError
```

Al igual que las listas, las tuplas también pueden ser anidadas.

```
tupla=1,2,('a','b'),3
print(tupla)#(1, 2, ('a', 'b'), 3)
print(tupla[2][0])#a
```

Y también es posible convertir una lista en tupla haciendo uso de al función tuple().

```
lista=[1,2,3]
tupla=tuple(lista)
print(type(tupla))#<class 'tuple'>
print(tupla)#(1, 2, 3)
```

Se puede **iterar** una tupla de la misma forma que se hacía con las listas.

```
tupla=[1,2,3]
for t in tupla:
print(t)#1, 2, 3
```

Y se puede también asignar el valor de una tupla con **n** elementos a **n** variables.

```
l=(1,2,3)
x,y,z=l
print(x,y,z)#1 2 3
```

Aunque tal vez no tenga mucho sentido a nivel práctico, es posible crear una tupla de un solo elemento. Para ello debes usar , antes del paréntesis, porque de lo contrario (2) sería interpretado como int.

```
tupla=(2,)
print(type(tupla))#<class 'tuple'>
```

El método count() cuenta el número de veces que el objeto pasado como parámetro se ha encontrado en la lista.

```
l=[1,1,1,3,5]
print(l.count(1))#3
```

El método index() busca el objeto que se le pasa como parámetro y devuelve el índice en el que se ha encontrado.

```
l=[7,7,7,3,5]
print(l.index(5))#4
```

En el caso de no encontrarse, se devuelve un ValueError.

```
l=[7,7,7,3,5]
#print(l.index(35)) #Error! ValueError
```

El método index() también acepta un segundo parámetro opcional, que indica a partir de que índice empezar a buscar el objeto.

```
l=[7,7,7,3,5]
print(l.index(7,2))#2
```

Empaquetado y desempaquetado de tuplas 🗳️

En Python, el empaquetado y desempaquetado de tuplas son técnicas que permiten asignar múltiples valores en una sola instrucción o extraer valores de una tupla de manera individual.

1. Empaquetado de Tuplas:

Empaquetar una tupla significa asignar varios valores a una sola variable en forma de tupla.

```
mi_tupla_empaquetada = 1, 2, 3
```

En este ejemplo, mi_tupla_empaquetada es una tupla que contiene los valores 1, 2 y 3.

2. Desempaquetado de Tuplas:

Desempaquetar una tupla significa asignar los valores individuales de una tupla a variables separadas.

```
a, b, c = mi_tupla_empaquetada
```

Ahora, a tiene el valor 1, b tiene el valor 2 y c tiene el valor 3.

Ejemplo Práctico:

```
# Empaquetado
mi_tupla_empaquetada = 4, 5, 6

# Desempaquetado
x, y, z = mi_tupla_empaquetada

print("Tupla Empaquetada:", mi_tupla_empaquetada)
print("Desempaquetado - x:", x)
print("Desempaquetado - y:", y)
print("Desempaquetado - z:", z)
```

Este ejemplo ilustra cómo empaquetar una tupla y luego desempaquetarla para asignar sus valores a variables individuales.

Diccionarios 

En Python, un diccionario es una estructura de datos que almacena pares clave-valor, permitiendo el acceso eficiente a los valores a través de sus claves.

Diccionarios y sus Claves:

```
mi_tupla_empaquetada = 1, 2, 3
```

En este ejemplo, mi_diccionario es un diccionario con tres pares clave-valor.

2. Métodos de los Diccionarios:

- get(): Obtiene el valor asociado a una clave.
- keys(): Devuelve una lista con todas las claves del diccionario.
- values(): Devuelve una lista con todos los valores del diccionario.
- items(): Devuelve una lista de tuplas (clave, valor).

3. Iteración de un Diccionario:

```
mi_tupla_empaquetada = 1, 2, 3
```

Este código itera sobre el diccionario, proporcionando tanto la clave como el valor en cada iteración.

Ejemplo Práctico:

```
mi_tupla_empaquetada = 1, 2, 3
mi_tupla_empaquetada = 1, 2, 3
```

Trabajando con diccionarios 🗳️

Agregar y Modificar Elementos:

```
mi_diccionario = {'clave1': 'valor1', 'clave2': 'valor2'}

# Agregar un nuevo elemento
mi_diccionario['clave3'] = 'valor3'

# Modificar un valor existente
mi_diccionario['clave1'] = 'nuevo_valor'
```

2. Eliminar Elementos:

```
# Eliminar un elemento por clave
```

```
del mi_diccionario['clave2']

# Eliminar y obtener un elemento
valor_eliminado = mi_diccionario.pop('clave1')
```

3. Comprobación de Claves:

```
# Verificar si una clave está en el diccionario
if 'clave1' in mi_diccionario:
    print('La clave existe en el diccionario')
```

4. Longitud del Diccionario:

```
# Obtener la cantidad de elementos en el diccionario
longitud = len(mi_diccionario)
```

Ejemplo Práctico:

```
# Crear un diccionario de contactos
contactos = {
    'Juan': {'telefono': '123456789', 'email': 'juan@email.com'},
    'María': {'telefono': '987654321', 'email': 'maria@email.com'}
}

# Agregar un nuevo contacto
contactos['Carlos'] = {'telefono': '555555555', 'email': 'carlos@email.com'}

# Modificar el teléfono de María
contactos['María']['telefono'] = '111111111'

# Eliminar a Juan de los contactos
del contactos['Juan']

# Mostrar información actualizada
for nombre, detalles in contactos.items():
```

```
        print(f'Contacto: {nombre}, Teléfono: {detalles["telefono"]}, Email: {detalles["email"]}')
```

Este ejemplo ilustra la manipulación de un diccionario de contactos, incluyendo agregar, modificar y eliminar elementos.

Métodos de los diccionarios 🗳️

Los diccionarios en Python no solo almacenan datos sino que también proporcionan métodos y técnicas para trabajar eficientemente con ellos. En esta sección, exploraremos algunos métodos comunes de diccionarios y cómo iterar a través de ellos.

1. Métodos de Diccionarios:

- keys(): Retorna una vista de todas las claves en el diccionario.
- values(): Retorna una vista de todos los valores en el diccionario.
- items(): Retorna una vista de tuplas (clave, valor) en el diccionario.
- get(clave, valor_default): Retorna el valor asociado con la clave o un valor predeterminado si la clave no existe.

```
mi_diccionario = {'clave1': 'valor1', 'clave2': 'valor2'}

# Obtener todas las claves
claves = mi_diccionario.keys()

# Obtener todos los valores
valores = mi_diccionario.values()

# Obtener pares clave-valor
pares = mi_diccionario.items()

# Obtener un valor con clave o valor predeterminado si no existe
valor = mi_diccionario.get('clave3', 'valor_default')
```

Iteración de los diccionarios 🗳️

```
mi_diccionario = {'clave1': 'valor1', 'clave2': 'valor2'}

# Iterar sobre las claves
for clave in mi_diccionario:
    print(clave)

# Iterar sobre los valores
for valor in mi_diccionario.values():
    print(valor)

# Iterar sobre pares clave-valor
for clave, valor in mi_diccionario.items():
    print(f'Clave: {clave}, Valor: {valor}')
```

Ejemplo Práctico:

```
# Crear un diccionario de notas
notas = {'Juan': 85, 'María': 92, 'Carlos': 78, 'Ana': 95}

# Calcular el promedio de las notas
promedio = sum(notas.values()) / len(notas)

# Imprimir nombres de estudiantes con notas superiores al promedio
for estudiante, nota in notas.items():
    if nota > promedio:
 print(f'{estudiante} tiene una nota superior al promedio.')
```

Este ejemplo muestra cómo utilizar métodos de diccionarios y realizar iteraciones para analizar las notas de los estudiantes. Los diccionarios ofrecen una flexibilidad significativa en la manipulación de datos en Python.

Claves 🔐

En Python, los diccionarios son estructuras de datos que almacenan pares clave-valor. Las claves son elementos únicos e inmutables que se utilizan para acceder a los valores asociados. En esta sección, exploraremos la importancia de las claves en los diccionarios de Python.

Unicidad e Inmutabilidad:

Las claves deben ser únicas en un diccionario, ya que proporcionan la forma de identificar y acceder a un valor específico.

Las claves deben ser inmutables, lo que significa que no pueden cambiar después de haber sido definidas. Esto garantiza que las claves conserven su identidad única.

Ejemplos de Claves:

Las claves pueden ser de diversos tipos, como cadenas, números, tuplas, etc. En un diccionario, las claves no pueden repetirse.

```
# Ejemplo de diccionario con claves de diferentes tipos
mi_diccionario = {'nombre': 'Juan', 42: 'respuesta', (1, 2): 'tupla'}

# Intento de definir un diccionario con claves duplicadas (error)
diccionario_erroneo = {'clave': 1, 'clave': 2}  # Generará un error de sintaxis
```

Acceso y Modificación:

Para acceder a un valor en un diccionario, se utiliza la clave correspondiente. Las claves permiten modificar los valores asociados.

```
# Acceso y modificación de valores utilizando claves
mi_diccionario = {'nombre': 'Juan', 'edad': 25, 'puntaje': 90}

# Acceder a un valor mediante clave
nombre = mi_diccionario['nombre']

# Modificar el valor asociado a una clave
mi_diccionario['edad'] = 26
```

Ejemplo Práctico:

```
# Crear un diccionario de productos y sus precios
productos = {'manzanas': 2.5, 'plátanos': 1.8, 'uvas': 3.2}

# Acceder al precio de las uvas mediante su clave
precio_uvas = productos['uvas']

# Modificar el precio de las manzanas
productos['manzanas'] = 2.0

# Imprimir el diccionario actualizado
print(productos)
```

En este ejemplo, las claves (manzanas, plátanos, uvas) permiten acceder y modificar los precios asociados en el diccionario de productos. Las claves desempeñan un papel fundamental en la eficacia y versatilidad de los diccionarios en Python.

Funciones 📑

Las funciones desempeñan un papel fundamental en Python y en la programación en general. Aquí te proporcionaré una explicación más detallada sobre las funciones en Python:

Importancia de las funciones en Python:

Las funciones son bloques de código reutilizables que realizan tareas específicas.

Permiten dividir un programa en partes más pequeñas y manejables, lo que hace que el código sea más organizado y mantenible.

Promueven la reutilización del código, lo que significa que puedes usar la misma función en diferentes partes de tu programa sin tener que volver a escribirla.

Ayudan a modularizar el código, lo que facilita el trabajo en equipo, ya que diferentes desarrolladores pueden trabajar en funciones individuales.

- **Ámbito de variables:** Las variables definidas dentro de una función tienen un ámbito local, lo que significa que solo son visibles dentro de la función. Si intentas acceder a una variable local desde fuera de la función, obtendrás un error. Sin embargo, puedes acceder a variables globales desde dentro de una función si las declaras como global.
- **Funciones anidadas:** Puedes definir funciones dentro de otras funciones. Estas funciones anidadas son útiles cuando deseas encapsular lógica específica dentro de una función externa. Las funciones anidadas tienen acceso al ámbito de la función externa.
- **Funciones lambda:** Las funciones lambda (también conocidas como funciones anónimas) son funciones pequeñas y anónimas que pueden tener cualquier número de argumentos pero solo una expresión. Se utilizan comúnmente para operaciones simples y se definen con la palabra clave lambda.
- **Recursión:** Python permite la recursión, lo que significa que una función puede llamarse a sí misma. La recursión es útil para resolver problemas que se pueden descomponer en casos más pequeños.
- **Parámetros predeterminados:** Puedes asignar valores predeterminados a los parámetros de una función. Esto permite que la función se llame con menos argumentos, ya que los valores predeterminados se utilizan en caso de que no se proporcionen argumentos.

Definición de funciones

Para definir una función en Python, utilizamos la palabra clave def, seguida del nombre de la función y paréntesis que pueden contener parámetros.

La definición de una función se inicia con dos puntos : y se requiere la indentación del código que forma el cuerpo de la función.

Sintaxis:

```
def nombre_de_la_funcion(parametro1, parametro2):
    # Código de la función
    # Puede incluir declaraciones return para devolver valores
```

Ejemplo de una función simple que suma dos números:

```
def suma(a, b):
    resultado = a + b
    return resultado
```

En este ejemplo, la función suma toma dos parámetros a y b, realiza la suma y devuelve el resultado.

Para llamar a una función, simplemente escribimos su nombre seguido de paréntesis y, si es necesario, proporcionamos los argumentos que requiere. Por ejemplo:

```
resultado = suma(5, 3)
print(resultado)  # Esto imprimirá 8
```

Las funciones son un concepto esencial en Python y te permiten dividir tu código en unidades lógicas y reutilizables, lo que facilita el desarrollo de programas complejos.

Las funciones anidadas, también conocidas como funciones internas o funciones dentro de funciones, son funciones que se definen dentro del cuerpo de otra función en Python. Estas funciones anidadas son accesibles solo dentro del ámbito de la función en la que están definidas. Aquí tienes algunas características y usos importantes de las funciones anidadas:

- **Ámbito local:** Las funciones anidadas tienen un ámbito local, lo que significa que solo pueden ser llamadas y utilizadas dentro de la función que las contiene. No pueden ser invocadas desde fuera de esa función.
- **Encapsulación de lógica:** Las funciones anidadas son útiles para encapsular lógica específica dentro de una función externa. Esto puede ayudar a modularizar y organizar el código, ya que permite definir funciones internas que son relevantes solo para la función principal.
- **Reutilización de código:** Puedes reutilizar las funciones anidadas en múltiples lugares dentro de la función principal sin necesidad de

copiar y pegar código. Esto promueve la reutilización de código y reduce la duplicación.

- **Acceso a variables externas**: Las funciones anidadas pueden acceder a las variables locales y argumentos de la función externa en la que están definidas. Esto permite que las funciones anidadas utilicen datos de la función contenedora.

- **Cierre léxico:** Las funciones anidadas tienen acceso al entorno léxico (ámbito) de la función que las rodea. Esto significa que pueden acceder a variables y argumentos de la función externa incluso después de que la función principal haya finalizado su ejecución.

Aquí tienes un ejemplo de cómo se define y utiliza una función anidada en Python:

```
def funcion_externa(x):
    def funcion_anidada(y):
        return x + y
# Accede a la variable x de la función externa

    resultado = funcion_anidada(10)
# Llama a la función anidada con argumento 10
    return resultado

print(funcion_externa(5))
# Imprime 15, ya que x=5 y y=10
```

En este ejemplo, funcion_anidada se define dentro de funcion_externa y tiene acceso a la variable x de la función externa. La función anidada se llama con el argumento 10, y el resultado se devuelve desde la función principal. Esto ilustra cómo las funciones anidadas pueden ser útiles para organizar y reutilizar código dentro de una función.

El ámbito de las variables, también conocido como alcance de las variables, se refiere a la región del código en la que una variable es válida y puede ser utilizada. En Python, el ámbito de una variable depende de cómo se declara. Aquí hay dos tipos principales de ámbito de variables en Python:

Ámbito Local (Local Scope): Una variable declarada dentro de una función tiene un ámbito local. Esto significa que solo es accesible dentro de la función en la que se declara. No se puede acceder a ella desde fuera de la función. Las variables locales se utilizan principalmente para almacenar datos temporales o intermedios en el contexto de la función.

```
def mi_funcion():
    variable_local = 10
    print(variable_local)

mi_funcion()  # Esto imprimirá 10
print(variable_local)  # Esto generará un error, ya que variable_local no es accesible aquí
```

Ámbito Global (Global Scope): Una variable declarada fuera de cualquier función, generalmente en el nivel superior del programa, tiene un ámbito global. Esto significa que es accesible en todo el programa, tanto dentro como fuera de las funciones. Las variables globales son útiles para almacenar datos que deben ser compartidos y utilizados en múltiples partes del código.

```
variable_global = 5

def mi_funcion():
    print(variable_global)
# Puedo acceder a variable_global aquí

mi_funcion()
```

```
print(variable_global)

# Puedo acceder a variable_global desde fuera de la función
```

Es importante tener en cuenta que si intentas modificar una variable global dentro de una función, Python asumirá que estás creando una nueva variable local en lugar de modificar la variable global. Para modificar una variable global desde una función, debes declararla explícitamente como global dentro de la función.

```
variable_global = 5


def modificar_variable():
    global variable_global
# Indica que queremos modificar la variable global
    variable_global = 10


modificar_variable()
print(variable_global)
# Ahora variable_global es 10
```

El ámbito de las variables es un concepto importante en la programación, ya que ayuda a controlar el acceso y la visibilidad de las variables en diferentes partes del código.

Importación y llamado de módulos 📑

En Python, los módulos son archivos que contienen definiciones y declaraciones de Python. Importar y llamar módulos es esencial para reutilizar código y ampliar las capacidades de Python.

Para importar un módulo en Python, utilizamos la palabra clave import seguida del nombre del módulo. Python busca el módulo en su ruta de

búsqueda de módulos. Una vez importado, puedes acceder a las funciones y variables definidas en el módulo utilizando la notación de punto. Por ejemplo, si deseas utilizar la función sqrt del módulo math, puedes hacerlo de la siguiente manera:

```
import math


resultado = math.sqrt(16)

# Utilizando la función sqrt del módulo math
```

Supongamos que deseas trabajar con fechas y horas en Python. Puedes importar el módulo datetime para acceder a funciones relacionadas con fechas y horas. Ejemplo:

```
import datetime


hoy = datetime.date.today()

# Obtiene la fecha actual

print("Fecha actual:", hoy)


ahora = datetime.datetime.now()

# Obtiene la fecha y hora actual

print("Fecha y hora actual:", ahora)
```

Invocación y rango de una función 📑

Las funciones en Python son bloques de código que realizan tareas específicas. Para utilizar una función, debes invocarla y proporcionar los argumentos necesarios. El rango de una función se refiere a las variables y datos que están disponibles dentro de la función.

Desarrollo: Cuando invocas una función, Python ejecuta el bloque de código dentro de la función, procesa los argumentos que le proporcionas y puede

devolver un resultado. El rango de una función incluye todas las variables definidas dentro de la función. Estas variables se denominan variables locales y no son accesibles desde fuera de la función.

Consideremos una función simple que suma dos números:

```
def suma(a, b):
    resultado = a + b
    return resultado


resultado_suma = suma(5, 3)
# Llamamos a la función suma con los argumentos 5 y 3
print("Resultado de la suma:", resultado_suma)
```

En este ejemplo, a, b y resultado son variables locales dentro de la función suma. El resultado de la suma se devuelve y se almacena en resultado_suma.

Funciones como módulos 📑

En Python, las funciones pueden definirse en módulos separados y luego importarse como cualquier otro módulo. Esto permite organizar y reutilizar funciones de manera eficiente.

Puedes crear tus propios módulos Python que contengan funciones y variables. Luego, puedes importar esas funciones en otros programas para utilizarlas. Esto es útil cuando deseas reutilizar código en múltiples proyectos o colaborar con otros desarrolladores.

Supongamos que has definido un módulo llamado mis_utilidades.py que contiene una función llamada saludar. Puedes importar esta función en otro programa y utilizarla de la siguiente manera:

```
# En mis_utilidades.py
def saludar(nombre):
    return "Hola, " + nombre + "!"
```

```
# En otro programa
import mis_utilidades

mensaje = mis_utilidades.saludar("Juan")
print(mensaje)  # Esto imprimirá "Hola, Juan!"
```

En este ejemplo, mis_utilidades es un módulo que contiene la función saludar, que luego se importa y utiliza en otro programa.

Cierre

En este manual de Fundamentos de Python, hemos cubierto una variedad de temas fundamentales que sientan las bases para tu viaje en el mundo de la programación con Python.

Comenzamos explorando la sintaxis de Python, aprendiendo cómo escribir y ejecutar programas. Aprendiste sobre variables, tipos de datos y cómo manipularlos en Python. Además, profundizamos en las estructuras de control, como las instrucciones condicionales (if-else) y los bucles (for y while), lo que te permitió controlar el flujo de ejecución de tus programas.

Continuamos estudiando las funciones en Python y cómo crear tus propias funciones para encapsular bloques de código reutilizables. Aprendiste sobre los parámetros de las funciones, los valores de retorno y cómo modularizar tu código en unidades más pequeñas y manejables.

Exploramos también las listas, que son estructuras de datos muy útiles para almacenar y manipular conjuntos de valores relacionados. Aprendiste a trabajar con índices y rebanadas, a agregar, eliminar y modificar elementos en las listas.

Referencias

- Python Software Foundation. (s.f.). Documentación de Python. https://docs.python.org
- Python Software Foundation. (s.f.). Python.org. https://www.python.org
- Matthes, E. (2019). Python Crash Course (2nd ed.). No Starch Press.
- Ramalho, L. (2015). Fluent Python. O'Reilly Media.

¡Muchas gracias!

Nos vemos en la próxima unidad 👋