DESCUBRIENDO EL PODER DE LA PROGRAMACION

CURSO INICIAL DE PYTHON

Jimmy Chung
Alexander Solis

CONTENIDO DEL CURSO

1. Introducción, Instalación y Conceptos básicos
2. Variables, Expresiones, Funciones y Operadores
3. Condicionales y Ciclos – Patrones de sintaxis válidos
4. Listas, tuplas y diccionarios
5. Errores y Excepciones
6. Clases y funciones en Python

Tema 3: Condicionales y Ciclos, Patrones de sintaxis válidos

Funciones «built-in»

Funciones «built-in»: Incorporadas por defecto en el propio lenguaje

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	any()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

https://docs.python.org/es/3/library/functions.html?highlight=built

Estructuras de control: Conceptos

Tabulación: En el contexto de la programación, la tabulación se refiere al uso de la tecla de tabulación para crear espacios en blanco en el código. En Python, la tabulación se utiliza para realizar la identación y organizar visualmente los bloques de código. Se recomienda utilizar cuatro espacios en blanco para cada nivel de identación en Python.

Identación: En Python, la identación se refiere a la estructura visual del código, es decir, la forma en que se indentan o sangran las líneas de código. La identación es utilizada para delimitar bloques de código y definir la estructura jerárquica del programa. En Python, se utiliza la identación en lugar de llaves o palabras clave para indicar la pertenencia de líneas de código a un bloque.

```
if True:
    print("True")
```

```
if True:
print("True")
```

Estructuras de control: Conceptos

Bloques de Código: Son conjuntos de instrucciones que se agrupan y se ejecutan juntas. Los bloques de código en Python están determinados por su identación. Un bloque de código puede ser parte de una función, un condicional, un bucle o cualquier estructura que agrupe instrucciones.

Control de flujo: Es la capacidad de un programa para tomar decisiones y ejecutar diferentes acciones según ciertas condiciones. Permite dirigir el flujo de ejecución de un programa hacia diferentes ramas o secciones de código.

Estructuras de control: Son bloques de código que permiten controlar el flujo de ejecución de un programa. Las estructuras de control incluyen condicionales y bucles, que permiten tomar decisiones y repetir acciones según ciertas condiciones.

Estructuras de control: Conceptos

Condicionales: Son estructuras de control que permiten ejecutar diferentes bloques de código según el cumplimiento de una condición. Los condicionales más comunes en Python son if, elif y else. Estas sentencias permiten ejecutar diferentes acciones en función de la evaluación de una expresión booleana.

Bucles o Ciclos: Son estructuras de control que permiten repetir un bloque de código múltiples veces. En Python, los bucles más comunes son for y while, y también la función range(). Estos bucles permiten ejecutar un conjunto de instrucciones de manera iterativa hasta que se cumpla una condición.

Estructuras de control: Conceptos

Un código es una secuencia de instrucciones.

Tenemos las siguientes acciones para cocinar un almuerzo:

```
poner_agua_hervir()
echar_arroz_hervir()
cortar_pollo()
freir_pollo()
mezclar_pollo_arroz()
```

Estructuras de control: Conceptos

Un código es una secuencia de instrucciones.

Tenemos las siguientes acciones para cocinar un almuerzo:

```
poner_agua_hervir()
echar_arroz_hervir()
cortar_pollo()
freir_pollo()
mezclar_pollo_arroz()
```

¿ Que pasa si nuestro comensal es vegetariano?

Estructuras de control: Conceptos

Que pasa si nuestro comensal es vegetariano?

Utilizamos el condicional if.

```
poner_agua_hervir()
echar_arroz_hervir()
if not vegetariano:
    cortar_pollo()
    freir_pollo()
    mezclar_pollo_arroz()
else:
    cortar_zanahoria()
    freir_zanahoria()
    mezclar_zanahoria_arroz()
```

Estructuras de control

Que pasa si nuestro comensal es vegetariano?

Utilizamos el condicional if.

```
poner_agua_hervir()
echar_arroz_hervir()
if not vegetariano:
    cortar_pollo()
    freir_pollo()
    mezclar_pollo_arroz()
else:
    cortar_zanahoria()
    freir_zanahoria()
    mezclar_zanahoria_arroz()
```

¿Qué pasa si queremos cocinar para 3 personas?

Estructuras de control

¿Qué pasa si queremos cocinar para 3 personas?

```
# Cocinamos para la primera persona
poner_agua_hervir()
echar_arroz_hervir()
cortar_pollo()
freir_pollo()
mezclar_pollo_arroz()
# Volvemos a cocinar para la segunda
poner_agua_hervir()
echar_arroz_hervir()
cortar_pollo()
freir_pollo()
mezclar_pollo_arroz()
# Y para la tercera
poner_agua_hervir()
echar_arroz_hervir()
cortar_pollo()
freir_pollo()
mezclar_pollo_arroz()
```

Estructuras de control

¿Qué pasa si queremos cocinar para 350 personas?

Es aquí donde juegan un papel importante los ciclos (for, while)

```
# Repite lo que hay dentro 100 veces
for i in range(100):
    # Cocinamos la receta
    poner_agua_hervir()
    echar_arroz_hervir()
    cortar_pollo()
    freir_pollo()
    mezclar_pollo_arroz()
```

Operadores de comparación

Nombre	Operador	Ejemplo	Resultado
# Igual que	==	1 == 1	True
# Diferente a	!=	“a” != “a”	False
# Mayor que	>	10 > 5	True
# Menor que	<	5 < 1	False
# Mayor o igual que	>=	30 >= 30	True
# Menos o igual que	<=	20 <= 10	False

Notas:

1. El resultado siempre es un Boolean (True/False).
2. Los objetos a comparar deben ser compatibles entre sí.

Operadores lógicos o booleanos

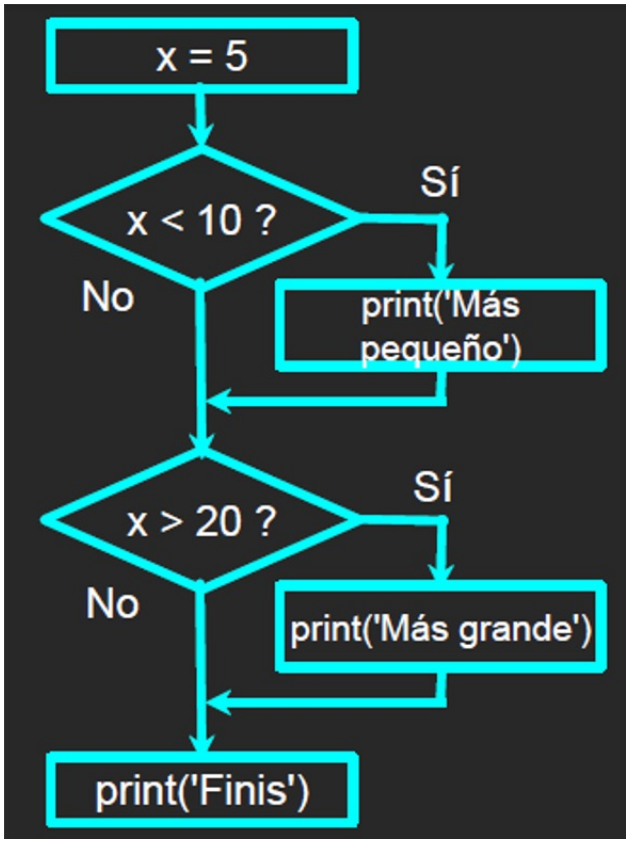
OR, AND, NOT

Operación	Resultado	Descripción
`a or b`	Si `a` se evalúa a falso, entonces devuelve `b`, si no devuelve `a`	Solo se evalúa el segundo operando si el primero es falso
`a and b`	Si `a` se evalúa a falso, entonces devuelve `a`, si no devuelve `b`	Solo se evalúa el segundo operando si el primero es verdadero
`not a`	Si `a` se evalúa a falso, entonces devuelve `True`, si no devuelve `False`	Tiene menos prioridad que otros operadores no booleanos

Los operadores and, or y not realmente no devuelven True o False, devuelven uno de los operandos.

CONDICIONALES IF, ELSE, ELIF

Flujo condicional con la sentencia if



Programa:

```
x = 5
If x < 10:
  print('Más pequeño')
If x > 20:
  print('Más grande')
print('Finis')
```

Resultado:

```
Más pequeño
Finis
```

CONDICIONALES IF, ELSE, ELIF

Sentencia else:

```
temperatura = 28
if temperatura < 20:
    if temperatura < 10:
        print('Nivel azul')
    else:
        print('Nivel verde')
else:
    if temperatura < 30:
        print('Nivel naranja')
    else:
        print('Nivel rojo')
```

CONDICIONALES IF, ELSE, ELIF

Sentencia elif (viene de else if):

```
temperatura = 28
if temperatura < 20:
    if temperatura < 10:
        print('Nivel azul')
    else:
        print('Nivel verde')
elif temperatura < 30:
    print('Nivel naranja')
else:
    print('Nivel rojo')
```

```
else:
    if ...
```
} `elif ...`

CONDICIONALES Sentencia match-case

Esta funcionalidad se adicionó en Python 3.10

```
edad = 21

if edad == 0 or edad is None:
    print('Not es una persona')
elif edad == 7:
    print('Tiene 7 años')
elif edad < 18:
    print('Menor de edad')
elif edad < 70:
    print('Mayor de edad')
else:
    print('Adulto mayor')
```

```
edad = 21
match edad:
    case 0 | None:
        print('Not es una persona')
    case 7:
        print('Tiene 7 años')
    case n if n < 18:
        print('Menor de edad')
    case n if n < 70:
        print('Mayor de edad')
    case _:
        print('Adulto mayor')
```

OPERADOR TERNARIO

Es una cláusula if, else que se define en una sola línea, y el resultado puede ser utilizado para asignarlo a una variable, entre otros usos.

Existen tres partes en un operador ternario:

```
# [código si se cumple] if [condición] else [código si no se cumple]
```

Usando el operador ternario:

```
x = 5
print("Es 5" if x == 5 else "No es 5")
#Es 5
```

```
a = 10
b = 5
c = a/b if b!=0 else -1
print(c)
#2
```

```
x = 5
x-=1 if x>0 else x
print(x)
```

Importante: Con el operador ternario no se puede utilizar elif

CICLOS (Iterables e iteradores)

Iterables: Son aquellos objetos que como su nombre indica pueden ser iterados, lo que dicho de otra forma es, que puedan ser indexados. Si piensas en un array (o una list en Python), podemos indexarlo con lista[1] por ejemplo, por lo que sería un iterable.

Iteradores: Son objetos que hacen referencia a un elemento, y que tienen un método next() que permite hacer referencia al siguiente.

En Python los principales iterables son los siguientes tipos de datos:

Cadenas, Listas, Tuplas y Diccionarios.

CICLOS (Iterables)

¿Pero cómo puedo saber si algo es iterable o no?

Para ello tenemos la función built-in isinstance()

```
from collections import Iterable
lista = [1, 2, 3, 4]
cadena = "Python"
numero = 10
print(isinstance(lista, Iterable))  #True
print(isinstance(cadena, Iterable)) #True
print(isinstance(numero, Iterable)) #False
```

CICLOS (iteradores)

En Python se utiliza el método iter() sobre un iterable para generar un iterador. El iterador es un objeto que tiene un apuntador o referencia al siguiente elemento de la secuencia. Y para obtener el siguiente elemento se utiliza el método o función next(), la cual devuelve el elemento sobre el cual está puesto el apuntador, y mueve el apuntador al siguiente. Si no hay más elementos, y se utiliza next(), se lanza una excepción StopIteration.

En resume un iterador es el iterable, pero con un apuntador al elemento actual.

```
libro = ['página1', 'página2', 'página3', 'página4']
iterador = iter(libro)  # Hemos creado un Iterador
print(next(iterador)) # página1
print(next(iterador)) # página2
print(next(iterador)) # página3
print(next(iterador)) # página4
#Como ya no hay mas páginas, si llamamos nuevamente next()...
print(next(iterador)) # Salida: StopIteration (Se generar un error o una excepción)
```

CICLOS (Listas)

En Python, una lista es una estructura de datos que permite almacenar y organizar múltiples elementos en una secuencia ordenada. Las listas son muy versátiles y se utilizan con frecuencia en programación debido a su capacidad para almacenar diferentes tipos de datos y su flexibilidad para realizar operaciones y manipulaciones.

```
lista_1 = [1, 2, 3, 4] # Hemos creado una lista de números.
lista_2 = list("1234") # Se puede crear una lista usando list() y pasando un objeto iterable.
lista_3 = [1, "Hola", 3.67, [1, 2, 3]] # Pueden almacenar tipos de datos distintos
print(lista_3[0])  # Imprime 1
print(lista_3[1])  # Imprime Hola
print(lista_3[-1]) # Imprime [1, 2, 3]
```

CICLOS: while

La sentencia while nos permite ejecutar una sección de código repetidas veces, el código se ejecutará mientras una condición se cumpla.

```
texto = "Python”
i = 0
while i < len(texto):
    print(texto[i])
    i += 1
```

Salida:

```
# P
# y
# t
# h
# o
# n
```

```
numero = 5
i = 0
while i <= numero:
    print(i)
    i += 1
```

Salida:

```
# 0
# 1
# 2
# 3
# 4
# 5
```

CICLOS: for

La sentencia for es una estructura de control en Python que permite iterar (repetir) sobre una secuencia de instrucciones.

Es especialmente útil cuando se conoce la cantidad exacta de repeticiones que se desea realizar.

Sintaxis:

```
#for <variable> in <iterable>:
#    <Código>
```

```
for i in "Python":
    print(i)

# Salida:
# P
# y
# t
# h
# o
# n
```

CICLOS (Clausula break)

La sentencia break es una instrucción utilizada en los ciclos (como for y while) en Python para interrumpir la ejecución del ciclo de forma anticipada. Cuando se encuentra la instrucción break, el flujo del programa sale inmediatamente del ciclo, sin ejecutar las iteraciones restantes.

La sintaxis básica del break es la siguiente:

```
for elemento in secuencia:
    # código a ejecutar en cada iteración
    if condición:
        break
    # más código
```

```
while condición:
    # código a ejecutar en cada iteración
    if condición:
        break
    # más código
```

```
numeros = [0, 1, 2, 3, 4, 5]
for numero in numeros:
    if numero == 3:
        break
    print(numero)

# Salida:
0
1
2
```

CICLOS (Clausula continue)

La sentencia continue es utilizada en los ciclos (como for y while) en Python para omitir el resto del código en una iteración particular y pasar a la siguiente iteración del ciclo. Cuando se encuentra la instrucción continue, el flujo del programa salta al inicio del ciclo y se evalúa la siguiente iteración sin ejecutar el código restante dentro del ciclo para esa iteración en particular.

La sintaxis básica del continue es la siguiente:

```
for elemento in secuencia:
    # código a ejecutar en cada iteración
    if condición:
        continue
    # más código
```

```
while condición:
    # código a ejecutar en cada iteración
    if condición:
        continue
    # más código
```

```
numeros = [1, 2, 3, 4, 5, 6]
for numero in numeros:
    if numero % 2 = 0:
        continue
    print(numero)

# Salida:
1
3
5
```

CICLOS: Uso de while y else

La sección de código que se encuentra dentro del else, se ejecutará cuando el bucle termine, pero solo si el while termina porque la condición se deja de cumplir, y no porque se ha hecho uso del break.

```
x = 5
while x > 0:
    x -=1
    print(x) #4, 3, 2, 1, 0
else:
    print("El while ha finalizado")
```

```
x = 5
while True:
    x -=1
    print(x) #4, 3, 2, 1, 0
    if x == 0:
        break
else:
    # El print no se ejecuta
    print("El while ha finalizado")
```

CICLOS: while anidados

Consiste en meter un while dentro de otro ciclo while.

Es algo que resulta especialmente útil si por ejemplo queremos generar permutaciones de números, es decir, si queremos generar todas las combinaciones posibles.

```
# Permutación
i = 0
j = 0
while i < 3:
    while j < 3:
        print(i,j)
        j += 1
    i += 1
    j = 0
```

CICLOS: for anidados

Consiste en meter un for dentro de otro for. Esto puede ser muy útil si queremos iterar algún objeto que en cada elemento, tiene a su vez otra clase iterable. Podemos tener por ejemplo, una lista de listas, una especie de matriz.

```
lista = [[56, 34, 1], [12, 4, 5], [9, 4, 3]]

for i in lista:
    print(i)
#[56, 34, 1]
#[12, 4, 5]
#[9, 4, 3]
```

```
for i in lista:
    for j in i:
        print(j)
# Salida: 56,34,1,12,4,5,9,4,3
```

CICLOS: iterar con range()

Función range(): Sintaxis range(inicio, fin, salto)

Genera una secuencia de números que van desde inicio (0 por defecto) hasta fin menos una (fin es el número que se pasa como parámetro), y el salto es de 1.

```
for i in range(6):
    print(i) #0, 1, 2, 3, 4, 5
```

Al utilizar range() con salto (ej: 5,20,2), generará los números del 5 al 20 de dos en dos:

```
for i in range(5, 20, 2):
    print(i) #5,7,9,11,13,15,17,19
```

Se pueden generar también secuencias inversas, utilizando un salto negativo:

```
for i in range (5, 0, -1):
    print(i) #5,4,3,2,1
```

CICLOS: iterar con enumerate()

La función enumerate() en Python se utiliza en los ciclos for para obtener tanto el índice como el valor de cada elemento de una secuencia. La función enumerate() toma una secuencia como argumento y devuelve un objeto iterable que genera pares de (índice, valor) en cada iteración del ciclo. El índice representa la posición del elemento en la secuencia, comenzando desde 0, y el valor es el elemento actual.

```
frutas = ['manzana', 'banana', 'cereza']

for indice, fruta in enumerate(frutas):
    print(f'Índice: {indice}, Fruta: {fruta}')

# Salida:
# Índice: 0, Fruta: manzana
# Índice: 1, Fruta: banana
# Índice: 2, Fruta: cereza
```

CICLOS: iterar con zip()

La función zip() en Python se utiliza para combinar elementos de dos o más secuencias en pares ordenados. Proporciona una forma conveniente de recorrer varias secuencias simultáneamente en un ciclo for.

La sintaxis básica de zip() es la siguiente:

```
for elemento1, elemento2, ... in zip(secuencia1, secuencia2, ...):
    # código a ejecutar en cada iteración
    # se pueden utilizar los elementos combinados
    # más código
```

```
nombres = ['Juan', 'María', 'Pedro']
edades = [25, 30, 35]

for nombre, edad in zip(nombres, edades):
    print(f'Nombre: {nombre}, Edad: {edad}')

# Salida:
# Nombre: Juan, Edad: 25
# Nombre: María, Edad: 30
# Nombre: Pedro, Edad: 35
```

La función zip() es útil cuando necesitas recorrer varias secuencias al mismo tiempo y realizar operaciones con los elementos combinados.

CICLOS: List comprehencions

Las List comprehensions (comprensiones de listas) en Python son una forma concisa y poderosa de crear listas de manera elegante y eficiente. Proporcionan una sintaxis compacta para generar listas basadas en una expresión y una o varias iteraciones.

Sintaxis básica de una List comprehension:

```
nueva_lista = [expresion for elemento in secuencia]
```

Donde:

nueva_lista: Es la lista resultante generada por la List comprehension.

expresion: Es una expresión que define cómo se calculará cada elemento de la lista resultante.

elemento: Es una variable que representa cada elemento de la secuencia en la que se está iterando.

secuencia: Es una secuencia, como una lista, tupla o rango, sobre la cual se realizará la iteración.

CICLOS: List comprehencions

```
cuadrados = [i**2 for i in range(5)] #[0, 1, 4, 9, 16]
```

Si no existieran las List comprehencios, podríamos hacer lo mismo de la siguiente forma, pero necesitamos más líneas de código.

```
cuadrados = []
for i in range(5):
    cuadrados.append(i**2)
#[0, 1, 4, 9, 16]
```

La expresión también puede ser una llamada a una función:

```
def eleva_al_2(i):
    return i**2

cuadrados = [eleva_al_2(i) for i in range(5)]
#[0, 1, 4, 9, 16]
```

CICLOS: List comprehencions

Las List comprehensions también pueden incluir cláusulas opcionales como if para filtrar elementos o realizar condiciones adicionales.

```
nueva_lista = [expresion for elemento in secuencia if condicion]
```

En este caso, la condicion se evalúa para cada elemento de la secuencia y solo se incluyen en la lista resultante aquellos elementos que cumplen con la condición especificada.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
cuadrados_pares = [num ** 2 for num in numeros if num % 2 == 0]

print(cuadrados_pares)
# Salida: [4, 16, 36, 64, 100]
```

List comprehensions para crear una lista de los cuadrados de los números pares de 1 a 10

Ejercicios

- Pida un texto por pantalla, luego un numero para extraer los primeros n caracteres.
- Pida un texto por pantalla y valide que no supere 20 caracteres.
- Pida un numero por pantalla determine si es positivo, negativo o cero.
- Capture un número por pantalla y determine si es par o impar.
- Escribir un programa que determine el día de la semana correspondiente a un número del 1 al 7.
- Determinar si una persona puede votar con base en su edad.

Ejercicios

- Calcular descuento a aplicar en una compra según el monto total:

  De 0 a 1.000, 0% de descuento.

  De 1.001 a 20.000 10% de descuento

  De 20.001 a 50.000 20% de descuento

  Mas de 40.000 30% de descuento

- Dada una lista de números, calcular la suma de todos los elementos.
- Dado un conjunto de nombres, imprimir cada nombre en una línea separada.

Ejercicios

- Dada una cadena de texto, contar cuántas veces aparece una letra específica.
- Dada una lista de palabras, crear una nueva lista con las palabras en mayúsculas.
- Dada una lista de estudiantes y sus calificaciones, imprimir el nombre del estudiante con la calificación más alta.
- Dado un rango de números, crear una lista con los números pares.

Ejercicios

- Dado un conjunto de números, encontrar el número mínimo y máximo sin utilizar las funciones min() y max().
- Dada una cadena de texto, imprimir las palabras que tienen más de 5 caracteres.
- Dado un diccionario de productos y sus precios, calcular el precio promedio de los productos.
- Dada una lista de números, crea una List comprehencion que devuelva solo los números pares.
- Dada una lista de nombres, crea una List comprehencion que devuelva solo los nombres que empiecen con la letra "A".

Nuestro segundo programa

Programa que ayuda a un usuario a administrar una lista de tareas.

El programa debe permitir al usuario realizar las siguientes acciones:

1. Agregar una tarea a la lista.
2. Ver todas las tareas en la lista.
3. Marcar una tarea como completada.
4. Eliminar una tarea de la lista.

Referencias bibliográficas

Documentación oficial Python.

https://docs.python.org/es/3/tutorial/index.html

https://docs.python.org/es/3/tutorial/classes.html#iterators

https://docs.python.org/es/3/reference/lexical_analysis.html#f-strings

https://docs.python.org/es/3/library/stdtypes.html#str.format