

MÓDULO

2

Área: **NEGOCIOS**

Curso: **PROGRAMACIÓN BÁSICA (PYTHON)**

Módulo: Estructuras básicas de datos y herramientas de control de flujo



IPP

SUEÑA • APRENDE • CRECE



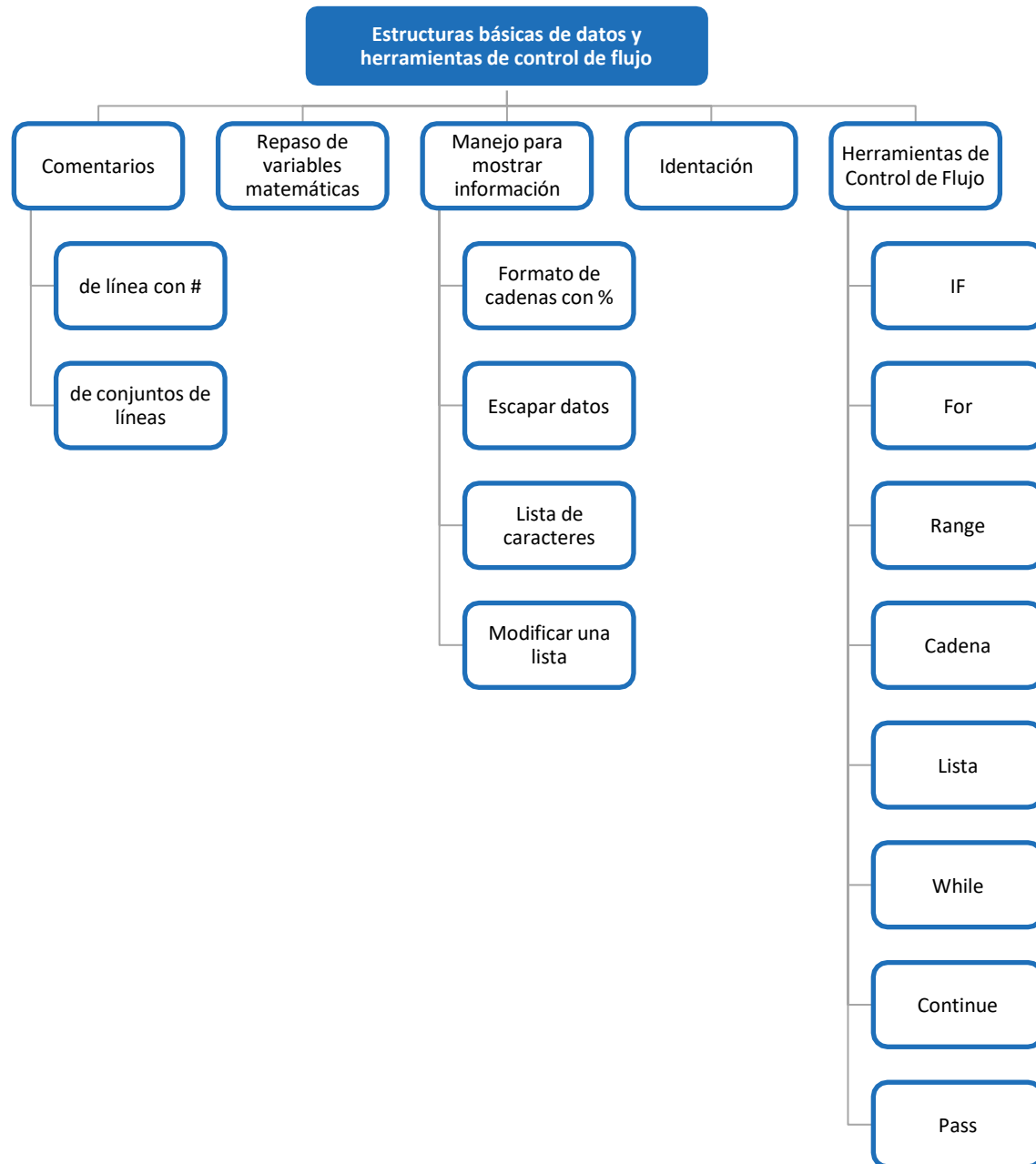
Índice

Contenido

Introducción	1
1. Comentarios	1
1.1 Comentarios de línea con #	1
1.2 Comentarios de conjuntos de líneas	2
2. Repaso de variables matemáticas	2
3. Manejo para mostrar información	3
3.1 Formato de cadenas con %	4
3.2 Escapar datos	6
3.3 Lista de caracteres	7
3.4 Cómo modificar una lista	11
3.4.1 Agregar datos a una lista	11
4. Identación	17
5. Herramientas de control de Flujo	18
5.1 IF	18
5.2 Condiciones de IF	21
5.2.1 Comando AND	21
5.2.2 Comando OR	22
5.2 For	23
5.3 Range	24
5.4 Cadena	24
5.5 Lista	25
5.6 While	26
5.7 Continue	27
5.8 Break	28
5.9 Pass	28
6. Cierre	30



Mapa de Contenido



Resultado de aprendizaje del módulo

- Crea programas simples a partir de estructuras de datos y herramientas de control de flujo, que responden a enunciados escritos.

Introducción

Una de las operaciones más importantes en la computación son las validaciones y los ciclos, en los cuales un código puede tomar una dirección diferente dependiendo de la condición que se le asigne; por ejemplo, escribir "niño" si el número es menor a 18, o "adulto" si es lo contrario.

Es igual de importante tener en cuenta los ciclos. Estos son utilizados cuando hay acciones que se repiten constantemente y no tienen variaciones. Los ciclos pueden ser infinitos, por una cantidad determinada o por una condición.

1. Comentarios

A medida que avanzamos en Python, nuestros códigos son cada vez más largos y complejos. Por lo tanto, es fundamental saber presentar la información correcta para que la siguiente persona que ocupe o vea tu programa lo entienda claramente. Para esto existen los comentarios en Python.

IMPORTANTE

Los comentarios son línea de códigos que el interpretador de Python omite por completo y así se pueden explicar qué tipo de procesos se realizan o para que sirve cada operación.

* Advertencia: en Python los comentarios no pueden llevar tilde

En Python existen dos tipos de comentarios, los comentarios de línea, que se identifican con el símbolo "#", y los comentarios de bloque, que van entre tres comillas dobles seguidas.

1.1 Comentarios de línea con

Ejemplo:

```
ipp.py
C: > Users > User > Desktop > ipp.py > ...
1  a = 5
2  b = 7
3
4  # C será la variable que realizará la suma de las variables a y b
5
6  c = a+b
7
8  print(c) #Acá se muestra el resultado de la suma
9
```

El `#` le dice al código que todo lo que se encuentra a la derecha de él, es un comentario.

1.2 Comentarios de conjuntos de líneas

Se inicia con `"""` (3 comillas dobles) y se termina con `"""` (3 comillas dobles).

Esto sirve para realizar explicaciones largas y por lo general acá van la firma del autor y sus explicaciones.

Ejemplo:

```
ipp.py
C: > Users > User > Desktop > ipp.py > ...
1  """
2  En este programa se realizará la suma de dos variables,
3  una de ellas será la variable "a" que contendrá un número entero y
4  la variable "b" que contendrá un número decimal. Para la realización
5  de una suma correcta es necesario que los valores asignados
6  tengan esos tipos de valores.
7
8  """
9  a = 5
10 b = 2.3
11 c = a + b
12 print(c)
13
```

2. Repaso de variables matemáticas

Ya conocimos anteriormente las siguientes funciones matemáticas en Python.

Sumar (+)

Restar (-)

Multiplicar
(*)

Dividir (/)

Resto (%)

A estas operaciones se le agrega una que tendrá gran utilidad: calcular potencias. Anteriormente y en otros lenguajes de programación, se tenían que agregar otras librerías matemáticas para poder realizar esta operación. pero Python3 viene con una operación ya incorporada que puede realizar potencias.

Potencias (**) (doble asterisco)

Ejemplo:

```

1 variable = 3**2
2 print(variable)
3
4 #el resultado será 9 ya que 3 elevado a 2 es 9
  
```

Print Output: 9

Nos faltan las raíces, pero recuerden que una **raíz es igual que una potencia fraccionaria**, es decir que $\sqrt{4} = 4^{1/2}$, entonces para términos de programación:

```

1 raiz_cuadrada = 4**1/2
2 print(raiz_cuadrada)
3
4 #el resultado será 2 ya que la raíz cuadrada de 4 es 2
  
```

Print Output: 2.0

Esto también ocurre para las raíces enésimas.

3. Manejo para mostrar información

Anteriormente, hemos trabajado con print para mostrar una variable o un número por separado, o si queremos mostrar una información más elaborada, generábamos una variable con la información. Por ejemplo:

Esto se mostrará por pantalla:

```

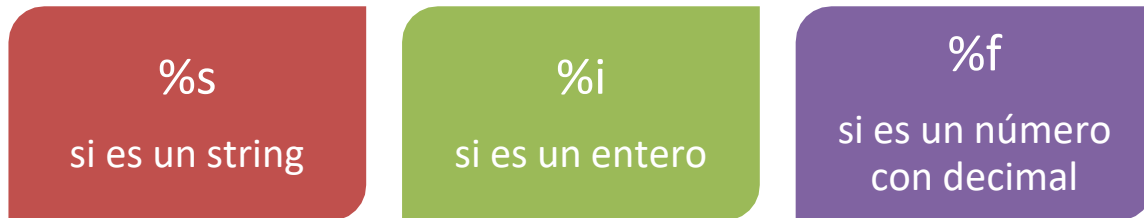
1 a = 5
2 Texto = "El valor de la variable 'a' es: ", a
3 print(Texto)
4
  
```

Print Output: ("El valor de la variable 'a' es: ", 5)

Esto es muy práctico, pero cuando ya tenemos un gran manejo de datos y queremos optimizar la memoria del computador lo máximo posible, existe otro método de mostrar mensajes, que es el método del %.

3.1 Formato de cadenas con %

En este formato sirve principalmente para explicarle al print donde irá el mensaje y de qué tipo será:



Si se quiere mostrar solo una cierta cantidad de decimales se escribe %.3f, en este caso solo mostrara 3 decimales y posterior a declarar todas las variables se agrega un % y después las variables a utilizar.

Ejemplo:

El siguiente código realiza las siguientes operaciones:

- Se asigna el valor 5 a la variable A.
- Se asigna el valor 2.3 a la variable B.
- Se realiza la suma de A y B y se guarda el resultado en la variable C.
- Se asigna un string a la variable Text: "Me encanta la programación".

Luego, se usa la función print para mostrar un mensaje formateado. Se usa el método % para especificar los valores a ser insertados en el string.

Los formatos de los valores a insertar son los siguientes:

- %i para el entero A.
- %.1f para los decimales B y C, que se mostrarán con un decimal, por eso se utiliza un 1.
- %s para el string Text.

```
ipp.py
C: > Users > User > Desktop > ipp.py > ...
1  # Se definen tres variables: A, B y C
2
3  A = 5
4  B = 2.3
5  C = A + B
6
7  # Se define una variable de texto
8
9  Text = "Me encanta la programación"
10
11
12  print("La operación es %i + %.1f = %.1f, Después se muestra el texto:
    %s" % (A, B, C, Text))
13
14
```

La respuesta que mostrará el computador será la siguiente:

```
AREPL - ipp.py X

Print Output:

La operación es 5 + 2.3 = 7.3, Después se muestra el texto: Me
encanta la programación
```

Otro ejemplo:

Este código asigna un string "Sebastián" a la variable Nombre. Después, utiliza la función print dos veces para mostrar por pantalla mensajes con sustituciones de valor.

En la primera llamada a print, se muestra una cadena de texto con una sustitución de valor utilizando el operador %. El valor sustituido es el contenido de la variable Nombre. La letra **s** dentro de las comillas **%s** indica que se espera un valor de tipo string (cadena de texto).

En la segunda llamada a print, se muestra una cadena de texto con varias sustituciones de valor utilizando el operador %. Los valores sustituidos son tres strings: "caballo", "napoleon" y "blanco". La letra **s** dentro de las comillas **%s** indica que se esperan valores de tipo string (cadena de texto).

```
ipp.py X
C: > Users > User > Desktop > ipp.py > ...
1  # Asignación de la variable "Nombre" con un string "Sebastian"
2  Nombre = "Sebastian"
3
4  # Uso de la función "print" para mostrar una cadena de texto con una
   sustitución de valor utilizando el operador %
5  print("Mi nombre es: %s" % Nombre)
6
7  # Uso de la función "print" para mostrar una cadena de texto con varias
   sustituciones de valor utilizando el operador %
8  print("El %s blanco de %s, no era %s" % ("caballo", "napoleon",
   "blanco"))
9
10
11
```

La respuesta que mostrará el computador será la siguiente:

```
AREPL - ipp.py X

Print Output:

Mi nombre es: Sebastian
El caballo blanco de napoleon, no era blanco
```

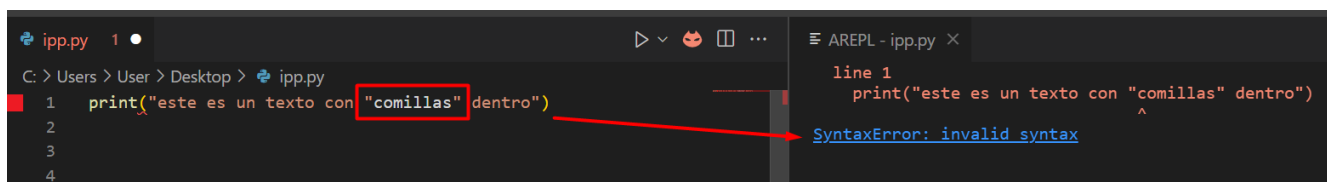

Además de estos ejemplos, hay muchas formas de usar print con el % que brindan mayor velocidad al mostrar los mensajes. A diferencia de la forma anterior en que teníamos que castear las variables a **str** para poder unirlos en una oración

3.2 Escapar datos

El "escapar datos" hace referencia a las veces en que queremos mostrar por pantalla un carácter que Python ya está ocupado en otra tarea.

Un ejemplo fácil de reconocer es el carácter comillas ("). Si escribimos en el print una ("), Python pensará que es el inicio o el final de un **str**. Para poder mostrar estos datos, solo necesitamos agregar una barra invertida (\) antes del carácter que queremos dejar entre comillas y luego una barra invertida al final de ese mismo caracter, por ejemplo:

En el primer caso no se utilizará la barra invertida y solo se escribirán las comillas, nos arrojará un error, porque Python entiende que la cadena termina luego de la palabra "con" ya que ahí se cierra la comilla doble del inicio. Python entiende que la palabra "comillas" no está dentro de la cadena a diferencia de las otras palabras que aparecen en naranja



```

1 print('este es un texto con "comillas" dentro')
2
3
4

```

SyntaxError: invalid syntax

En el segundo caso se utilizará la barra invertida y nos arrojará el mensaje de forma correcta incluyendo las comillas dentro de la respuesta. En este caso la barra invertida le dice a Python que el caracter que sigue después de la barra invertida debe ser tomado en cuenta al momento de enviar el mensaje por pantalla



```

1 print('este es un texto con \"comillas\" dentro')
2
3
4
5

```

Print Output:
este es un texto con "comillas" dentro

3.3 Lista de caracteres

Pensemos que en una sola variable queremos guardar varios datos que tiene relación con lo mismo, como por ejemplo las notas de un alumno.

Nota1 = 4.0

Nota2 = 3.2

Nota3 = 5.0

Nota4 = 6.0

Pero si se dan cuenta tienen la misma relación, por este motivo es que se crearon las listas.

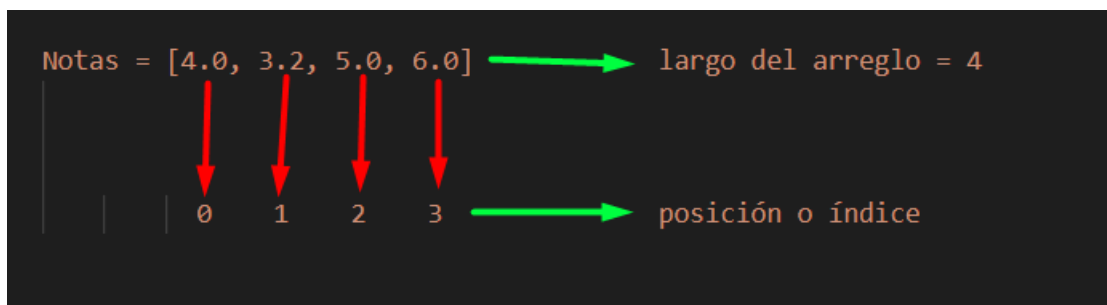
Las listas en Python son un tipo de datos que permite almacenar varios elementos en una sola estructura de datos. Las listas son similares a los arreglos en otros lenguajes de programación, pero en Python son más flexibles y versátiles. Cada elemento de una lista puede ser de cualquier tipo de dato, incluyendo otras listas, lo que permite almacenar una gran cantidad de información en un solo objeto.

Además, las listas en Python son objetos mutables, lo que significa que los elementos de una lista pueden ser cambiados o modificados después de haber sido creados. Por ejemplo, puedes agregar o quitar elementos de una lista, modificar valores existentes, etc.

las listas se identifican generalmente mediante corchetes `[]`.

Notas = [4.0, 3.2, 5.0, 6.0]

Los arreglos en Python son leídos desde 0 hasta n-1, siendo n el largo del arreglo. En nuestro ejemplo la primera nota sería la posición [0] con un 4.0 y la última posición sería [3] con un 6.0.



Otro ejemplo:

Esta lista contiene 6 elementos, cada uno correspondiente a una letra de la palabra "Python". Puedes acceder a cualquier elemento de la lista usando su índice (el número que indica su posición en la lista), empezando desde 0. Por ejemplo, para acceder a la primera letra, podrías usar **primer_indice= lista[0]**, que devolvería 'P'.

```

ipp.py
C: > Users > User > Desktop > ipp.py > ...
1  # Este código muestra el primer índice de una lista de caracteres
2  # En primer lugar, se define una lista llamada "lista" que contiene los caracteres "P",
   # "y", "t", "h", "o" y "n".
3
4  lista = ["P", "y", "t", "h", "o", "n"]
5
6  # Luego, se define una variable "primer_indice" que almacena el primer índice de la
   # lista, que es 0.
7  # La sintaxis para acceder a un índice específico de una lista es la siguiente: lista
   # [índice].
8
9  primer_indice = lista[0]
10
11 # Finalmente, se imprime el resultado usando una sentencia de impresión y se concatena
   # la variable "primer_indice".
12
13 print("El primer índice de la lista es:", primer_indice)
14
15

```

IMPORTANTE

Nota: Las listas de datos se pueden escribir tanto con () o con []. La mayoría de la literatura se escribe con [], pero ambas son válidas.

Las listas definidas con () son conocidas como **tuplas**, pero a diferencia de las listas (definidas con []), las tuplas **no se pueden modificar una vez que se han definido**. Las listas, por otro lado, son fácilmente modificables.

Es importante tener en cuenta que el índice de las listas siempre empieza desde 0 hasta n-1. Por eso es una buena práctica siempre mantener la misma estructura de datos en una lista. Si la lista contiene números, entonces es recomendable no incluir datos de otro tipo, como palabras, ya que, si se incluye una palabra, toda la lista se convierte automáticamente en una cadena de caracteres.

Ahora veremos cómo mostrar información:

En Python, podemos acceder a un carácter de una lista al encerrar su posición entre corchetes. Pero ¿cómo podemos mostrar varios caracteres de la lista? Para esto, existe el concepto de rango, que nos permite seleccionar una serie de posiciones en la lista, especificando un rango desde una posición hasta una posición anterior a la deseada, esto se hará dentro de corchetes, separados por dos puntos. Primero irá el índice que deseas buscar primero y luego donde quieres que termine [inicial : final-1]

Siguiendo con los ejemplos, esto sería:

```
ipp.py
C: > Users > User > Desktop > ipp.py > ...
1 lista = ["Python", "es", "un", "lenguaje", "de", "programación"]
2 segmento = lista[1:4]
3 print(segmento)
4
5
```

Y este es el mensaje que nos arrojará por pantalla

```
Print Output:
['es', 'un', 'lenguaje']
```

Si te das cuenta, la palabra “Python” comienza en el índice [0], por lo que en la posición [1] está la palabra “es”. Pero al querer que nos de la palabra que está en la posición [4] que es la palabra “de” este nos arroja la palabra ubicada en el índice [3] que es la palabra “lenguaje” esto pasa porque el rango nos permite seleccionar una serie de posiciones en la lista, especificando un rango desde una posición hasta una posición **anterior a la deseada** por lo que si deseas obtener la palabra “de” deberías escribir en tu código el índice [5]

```
ipp.py
C: > Users > User > Desktop > ipp.py > ...
1 lista = ["Python", "es", "un", "lenguaje", "de", "programación"]
2 segmento = lista[1:5]
3 print(segmento)
4
5
```

Y esta es la salida:

```
Print Output:
['es', 'un', 'lenguaje', 'de']
```

Por el mismo concepto del anterior que número empieza en 1 y es menor que 3, las únicas posiciones que cumplen esta condición son 1 y 2 solamente.

Ahora digamos que quiero mostrar los primeros 4 datos, esto se tendría que hacer.

```
ipp.py
C: > Users > User > Desktop > ipp.py > ...
1  lista = ["Python", "es", "un", "lenguaje", "de", "programación"]
2  segmento = lista[:4]
3  print(segmento)
4
```

El computador nos mostrará

Print Output:

```
['Python', 'es', 'un', 'lenguaje']
```

Esto ocurre porque le dijimos al computador que nos muestre los datos desde el inicio hasta el número menor a 4, es decir 0, 1, 2 y 3 que fueron lo que se desplegaron.

Ahora si quisiéramos mostrar los últimos 2 datos de la lista se realizaría de la siguiente manera.

```
ipp.py
C: > Users > User > Desktop > ipp.py > ...
1  lista = ["Python", "es", "un", "lenguaje", "de", "programación"]
2  segmento = lista[4:]
3  print(segmento)
4
5
```

El computador nos mostrará

Print Output:

```
['de', 'programación']
```

3.4 Cómo modificar una lista

IMPORTANTE

NOTA: Los comandos que se presentarán a continuación solo serán efectivos si se han declarado las listas utilizando []. Esto se debe a que si se utilizan paréntesis (), Python las interpreta como Tuplas, las cuales solo permiten realizar las operaciones básicas que se han visto previamente.

3.4.1 Agregar datos a una lista

1. append

append(elemento que quieres agregar): Agrega un elemento al final de la lista.

```
python

>>> lista = [1, 2, 3]
>>> lista.append(4)
>>> print(lista)
[1, 2, 3, 4]
```

2. insert

insert(índice, elemento): Inserta un elemento en una posición específica de la lista. El primer argumento de insert es el índice o posición en el que se desea insertar el elemento, y el segundo argumento es el elemento en sí.

```
python

>>> lista = [1, 2, 3]
>>> lista.insert(1, 4)
>>> print(lista)
[1, 4, 2, 3]
```

```
python

# Crea una lista con los números del 1 al 5
>>> lista = [1, 2, 3, 4, 5]

# Inserta un 0 al principio de la lista
>>> lista.insert(0, 0)

# Imprime la lista resultante
>>> print(lista)
[0, 1, 2, 3, 4, 5]
```

3. remove

remove(elemento): Elimina el primer elemento de la lista (de izquierda a derecha) que coincida con el valor especificado en los argumentos. *Al borrar un dato que no existen en el arreglo Python arrojará un error y dejará de ejecutar todo el código.*

```
python

>>> lista = [1, 2, 3, 2]
>>> lista.remove(2)
>>> print(lista)
[1, 3, 2]
```

4. clear

clear(): Elimina todos los elementos de la lista.

```
python

>>> lista = [1, 2, 3]
>>> lista.clear()
>>> print(lista)
[]
```

5. Búsqueda

En Python, puedes realizar una búsqueda en una lista usando la sentencia **in**. La sentencia **in** devuelve **True** si un elemento especificado está en la lista y **False** en caso contrario. Aquí hay un ejemplo:

```
python

>>> lista = [1, 2, 3, 4, 5]
>>> print(3 in lista)
True
>>> print(6 in lista)
False
```

En el ejemplo anterior, se crea una lista con los números del 1 al 5 y luego se usa la sentencia **in** para buscar si el número 3 y el número 6 están en la lista. La sentencia **in** devuelve **True** si el número 3 está en la lista y **False** si el número 6 no está en la lista.

Otra forma de buscar un elemento en una lista es usar el método **index**. Este método devuelve el índice del primer elemento que coincide con el valor especificado. Aquí hay un ejemplo:

```
python

>>> lista = [1, 2, 3, 4, 5]
>>> print(lista.index(3))
2
>>> print(lista.index(6))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 6 is not in list
```

En el ejemplo anterior, se crea una lista con los números del 1 al 5 y luego se usa el método **index** para buscar el índice del número 3 y el número 6 en la lista. El método **index** devuelve el índice 2 si el número 3 está en la lista y lanza una excepción **ValueError** si el número 6 no está en la lista.

6. extend

extend(lista2): Agrega todos los elementos de otra lista al final de la lista actual.

```
python

>>> lista1 = [1, 2, 3]
>>> lista2 = [4, 5, 6]
>>> lista1.extend(lista2)
>>> print(lista1)
[1, 2, 3, 4, 5, 6]
```

7. pop

pop(índice): Elimina el elemento en la posición especificada y lo devuelve. Si no se proporciona un índice, se elimina el último elemento de la lista.

```
python

>>> lista = [1, 2, 3]
>>> item = lista.pop(1)
>>> print(lista)
[1, 3]
>>> print(item)
2
```


i. Obtener índice de una búsqueda

El método **index** en Python es un método que se puede usar en listas para buscar el índice de un elemento específico en la lista. El método devuelve el índice del primer elemento que coincide con el valor especificado. Si el elemento no se encuentra en la lista, se lanzará una excepción **ValueError**.

```
python

# Crea una lista con los números del 1 al 5
>>> lista = [1, 2, 3, 4, 5]

# Busca el índice del número 3 en la lista
>>> print(lista.index(3))
2

# Busca el índice del número 6 en la lista
>>> print(lista.index(6))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 6 is not in list

# Busca el índice de la primera ocurrencia de 3 en la lista
>>> lista.append(3)
>>> print(lista.index(3))
2
```

En el ejemplo anterior, se crea una lista con los números del 1 al 5 y luego se usa el método **index** para buscar el índice del número 3 y el número 6 en la lista. El método **index** devuelve el índice 2 si el número 3 está en la lista y lanza una excepción **ValueError** si el número 6 no está en la lista. También se muestra cómo usar **index** para buscar la primera ocurrencia de un elemento en la lista, después de agregar una segunda ocurrencia del número 3 a la lista.

ii. Cadena de Texto (string)

En Python, una variable de tipo string es en realidad una secuencia de caracteres (char), que se puede pensar como una lista de caracteres individuales. Cada carácter en un string puede ser accedido y manipulado por su índice, lo que permite una gran cantidad de operaciones útiles sobre strings en Python.

IMPORTANTE

Nota: Una cadena de texto se comporta como una Dupla, teniendo comandos específicos para su modificación.

Las funciones y métodos que se proporcionan en Python para trabajar con strings son herramientas muy útiles que te permiten realizar una variedad de tareas comunes con strings, como manipular su contenido, verificar su longitud, reemplazar subcadenas, dividir strings en subcadenas, y mucho más. Estas funciones y métodos se pueden aplicar de manera muy sencilla y efectiva a tus strings, lo que te permite trabajar con ellos de manera más eficiente y efectiva. Además, estas funciones y métodos son ampliamente compatibles con otras estructuras de datos en Python, lo que significa que puedes usarlos para trabajar con strings en combinación con otras estructuras de datos, como listas y diccionarios, para crear soluciones más complejas.

1. **len():** Devuelve la longitud de un string

```
python

>>> nombre = "Juan"
>>> print(len(nombre))
4
```

2. **str.lower():** Convierte todos los caracteres de un string a minúsculas.

```
python

>>> frase = "HOLA MUNDO"
>>> print(frase.lower())
hola mundo
```

3. **str.upper():** Convierte todos los caracteres de un string a mayúsculas.

```
python

>>> frase = "hola mundo"
>>> print(frase.upper())
HOLA MUNDO
```

4. **str.replace()**: Reemplaza una subcadena dentro de un string con otra subcadena.

```
python

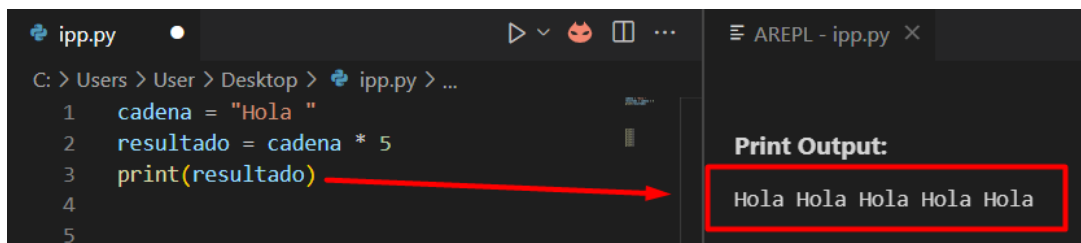
>>> frase = "hola mundo"
>>> print(frase.replace("mundo", "amigo"))
hola amigo
```

5. **str.split()**: Divide un string en una lista de subcadenas en función de un separador específico.

```
python

>>> frase = "hola,mundo"
>>> print(frase.split(","))
['hola', 'mundo']
```

También las cadenas de caracteres se pueden multiplicar en Python3. Ejemplo:



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

```
1 cadena = "Hola "
2 resultado = cadena * 5
3 print(resultado)
4
5
```

A red arrow points from the `print(resultado)` line to the output window. The output window, titled 'Print Output:', displays the result: 'Hola Hola Hola Hola Hola'.

ACTIVIDAD

Ejercicios formativos:

Crear un pequeño texto, no más de 5 palabras.

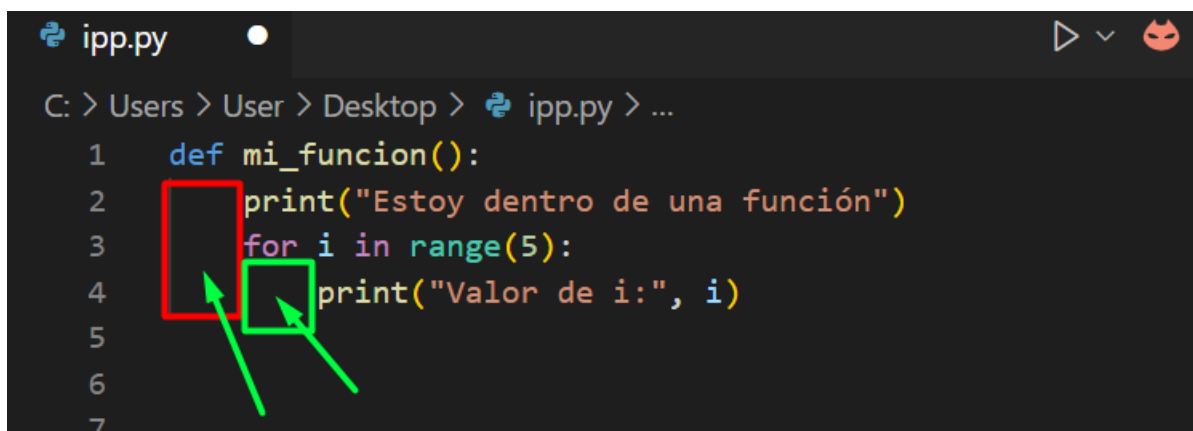
1. Mostrar por pantalla los primeros 6 caracteres del texto creado.
2. Mostrar por pantalla los últimos 4 caracteres del texto creado.
3. Mostrar por pantalla entre tercer y décimo carácter del texto creado.
4. Al texto creado reemplace la letra "a" con "ipp"
5. Busque en la cadena de caracteres la posición de la primera i.
6. Cree una lista de datos, que contenga string y números.
7. Agrega una variable en el texto, en la 4 posición.
8. Elimine la información de la posición 3.

4. Identación

La indentación es un aspecto importante de la sintaxis en Python. Se refiere a la cantidad de espacio en blanco al inicio de una línea de código. La indentación se utiliza en Python para definir bloques de código, tales como funciones, bucles y condicionales.

En Python, se recomienda usar una indentación de 4 espacios por nivel. Esto significa que, por ejemplo, si tienes un bloque de código dentro de una función, todas las líneas de código dentro de esa función deben estar indentadas con 4 espacios en relación con la línea que define la función. Aquí hay un ejemplo:

Ejemplo:



```

1  def mi_funcion():
2      print("Estoy dentro de una función")
3      for i in range(5):
4          print("Valor de i:", i)
5
6
7

```

En este ejemplo se ve que el **print("Estoy dentro de una función")** y el ciclo **for** (que lo veremos pronto), tienen indentación, y a su vez el siguiente **print("valor de i: ",i)** también se encuentra indentado.

IMPORTANTE

Hay una advertencia importante a tener en cuenta al trabajar con indentación en Python: Si has estado utilizando TAB para indentar tu código, debes saber que no puedes cambiar a 4 espacios y viceversa. Esto se debe a que el interpretador de Python utiliza la indentación para determinar la estructura del código y, si cambias la forma en que la indentación se realiza (ya sea usando TAB o 4 espacios), el interpretador no será capaz de reconocer correctamente la estructura del código. Por lo tanto, es importante elegir una forma de indentación y mantenerla consistente a lo largo de todo el código.

5. Herramientas de control de Flujo

IMPORTANTE

Se recomienda trabajar en un archivo con extensión .py durante todo este módulo. Esto se debe a que probablemente tendrás que escribir varias líneas de código antes de obtener algún resultado y escribir directamente en la consola de Python puede resultar un poco incómodo. Ejecuta el archivo .py para ver los resultados.

Todo lo que hemos visto hasta ahora es cómo el programa trabaja linealmente, es decir, solo avanza en orden. A continuación, crearemos programas más inteligentes, capaces de tomar decisiones, repetir un flujo un número determinado de veces o ejecutar algo hasta que se cumpla una condición. Para ello, existen diversas herramientas de control de flujo.

Las herramientas de control de flujo son elementos en un lenguaje de programación que te permiten controlar el flujo de ejecución de un programa. Esto significa que puedes decirle a tu programa que haga una tarea determinada **solo si se cumple cierta condición**, o que repita una tarea múltiples veces **hasta que se cumpla otra condición**.

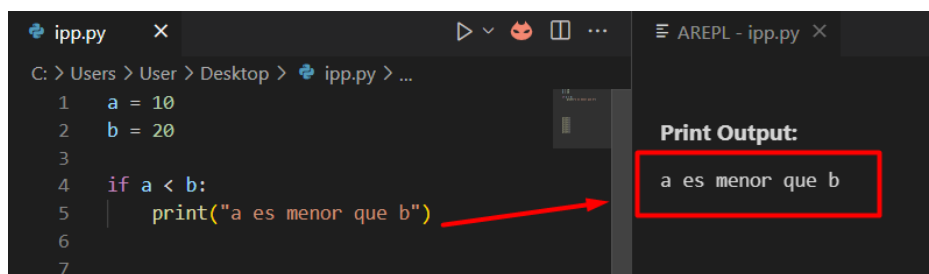
Hay tres herramientas de control de flujo principales en Python: **if**, **for** y **while**.

5.1 IF

IF, en Python, es la representación de "SI", **If** es una sentencia que te permite controlar la ejecución de un bloque de código **solo si se cumple una determinada condición**. Aquí tienes un ejemplo:

Después de cada if se debe terminar la línea en : (dos puntos), y después colocar el salto de línea y indentar.

Ejemplo:



```

1  a = 10
2  b = 20
3
4  if a < b:
5      print("a es menor que b")
6
7

```

The screenshot shows a code editor with the above Python code. A red arrow points from the `print` statement to the output window on the right, which displays "a es menor que b" under the heading "Print Output:". The output is enclosed in a red box.

En este ejemplo, la sentencia **if** comprueba si **a** es menor que **b**. Si se cumple la condición, el código dentro del bloque **if** se ejecutará. En este caso, se imprimirá "a es menor que b"

Pero esto por sí solo no es muy útil, porque ¿qué pasa si no se cumple la condición? Para esto existe un comando que es **ELSE**, es la representación de “**SINO**”.

El **else** se ejecuta en caso de que la condición especificada en el **if** sea falsa. Es decir, el código dentro del bloque **else** se ejecutará solo si la condición especificada en el **if** es falsa. Aquí hay un ejemplo de código:

```

1  x = 5
2  if x > 10:
3      print("x es mayor que 10")
4  else:
5      print("x es menor o igual a 10")
6
7

```

Print Output:
x es menor o igual a 10

En este ejemplo, la condición en el **if** es "**x > 10**", por lo que x es menor o igual a 10. Por lo tanto, el código dentro del bloque **else** se ejecutará y se imprimirá "x es menor o igual a 10".

De momento solo hemos visto la condición que dice **>=** (mayor o igual), pero **if** tiene varias condicionantes, las cuales son:

< (Menor)	<= (Menor igual)	> (Mayor)
>= (Mayor igual)	== (Igual)	!= (Distinto)

IMPORTANTE

Recuerden que las comparaciones de datos deben ser del mismo tipo, si estamos viendo mayor que o menor que, significa que es un dato de tipo numérico entonces la variable a comparar debe ser **INT** o **FLOAT**.

Los operadores de comparación en Python son similares a los de otros lenguajes de programación y se utilizan para comparar dos valores y determinar si se cumplen o no ciertas condiciones.

A continuación, se describen los operadores de comparación más comunes en Python:

```

ipp.py
C: > Users > User > Desktop > ipp.py
1  # Operadores de comparación en Python
2
3  # Ejemplo 1: Uso del operador == (igual a)
4  print(5 == 5) # Imprime True
5  print(5 == 6) # Imprime False
6
7  # Ejemplo 2: Uso del operador != (diferente a)
8  print(5 != 5) # Imprime False
9  print(5 != 6) # Imprime True
10
11 # Ejemplo 3: Uso del operador > (mayor que)
12 print(5 > 4) # Imprime True
13 print(5 > 5) # Imprime False
14
15 # Ejemplo 4: Uso del operador < (menor que)
16 print(5 < 6) # Imprime True
17 print(5 < 5) # Imprime False
18
19 # Ejemplo 5: Uso del operador >= (mayor o igual que)
20 print(5 >= 5) # Imprime True
21 print(5 >= 6) # Imprime False
22
23 # Ejemplo 6: Uso del operador <= (menor o igual que)
24 print(5 <= 5) # Imprime True
25 print(5 <= 4) # Imprime False
26
27
28

```

Pero pensemos que queremos colocar una nueva condición si no se cumple la primera, siguiendo el ejemplo, si la persona no tiene 15 ver si es menor o mayor de edad, para poder hacer esto existe comando ELIF (que es una mezcla de ELSE y IF), ejemplo:

```

ipp.py X
> Users > User > Desktop > ipp.py > ...
1  # Solicitamos la edad del usuario
2
3  edad = int(input("Ingrese su edad: "))
4
5  # Comparamos la edad del usuario
6
7  if edad == 15:
8      print("Feliz cumpleaños de 15 años")
9  elif edad >= 18:
10     print("Feliz cumpleaños, eres mayor de edad")
11 else:
12     print("Feliz cumpleaños, eres menor de edad")
13

```

5.2 Condiciones de IF

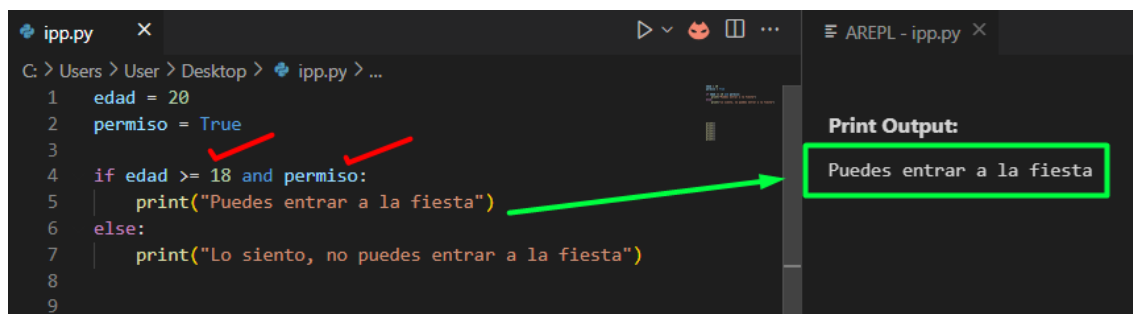
Digamos que quieres hacer un programa que tenga más de una validación para una ejecución. Ejemplo, quiero conducir un auto, pero para esta acción necesito ser mayor de edad y también tener licencia de conducir o sea necesito que se cumplan dos condiciones para poder manejar

Para estos casos Python tiene el comando **and** (AND, en español “y”) y el comando **or** (OR, en español “o”).

5.2.1 Comando AND

El operador **and** se utiliza para verificar si ambas condiciones son verdaderas. En otras palabras, **solo si ambas condiciones son verdaderas**, se considera que la expresión lógica completa es verdadera.

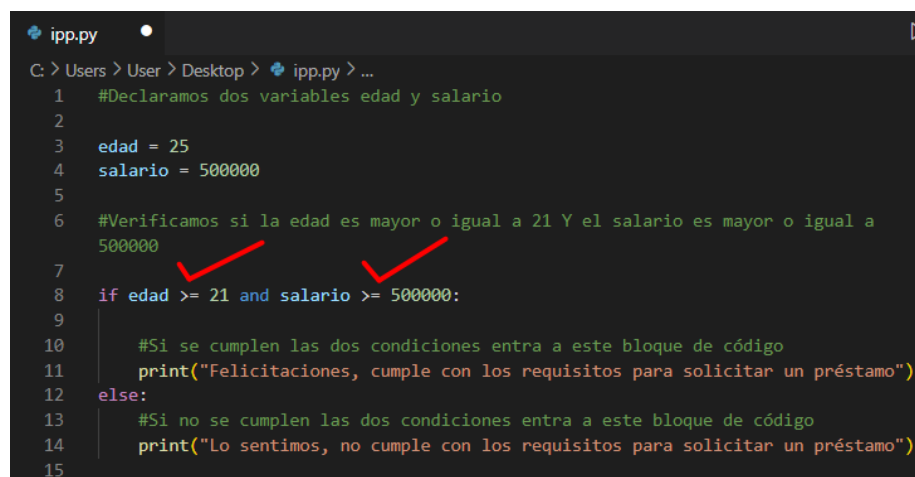
Ejemplo:



```

ipp.py
C: > Users > User > Desktop > ipp.py > ...
1 edad = 20
2 permiso = True
3
4 if edad >= 18 and permiso:
5     print("Puedes entrar a la fiesta")
6 else:
7     print("Lo siento, no puedes entrar a la fiesta")
8
9
Print Output:
Puedes entrar a la fiesta
  
```

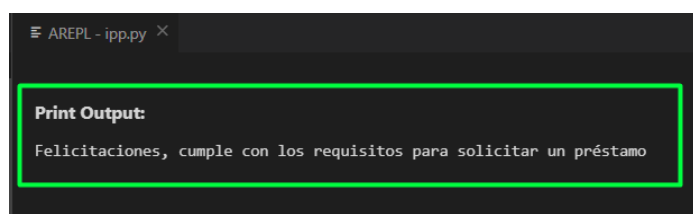
En el ejemplo anterior, la persona solo podrá entrar a la fiesta si su edad es mayor o igual a 18 años y tiene el permiso.



```

ipp.py
C: > Users > User > Desktop > ipp.py > ...
1 #Declaramos dos variables edad y salario
2
3 edad = 25
4 salario = 500000
5
6 #Verificamos si la edad es mayor o igual a 21 Y el salario es mayor o igual a 500000
7
8 if edad >= 21 and salario >= 500000:
9
10     #Si se cumplen las dos condiciones entra a este bloque de código
11     print("Felicitaciones, cumple con los requisitos para solicitar un préstamo")
12 else:
13     #Si no se cumplen las dos condiciones entra a este bloque de código
14     print("Lo sentimos, no cumple con los requisitos para solicitar un préstamo")
15
  
```

Y esta es la salida:



```

AREPL - ipp.py
Print Output:
Felicitaciones, cumple con los requisitos para solicitar un préstamo
  
```


5.2.2 Comando OR

El operador **or** se utiliza para verificar **si al menos una de las condiciones es verdadera**. En otras palabras, solo si al menos una de las condiciones es verdadera, se considera que **la expresión lógica completa es verdadera**.

Ejemplo:

```

1  edad = 17
2  invitacion = True
3
4  if edad >= 18 or invitacion:
5      print("Puedes entrar a la fiesta")
6  else:
7      print("Lo siento, no puedes entrar a la fiesta")
8
9

```

Print Output:
Puedes entrar a la fiesta

En este ejemplo, la persona podrá entrar a la fiesta si su edad es mayor o igual a 18 años o si tiene una invitación, si cumple con al menos una de esas condiciones, podrá entrar a la fiesta.

Como sería ese mismo ejemplo, pero utilizando **and**, recordemos que el operador **and** necesita que **todas** las condiciones se cumplan.

```

1  edad = 17
2  invitacion = True
3
4  if edad >= 18 and invitacion:
5      print("Puedes entrar a la fiesta")
6  else:
7      print("Lo siento, no puedes entrar a la fiesta")
8
9

```

Print Output:
Lo siento, no puedes entrar a la fiesta

Al hacer un **if**, siempre se está buscando una validación **True** o **False**.

Si escriben en la consola de Python, `10 >= 3`, se darán cuenta que retornará un **True**, o si lo escriben al revés `3 >= 10` retornará un **False**.

En Python, trabajar con variables booleanas no requiere especificar `"== True"` o `"== False"` ya que la sola presencia de la variable indicará si es verdadera o falsa. Por lo tanto, se puede simplificar la escritura de las condiciones en el código

5.2 For

Imagínense que les pido que impriman 200 números correlativos, es decir, del 1 al 200. Podrían pensar que tendrían que escribir 200 veces `print()`, pero en cualquier tipo de programación se pueden realizar ciclos, y uno de ellos es el **for**, que en pseudo código se traduciría como "**para**"

El ciclo **for** en Python es una estructura de control que nos **permite repetir un bloque de código una cantidad determinada de veces**. Esta estructura se basa en una lista o un rango de valores y se utiliza para recorrer estos valores y realizar una tarea específica por cada uno de ellos.

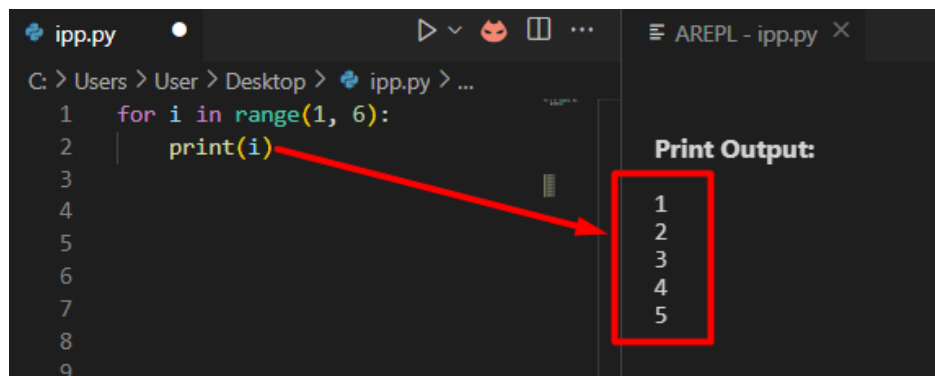
La sintaxis general del ciclo **for** es la siguiente:

```
for variable in secuencia:  
    # Aquí se escriben las instrucciones que se deben repetir
```

Donde:

- **variable**: es una variable que se usa para recorrer cada uno de los elementos de la **secuencia**
- **secuencia** puede ser una lista, una tupla, un conjunto, un rango o cualquier objeto iterable.
- Las instrucciones dentro del ciclo deben estar indentadas con al menos un espacio o tabulación, para indicar que forman parte del ciclo.

Por ejemplo, para imprimir los números del 1 al 5, podemos escribir el siguiente código:



The screenshot shows a code editor with a file named `ipp.py`. The code is as follows:

```
1 for i in range(1, 6):  
2     print(i)  
3  
4  
5  
6  
7  
8  
9
```

A red arrow points from the `print(i)` statement to the output area on the right. The output area is titled "Print Output:" and displays the numbers 1 through 5, each on a new line. The output is enclosed in a red rectangular box.

Donde **i** es la variable que se usa para recorrer los elementos del rango (1, 6), y dentro del ciclo, se imprime el valor de **i** en cada iteración.

5.3 Range

Este es un objeto que se utiliza para especificar la secuencia de números que se van a repetir en un ciclo for

Sigamos el ejemplo de imprimir 200 números seguidos, esto sería:

```
ipp.py
C: > Users > User > Desktop > ipp.py > ...
1 # Este código imprime los números del 0 al 199
2 for i in range(200):
3     print(i)
4
5 # Este código imprime los números del 1 al 200
6 for i in range(1,201):
7     print(i)
8
9 # Este código imprime los números pares del 0 al 200
10 for i in range(0,201,2):
11     print(i)
12
13
```

5.4 Cadena

Ahora queremos recorrer una cadena de texto. Ejemplo:

```
ipp.py
C: > Users > User > Desktop > ipp.py > ...
1 # Define la variable "Palabra" como una string
2 Palabra = "Hola mundo IPP"
3
4 # Recorre cada carácter de la variable "Palabra"
5 for i in Palabra:
6     # Imprime cada carácter de la variable "Palabra" con un
    mensaje
7     print("La palabra separada por caracteres es: " + i)
8
```

Lo que el computador mostraría sería:

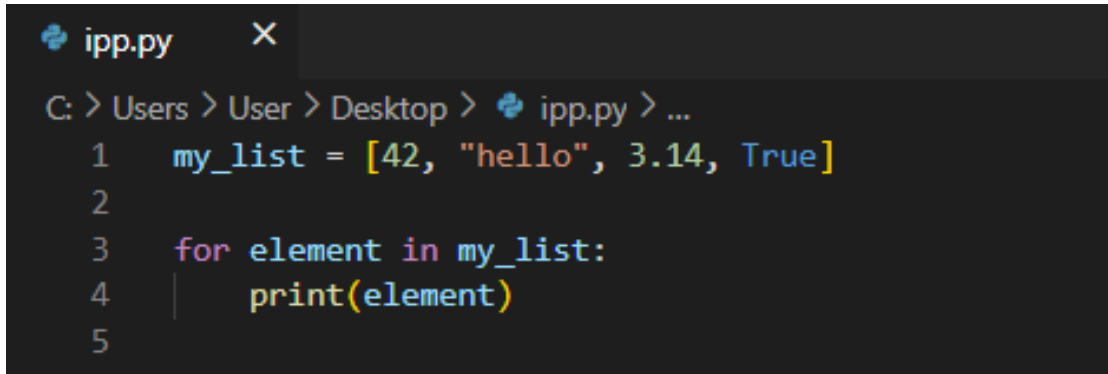
```
Print Output:
La palabra separada por caracteres es: H
La palabra separada por caracteres es: o
La palabra separada por caracteres es: l
La palabra separada por caracteres es: a
La palabra separada por caracteres es: m
La palabra separada por caracteres es: u
La palabra separada por caracteres es: n
La palabra separada por caracteres es: d
La palabra separada por caracteres es: o
La palabra separada por caracteres es: I
La palabra separada por caracteres es: P
La palabra separada por caracteres es: P
```

5.5 Lista

Ahora queremos recorrer cada carácter de una lista.

```
my_list = [42, "hello", 3.14, True]
```

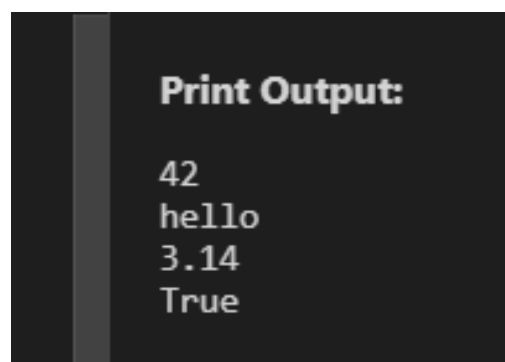
Tenemos esta siguiente lista que es mixta con caracteres, números y operaciones booleanas. La queremos recorrer, se recorre de la siguiente forma:



```
ipp.py X
C: > Users > User > Desktop > ipp.py > ...
1  my_list = [42, "hello", 3.14, True]
2
3  for element in my_list:
4      print(element)
5
```

En este ejemplo, la lista `my_list` contiene cuatro elementos: un número entero, una cadena de caracteres, un número decimal y un valor booleano. El bucle `for` itera a través de cada elemento de la lista, y en cada iteración, imprime el elemento

El resultado entregado por el computador es:



```
Print Output:
42
hello
3.14
True
```

5.6 While

El ciclo while es un tipo de estructura de control de flujo en programación que **se utiliza para repetir una acción mientras se cumpla una determinada condición**. Es decir, mientras se cumpla la condición, el código dentro del ciclo **se ejecutará repetidamente**. La sintaxis básica de un ciclo while es la siguiente:

Sintaxis:

```
while condición:
    # código a ejecutar mientras se cumpla la condición
```

La condición es una expresión que se evalúa como verdadera o falsa. Si la condición es verdadera, el bloque de código se ejecuta. Después de la ejecución del bloque de código, la condición se vuelve a evaluar. Si sigue siendo verdadera, el bloque de código se ejecuta de nuevo. Este proceso se repite hasta que la condición se evalúe como falsa.

Es importante tener cuidado con el uso del ciclo while, ya que, **si no se proporciona una forma de detener la ejecución, el ciclo se repetirá indefinidamente, causando un bucle infinito**.

Ejemplo:

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

```
1 # Este código imprime los números del 1 al 5
2 # Se inicializa la variable i con 1
3
4 i = 1
5
6 # Mientras i sea menor o igual a 5, se ejecutará el ciclo
7
8 while i <= 5:
9     # Se imprime el valor actual de i
10    print(i)
11    # Se aumenta i en 1 para que el ciclo avance
12    i += 1
13
```

On the right side of the editor, there is a 'Print Output' window showing the results of the program:

```
1
2
3
4
5
```

A red arrow points from the 'print(i)' statement in the code to the output window, indicating that the code is executing and printing the values 1 through 5.

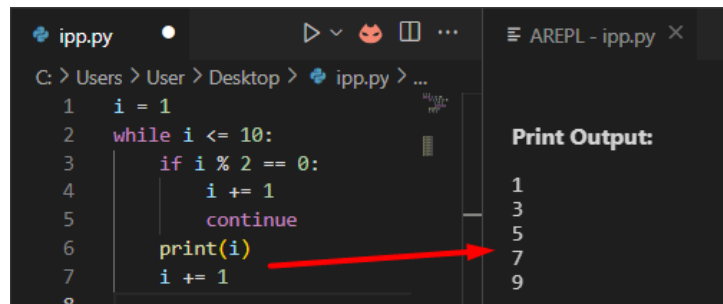
Este código **detiene su ejecución porque se ha cumplido la condición** que establece el ciclo while. La estructura de un ciclo while es: "mientras se cumpla esta condición, ejecuta este bloque de código". En este caso, la condición es " $i \leq 5$ ".

El ciclo while comienza con la variable "i" iniciada en 1, y se ejecuta mientras "i" sea menor o igual a 5. Cada vez que se ejecuta el ciclo, se imprime el valor de "i" y se suma 1 a su valor ($i += 1$). Cuando "i" alcanza el valor de 6, ya no se cumple la condición " $i \leq 5$ " y el ciclo se detiene.

5.7 Continue

El `continue` en un ciclo `while` de Python se utiliza para saltar a la siguiente iteración del ciclo, sin ejecutar las sentencias restantes en la iteración actual. Esto significa que, cuando se encuentra una sentencia `continue`, se abandonará la iteración actual y se pasará directamente a la siguiente.

Por ejemplo, supongamos que queremos imprimir todos los números impares del 1 al 10. Podemos hacer esto con el siguiente código:



```

1 i = 1
2 while i <= 10:
3     if i % 2 == 0:
4         i += 1
5         continue
6     print(i)
7     i += 1
8

```

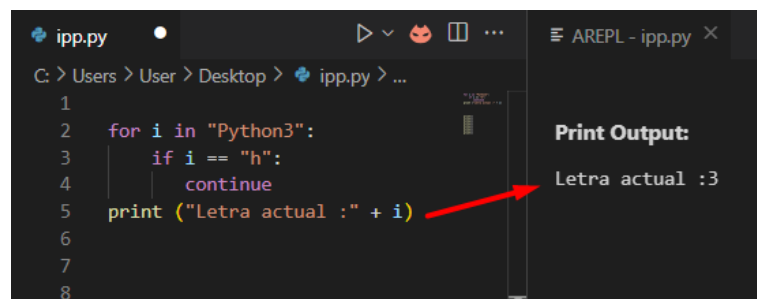
Print Output:

```

1
3
5
7
9

```

En el siguiente ejemplo se ve más claro:



```

1
2 for i in "Python3":
3     if i == "h":
4         continue
5     print ("Letra actual :" + i)
6
7
8

```

Print Output:

```

Letra actual :3

```

Este código es un ejemplo de un bucle `for` en Python, que se utiliza para recorrer una secuencia de elementos. En este caso, la secuencia es una cadena de texto `"Python3"`.

En cada iteración del bucle, la variable `"i"` toma el valor de una letra en la cadena `"Python3"`. La instrucción `"if i == 'h'"` comprueba si la letra actual es `"h"`. Si es así, se ejecuta la instrucción `"continue"`, que hace que el programa salte a la siguiente iteración del bucle, sin ejecutar ninguna otra instrucción dentro del bucle.

Si la letra actual no es `"h"`, el programa imprime `"Letra actual: [letra actual]"`.

La salida de este programa sería:

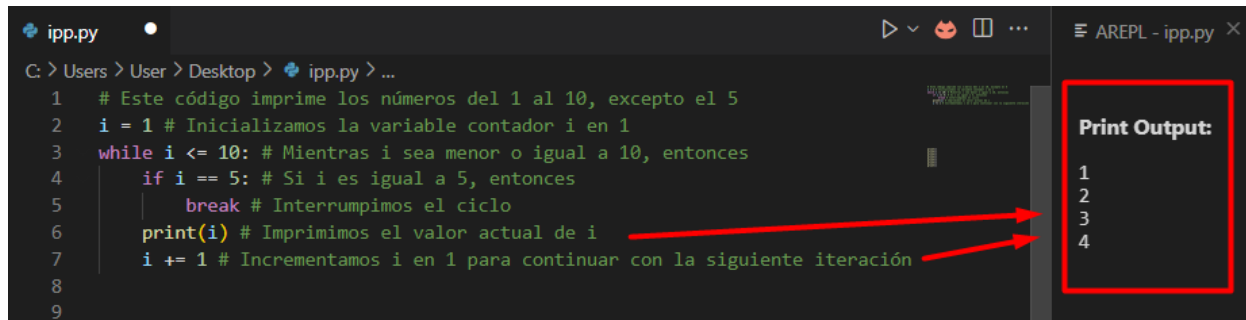
```

"Letra actual: P
Letra actual: y
Letra actual: t
Letra actual: o
Letra actual: n
Letra actual: 3"

```

5.8 Break

El ciclo while con la instrucción break se utiliza para interrumpir el ciclo cuando se cumple una determinada condición. Aquí te muestro un ejemplo:



```

1  # Este código imprime los números del 1 al 10, excepto el 5
2  i = 1 # Inicializamos la variable contador i en 1
3  while i <= 10: # Mientras i sea menor o igual a 10, entonces
4      if i == 5: # Si i es igual a 5, entonces
5          break # Interrumpimos el ciclo
6      print(i) # Imprimimos el valor actual de i
7      i += 1 # Incrementamos i en 1 para continuar con la siguiente iteración
8
9

```

Print Output:

```

1
2
3
4

```

Este código utiliza un ciclo "while" para imprimir los números del 1 al 10, excepto el número 5. La variable "i" es inicializada en 1 y el ciclo "while" continuará mientras "i" sea menor o igual a 10. Dentro del ciclo "while" hay una sentencia "if" que verifica si "i" es igual a 5. Si es así, se ejecuta la sentencia "break", lo que hace que el ciclo se detenga inmediatamente y el programa continúe con la siguiente instrucción después del ciclo. Si "i" no es igual a 5, se imprime el valor de "i" y luego se incrementa en 1 para continuar con la siguiente iteración. La salida será los números del 1 al 4, ya que el número 5 se detiene con la sentencia "break".

5.9 Pass

Pass, es una variable pasiva que se ocupa cuando no quieres que se haga nada dentro de un ciclo.

Viendo los mismos ejemplos anteriores:

```

for i in "Python3":
    if i == "h":
        pass
    print("Letra actual: " + i)

```

Este código itera a través de la cadena de texto "Python3" y en cada iteración imprime la letra actual de la cadena junto con el mensaje "Letra actual: ". Sin embargo, si la letra actual es "h", la palabra clave **pass** se ejecutará, lo que significa que no se hará nada para esa letra en particular. Luego, el bucle continuará iterando a través de la cadena y, en la siguiente iteración, imprimirá la letra actual, como de costumbre.

Ejemplo de Continue vs Pass:

```

1  print("for con continue")
2  for x in "Hola":
3      print(x)
4      continue # Con la palabra clave continue, saltamos al siguiente
5                  elemento en la iteración actual del bucle.
6  print (str(x) + " denuevo :D ") # x todavía retiene el valor del último
7                  elemento de la cadena, por lo que lo imprimimos fuera del bucle.
8
9
10 print() #este print vacío nos genera un espacio entre los cuerpos de código
11
12
13
14 print("for con pass")
15 for x in "Hola":
16     print(x)
17     pass # La palabra clave pass no hace nada y se utiliza simplemente para
18           mantener el código sintácticamente válido.
19 print (str(x) + " denuevo :D ") # x todavía retiene el valor del último
20           elemento de la cadena, por lo que lo imprimimos fuera del bucle.

```

La salida del código será

```

for con continue
H
o
l
a
a denuevo :D

for con pass
H
o
l
a
a denuevo :D

```

En el primer bucle for, utilizamos la palabra clave **continue** dentro del bucle para saltar al siguiente elemento en la iteración actual. Debido a esto, el último elemento de la cadena nunca se imprime dentro del bucle, y cuando el bucle termina, todavía podemos imprimir el último valor de x usando la variable en una cadena de caracteres fuera del bucle.

En el segundo bucle for, utilizamos la palabra clave **pass** dentro del bucle. **pass** es una sentencia nula en Python, lo que significa que no hace nada. En este caso, se usa simplemente para mantener el código sintácticamente válido. Como resultado, cada elemento de la cadena se imprime dentro del bucle, y cuando el bucle termina, todavía podemos imprimir el último valor de x usando la variable en una cadena de caracteres fuera del bucle.

6. Cierre

La importancia de tener un control más preciso de los datos se puede ver aplicada en situaciones como: recorrer una palabra, determinar si contiene una letra o un número, crear una lista de datos, etc.

A medida que se trabaja más con lenguajes de programación como Python, el manejo de estos datos se vuelve cada vez más importante. Sin embargo, lo que se aprende en este módulo es útil no solo en Python sino también en otros lenguajes de programación, ya que solo la sintaxis varía.

Como se mencionó en el módulo anterior, cuando se unifican actividades, se sigue siempre la lógica: "si sucede esto, entonces hago esto", "realizo esta actividad mientras esto suceda", "repito esto tantas veces". Ahora, con el manejo de condiciones y ciclos, el análisis que se hizo en el módulo anterior se puede traducir a la programación.

APORTE A TU FORMACIÓN

El estudiante tendrá una comprensión más profunda en el análisis de situaciones, ya sea en el ámbito de la programación o no. Esto le permitirá aplicar su perspectiva en la toma de decisiones en su carrera profesional.