ÍNDICE

1. Introducción 2
1.1. Objetivos 3
2. Sintaxis Python 4
2.1. Identificadores 4
2.2. Palabras Reservadas 5
2.3. Sintaxis 6
3. Almacenar Valores 8
3.1. Variables 8
3.2. Conversión de tipo de Dato 9
4. Tipos de Datos y Operadores 10
4.1. Datos Numéricos 10
A. Operadores aritméticos 10
B. Operadores de asignación 11
C. Operaciones de bits 12
4.2. Datos Booleanos 14
A. Operadores lógicos 14
B. Operadores de Comparación 14
C. Operadores de Identidad 15
4.3. Cadenas de caracteres 16
A. Operadores de cadenas de texto 16
B. Funcionalidades para el manejo de cadenas 21
ANEXO 23

1. Introducción

Comenzaremos hablando sobre la sintaxis en Python. Es importante destacar que Python se distingue de otros lenguajes, como Java y C, en términos de cómo se deben escribir los elementos del lenguaje. Esto se hace con el fin de mejorar la legibilidad del código y hacerlo más comprensible. A lo largo de este tema, exploraremos las reglas específicas sobre cómo escribir identificadores, comentarios y cómo utilizar la sangría adecuada.

Continuaremos con el almacenamiento de valores en Python. En Python, existen dos formas principales de almacenar valores: mediante variables y constantes. Aprenderemos cómo declarar variables y asignarles valores, así como también cómo trabajar con constantes, que son valores inmutables.

Una vez que tengamos claros los conceptos básicos de almacenamiento de valores, pasaremos a los tipos de datos fundamentales en Python. Estos incluyen los tipos numéricos (como enteros y números de punto flotante), los valores booleanos (True o False), las cadenas de caracteres (texto) y el tipo None, que representa la ausencia de un valor. Exploraremos cómo trabajar con cada uno de estos tipos de datos y cómo realizar operaciones y comparaciones con ellos.

Además, profundizaremos en los operadores disponibles en Python. Cada tipo de dato tiene sus propios operadores específicos que se pueden aplicar. Discutiremos los operadores aritméticos, lógicos, de comparación y de asignación, y aprenderemos cómo utilizarlos correctamente en nuestros programas.

Por último, abordaremos la posibilidad de cambiar el tipo de dato de una variable. En Python, es posible convertir una variable de un tipo de dato a otro, siempre y cuando sea compatible. Exploraremos cómo realizar estas conversiones y entenderemos las implicaciones que pueden tener en nuestros programas.

1.1. Objetivos

- Familiarizarse con la sintaxis de Python para los identificadores, comentarios y la indentación.
- Aprender cómo almacenar valores en variables y constantes.
- Comprender los tipos básicos de Python, como los numéricos, booleanos, cadenas de caracteres y el tipo None.
- Explorar los diversos operadores que se pueden aplicar a cada tipo de dato.
- Descubrir cómo cambiar el tipo de dato de las variables.

2. Sintaxis Python

En este primer apartado vamos a explorar las normas básicas de la sintaxis de Python. Como mencionamos al inicio del curso, Python sigue una guía de estilo que busca facilitar la lectura de los programas. A continuación, detallaré estas normas básicas para que puedas familiarizarte con ellas:

2.1. Identificadores

Son los nombres asignados a los elementos, tales como funciones, variables, etc., para hacerles referencias más adelante en el desarrollo del código. Estos identificadores siguientes las siguientes reglas

- Pueden ser combinaciones de números (0-9).
- Pueden ser combinaciones de letras mayúsculas y minúsculas (Aa -Zz).
- Puede usarse el símbolo de guion abajo (_).
- Pueden tener cualquier longitud.
- No pueden comenzar por un digito.
- No pueden utilizarse símbolos especiales: `@, !, #`, etc.

Un punto importante que definir es que Python distingue las mayúsculas de las minúsculas (key sensitive); esto significa que, si tuviéramos los identificadores variable y VARIABLE, Python los consideraría diferentes

```
# Identificadores correctos
variable = 324
algo = 'algo'
_variable_ = [1, 2, 3]
Variable = True
```

```
# Identificadores incorrectos.
@variable = 3
2_variable = "dará error"
!variable = "no reconocible"
```

2.2. Palabras Reservadas

Python al igual que todo lenguaje de programación, posee un conjunto de palabras reservadas en donde estas no se pueden utilizar como identificadores; a continuación, se lista las palabras reservadas

False	None	True	and	as	assert
elif	del	def	continue	class	break
else	except	finally	for	from	global
nolocal	lambda	is	in	import	if
not	or	pass	raise	return	while
async	await	try	with	yield	

```
# SyntaxError: cannot assign to True
True = 5

# SyntaxError: invalid syntax
from = 'asignar valor fallido'
```

El módulo keyword en Python proporciona una lista de palabras reservadas del lenguaje. Puedes importar el módulo keyword y utilizar sus funciones para obtener información sobre las palabras reservadas en Python.

La función kwlist del módulo keyword devuelve una lista que contiene todas las palabras reservadas en Python. Puedes utilizar esta lista para verificar si una palabra determinada es una palabra reservada de Python o para cualquier otro propósito relacionado con el análisis léxico.

```
import keyword


# Obtener todas las palabras reservadas
palabras_reservadas = keyword.kwlist
print(palabras_reservadas)
# Salida: ['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await',
'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except',
'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with',
'yield']
```

2.3. Sintaxis

- **Indentación**: En Python, la indentación (sangría) es muy importante, ya que define la estructura del código. Se utiliza para delimitar bloques de código, como loops y funciones. Se recomienda utilizar espacios en lugar de tabulaciones para la indentación, y la cantidad de espacios generalmente se establece en 4.

```
# bucle for
for i in range(5):
    print(i)
    print("Este código está dentro del bucle for")
print("Este código está fuera del bucle for")
```

- **Comentarios**: Los comentarios son líneas de texto que se utilizan para explicar el código y hacerlo más comprensible. En Python, los comentarios se inician con el símbolo #. Todo lo que sigue después de este símbolo se considera un comentario y es ignorado por el intérprete de Python.

    - De línea, uso del símbolo #

```
# Ejemplo de comentario de línea.
```

    - De bloque, comillas triples (dobles o simples)

```
"""
Ejemplo de comentario de bloque.
Todas las líneas pertenecen al comentario.
"""
'''
Otro ejemplo de
comentario de bloque con comillas simples.
'''
```

- **Convenciones de nombres**: En Python, se recomienda seguir ciertas convenciones de nombres para que el código sea más legible. consiste en escribir las palabras en minúsculas y separarlas con guiones bajos; y Mayúsculas separadas con guion(es) bajo para declarar constantes.

```
nombre_completo = "Juan Pérez"
numero_entero = 42
calificacion_final = 8.5
SEMILLA = 123123 # Constante
```

```
# Ejemplo de constantes:
IVA = 0.21  # constante

precio = float(input("Ingresa el precio: "))  # Convierte la entrada
a float
precio_final = precio + (precio * IVA)
print("El precio final es:", precio_final)
```

- **Uso de paréntesis y puntos**: En Python, se utilizan paréntesis para agrupar y controlar la precedencia de las operaciones matemáticas. Además, se utilizan puntos para acceder a los atributos y métodos de un objeto. Por ejemplo: resultado = (2 + 3) * 4 y cadena = "Hola".upper().

```
resultado = (2 + 3) * 4
print(resultado)  # Salida: 20
```

```
cadena = "Hola"
longitud = cadena.upper().count("O")
print(longitud)  # Salida: 1
```

- **Fin de línea**: En Python, no es necesario utilizar punto y coma al final de cada línea de código, a diferencia de otros lenguajes. El intérprete de Python reconoce el final de una línea de código cuando encuentra un salto de línea.

```
nombre = "Ana"
edad = 25
```

```
nombre = "Juan"; edad = 30;
```

3. Almacenar Valores

En Python, tenemos dos formas principales de almacenar valores: utilizando variables y constantes. En este apartado, explicaré cómo se declaran y utilizan tanto las variables como las constantes en Python.

1. **Reglas de nombramiento:** Al nombrar variables y constantes en Python, es importante seguir algunas reglas y convenciones. Los nombres de variables deben ser descriptivos y seguir el estilo snake_case (palabras en minúsculas separadas por guiones bajos). También es importante evitar utilizar palabras reservadas del lenguaje como nombres de variables.
2. **Alcance de las variables:** Las variables en Python tienen un alcance (scope) que determina dónde se pueden acceder y utilizar. Hay variables locales, que solo existen dentro de un bloque o función específicos, y variables globales, que se definen fuera de cualquier función y se pueden acceder desde cualquier parte del programa. Es importante entender cómo funciona el alcance de las variables para evitar problemas de colisión de nombres y asegurarse de que las variables estén disponibles donde se necesitan.
3. **Tipado dinámico:** En Python, no es necesario declarar explícitamente el tipo de dato de una variable al momento de su creación. Python es un lenguaje de tipado dinámico, lo que significa que el tipo de dato de una variable se determina automáticamente en tiempo de ejecución según el valor que se le asigna. Esto permite una mayor flexibilidad y facilidad de uso al trabajar con variables.

3.1. Variables

Las variables en Python son contenedores que nos permiten almacenar y manipular datos. Para declarar una variable, simplemente asignamos un valor utilizando el operador de asignación (=). No es necesario especificar el tipo de dato de la variable, ya que Python es un lenguaje de tipado dinámico.

A continuación, veremos un ejemplo de variables y el tipo de datos que se ha asignado; para preguntar que tipo de dato ha almacenado una variable hacemos uso de la función **`type().`**

```
# Declaración de variables
nombre = "Juan"
edad = 25
pi = 3.14159
es_estudiante = True
```

```
print (f"Tipo de dato: {type(nombre)}")
print (f"Tipo de dato: {type(edad)}")
print (f"Tipo de dato: {type(pi)}")
print(f"Tipo de dato: {type(es_estudiante)}")
```

3.2. Conversión de tipo de Dato

En Python, es posible convertir el tipo de dato de una variable de un tipo a otro utilizando funciones incorporadas como **`int(), float(), str(),`** entre otras. Esto es útil cuando se necesita realizar operaciones específicas o cuando se requiere que una variable tenga un tipo de dato particular.

```
# Conversión de tipo de datos
edad = "25"  # La edad es una cadena de caracteres
print(type(edad))  # Salida: <class 'str'>
```

```
# Convertir la edad a un entero
edad_entero = int(edad)
print(type(edad_entero))  # Salida: <class 'int'>
```

```
# Realizar operaciones con la edad como entero
doble_edad = edad_entero * 2
print(doble_edad)  # Salida: 50
```

```
# Convertir la edad nuevamente a una cadena de caracteres
edad_str = str(edad_entero)
print(type(edad_str))  # Salida: <class 'str'>
```

4. Tipos de Datos y Operadores

Como hemos visto, cuando declaramos un elemento para almacenar un valor en Python, ya sea una variable o constante, no necesitamos definir el tipo de dato que almacenara; sin embargo, es necesario conocer los tipos de datos que podemos utilizar en Python. A continuación, describiremos los tipos de datos y operadores que podemos aplicar sobre cada uno de estos tipos.

4.1. Datos Numéricos

- **Enteros (int):** Representan números enteros, como -5, 0, 10, etc. Se pueden realizar operaciones aritméticas básicas (suma, resta, multiplicación, división, etc.) con ellos utilizando los operadores +, -, *, /, // (división entera), % (módulo) y ** (exponente).
- **Números de punto flotante (float):** Representan números decimales, como 3.14, -0.5, 1.0, etc. Se pueden realizar las mismas operaciones aritméticas básicas que con los enteros.
- **Complejos:**

A. Operadores aritméticos

	Oper.	Descripción
Suma	+	Devuelve como resultado la suma de dos números
Resta	-	Devuelve como resultado la resta de dos números
multiplicación	*	Devuelve como resultado la multiplicación de dos números
División	/	Devuelve como resultado la división de dos números
División Entera	//	Devuelve como resultado la división entera de dos números
Módulo	%	Devuelve como resultado el valor del resto obtenido de la división entera entre dos números
Exponente	**	Devuelve como resultado el valor exponencial de una base con respecto al exponente

	Descripción
Suma	`5.12 + 3.1 # Devuelve 8.22`
Resta	`3.13 - 6 # Devuelve -2.87`
multiplicación	`3.23 * 10 # Devuelve 32.3`
División	`3.15 / 10 # Devuelve 0.315`
División Entera	`2 // 10 # Devuelve 0`
Módulo	`3 % 10 # Devuelve 3`
Exponente	`5 ** 2 # Devuelve 25`

B. Operadores de asignación

Los operadores de asignación nos permiten asignar el resultado de la operación a una variable incluyendo el símbolo = en el operador; estos nos permiten modificar el valor de una variable sin tener que definirla nuevamente.

	Oper.	Descripción
Simple	=	Asigna el valor a la variable definida
Suma y asignación	+=	el operador suma, al valor de la variable, el valor definido en el lado derecho
Resta y asignación	-=	el operador resta, al valor de la variable, el valor definido en el lado derecho
Multiplicación y asignación	*=	el operador multiplica, al valor de la variable, el valor definido en el lado derecho
División y asignación	/=	el operador divide, al valor de la variable, el valor definido en el lado derecho
División Entera y asignación	//=	el operador realiza la división entera al valor de la variable con respecto al valor definido en el lado derecho
Módulo y asignación	%=	el operador asigna a la variable el resto de la división entera entre el valor de la variable y el valor definido en el lado derecho de la operación
Exponente y asignación	**=	el operador asigna a la variable el resultado del exponente entre el valor de la variable y el valor de la derecha de la operación

	Operador	Descripción
Simple	=	`valor = 10 # vale 10`
Suma y asignación	+=	`valor += 10 #  vale 20` `valor`
Resta y asignación	-=	`valor -= 10 # vale 10` `valor`
Multiplicación y asignación	*=	`valor *= 10 # vale 100` `valor`
División y asignación	/=	`valor /= 10 # vale 10.0` `valor`
División Entera y asignación	//=	`valor //= 10 # vale 1.0` `valor`

Módulo y asignación	%=	`valor = 14`
		`valor %= 10 # vale 4`
		`valor`

Exponente y asignación	**=	`valor **= 2 # El vale 16` `valor`

C. Operaciones de bits

Se pueden realizar operaciones de bits utilizando los operadores bit a bit. Estos operadores actúan a nivel de bits en números enteros y permiten manipular y realizar operaciones lógicas con los bits individuales de los números. A continuación, te presento los operadores de bits más comunes en Python:

- Operador AND a nivel de bits (&):
    - Realiza una operación AND bit a bit entre los números binarios que representan los operandos.
    - Para cada par de bits correspondientes de los operandos, si ambos bits son 1, el resultado tendrá un bit activado (1); de lo contrario, el resultado tendrá un bit desactivado (0).
- Operador OR a nivel de bits (|):
    - Realiza una operación OR bit a bit entre los números binarios que representan los operandos.
    - Para cada par de bits correspondientes de los operandos, si al menos uno de los bits es 1, el resultado tendrá un bit activado (1); de lo contrario, el resultado tendrá un bit desactivado (0).
- Operador XOR a nivel de bits (^):
    - Realiza una operación XOR (OR exclusivo) bit a bit entre los números binarios que representan los operandos.
    - Para cada par de bits correspondientes de los operandos, si exactamente uno de los bits es 1, el resultado tendrá un bit activado (1); de lo contrario, el resultado tendrá un bit desactivado (0).
- Operador NOT a nivel de bits (~):
    - Realiza una operación NOT (complemento) bit a bit en el número binario que representa el operando.
    - Invierte cada bit del número: los bits activados (1) se convierten en bits desactivados (0), y los bits desactivados (0) se convierten en bits activados (1).
- Desplazamiento a la izquierda (<<) y desplazamiento a la derecha (>>):
    - Estos operadores desplazan los bits del número binario que representa el operando hacia la izquierda o hacia la derecha, respectivamente, en la cantidad de posiciones especificadas.

- Al desplazar a la izquierda, se añaden ceros a la derecha y se descartan los bits desplazados fuera del rango.
- Al desplazar a la derecha, se añaden ceros o unos, dependiendo del signo del número, a la izquierda y se descartan los bits desplazados fuera del rango.

A continuación, se presentan 2 ejemplos (baso en bits y con números enteros).

```
# Operaciones de bits
a = 0b10101010  # Representación binaria de 170
b = 0b11001100  # Representación binaria de 204

# Operador AND a nivel de bits
resultado_and = a & b
print(bin(resultado_and))  # Salida: 0b10001000 (136 en decimal)

# Operador OR a nivel de bits
resultado_or = a | b
print(bin(resultado_or))  # Salida: 0b11101110 (238 en decimal)

# Operador XOR a nivel de bits
resultado_xor = a ^ b
print(bin(resultado_xor))  # Salida: 0b01100110 (102 en decimal)

# Operador NOT a nivel de bits
resultado_not = ~a
print(bin(resultado_not))  # Salida: -0b10101011 (-171 en decimal)

# Desplazamiento a la izquierda
desplazamiento_izq = a << 2
print(bin(desplazamiento_izq))  # Salida: 0b1010101000 (680 en decimal)

# Desplazamiento a la derecha
desplazamiento_der = b >> 3
print(bin(desplazamiento_der))  # Salida: 0b11001 (25 en decimal)
```

```
# Operaciones a nivel de enteros
# AND
3 & 2 # El resultado será 2

# OR
3 | 2 # el resultado será 3

# XOR
4 ^ 5 # el resultado será 1

# Mover bits a la izquierda
4 << 1 # el resultado será 8

# Mover bits a la derecha
4 >> 1 # el resultado será 2
```

4.2. Datos Booleanos

Representan valores de verdadero (True) o falso (False).

A. Operadores lógicos

Se utilizan en expresiones lógicas y condiciones. Los operadores lógicos disponibles son and (y lógico), or (o lógico) y not (negación).

AND	`True and False # Devolverá False`
OR	`True or False # Devolverá True`
NOT	`not True # Devolverá False`

B. Operadores de Comparación

Los operadores de comparación se utilizan para comparar dos valores y evaluar si se cumple una determinada relación. A continuación, se describen los operadores de comparación más comunes en Python:

- == (igual a): Verifica si dos valores son iguales.
- != (diferente de): Verifica si dos valores no son iguales.
- (mayor que): Verifica si el valor de la izquierda es mayor que el de la derecha.

- < (menor que): Verifica si el valor de la izquierda es menor que el de la derecha.
- >= (mayor o igual que): Verifica si el valor de la izquierda es mayor o igual que el de la derecha.
- <= (menor o igual que): Verifica si el valor de la izquierda es menor o igual que el de la derecha.

```
# Definimos los valores
x = 5
y = 10
```

==	print(x == y) # Salida: False
!=	print(x != y) # Salida: True
>	print(x > y) # Salida: False
<	print(x < y) # Salida: True
>=	print(x >= y) # Salida: False
<=	print(x <= y) # Salida: True

C. Operadores de Identidad

Los operadores de identidad se utilizan para comparar la identidad de dos objetos, es decir, si hacen referencia al mismo objeto en la memoria. A continuación, se describen los operadores de identidad en Python:

- is: Verifica si dos objetos son el mismo objeto.
- is not: Verifica si dos objetos no son el mismo objeto.

```
# Definimos los valores
x = [1, 2, 3]
y = [1, 2, 3]
z = x
```

is	print(x is y) # Salida: False
Is not	print(x is not y) # Salida: True
is	print(x is y) # Salida: True

En el ejemplo anterior, x e y son dos listas distintas, aunque contengan los mismos elementos, por lo que x is y devuelve False. Sin embargo, x y z hacen referencia al mismo objeto en la memoria, por lo que x is z devuelve True.

4.3. Cadenas de caracteres

Representan secuencias de caracteres, como `"Hola", 'Python', "123"`, etc. Se pueden concatenar utilizando el operador +, acceder a caracteres individuales utilizando la notación de corchetes [], obtener sub-cadenas utilizando la notación de rebanadas [:] y realizar comparaciones alfabéticas utilizando los operadores ==, !=, <, >, <= y >=.

A. Operadores de cadenas de texto

- **Concatenación**, El operador de concatenación + se utiliza para unir dos cadenas en una sola cadena.

```
cadena1 = "Hola"
cadena2 = "Mundo"
resultado = cadena1 + " " + cadena2
print(resultado)  # Salida: "Hola Mundo"
```

- **Repetición**, El operador de repetición * se utiliza para repetir una cadena un número determinado de veces.

```
cadena = "Hola "
repetida = cadena * 3
print(repetida)  # Salida: "Hola Hola Hola "
```

- **Indexación**, La indexación se utiliza para acceder a caracteres individuales de una cadena mediante su posición; se utiliza el operador de corchetes [] y se proporciona el índice del carácter deseado (comenzando desde 0).

```
cadena = "Python"
print(cadena[0])     # Salida: "P"
print(cadena[2])     # Salida: "t"
print(cadena[-1])    # Salida: "n"
```

- **Slicing (Rebanado),** El slicing se utiliza para obtener una porción (subcadena) de una cadena; se utiliza el operador de rebanado [:] y se proporcionan los índices de inicio y fin (no inclusivo) de la porción deseada.

```
cadena = "Python"
```

```
print(cadena[1:4])      # Salida: "yth"
print(cadena[:3])       # Salida: "Pyt"
print(cadena[2:])       # Salida: "thon"
print(cadena[::2])      # Salida: "Pto"
```

- **Longitud**, La función len() se utiliza para obtener la longitud (cantidad de caracteres) de una cadena.

```
cadena = "Python"
longitud = len(cadena)
print(longitud)  # Salida: 6
```

- **Split**, El método split() se utiliza para dividir una cadena en una lista de subcadenas, utilizando un carácter delimitador; Toma como argumento el carácter delimitador y devuelve una lista de subcadenas separadas por ese delimitador.

```
cadena = "Hola,Python,Mundo"
lista = cadena.split(",")
print(lista)  # Salida: ['Hola', 'Python', 'Mundo']
```

- **Join**, El método join() se utiliza para concatenar una lista de cadenas en una sola cadena, utilizando un separador. Toma como argumento una lista de cadenas y devuelve una cadena que las concatena utilizando el separador especificado.

```
lista = ['Hola', 'Python', 'Mundo']
cadena = ",".join(lista)
print(cadena)  # Salida: "Hola,Python,Mundo"
```

- **Str**, El método str() se utiliza para convertir valores de otros tipos de datos en cadenas de texto. Toma un valor y devuelve su representación en forma de cadena.

```
numero = 123
cadena = str(numero)
print(cadena)  # Salida: "123"
```

- **Lower**, convierte todos los caracteres a minúsculas.

```
cadena = "Hola Mundo"
```

```
print(cadena.lower())        # Salida: "hola mundo"
```

- **Upper**, convierte todos los caracteres a mayúsculas.

```
cadena = "Hola Mundo"
print(cadena.upper())        # Salida: "HOLA MUNDO"
```

- **Capitalize**, convierte el primer carácter a mayúscula y el resto a minúsculas.

```
cadena = "hola mundo"
print(cadena.capitalize())   # Salida: "Hola mundo"
```

Algunos ejemplos adicionales que se pueden realizar con métodos de cadena STR

- **Búsqueda de sub-cadena**, La función find() en Python se utiliza para buscar la primera aparición de una subcadena dentro de una cadena de texto. Devuelve la posición (índice) de la subcadena si se encuentra, o -1 si no se encuentra. También se puede especificar un segundo argumento opcional en la función find() para indicar la posición inicial de búsqueda:

```
texto = "Hola Mundo"

posicion = texto.find("Mundo")
print(posicion)  # Salida: 5

existe = "Mundo" in texto
print(existe)  # Salida: True

posicion = texto.find("o", 3)
print(posicion) # Salida: 9
```

- **Eliminación de espacios en blanco**, La función strip() en Python se utiliza para eliminar los espacios en blanco (u otros caracteres especificados) al inicio y al final de una cadena de texto. Devuelve una nueva cadena con los espacios en blanco eliminados.

```
texto = "   Hola Mundo   "

texto_sin_espacios = texto.strip()
```

```
print(texto_sin_espacios)  # Salida: "Hola Mundo"
```

- **Formateo de cadenas**, La función format() en Python se utiliza para formatear cadenas de texto y combinar valores en ellas. Permite crear cadenas dinámicas y reemplazar marcadores de posición con valores proporcionados.

```
nombre = "John"
edad = 30

mensaje = "Hola, mi nombre es {} y tengo {} años.".format(nombre, edad)
print(mensaje)  # Salida: "Hola, mi nombre es John y tengo 30 años."
```

También puedes especificar índices o nombres para los marcadores de posición, lo que te permite reutilizar los valores en diferentes posiciones:

```
nombre = "Alice"
edad = 25

mensaje = "Hola, mi nombre es {0} y tengo {1} años. Me llamo {0}.".format(nombre, edad)
print(mensaje)
# Salida: "Hola, mi nombre es Alice y tengo 25 años. Me llamo Alice."
```

Además de los índices, también puedes utilizar nombres para los marcadores de posición utilizando la sintaxis {nombre}:

```
persona = {"nombre": "Alice", "edad": 25}

mensaje = "Hola, mi nombre es {nombre} y tengo {edad} años.".format(**persona)
print(mensaje)
# Salida: "Hola, mi nombre es Alice y tengo 25 años."
```

- **Contar ocurrencias de una subcadena**, La función count() en Python se utiliza para contar el número de ocurrencias de una subcadena dentro de una cadena de texto. Devuelve el número de veces que la subcadena aparece en la cadena principal.

```
texto = "Hola Mundo"

ocurrencias = texto.count("o")
print(ocurrencias)  # Salida: 2
```

- **Verificar si la cadena comienza o termina con una subcadena específica,**

La función **startswith**() en Python se utiliza para verificar si una cadena de texto comienza con una subcadena específica. Retorna True si la cadena empieza con la subcadena, y False en caso contrario.

```
texto = "Hola Mundo"

comienza_con_hola = texto.startswith("Hola")
print(comienza_con_hola)  # Salida: True
```

La función **endswith**() en Python se utiliza para verificar si una cadena de texto termina con una subcadena específica. Retorna True si la cadena termina con la subcadena, y False en caso contrario.

```
texto = "Hola, ¿cómo estás?"

termina_con_estas = texto.endswith("estás?")
print(termina_con_estas)
# Salida: True
```

también puedes utilizar **endswith**() y **startswith()** con múltiples subcadenas para realizar verificaciones más complejas:

```
texto = "Hola, ¿cómo estás?"

termina_con_saludo = texto.endswith(("adiós", "bye", "hasta luego"))
print(termina_con_saludo)
# Salida: False
```

- Verificar si la cadena contiene solo caracteres alfabéticos o numéricos:

```
texto1 = "HolaMundo123"
texto2 = "Hola Mundo 123"
```

```
es_alnum1 = texto1.isalnum()
print(es_alnum1)  # Salida: True

es_alnum2 = texto2.isalnum()
print(es_alnum2)  # Salida: False
```

- Verificar si la cadena contiene solo caracteres alfabéticos:

```
texto1 = "HolaMundo"
texto2 = "Hola Mundo 123"

es_alpha1 = texto1.isalpha()
print(es_alpha1)  # Salida: True

es_alpha2 = texto2.isalpha()
print(es_alpha2)  # Salida: False
```

- Verificar si la cadena contiene solo caracteres numéricos:

```
texto1 = "12345"
texto2 = "Hola Mundo 123"

es_digit1 = texto1.isdigit()
print(es_digit1)  # Salida: True

es_digit2 = texto2.isdigit()
print(es_digit2)  # Salida: False
```

B. Funcionalidades para el manejo de cadenas

`import string;` Al importar el módulo `string`, se pueden acceder a diferentes utilidades y funciones relacionadas con el manejo de cadenas. Algunas de las funcionalidades más comunes que se pueden utilizar a través del módulo string incluyen:

- **string.ascii_letters:** Una cadena que contiene todas las letras ASCII en minúsculas y mayúsculas.

```
import string
```

```
print(string.ascii_letters)
# Salida: 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

- **string.ascii_lowercase:** Una cadena que contiene todas las letras ASCII en minúsculas.

```
import string
print(string.ascii_lowercase)
# Salida: 'abcdefghijklmnopqrstuvwxyz'
```

- **string.ascii_uppercase:** Una cadena que contiene todas las letras ASCII en mayúsculas.

```
import string
print(string.ascii_uppercase)
# Salida: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

- **string.digits:** Una cadena que contiene todos los dígitos decimales.

```
import string
print(string.digits)
# Salida: '0123456789'
```

- **string.punctuation:** Una cadena que contiene todos los caracteres de puntuación.

```
import string
print(string.punctuation)
# Salida: '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

ANEXO

Cheatography

Python Cheat Sheet

by yunshu (xys) via cheatography.com/130138/cs/25798/

Basics

abs()	absolute value
hash()	hash value
set()	creat a Set object
all()	True if all elements are true
any()	True if any element is true
min()	return minimum
max()	return maximum
divmod(a,b)	return (a//b,a%b)
hex()	hexadecimal
oct()	octal
bin()	binary
dir()	return all attributes of obj
sorted(*iter*)	return a new sorted list from iter
open(p-ath,mode)	open a file
int()	creat an Int object
str()	return the string form
float()	creat a float obj
list()	creat a List obj
isinstance(o-bj,class)	check if obj belongs to class
ord(c)	return ASCII of c
chr(n)	return char of ASCII
sum(iter)	return sum of iter
filter(pred,iter)	return list of elements meeting pred
pow(a,b)	return a^b
callable(obj)	True if obj is callable
type()	return type of obj
zip()	zip('ab','12') -> a1,b2
map(f,xs)	return ys = f(xs)
round()	rounded number

thread

import threading	
t = threading.Thread(t-arget=fun, args = iter,-name=thread name)	creat a thread
t.start()	start thread t
t.join()	join thread t, other threads waiting until t finishes
lock = threading.Lock()	create a lock
lock.acquire()	current thread acquires the lock
lock.release()	current thead release lock

str-library

s1+s2	string concatenation
s*5	repeating 5 times of s
[0] or [:]	subscription and slice
in/not in	member test
r/R	no escape: r'\n' = '\n' (no new line)
%	string formatting (%d <=> integer))
s.capitalize()	capitalize the first char in s
s.count(x,be-g=0,end=len(s)))	count the number of occurence of x in s
s.endswith(suff-ix,beg=0,end=le-n(s))	check if s ends with suffix (within the given area)
s.startswith(prefi-x,beg=0,end=-len(s))	check if s starts with x
s.expandtabs(ta-bsize=8)	expand the "tab" in s to space
s.find(x,beg=0,-end=len(s))	return start index of x in s if x is in s, else -1

str-library (cont)

s.inde-x(x,be-g=0,en-d=len(s))	similar to find, but raises an exception if x is not in s
s.rindex()	
s.rfind()	
s.isal-num()	True if every char(>=1) in s is number or letter
s.isalpha()	True if every char(>=1) in s is letter
s.isdigit()	True if every char(>=1) in s is number
s.isnu-meric()	True if all characters in the string are numeric(>=1)
s.isde-cimal()	Return True if the string is a decimal string(>=1), False otherwise.
s.issp-ace()	True if s only contains space
s.join()	Concatenate any number of strings using s as delimiter
s.upper()	all to uppercase
s.isupper()	True if all cased chars are supercase(>=1)
s.lower()	all to lowercase
s.islower()	True if all cased chars are lowercase(>=1)
s.lstrip()	return a new string leading whitespace removed
s.strip()	Return a copy of the string with leading and trailing whitespace removed

By **yunshu** (xys)
cheatography.com/xys/

Published 20th December, 2020.
Last updated 20th December, 2020.

Cheatography

Python Cheat Sheet

by yunshu (xys) via cheatography.com/130138/cs/25798/

str-library (cont)

s.rstrip()	Return a copy of the string with trailing whitespace removed.
s.split(del,max-split = s.count(del))	Return a list of the words in the string, using del as the delimiter string
s.splitlines(keepends)	Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.
s.swapcase()	lower <-> upper
s.title()	titilization: all words are capitalized
s.replace(old,new,*max*)	Return a copy with all occurrences of substring old replaced by new

list

[1,2,3]+[4,5,6]	[1,2,3,4,5,6]
arr = [0]*10	Array arr = new Array[10]
l.append(obj)	append obj at end of l
l.count(obj)	count occurence number of obj in l
l.extend(*iter*)	Extend list by appending elements from the iterable
l.index(obj,beg=0,end=len(l))	Return first index of value. Raises ValueError if the value is not present

list (cont)

l.remove(obj)	Remove first occurrence of value. Raises ValueError if the value is not present

l.sort(cmp=None,key=None,reverse=False)

tuple

(1,2)+(3,4)	(1,2,3,4)
(0)*10	(0,0,0,0,0,0,0,0,0,0)

dict (hashtable)

d = {'age':20}	create a dict
d['age'] = 30	add/update value
d.pop(key)	deleting key and value
d.clear()	create a dict
d.get(key,default=None)	get value by key, or default if key not exists
d.has_key(key)	True if d has key
d.items()	a list of (key,value) of d
d.update(d2)	updating (k,v) of d2 to d1
d.pop(key)	delete and return the value pointed by the key
d.popitem()	delete and return a pair of (k,v) randomly

dict features:
1. fast for searching and inserting, which won't be affected by the number of keys
2. occupy a lot of memory

set

s = set([1,2,3])	creat a set
s.add(4)	adding element
s.remove(4)	deleting element
s1 & s2	intersection of sets
s1 | s2	union of sets
s.clear()	clear the set
s.pop()	remove one element randomly

s1.symmetric_difference(s2)

copy

a = li	a: new pointer to li
a = li[:]	first level copy
a = list(li)	first level copy
a = copy.copy(li)	first level copy
a = copy.deepcopy(li)	recursive copy

```
import copy
li = [1,2,3,[4,5]]
```

list generation expression

```
[a+b for a in list1 for b in list2]
```

@property

```
class Student(object):
  @property
  def score(self): return 100
  @score.setter
  def score(self,value): pass
```

the three names (score) should be consistent

regular expression

```
import re
```

re.match(pattern,string,*flags*)	Try to apply the pattern at the start of the string, returning a Match object, or None if no match was found.
re.search(pattern,string,*flag*)	Scan through string looking for a match to the pattern, returning a Match object, or None if no match was found.
matchObject.span()	return (a,b) where a is the start inex and b is the end index of the matching
re.compile(pattern,*flag*)	Compile a regular expression pattern, returning a Pattern object, which can be used in re.match/re.search

Cheatography

Python Cheat Sheet

by yunshu (xys) via cheatography.com/130138/cs/25798/

parameters

func(*args)	accepting any parameters
func(**kw)	accepting only key word parameters

closure

```
def
create_myFunc_at_runtime(*runtime_para):
  def myFunc(x):
    (return x + runtime_para)
    pass
  return myFunc
```

Build A Class: Test

__slots__ = ('name','age')	this class have only 2 attributes now: name & age
__eq__(self,obj)	override "==" operator
__ne__(self,obj)	!=
__le__(self,o)	<=
__ge__(self,o)	>=
__lt__(self,o)	<
__gt__(self,o)	>
__str__(self)	override str()
__repr__(self)	repr()
__len__(self)	len()
__getitem__(-self,n)	subscritable and slice-able
__setitem__(self,-key,value)	supporting item assignment
__call__(self)	-> callable

inheritance

```
overriding __init__:
super(child class,self).__init__(*para)
```

datetime

from datetime import datetime

dt = datetime(201-5,4,19,12,20)	2015-04-19 12:20:00
datetime.now()	current date and time
datetime.strptime('2015-6-1 18:19:59','%Y-%m-%d %H:%M:%S')	str -> datetime
dt.strftime('%a,%b %d %H %M')	datetime -> str
from datetime import timedelta	datetime addition and subtraction

now + timedelta(hours = 10)

now + timedelta(days=1)

JSON

import json

js=json.dump-s(py)	convet from python obj to json
py = json.l-oads(js)	convert from json to python obj