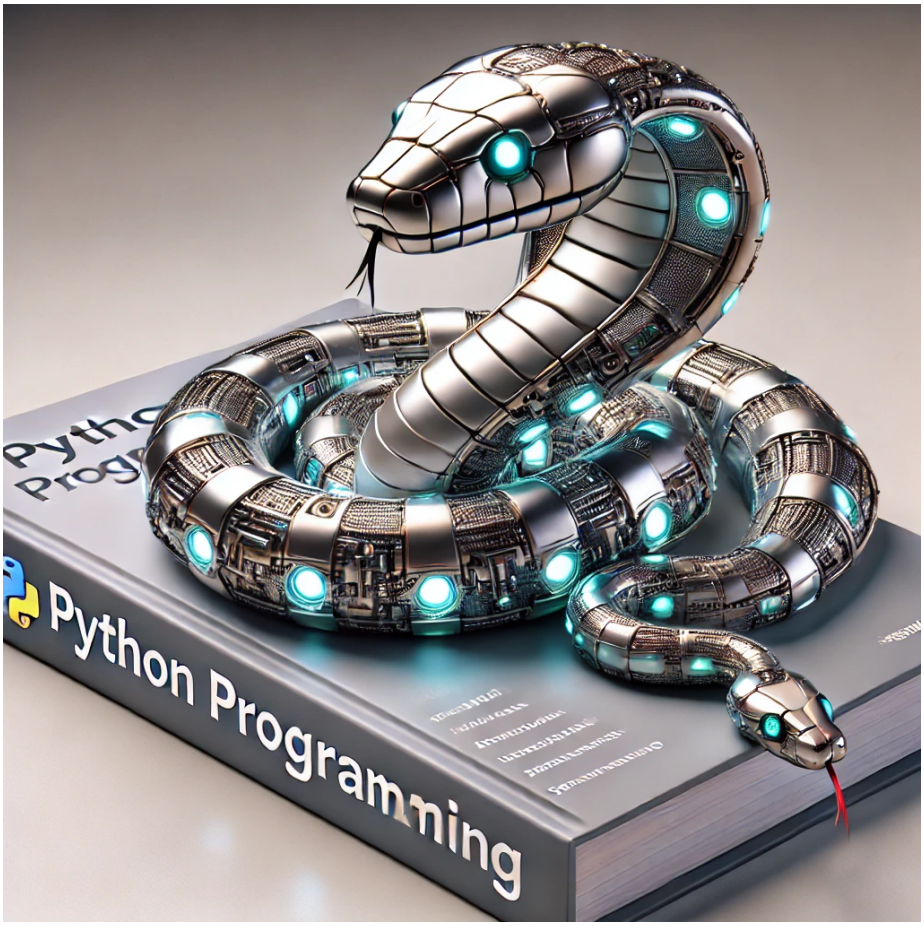
UNIDAD 2

EL LENGUAJE DE PROGRAMACIÓN PYTHON

Programación de Inteligencia Artificial

Curso de Especialización en Inteligencia Artificial y Big Data



CHATGPT prompt: Crea una imagen con un libro sobre programación en Python y que tenga sobre él una serpiente pitón robotizada

Carlos M. Abrisqueta Valcárcel
IES Ingeniero de la Cierva 2024/25

ÍNDICE

1. INTRODUCCIÓN A PYTHON

Python, concebido por Guido van Rossum y liberado en 1991, ha sido desarrollado bajo una filosofía que enfatiza la simplicidad y la legibilidad del código. Su sintaxis permite a los programadores expresar conceptos en menos líneas de código que en lenguajes como C++ o Java.

1.2. FILOSOFÍA: EL ZEN DE PYTHON

El Zen de Python, escrito por Tim Peters, enfatiza la belleza y la simplicidad del código. Algunos de sus aforismos incluyen: "Lo explícito es mejor que lo implícito", "La simplicidad es mejor que la complejidad". Estos principios guían el diseño y desarrollo del lenguaje. Las características de python son las siguientes:

Tipado Dinámico: Python es un lenguaje de tipado dinámico, lo que significa que el tipo de dato de una variable se determina en tiempo de ejecución, permitiendo una mayor flexibilidad al asignar y manipular variables.

Soporte Multiparadigma: Python es un lenguaje de programación multiparadigma que soporta programación orientada a objetos, imperativa y funcional, ofreciendo así diversas opciones para resolver problemas y desarrollar sistemas.

Portabilidad: La portabilidad de Python se refiere a la capacidad del lenguaje para ejecutarse en diversas plataformas y sistemas operativos sin necesidad de realizar modificaciones significativas en el código fuente.

Garbage Collection: Python cuenta con un recolector de basura ("garbage collector") que gestiona automáticamente la memoria, liberando aquellos objetos que ya no están siendo utilizados y asegurando que la memoria utilizada sea la mínima necesaria.

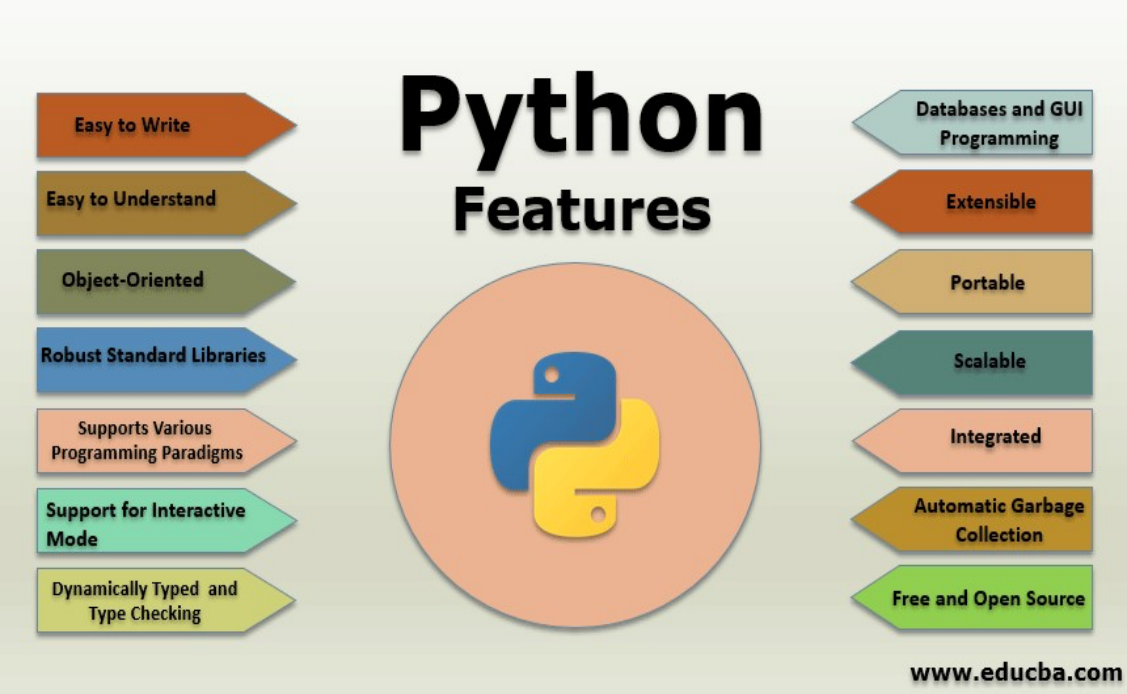


Figura 1: Las características de Python

1.3. HISTÓRICO DE VERSIONES

1.3.1. Python 2.0

Lanzado en el año 2000, Python 2.0 introdujo varias nuevas características, tales como la recolección de basura y soporte completo para Unicode. Durante su desarrollo, se colocó un énfasis significativo en la retrocompatibilidad con versiones anteriores.

Características notables:

- List comprehensions: proporcionando una sintaxis más concisa para crear listas.
- Recolección de basura: sistema de gestión automática de memoria.

1.3.2. Python 3.0

Python 3.0, lanzado en 2008, fue diseñado para rectificar los defectos percibidos del lenguaje. No se diseñó para ser compatible con las versiones anteriores, marcando un cambio importante en la evolución del lenguaje.

Características notables:

- `print()` se convierte en una función, alterando su uso en comparación con Python 2.
- Introducción de la sintaxis "nonlocal" para usar variables no locales en funciones anidadas.
- Nuevos operadores de división: / realiza la división clásica y // la división entera.

1.3.3. Python 3.6

Python 3.6, lanzado en 2016, introdujo varias características nuevas, optimizaciones y actualizaciones.

Características notables:

- Formateo de cadenas de texto (f-strings): una nueva forma de incrustar expresiones dentro de literales de cadena usando `f"texto {expresion}"`.
- Tipo de dictado y anotaciones de variables: permitiendo la especificación explícita de los tipos de datos.

1.3.4. Python 3.8

Lanzado en 2019, Python 3.8 trajo varias optimizaciones y algunas características nuevas significativas.

Características notables:

- Asignación con expresión (operador Walrus): permite asignar valores a variables como parte de una expresión usando la sintaxis `NAME := expr`.
- Protocolos de tipos: soporte mejorado para tipado estático y verificaciones de tipos en tiempo de ejecución.

1.3.5. Python 3.9

La versión 3.9 del lenguaje, lanzada en 2020, introdujo varias nuevas sintaxis y actualizaciones de la API.

Características notables:

- Operadores de fusión y actualización de diccionarios, que permiten unir y actualizar diccionarios con `|` y `|=`.
- Análisis sintáctico de Zonas Horarias: El módulo zoneinfo trae soporte para el análisis de zonas horarias IANA al estándar de la biblioteca.

1.3.6. Python 3.10

Lanzada en octubre de 2021, introdujo varias nuevas sintaxis y actualizaciones de la API.

Características notables:

- Pattern matching: Se introduce el equivalente a `switch/case`, pero con más potencia gracias a la sintaxis `match-case`.
- Introducción de las union-types

1.3.7. Python 3.11

Lanzado en octubre de 2022, Python 3.11 se enfocó en mejorar el rendimiento y la simplicidad en el manejo de excepciones y tipos.

Características notables:

- Mejoras en el rendimiento: Python 3.11 ofrece mejoras en el rendimiento, con una aceleración general de las ejecuciones en torno al 10-60% en varios casos de uso.
- Nuevas excepciones: Se introduce ExceptionGroup, una nueva clase que permite agrupar múltiples excepciones en una sola, y except* para manejar excepciones agrupadas.

1.3.8. Python 3.12

Lanzado en octubre de 2023, esta versión se centró en seguir mejorando el rendimiento y refinar la funcionalidad de los tipos y las excepciones introducidas en versiones anteriores.

Características notables:

- Iteración y rendimiento: Se continúan optimizando algunas operaciones comunes para que sean más rápidas, como la concatenación de cadenas y el manejo de estructuras de datos grandes.
- Depuración avanzada: Mejoras en las herramientas de depuración, que permiten un análisis más detallado del código, especialmente en entornos de desarrollo.

1.3.9. Python 3.13

Lanzado en Octubre de 2024, Python 3.13 introduce varias funcionalidades que mejoran la experiencia del desarrollador, optimizan el rendimiento y amplían el soporte de plataformas.

Características notables:

- Una de las actualizaciones más notables es el nuevo intérprete interactivo, inspirado en PyPy. Ahora cuenta con edición de líneas múltiples, soporte para color y trazas de excepción colorizadas, lo cual facilita la depuración y mejora la experiencia de codificación en tiempo real.
- Compatibilidad y deprecaciones: Algunos métodos y funciones que han sido reemplazados en las últimas versiones comienzan a ser oficialmente eliminados, mejorando la consistencia del lenguaje a largo plazo.

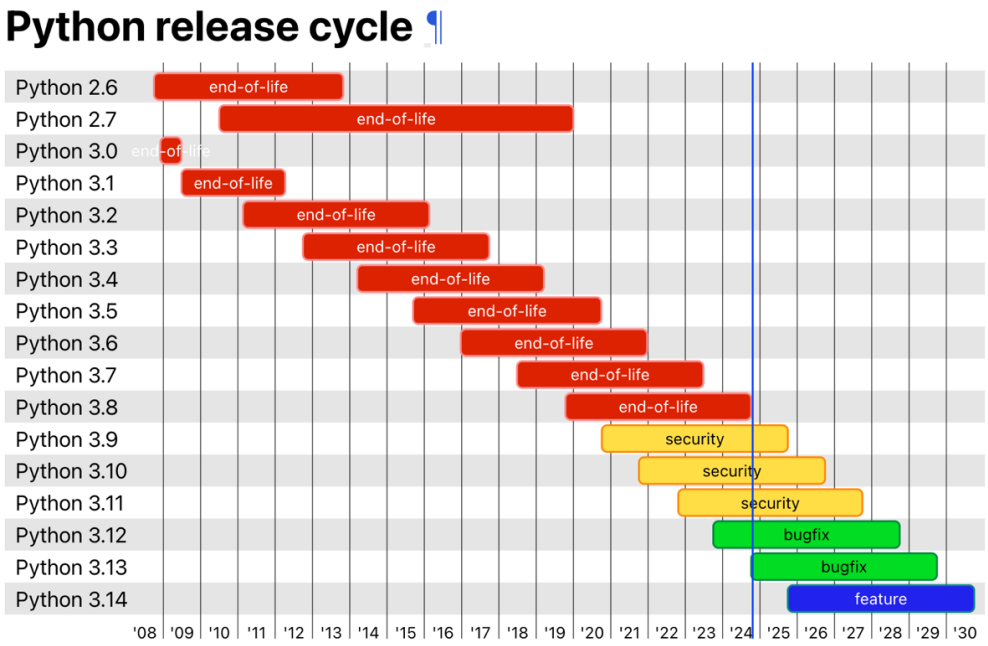


Figura 2: Evolución de las versiones de python

1.4. ENTORNOS DE DESARROLLO PARA PYTHON

Existen diversos IDEs y distribuciones que facilitan la programación en Python, proporcionando herramientas útiles para desarrollo, prueba y depuración de código.

1.4.1. Anaconda

Anaconda es una distribución gratuita y de código abierto que busca simplificar el manejo de paquetes y su despliegue. Su plataforma conda facilita la gestión y despliegue de software en diferentes sistemas operativos.

1.4.2. PyCharm

Desarrollado por JetBrains, PyCharm es un IDE utilizado comúnmente para desarrollo en Python. Ofrece potentes herramientas para desarrollo web, ciencia de datos, inteligencia artificial y desarrollo de software en general. Con tu cuenta de alumno, tienes gratuitamente una licencia de desarrollo, tan sólo tienes que utilizar el correo electrónico @alu.murciaeduca.es a la hora de descargarlo.

1.4.3. Jupiter Notebooks y Google Colab

Jupyter Notebooks es un entorno de desarrollo que facilita la ciencia de datos al permitir combinar código, gráficos y texto en un mismo documento. Google Colab, por su parte, permite ejecutar estos notebooks en la nube, facilitando el acceso a hardware potente y compartición de trabajo.

1.4.4. Importancia de los entornos virtuales (envs) en Python

Los entornos virtuales, también conocidos por sus abreviaciones env o venv, han surgido como una solución esencial para gestionar dependencias y versiones en los proyectos de Python. Un env en Python es un espacio aislado donde se pueden instalar paquetes y mantener dependencias sin interferir con el sistema de Python global o con otros proyectos.

Utilidades y ventajas:

- **Gestión de Dependencias**: Los envs permiten tener un control preciso sobre las versiones de los paquetes utilizados en un proyecto, asegurando que las dependencias documentadas son coherentes y reproducibles.
- **Aislamiento**: Cada proyecto puede tener sus propios requisitos y, gracias a los envs, es posible mantenerlos aislados para evitar conflictos entre versiones de paquetes.
- **Compatibilidad**: Al utilizar envs, los desarrolladores pueden trabajar con diferentes versiones de Python y sus respectivas bibliotecas sin enfrentarse a incompatibilidades.
- **Facilita la Colaboración**: Los colaboradores de un proyecto pueden replicar fácilmente el entorno de desarrollo al utilizar un env con las dependencias especificadas.

1.4.5. Creación y gestión de entornos virtuales con Conda.

Conda es un sistema de gestión de paquetes y también un sistema de gestión de entornos, proporcionando herramientas para instalar, correr y actualizar bibliotecas y dependencias dentro de diferentes entornos en tu sistema de desarrollo.

Ventajas de Conda:

- Gestión de Paquetes: Conda permite instalar y gestionar paquetes de diferentes versiones y configuraciones.
- Manejo de Entornos: Permite crear, exportar, listar, eliminar e intercambiar entornos que tienen diferentes versiones de Python y/o paquetes instalados.
- Compatibilidad: Asegura que los paquetes dentro de un entorno sean compatibles.
- Reproducibilidad: Los entornos y paquetes se pueden compartir, facilitando la reproducción de resultados y configuraciones en diferentes sistemas.

Creación de un Entorno Virtual con Conda:

Para crear un entorno virtual utilizando Conda, puedes utilizar la siguiente sintaxis básica:

```
conda create --name myenv # Crear un entorno llamado 'myenv'
```

En este caso, ‘myenv‘ es el nombre del nuevo entorno. Si también deseas especificar la versión de Python, puedes hacerlo de la siguiente manera:

```
conda create --name myenv python=3.8 # Especificando versión de Python
```

Activación y Desactivación de Entornos:

Activar y desactivar entornos es esencial para gestionar múltiples proyectos y asegurar que las dependencias utilizadas no entren en conflicto.

```
conda info --envs     # listar los entornos disponibles
conda activate myenv # Activar el entorno 'myenv'
conda deactivate      # Desactivar el entorno activo
```

Gestión de Paquetes:

Conda también permite gestionar paquetes, permitiendo instalar, actualizar y eliminar paquetes:

```
conda install numpy   # Instalar un paquete (ej. numpy)
conda update numpy    # Actualizar un paquete
conda remove numpy    # Desinstalar un paquete
```

Los entornos virtuales y la gestión de paquetes son cruciales para un desarrollo de software efectivo y eficiente. Conda proporciona herramientas robustas para manejar tanto paquetes como entornos, asegurando que las dependencias sean manejadas de manera coherente y reproducible. Conda se ha destacado en la comunidad de ciencia de datos, pero su utilidad es amplia y aplicable a una variedad de proyectos y disciplinas en desarrollo de software en Python.

1.5. VARIABLES, CONSTANTES Y TIPOS DE DATOS EN PYTHON

Python ofrece una rica variedad de tipos de datos y estructuras de datos incorporadas para facilitar la manipulación de datos y la implementación de algoritmos.

1.5.1. Variables y Constantes

En Python, las variables no requieren declaración explícita. Una variable se crea en el momento que le asigna un valor por primera vez. Por otro lado, las constantes idealmente no deben cambiar durante la ejecución del programa.

```
variable = 15   # Ejemplo de variable
CONSTANTE = 15  # Ejemplo de constante
```

1.5.2. Tipos de datos básicos en Python

Los tipos de datos numéricos en Python incluyen enteros, números de punto flotante y números complejos. Las cadenas de caracteres pueden ser definidas usando comillas simples o dobles y pueden contener una secuencia de caracteres de cualquier longitud.

```
entero = 10
flotante = 20.5
complejo = 3 + 5j
```

```
cadena = "Hola, Python"
```

1.5.3. Reglas de identificadores en Python

Los identificadores en Python, que incluyen los nombres de variables, siguen un conjunto de reglas y convenciones definidas para garantizar que el código sea legible y evite conflictos con las palabras clave y elementos internos del lenguaje. A continuación, se describen las reglas básicas para formar identificadores de variables en Python:

- **Caracteres permitidos**: Los identificadores deben estar compuestos únicamente por letras (mayúsculas o minúsculas), dígitos y/o el carácter guion bajo ( ).
- **Inicio del identificador**: No pueden comenzar con un dígito. Generalmente, un identificador debe comenzar con una letra (a-z, A-Z) o un guion bajo ( ).
- **Sensibilidad a mayúsculas y minúsculas**: Python es sensible a mayúsculas y minúsculas, por lo que, Variable y variable serían dos identificadores distintos.
- **Evitar palabras clave**: Los identificadores no deben coincidir con las palabras clave de Python, como if, else, while, etc., ya que estas están reservadas para funcionalidades específicas del lenguaje.
- **Sin límite de longitud**: No hay un límite para la longitud de los identificadores, pero es recomendable utilizar nombres concisos y que indiquen claramente el propósito de la variable.
- **Convenciones**: Aunque no es una regla, es común usar el estilo snake case (todo en minúsculas con palabras separadas por guiones bajos) para los nombres de variables, mientras que CamelCase se utiliza para nombres de clases.

Ejemplos de identificadores válidos:

```
mi_variable = 10
otraVariable = 20
_variable3 = 30
```

Ejemplos de identificadores inválidos:

```
3variable = 40 # Inicia con un dígito
mi-variable = 50 # Contiene un carácter no permitido (-)
if = 60 # Coincide con una palabra clave
```

Estas reglas y convenciones ayudan a mantener el código claro y a evitar errores semánticos en la programación. Es vital para los programadores estar familiarizados con estas para escribir código Python efectivo y fácil de entender.

1.5.4. El Estilo CamelCase en los identificadores

El camelCase es una convención para la escritura de identificadores como variables, funciones y otras entidades en la programación y otros contextos relacionados. Es especialmente prominente en lenguajes como Java y JavaScript, aunque también se usa en Python principalmente para nombrar clases. La nomenclatura camelCase se caracteriza por las siguientes propiedades:

- Sin espacios: Los identificadores escritos en camelCase no contienen espacios.

- Capitalización: La primera letra de cada palabra, excepto la inicial (a veces), se escribe con mayúscula.
- Legibilidad: La capitalización de las iniciales de cada palabra ayuda a mejorar la legibilidad, haciendo que las fronteras de las palabras sean fácilmente identificables.

Ejemplo de uso:

```
miPrimeraVariable = 10
calcularAreaTriangulo = 0.5 * baseTriangulo * alturaTriangulo
```

En estos ejemplos, `miPrimeraVariable` y `calcularAreaTriangulo` están escritos en camelCase. Note cómo cada palabra después de la primera comienza con una letra mayúscula, lo que facilita la identificación de las palabras individuales que componen el identificador, incluso sin la presencia de espacios o guiones bajos.

Es esencial mencionar que diferentes comunidades y lenguajes de programación pueden tener preferencias diversas en cuanto a las convenciones de nomenclatura. En Python, por ejemplo, mientras que camelCase es común para los nombres de las clases, para variables y funciones se suele preferir el uso de snake case, donde las palabras se escriben en minúsculas y se separan por guiones bajos.

```
class MiClase:
    def metodoEjemplo(self): pass
    variable_ejemplo = 20
```

La elección de la convención a menudo depende del estilo de codificación adoptado por un proyecto o equipo en particular. Es crucial ser consistente en la adopción de estas convenciones para mantener la base de código clara y fácil de leer para todos los colaboradores.

1.5.5. Expresiones y Asignaciones en Python

Las expresiones en Python son combinaciones de valores y operadores que pueden ser evaluadas para obtener un resultado. Python proporciona una variedad de operadores que permiten construir expresiones para realizar operaciones matemáticas, de comparación, lógicas, y de asignación, entre otras.

Operadores Aritméticos Básicos:

- Suma: +
- Resta: −
- Multiplicación: ∗
- División: /
- Módulo: %
- Exponente: ∗∗
- División Entera: //

Por ejemplo,

```
result = 3 * (4 + 5) # result se evaluá como 27
```

Operadores de Comparación y Lógicos:

- Igual: ==
- Diferente: ! =
- Mayor que: >
- Menor que: <
- Mayor o igual que: >=
- Menor o igual que: <=
- Y lógico: and
- O lógico: or
- No lógico: not

Operador de Asignación:

El operador de asignación (=) se utiliza para asignar un valor a una variable. La sintaxis general es:

```
variable = expresion
```

Donde el valor de la expresión se asigna a variable. Es imperativo notar que la asignación no es una relación de igualdad matemática, sino una operación que establece un vínculo entre un identificador (nombre de variable) y un valor o expresión.

Operadores de Asignación Compuesta:

Python también admite operadores de asignación compuesta, que combinan una operación aritmética y una operación de asignación en una única expresión.

- suma en asignación: +=
- resta en asignación: -=
- multiplicación en asignación: *=
- división en asignación: /=
- módulo en asignación: %=
- exponente en asignación: **=
- división entera en asignación: //=

```
# Ejemplo de uso de operadores y variables en Python

# Declaracion de variables
a = 10
b = 20

# Operadores aritmeticos
suma = a + b
resta = a - b
multiplicacion = a * b
division = a / b # Resultado: 0.5 (en Python 3.x)

# Operadores de comparacion
es_igual = a == b
```

```
    es_diferente = a != b
    es_mayor = a > b
    es_menor = a < b

    # Operadores logicos
    y_logico = es_igual and es_diferente
    o_logico = es_mayor or es_menor
    negacion = not es_igual

    # Operadores de asignacion
    a += 5 # Equivalente a: a = a + 5
    b *= 2 # Equivalente a: b = b * 2

    # Impresion de resultados
    print("Suma:", suma)
    print("Resta:", resta)
    print("Y Logico:", y_logico)
    print("O Logico:", o_logico)
```

La función `type()` en Python es utilizada para obtener el tipo de un objeto. Cuando se pasa un objeto como argumento a esta función, retorna el tipo del mismo. Ejemplo:

```
    x = 10
    print(type(x))        # <class 'int'>
```

Conversión de Tipos:

La conversión de tipos, por otro lado, se refiere al proceso de convertir un valor de un tipo de dato a otro. En Python, es posible realizar la conversión de tipos utilizando funciones predefinidas como `int()`, `float()`, y `str()` para convertir valores a enteros, números de punto flotante y cadenas de texto respectivamente.

Sin embargo, es crucial mencionar que no todas las conversiones de tipo son posibles. Si Python no puede determinar cómo convertir un valor de un tipo a otro, se generará un `ValueError` o un `TypeError`.

Ejemplo de una conversión de tipo inválida:

```
    x = "texto"
    y = int(x) # Generará un ValueError, ya que "texto" no puede convertirse
a entero
```

1.5.6. Manejo de cadenas de texto en Python

Las cadenas de texto (`strings`) en Python son arrays o vectores de caracteres y se encuentran entre los tipos de datos más utilizados. Python proporciona una amplia variedad de métodos y operadores para manipular cadenas, los cuales facilitan la realización de operaciones comunes de una manera fácil y directa.

Creación de Cadenas:

Las cadenas se pueden crear utilizando comillas simples o dobles, lo que permite incluir comillas y otros caracteres especiales dentro de ellas.

```
s1 = 'Hello, world!'
s2 = "It's a wonderful day!"
```

Cadenas Multilínea y Especiales:

Python permite la creación de cadenas multilínea y también el uso de caracteres especiales mediante el uso de secuencias de escape, tales como `\n` para nuevas líneas o `\t` para tabulaciones.

```
multiline_string = """This is a multiline
string spanning several lines."""
special_chars = "New line: \\n Tab: \\t"
```

Operaciones Básicas con Cadenas:

- Concatenación: Utilizando el operador +, se pueden unir dos o más cadenas.
- Repetición: Utilizando el operador *, se puede repetir una cadena un número específico de veces.

```
s3 = s1 + " " + s2          # Indexación: obtiene un caracter
s4 = "Repeat me! " * 3      # Segmentación: obtiene una subcadena
```

La indexación es una característica fundamental en Python para acceder a los elementos de estructuras de datos, tales como listas y cadenas, entre otros. La indexación permite acceder, modificar y referenciar los elementos de una secuencia mediante el uso de números enteros, los cuales representan la posición de un elemento dentro de la estructura.

Indexación Positiva:

La indexación en Python comienza desde 0, es decir, el primer elemento de una cadena tiene el índice 0, el segundo elemento tiene el índice 1, y así sucesivamente.

```
mi_cadena = "Programación"
primer_caracter = mi_cadena[0]    # Valor: 'P'
```

Indexación Negativa:

Python también permite la indexación negativa, donde el índice −1 se refiere al último elemento, −2 al penúltimo, y así sucesivamente.

```
ultimo_caracter = mi_cadena[-1] # Valor: 'n'
```

Indexación de Slicing:

Python permite rebanar cadenas usando la notación `[inicio : fin : paso]`, donde inicio es el índice inicial, fin es el índice hasta donde se quiere obtener la subcadena (sin incluir), y paso es la distancia entre índices consecutivos.

```
sub_cadena = mi_cadena[1:4] # Valor: 'yth'

mi_cadena = "Python"

# Slicing con índices negativos
sub_cadena = mi_cadena[-3:-1]
print(sub_cadena) # Salida: ho
```

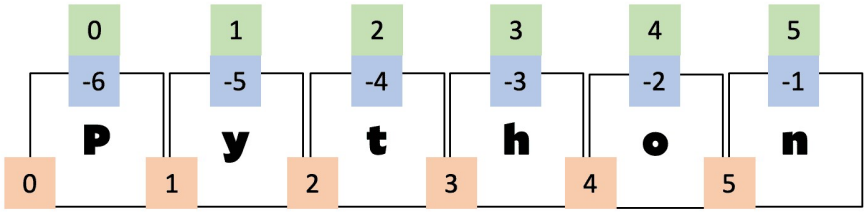


Figura 3: Python slicing

Es fundamental mencionar que el acceso a un índice fuera del rango de la estructura resultará en un error de tipo `IndexError`. Por lo tanto, es crucial asegurarse de que los índices utilizados estén dentro del rango válido para la estructura de datos con la que se está trabajando.

1.5.7. Uso de literales de cadena en Python

En Python, los `string literals`, o literales de cadena, son caracteres o secuencias de caracteres que representan valores de texto y son interpretados de manera especial cuando están precedidos por una barra invertida (\). Estos literales se utilizan para insertar caracteres especiales dentro de las cadenas de texto.

Uno de los literales de cadena más utilizados es `\n`, que se usa para insertar un nuevo renglón o salto de línea en una cadena de texto.

Ejemplo básico de uso:

```
print("Hola\nMundo")
```

La salida de este código sería:

```
Hola
Mundo
```

Otro ejemplo común es `\t`, que inserta un tabulador horizontal (espaciado). Algunos literales de cadena comunes en Python incluyen:

- \n - Nueva línea
- \t - Tabulador horizontal
- \" - Comillas dobles
- \' - Comillas simples
- \\ - Barra invertida

Estos literales permiten a los desarrolladores manejar convenientemente caracteres especiales y formatos dentro de las cadenas de texto, permitiendo una mayor flexibilidad y control sobre la salida del texto en sus programas.

Nota: Es importante considerar que si se desea utilizar una barra invertida como carácter común y no como prefijo de un literal, esta debe ser escapada como `\\` para evitar interpretaciones erróneas.

Los literales de cadena son herramientas esenciales, especialmente en situaciones donde el formato y la presentación del texto son críticos para la comunicación de información al usuario.

1.5.8. Funciones para manipular cadenas

Python ofrece una multitud de métodos para manipular y consultar cadenas, por ejemplo:

- `capitalize()`: Convierte el primer carácter de la cadena a mayúsculas y el resto a minúsculas.
- `swapcase()`: Invierte las mayúsculas por minúsculas y viceversa en toda la cadena.
- `replace(old, new)`: Reemplaza todas las ocurrencias del substring old por new.
- `split(separator)`: Divide la cadena en una lista de palabras utilizando separator como delimitador.
- `strip()`: Elimina los espacios en blanco en ambos extremos de la cadena.
- `rstrip()`: Elimina los espacios en blanco en el extremo derecho de la cadena.
- `lstrip()`: Elimina los espacios en blanco en el extremo izquierdo de la cadena.
- `find(substring)`: Retorna el índice de la primera ocurrencia de substring (-1 si no se encuentra).
- `index(substring)`: Similar a find, pero lanza una excepción si substring no se encuentra.
- `rindex(substring)`: Retorna el índice de la última ocurrencia de substring.
- `len()`: Retorna la longitud de la cadena.
- `title()`: Convierte el primer caracter de cada palabra a mayúsculas y el resto a minúsculas.
- `count(substring)`: Retorna el número de ocurrencias de substring en la cadena.
- `upper()`: Convierte todos los caracteres de la cadena a mayúsculas.
- `lower()`: Convierte todos los caracteres de la cadena a minúsculas.

Mira los ejemplos propuestos a continuación:

```
cadena_original = " Python es un lenguaje de programacion. "

# capitalize(): Convierte el primer caracter en mayusculas
cadena_capitalize = cadena_original.capitalize()

# swapcase(): Invierte las mayusculas y minusculas
cadena_swapcase = cadena_original.swapcase()

# replace(old, new): Reemplaza un substring por otro
cadena_replace = cadena_original.replace("Python", "Java")
```

```
    # split(separator): Divide la cadena en una lista utilizando un separador
    cadena_split = cadena_original.split(" ")

    # strip(): Elimina espacios en blanco en ambos extremos de la cadena
    cadena_strip = cadena_original.strip()

    # rstrip() y lstrip(): Elimina espacios a la derecha e izquierda respectivamente
    cadena_rstrip = cadena_original.rstrip()
    cadena_lstrip = cadena_original.lstrip()

    # find(substring): Retorna el indice de la primera ocurrencia de un substring (-1 si no se encuentra)
    indice_find = cadena_original.find("lenguaje")

    # index(substring): Similar a find, pero lanza una excepcion si no se encuentra el substring
    indice_index = cadena_original.index("lenguaje")

    # rindex(substring): Retorna el indice de la ultima ocurrencia de un substring
    indice_rindex = cadena_original.rindex("a")

    # len(): Retorna la longitud de la cadena
    longitud_cadena = len(cadena_original)

    # title(): Convierte la primera letra de cada palabra a mayúsculas
    cadena_title = cadena_original.title()

    # count(substring): Cuenta las ocurrencias de un substring en la cadena
    conteo_a = cadena_original.count("a")

    # upper() y lower(): Convierte la cadena a mayusculas y minusculas respectivamente
    cadena_upper = cadena_original.upper() cadena_lower = cadena_original.lower()

    # Impresion de algunos resultados
    print("Original:", cadena_original)
    # Salida: " Python es un lenguaje de programacion. "

    print("Capitalize:", cadena_capitalize)
    # Salida: " python es un lenguaje de programacion. "

    print("Swapcase:", cadena_swapcase)
    # Salida: " pYTHON ES UN LENGUAJE DE PROGRAMACION. "

    print("Replace:", cadena_replace)
    # Salida: " Java es un lenguaje de programacion. "
```

```
print("Split:", cadena_split)
# Salida: ['', '', '', 'Python', 'es', 'un',
# 'lenguaje', 'de', 'programacion.', '', '', '']

print("Strip:", cadena_strip)
# Salida: "Python es un lenguaje de programacion."
```

1.5.9. La función `str` en Python

La función `str()` en Python se utiliza para convertir un objeto dado en una cadena de texto (`string`). Esta función es especialmente útil en situaciones donde es necesario realizar operaciones de concatenación de texto o simplemente para mejorar la legibilidad de los datos de salida en la consola o interfaz de usuario.

La sintaxis básica de la función es la siguiente:

```
str(objeto)
```

donde `objeto` es el objeto que se desea convertir a una cadena de texto.

Ejemplo básico de uso:

```
numero = 123
texto_numero = str(numero)
print("El número convertido a texto es: " + texto_numero)
```

En este código, el número 123 es convertido en la cadena de texto "123" utilizando la función str(). Posteriormente, se concatena con otro texto para formar una cadena más larga que es impresa en la consola.

Uso con diferentes tipos de objetos:

```
mi_lista = [1, 2, 3]
texto_lista = str(mi_lista)
print("La lista convertida a texto es: " + texto_lista)
```

Aquí, la lista [1, 2, 3] se convierte en la cadena de texto "[1, 2, 3]" y posteriormente se concatena con otro `string` para imprimir el resultado en la consola.

Es importante mencionar que la función `str()` utiliza el método `str ()` definido en el objeto para realizar la conversión. Si un objeto no tiene este método, Python intentará utilizar el método `repr ()` como respaldo para obtener una representación en `string` del objeto.

La función `str()` es fundamental para la manipulación de texto en Python, permitiendo a los desarrolladores trabajar de manera más flexible con diferentes tipos de datos al convertirlos en su representación de texto.

Formato de Cadenas:

El formateo de cadenas permite insertar datos variables en una cadena de manera estructurada.

```
name = "World"
greeting = f"Hello, {name}!"
```

Las cadenas con formato, o `f-strings`, introducidas en Python 3.6, proporcionan una forma expresiva y cómoda de incrustar expresiones dentro de cadenas literales, utilizando una sintaxis minimalista. La sintaxis básica de las `f-strings` es preceder la cadena con la letra `f` o `F` y utilizar llaves `{}` para encerrar las expresiones que serán evaluadas en tiempo de ejecución y formateadas utilizando el formato especificado.

Ejemplo Básico:

```
name = "World"
greeting = f"Hello, {name}!"
```

En este caso, la variable `name` es referenciada directamente dentro de la cadena utilizando las llaves. Python evaluará la expresión dentro de las llaves y la insertará en la cadena en ese lugar.

Especificando la Precisión:

Las `f-strings` permiten especificar la precisión con la que se desea mostrar un número flotante, utilizando la sintaxis `{variable : .nf}`, donde `n` indica el número de decimales deseados.

```
pi = 3.141592653589793
formatted_pi = f"Pi, rounded to three decimal places: {pi:.3f}"
```

Expresiones Complejas:

Las `f-strings` también pueden contener expresiones más complejas, incluyendo operaciones aritméticas y llamadas a métodos.

```
radius = 5
area = f"Area of a circle with radius {radius} is: {3.14159 * (radius ** 2):.2f}"
```

Alineación y Relleno:

Es posible especificar la alineación y el carácter de relleno de las expresiones dentro de una `f-string`. Por ejemplo, alinear a la derecha con un ancho de 10 caracteres y rellenar con ceros.

```
number = 42
padded_number = f"{number:>010}"
```

Las `f-strings` en Python proporcionan una forma poderosa y flexible de trabajar con cadenas y expresiones, facilitando la generación de salidas formateadas y la interpolación de variables y expresiones dentro de cadenas de texto.

1.5.10. Formateo de cadenas con el método `format()`

El método *`format()`* en Python permite un control extenso sobre el formateo y alineación de cadenas de caracteres, facilitando la inserción y maquetación de variables y expresiones dentro de una cadena de texto. Este método puede ser particularmente útil para situaciones donde se requiere estructurar el texto de manera precisa o para mejorar la legibilidad del código.

Sintaxis Básica:

```
template_string = "Hello, {}!"
output_string = template_string.format("World")
```

En este caso, las llaves *`{}`* dentro de la cadena sirven como marcadores de posición para los argumentos proporcionados al método *`format()`*.

Formateo Posicional:

Es posible especificar el índice de los argumentos proporcionados para controlar su posición en la cadena de salida.

```
template_string = "From {1} to {0}"
output_string = template_string.format("B", "A")
```

Aquí, los números dentro de las llaves indican la posición de los argumentos a insertar.

Formateo de Números:

El método *`format()`* también permite un control detallado sobre la representación de números, incluyendo especificación de decimales y formateo como porcentaje.

```
pi = 3.141592653589793
formatted_string = "Value of Pi to 3 decimal places: {:.3f}".format(pi)
```

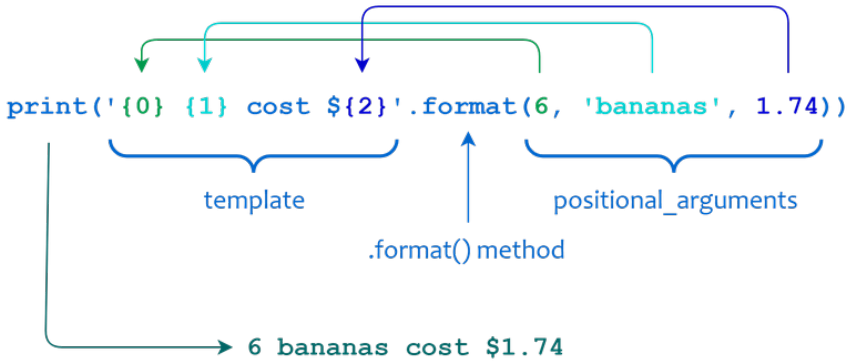


Figura 4: Ejemplo de uso de format strings

1.5.11. Símbolos utilizados en el método *`format()`*

El método *`format()`* en Python permite a los programadores especificar el formato de los valores de salida mediante el uso de varios símbolos y caracteres especiales, conocidos como especificadores de formato. Algunos de los más relevantes incluyen:

- `{}` Llaves utilizadas para definir campos de sustitución.
- `:` Dos puntos, introduce la especificación de formato.
- `.` Punto, especifica la precisión decimal en los formatos de punto flotante.
- `,` Coma, utilizada como separador de mil en números.
- `%` Porcentaje, convierte el número a una representación porcentual.
- `b` Binario, forma el número como binario.
- `o` Octal, forma el número como octal.
- `x/X` Hexadecimal, forma el número como hexadecimal.
- `e/E` Notación científica.
- `f/F` Punto flotante.
- `<`, `>`, `^` Símbolos para alineación izquierda, derecha y centro respectivamente.
- `+` Fuerza la aparición del signo.
- `-` Solo signo negativo.
- Espacio, inserta un espacio en blanco antes de números positivos.
- `#` Agrega un prefijo que indica la base de números.

Por ejemplo:

```
number = 1234567.890123456
formatted_number = "{:,}".format(number) # 1,234,567.890123456

print("binary value of 255 is {0:b}".format(255))
>>>binary value of 255 is 11111111
```

Es pertinente anotar que ciertos especificadores son exclusivos para ciertos tipos de formato y pueden ser combinados para ajustarse a las necesidades específicas del programador. La documentación oficial de Python proporciona una guía exhaustiva sobre estos símbolos y su utilización en diferentes contextos.

Alineación y Relleno:

Es posible controlar la alineación y caracteres de relleno de los valores insertados.

```
aligned_string = "|{:<10}|{:^10}|{:>10}|".format("Left", "Center", "Right")
```

En el ejemplo, los caracteres dentro de las llaves son utilizados para especificar la alineación de las cadenas a la izquierda, centro, y derecha respectivamente, mientras que el número ‘10’ indica el ancho del campo.

A través del adecuado uso del método `format()`, los desarrolladores pueden crear cadenas de salida estructuradas y legibles, asegurando que la presentación de los datos sea clara y coherente.

1.5.12. La función `print` en Python

La función `print()` es una de las funciones más utilizadas en Python y sirve para imprimir datos en la salida estándar, comúnmente la pantalla. Acepta varios parámetros que permiten configurar la impresión de los datos.

- **sep**: Este parámetro determina el separador entre los elementos impresos y por defecto es un espacio en blanco.
- **end**: Define qué se imprimirá al final de la llamada a la función. Por defecto es un salto de línea.
- **file**: Permite definir un flujo de salida diferente al estándar.
- **flush**: Un booleano que indica si se debe forzar la escritura de la salida (por defecto es False).

Ejemplo básico de uso:

```
print("Hola", "Mundo", sep="-", end="!")
```

En este caso, se imprime "Hola-Mundo!" ya que hemos configurado el separador como '-' y el final de la impresión como '!'.

1.6. LAS FECHAS EN PYTHON

Python proporciona la biblioteca *datetime* para trabajar con fechas y tiempos. Esta biblioteca incluye varios tipos de datos para manejar tanto fechas como combinaciones de fechas y tiempos.

A continuación, se ofrece una breve introducción sobre cómo usar algunos de los componentes principales de la biblioteca *datetime*:

```
import datetime

# Crear un objeto date (fecha)
fecha = datetime.date(2023, 10, 23)
print(fecha) # Salida: 2023-10-23

# Obtener la fecha actual
hoy = datetime.date.today() print(hoy)

# Crear un objeto datetime (fecha y hora)
fecha_hora = datetime.datetime(2023, 10, 23, 14, 30)
print(fecha_hora) # Salida: 2023-10-23 14:30:00

# Sumar y restar fechas con timedelta
dias = datetime.timedelta(days=5)
nueva_fecha = hoy + dias print(nueva_fecha)

# Formatear fechas
fecha_str = hoy.strftime("%d-%m-%Y") print(fecha_str) # Salida: 23-10-
2023

# Convertir cadena a fecha
fecha_obj = datetime.datetime.strptime("23-10-2023", "%d-%m-%Y")
print(fecha_obj.date())
```

Los puntos clave de la biblioteca `datetime` incluyen:

- `date`: Representa una fecha (año, mes, día).
- `datetime`: Representa una combinación de fecha y hora.
- `timedelta`: Representa una duración, la diferencia entre dos fechas o tiempos.
- `strftime`: Método para formatear objetos de fecha/hora como cadenas.
- `strptime`: Método para convertir cadenas en objetos de fecha/hora según un formato especificado.

1.6.1. Utilizando la función `import` en Python

En Python, la función `import` es una declaración que permite incluir módulos y paquetes, brindando acceso a recursos, funciones y clases adicionales que no están disponibles en el espacio de nombres principal del script. La utilización de `import` no sólo facilita la organización del código al permitir la segregación de funciones en diferentes módulos, sino que también enriquece los programas al proporcionar acceso a una amplia gama de módulos tanto integrados en Python como de terceros.

Sintaxis básica:

```
import modulo
```

Donde `modulo` es el nombre del módulo que se desea importar.

Ejemplo básico de uso:

```
import math

# Uso de una función del módulo math
resultado = math.sqrt(25)
print(resultado) # Salida: 5.0
```

En este ejemplo, el módulo `math` es importado, permitiendo el uso de sus funciones, en este caso, `sqrt()`, para calcular la raíz cuadrada de un número.

La función `import` también se puede utilizar con las declaraciones `from ... import ...`, permitiendo importar específicamente una función o clase de un módulo y reduciendo la necesidad de utilizar el nombre del módulo como prefijo al acceder a sus funciones.

Ejemplo de uso de `from ... import ...`:

```
from math import sqrt

# Uso directo de la función sqrt
resultado = sqrt(25)
print(resultado) # Salida: 5.0
```

Renombrando Importaciones:

Se puede renombrar un módulo o función importada usando la palabra clave `as` para mejorar la legibilidad del código o evitar conflictos de nombres.

```
import math as m

resultado = m.sqrt(25)
print(resultado) # Salida: 5.0
```

1.6.2. La función `input` en Python

La función `input()` es una de las funciones integradas en Python utilizada para obtener entrada del usuario en forma de cadena de texto (`string`). Esta función puede aceptar un argumento opcional, el cual es la cadena que se desea imprimir en pantalla como mensaje o indicación para el usuario, pero es comúnmente usado para proporcionar un mensaje de indicación o `prompt`.

Sintaxis básica de la función:

```
input([prompt])
```

donde [`prompt`] es el mensaje que se desea mostrar. Es importante mencionar que la respuesta del usuario siempre se trata como una cadena de texto, incluso si el usuario escribe un número.

Ejemplo básico de uso:

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

En este ejemplo, se le pide al usuario que introduzca su nombre. La función `input()` espera a que el usuario escriba su entrada y la retorna como una cadena de texto, que en este caso se almacena en la variable nombre. Posteriormente, se utiliza la función `print()` para saludar al usuario utilizando el nombre ingresado.

Cuando se necesita convertir la entrada del usuario a un tipo de dato diferente (por ejemplo, a un número entero o flotante), se puede utilizar funciones de conversión de tipos como `int()` o `float()`.

Ejemplo de conversión de tipo de dato:

```
edad = int(input("Por favor, introduce tu edad: "))
```

En este caso, se le pide al usuario su edad y se convierte la cadena de texto proporcionada por el usuario a un número entero antes de asignarlo a la variable edad. Si el usuario ingresa un valor que no se puede convertir a un número entero, Python generará una excepción de tipo `ValueError`.

1.6.3. Comentarios en Python

Los comentarios en Python son fragmentos de texto que no son ejecutados como parte del código. Son esenciales para explicar y aclarar el código para que los desarrolladores que lo lean puedan entender fácilmente su funcionalidad y propósito.

En Python, los comentarios se crean utilizando el símbolo `#` para los comentarios de una sola línea y las comillas triples `''' '''` o para los comentarios de múltiples líneas.

- **Comentarios de una sola línea**: Se utilizan para hacer anotaciones cortas y concisas.
- **Comentarios de múltiples líneas**: Se utilizan para proporcionar descripciones más detalladas o para bloquear temporalmente bloques de código durante la depuración y el desarrollo.

A continuación se presentan ejemplos:

```
# Este es un comentario de una sola línea en Python

'''
Este es un comentario
de m últiples l íneas en Python
'''

"""
También este es un comentario de múltiples líneas en Python """

def funcion_ejemplo():
    # Este comentario describe la siguiente línea de código
    print("¡Hola, mundo!") # Este comentario está al final de la línea
```

Es crucial usar comentarios para mejorar la legibilidad del código, explicando la funcionalidad de segmentos de código complejos o describiendo la lógica de negocio subyacente. Los comentarios también son útiles para explicar las decisiones de codificación y cualquier limitación conocida del código. Al escribir comentarios, es importante ser claro y conciso, proporcionando información útil sin ser demasiado exhaustivo (verborrágico).

1.6.4. Utilización de Docstrings en IDEs de Python

La documentación es un pilar fundamental en el desarrollo de software, facilitando la comprensión y utilización de código por parte de otros desarrolladores. En Python, las docstrings (cadenas de documentación) representan una forma estándar y accesible de documentar el código. Una docstring es una cadena de texto que se encuentra en la primera línea de una función, método, clase o módulo y se utiliza para describir su propósito y comportamiento.

```
def ejemplo_funcion():
    """
    Esta es una docstring que describe la función ejemplo_funcion.
    """
    pass
```

Los IDEs (Entornos de Desarrollo Integrado) de Python capitalizan activamente las docstrings. Al integrarlas, los IDEs permiten a los desarrolladores acceder a la documentación directamente desde el código fuente, facilitando así la comprensión y la utilización del código de una manera eficiente y efectiva.

Funcionalidades proporcionadas por los IDEs a través de docstrings:

- **Autocompletado Inteligente**: Los IDEs proponen automáticamente nombres de funciones, métodos y variables al escribir código, utilizando docstrings para mostrar información relevante acerca de los objetos sugeridos.
- **Ayuda Contextual**: Al colocar el cursor sobre una función o método, algunos IDEs muestran la docstring correspondiente en una ventana emergente, proporcionando información inmediata sobre su uso y requerimientos.
- **Generación Automática de Documentación**: Herramientas como Sphinx pueden utilizar docstrings para generar automáticamente la documentación del código en formatos accesibles como HTML o PDF.
- **Control de Calidad del Código**: Algunos IDEs y herramientas de control de calidad del código verifican la existencia y calidad de las docstrings, promoviendo las mejores prácticas en la documentación del código.

La utilización de docstrings no solo mejora la legibilidad del código, sino que también enriquece la experiencia de desarrollo al proporcionar acceso instantáneo a la documentación, directamente dentro del IDE. Es una práctica recomendada documentar sistemáticamente funciones, métodos, clases y módulos utilizando docstrings para aprovechar estas capacidades integradas y mejorar la sostenibilidad y comprensión del código a lo largo del tiempo.

1.6.5. Dividiendo una instrucción en varias líneas en Python

Python permite dividir una instrucción en varias líneas de código para mejorar la legibilidad y organización del mismo. Esto puede ser especialmente útil cuando se trabaja con expresiones o sentencias muy largas. Existen varias maneras de realizar esta división.

Usando el Carácter de Barra Invertida (*textbackslash*). En Python, podemos utilizar el carácter de barra invertida (\) para indicar que una línea continuá en la siguiente. Veamos un ejemplo con una expresión aritmética larga:

```
longExpressionResult = 3 + 4 + 5 - 6 + 7 + 8 - 9 + \
                       10 - 11 + 12 + 13 + 14 - 15
```

Usando Paréntesis, Corchetes o Llaves. Cuando una expresión está entre paréntesis (), corchetes [] o llaves {}, la expresión puede continuar en la siguiente línea sin necesidad de usar la barra:

```
longSumResult = (
    3+4+5-6+7+8-9+
    10 - 11 + 12 + 13 + 14 - 15
)

longList = [
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
]

longDictionary = {
```

```
        "keyOne": 1,
        "keyTwo": 2,
        "keyThree": 3,
        "keyFour": 4
    }
```

En ambos ejemplos, la expresión o declaración es legible y clara, y Python entiende que la instrucción se extiende a través de varias líneas debido al uso de paréntesis y corchetes/llaves, respectivamente.

2. SENTENCIAS DE CONTROL EN PYTHON

Las sentencias de control, también conocidas como estructuras de control, se utilizan en la programación para manejar el flujo de ejecución del código. Permiten que el código tome decisiones y repita bloques de código bajo ciertas condiciones. Hay dos tipos principales de estructuras de control: las estructuras de control de flujo condicional y las estructuras de control de flujo iterativo (bucles).

2.1. EXPRESIONES BOOLEANAS EN PYTHON

Las expresiones booleanas en Python se utilizan para realizar operaciones que involucran valores verdaderos o falsos (`True` o `False`). Python ofrece operadores booleanos típicos como AND, OR y NOT que permiten realizar operaciones lógicas.

```
# Ejemplo de expresiones booleanas en Python
x = True
y = False

# Operadores AND , OR y NOT
print(x and y) # Salida: False
print(x or y) # Salida: True
print(not x) # Salida: False
```

2.1.1. Operador `is` en Python

El operador `is` se utiliza para comparar si dos variables apuntan al mismo objeto, mientras que el `is not` se utiliza para verificar si dos variables no apuntan al mismo objeto. Es importante mencionar que `is` no es lo mismo que `==`, ya que este último compara los valores de las variables y no sus identidades.

```
# Ejemplo de uso del operador 'is'
x = [1, 2, 3]
y = [1, 2, 3]

# 'is' compara identidades
print(x is y) # Salida: False
```

```
# '==' compara valores
print(x == y) # Salida: True

# Ejemplo de uso de 'is' con 'type'
print(type(x) is list) # Salida: True

# Ejemplo de uso de 'is not'
z=x
print(z is not y) # Salida: True
```

2.2. CONDICIONALES EN PYTHON

Las sentencias condicionales en Python nos permiten ejecutar bloques específicos de código basados en una condición dada. La sintaxis básica para una sentencia condicional en Python es la siguiente:

```
if condicion:
    # Código a ejecutar si la condición es verdadera
```

La sintaxis para una sentencia condicional doble en Python es la siguiente:

```
if condicion:
    # Código a ejecutar si la condición es verdadera
else
    # Código a ejecutar si la condición es falsa
```

La sintaxis para una sentencia condicional múltiple en Python es la siguiente:

```
if condicion1:
    # Código a ejecutar si condición1 es verdadera
elif condicion2:
    # Código a ejecutar si la condición2 es verdadera
elif ...
    # Código a ejecutar si la condiciónN es verdadera
else
    # Código a ejecutar si las anteriores condiciones fueron falsas
```

En este fragmento, *condicion* es una expresión que se evalúa como *True* o *False*. Si *condicion* es *True*, entonces el bloque de código indentado debajo de la sentencia *if* se ejecutará. A continuación, un ejemplo práctico:

```
studentGrade = 85

if studentGrade >= 90:
    letterGrade = "A"
elif studentGrade >= 80:
     letterGrade = "B"
elif studentGrade >= 70:
```

```
        letterGrade = "C"
    elif studentGrade >= 60:
        letterGrade = "D"
    else:
        letterGrade = "F"

    print("The letter grade is: " + letterGrade)
```

Explicación:

La variable `studentGrade` es verificada contra varias condiciones utilizando la estructura `if/elif/else`. Dependiendo de la primera condición que sea `True`, el bloque de código asociado se ejecutará, asignando un valor correspondiente a `letterGrade`. Si `studentGrade` es `85`, entonces `letterGrade` se asignará como "`B`" y la salida será: "`The letter grade is: B`".

2.3. SENTENCIAS ITERATIVAS EN PYTHON

Las sentencias iterativas, también conocidas como bucles o ciclos, se utilizan en programación para repetir un bloque de código múltiples veces. Python proporciona dos estructuras de bucle principales: el bucle `for` y el bucle `while.`

Bucle `for`:

La sintaxis del bucle `for` es

```
for variable in iterable:
    # Cuerpo del bucle
    # ...
```

Aquí, `variable` es la variable de iteración que toma el valor de cada elemento en `iterable` en cada iteración del bucle.

Bucle `while`:

La sintaxis del bucle `while` es

```
while condicion:
    # Cuerpo del bucle
    # ...
```

Bucle `while` con `else`:

Python también permite el uso de una cláusula `else` con el bucle `while`, que se ejecuta cuando la condición del `while` se vuelve `false.`

```
while condicion:
    # Cuerpo del bucle
    # ...
```

```
    else:
        # Código a ejecutar cuando
        # la condición del while es False
        # ...
```

El bloque de código dentro de la cláusula `else` se ejecutará cuando el bucle `while` finalice, es decir, cuando `condicion` se evalúe como `False`. Nota que si el bucle termina con un break, el bloque `else` no se ejecutará.

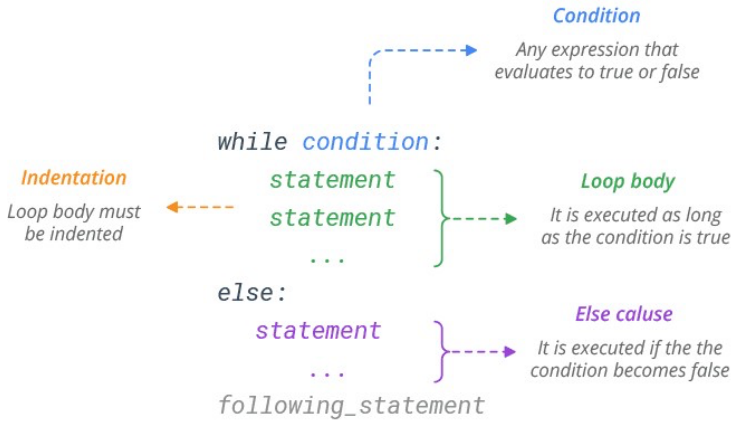


Figura 6: Sintaxis del bucle `while`

Control de Bucles:

Python también proporciona declaraciones de control de bucles, break y continue, que permiten una gestión más fina del comportamiento de los bucles.

```
    for variable in iterable:
        if condicion_para_salir:
            break
        elif condicion_para_continuar:
            continue
        # Cuerpo del bucle
        # ...
```

La instrucción `break` permite salir anticipadamente del bucle, mientras que `continue` salta al siguiente ciclo del bucle, omitiendo el resto del código que sigue en la iteración actual.

2.3.1. Ejemplos de Bucles

El bucle `while` repite un bloque de código mientras una condición dada sea verdadera

```
    # Ejemplo de bucle while
    count = 0

    while count < 5:
        print("Iteraci ón n úmero:", count)
        count += 1
```

```
# EJEMPLO: sacar las tablas de multiplicar del 1 al 10
# Bucle para iterar a través de los números del 1 al 10 (inclusive)
# para generar las tablas de multiplicar
for i in range(1, 11):
    print(f"\nTabla del {i}:")

    # Bucle anidado para calcular e imprimir
    # cada línea de la tabla de multiplicar
    for j in range(1, 11):
        print(f"{i} x {j} = {i*j}")
```

En este ejemplo, el bucle `while` seguirá ejecutándose mientras la variable `count` sea menor que `5`, imprimiendo el número de iteración en cada paso y aumentando el valor de `count` por `1` en cada iteración.

2.3.2. Bucle `for`

El bucle `for` es utilizado para iterar sobre una secuencia iterable:

```
# Ejemplo de bucle for con range
for number in range(5):
    print("Número: ", number)
```

Aquí, la función `range(5)` genera una secuencia de números desde `0` hasta `4`, y el bucle `for` imprime cada número en su respectiva iteración.

2.3.3. Bucles Anidados

Los bucles también pueden estar anidados dentro de otros bucles.

```
# Ejemplo de bucles anidados
for i in range(3):
    for j in range(2):
        print("i:", i, ", j:", j)
```

En este ejemplo, para cada valor de `i` en el bucle externo, el bucle interno se ejecutará completamente, generando todas las posibles combinaciones de `i` y `j` y imprimiéndolas.

```
# Ejemplo de combinación de for y while
for i in range(3):
    print("i:", i)
    j=0
    while j < 2:
        print(" j:", j)
        j += 1
```

En este ejemplo, para cada iteración del bucle `for`, el bucle `while` se ejecuta mientras `j` sea menor que `2`, imprimiendo los valores de `i` y `j`.

2.3.4. La función `range`

La función `range` en Python es una función incorporada que se utiliza para generar una secuencia de números, la cual es especialmente útil para iterar sobre un bloque de código un número específico de veces mediante un bucle for. La función range puede aceptar entre uno y tres argumentos numéricos: `start`, `stop`, y `step`.

- **start**: Especifica desde qué número comenzar la secuencia. Si se omite, la secuencia comenzará desde 0.
- **stop**: Especifica hasta qué número generar la secuencia. Este número no está incluido en la salida.
- **step**: Especifica la diferencia entre cada número en la secuencia. Si se omite, la diferencia será 1.

Si se proporciona un solo argumento a `range`, se utilizará como el valor de `stop`, y la secuencia generada será desde `0` hasta `stop-1`.

```
# Uso básico de range con un argumento
for i in range(5):
    print(i) # Salida: 0 1 2 3 4
```

Cuando se proporcionan dos argumentos, se utilizan como los valores de `start` y `stop` respectivamente, generando números desde `start` hasta `stop-1`

```
# Uso de range con dos argumentos
for i in range(2, 5):
    print(i) # Salida: 2 3 4
```

Cuando se utilizan tres argumentos, se interpretan como los valores de `start`, `stop`, y `step`, respectivamente. La secuencia generada comenzará en `start`, incrementará en `step`, y terminará justo antes de llegar a `stop`.

```
# Uso de range con tres argumentos
for i in range(1, 10, 2):
    print(i) # Salida: 1 3 5 7 9
```

3. ESTRUCTURAS DE DATOS EN PYTHON

"Una estructura de datos es un conjunto de objetos, con operaciones definidas sobre ellos, un comportamiento especificado para estas operaciones, y relaciones definidas entre los objetos y las operaciones".

— Niclaus Wirth

Python incluye varias estructuras de datos incorporadas: listas, diccionarios, tuplas y conjuntos, las cuales permiten crear, almacenar y manipular datos de manera eficiente y fácil.

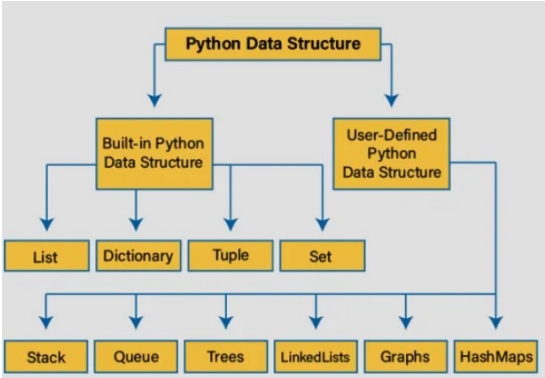


Figura 6: Estructuras de datos en Python

3.1. LISTAS

Las listas en Python son colecciones ordenadas de elementos que pueden ser de cualquier tipo y son mutables, es decir, los elementos pueden ser cambiados después de que la lista ha sido creada.

```
myList = [1, 2, 3, 4, 5]
myList[1] = 10 # Modificar un elemento
myList.append(6) # Añadir un elemento
```

Las listas en Python son una de las estructuras de datos más versátiles y utilizadas. Consisten en una colección ordenada de elementos, los cuales pueden ser de cualquier tipo, como enteros, cadenas, y hasta otras listas. Las listas son mutables, lo que significa que los elementos pueden ser modificados incluso después de que la lista ha sido creada.

Creación y acceso a las listas:

Para crear una lista, se utilizan corchetes `[]` y se separan los elementos con comas. Para acceder a los elementos de la lista, se utilizan índices, que comienzan desde 0.

```
myList = [1, 2, 3, 4, 5] # Crear una lista
firstElement = myList[0] # Acceder al primer elemento
```

Métodos importantes en listas:

A continuación, se presentan algunos métodos útiles que están disponibles para las listas en Python.

- *`append(element)`*: Añade un elemento al final de la lista

```
myList.append(6) # [1, 2, 3, 4, 5, 6]
```

- *`insert(index, element)`*: Inserta un elemento en la posición especificada

```
myList.insert(0, 0) # [0, 1, 2, 3, 4, 5, 6]
```

- *`remove(element)`*: Elimina la primera ocurrencia del elemento especificado

```
myList.remove(0) # [1, 2, 3, 4, 5, 6]
```

- *pop(index)*: Elimina y devuelve el elemento en la posición especificada
- *reverse()*: Invierte el orden de los elementos en la lista
- *sort()*: Ordena los elementos de la lista
- *count(elemento)*: Cuenta las apariciones de un elemento en la lista