

Algoritmos y Estructuras de Datos III

Profesor:
Rómulo Arceri

Tecnicatura Superior en Análisis de Sistemas

Clase 1

Contenido

Objetivos de la materia	4
Contenidos de la materia:	4
Temas de repaso.....	4
Introducción a PYTHON.....	5
Python. Características	5
Python. Palabras Claves.....	6
Python. Identificadores y recomendaciones	6
Python. Sangría y Comentarios	8
Tipos de datos simples. Numéricos.....	9
Tipos de datos simples. Cadenas	10
Tipos de datos simples. Lógicos o booleanos	11
Conversión implícita de tipos	11
Conversión explícita de tipos	12
Salida.....	13
print().....	13
Entrada.....	14
input().....	14
Entrada. input() y conversión explícita.....	15
Salida. Dando formato.....	15
Operadores.....	16
Condicionales dobles if else	17
Condicionales dobles if else	17
Condicionales anidadas elif	18
Estructuras iterativas.	19
Estructuras iterativas. while (mientras).....	19
Estructuras iterativas. while (mientras) else, continue.....	20
Estructuras iterativas. while (mientras) True, break.	20
Estructuras iterativas. while (mientras) continue	21
Estructuras iterativas. While (mientras)	22
Estructuras iterativas. while (mientras) anidadas.....	22
Estructuras iterativas. for (para)	23
Estructuras iterativas. for (para)	24
Estructuras iterativas. for y range().....	25

Estructuras iterativas. for (para) range - Ejercicios	26
Tipos de datos.....	26
Lista []	26
Tipos de datos. for y listas []	27
Estructuras iterativas. for listas, indexadas len, enumerate.....	1
Estructuras iterativas. for anidados.....	1
Estructuras de datos. Números aleatorios. Random.....	1

Objetivos de la materia

- Proporcionar al estudiante las herramientas para desarrollar programas correctos, eficientes, bien estructurados y con estilo, que sirvan de base para la construcción de fundamentos teóricos y prácticos que le permitan continuar con éxito sus estudios de los cursos superiores de la carrera.
- Enseñar técnicas de análisis, diseño y construcción de algoritmos, estructuras de datos y objetos, así como reglas para la escritura de programas, eficientes y orientados a objetos.
- Enseñar al estudiante técnicas de abstracción que le permitan resolver los problemas de programación del modo más simple y racional, pensando no sólo en el aprendizaje de reglas de sintaxis y construcción de programas, sino, y sobre todo, aprender a pensar para conseguir la resolución del problema en cuestión de forma clara, eficaz y fácil de implementar en un lenguaje de programación y su ejecución posterior en un computador.

Contenidos de la materia:

- Estructuras de datos lineales
- Abstracción de datos.
- Encapsulamiento de datos.
- Pilas, listas y colas.
- Recursividad.
- Manejo de memoria en ejecución.
- Corrección y verificación.
- Análisis de Algoritmos.
- Cálculo de tiempo y orden de ejecución,
- Estructuras de datos no lineales.
- Algoritmos de recorrido, búsqueda y actualización de árboles.
- Algoritmos de recorrido de grafos.
- Archivos.
- Procesamiento de un lenguaje

Temas de repaso

- Tipos de datos
- Variables
- Condicionales
- Ciclos
- Listas
- Vectores
- Funciones

Introducción a PYTHON

¿Qué es Python?

Python es un lenguaje de programación de propósito general muy poderoso y flexible, a la vez que sencillo y fácil de aprender.

Este lenguaje fue creado a principios de los noventa por Guido van Rossum en los Países Bajos.



By MaartenschrijftOorspronkelijk: Doc Searls op Flickr - File:Guido van Rossum OSCON 2006.jpg, CC BY-SA 2.0,
<https://commons.wikimedia.org/w/index.php?curid=45984776>

Python es relativamente joven, nació a comienzos de los años 90, (Fortran 1957, Pascal 1970, C 1972, Modula-2 1978, Java 1991).

Es software libre, y está implementado en todas las plataformas y sistemas operativos habituales.



Python. Características

Es **multiparadigma**: significa que soporta la programación imperativa, programación orientada a objetos y programación funcional.

- La **programación imperativa** encadena instrucciones una detrás de otra que determinan lo que debe hacer el computador en cada momento para alcanzar un resultado deseado. Los valores utilizados en las variables se modifican durante la ejecución del programa. Para gestionar las instrucciones, se integran estructuras de control como bucles o estructuras anidadas en el código. Esto se hace para mitigar o evitar por completo las instrucciones de salto que añaden una complejidad innecesaria al código.
- En la **programación orientada a objetos (POO)** los programas se organizan en torno a objetos. Un objeto es una entidad que combina datos (atributos) y comportamiento (métodos). Por ejemplo, si pensamos en un coche, podemos considerarlo como un objeto con atributos como color, modelo y métodos como “acelerar” o “frenar”. El objeto coche es una instancia de una clase coche (plantilla o plano para crear objetos coches). Podemos crear múltiples objetos a partir de la misma clase. En Python, todo (números, cadenas, funciones, etc.) es un objeto.
- La **programación funcional** descompone un problema en un conjunto de funciones. Podemos tratar las funciones como objetos y pasarlas como argumentos a otras funciones.

Es **multiplataforma**: Se puede encontrar un intérprete de Python para los principales sistemas operativos: Windows, Linux y Mac OS. Además, se puede reutilizar el mismo código en cada una de las plataformas.

Es **dinámicamente tipado**: El tipo de las variables se decide en tiempo de ejecución.

Es **fuertemente tipado**: No se puede usar una variable en un contexto fuera de su tipo. Si se quisiera, habría que hacer una conversión de tipos.

Es **interpretado**: El código no se compila a lenguaje máquina. Quiere decir que hace falta un intérprete que permita ejecutar un programa o script escrito en Python sin necesidad de compilarlo.

Es **estructural** y se refiere a cómo organizamos y presentamos nuestro código. Ejemplo: Python utiliza la indentación (espacios o tabulaciones al principio de una línea) para definir bloques de código.

Python. Palabras Claves

Las palabras claves son las palabras reservadas en Python. Se utilizan para definir la sintaxis y la estructura del lenguaje Python.

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

No podemos usar una **palabra clave** como nombre de variable, nombre de función o cualquier otro identificador.

En Python, las **palabras clave** distinguen entre mayúsculas y minúsculas.

Todas las **palabras clave** excepto **True**, **False** y **None** están en minúsculas y deben escribirse tal cual.

Python. Identificadores y recomendaciones

Un **identificador** es un nombre dado a entidades como **clases**, **funciones**, **variables**, etc. Ayuda a diferenciar una entidad de otra.

Los identificadores pueden ser una combinación de letras en minúsculas (**a** a **z**) o mayúsculas (**A** a **Z**) o dígitos (**0** a **9**) o un guión bajo **_**.

Nombres como: **myClass**, **var_1** y **print_this_to_screen**, son ejemplos válidos.

Un identificador no puede comenzar con un dígito.

1variable no es válido, pero **variable1** es un nombre válido.

Las **palabras clave** no se pueden utilizar como identificadores.

global = 1

No podemos usar símbolos especiales como **!**, **@**, **#**, **\$**, **%** etc. en nuestro identificador.

`a@ = 0`

Un identificador puede tener cualquier longitud.

Tengamos en cuenta:

- **Case sensitive:** Python es un lenguaje que distingue entre mayúsculas y minúsculas, por lo tanto, debemos prestar atención cuando identificamos una variable, función o clase, etc, Esto significa que, **Variable** y **variable** no son lo mismo.

```
>>> Variable = 0
>>> variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable' is not defined.
Did you mean: 'Variable'?
```

- **Identificación clara:** Siempre asigne a los identificadores un nombre que tenga sentido.
- Si bien `c = 10` es un nombre válido, escribir `contador = 10` tendría más sentido y sería más fácil descubrir qué representa cuando observamos su código después de un largo intervalo.
- Uso del guión bajo: Se pueden separar varias palabras con un guion bajo, como `this_is_a_long_variable`.

Python. Sangría y Comentarios

La mayoría de los lenguajes de programación como C, C++ y Java usan llaves { }, para definir un bloque de código. Python, sin embargo, usa sangría.

- ¿Cómo aplicar sangría y porque?

- Bloque de código
- 4 espacios
- consistentes
- Limpio y ordenado

```
if True:
    print('Hello')
    a = 5
```

- Continuación de la línea

- Desordenado

```
if True: print('Hello'); a = 5
```

- Sangría incorrecta

- Error

Los comentarios son muy importantes al escribir un programa.

- Comentario por línea #

```
#Este es un comentario
print('Hola UNPAZ')
```

- Comentarios de bloques de líneas

- ''' , triple comilla simple

```
''' Este es un comentario
de bloque de 3 líneas y
Realizo un saludo '''
print('Hola UNPAZ')
```

- Ignorados

- Un **bloque de código** (cuerpo de una función, ciclo, etc.) comienza con sangría y termina con la primera línea sin sangría
- **¿Cómo aplicar sangría y por qué?**
Generalmente se utilizan cuatro espacios en blanco para la sangría y se prefieren antes que los tabulados.
La aplicación de la sangría en Python hace que el código se vea limpio y ordenado.
- **Continuación de la línea:** La sangría se puede ignorar en la continuación de la línea, pero siempre es una buena idea sangrar. Hace que el código sea más legible, vea como se ve si se utiliza el código anterior en una sola línea continua, Ambos códigos son válidos y hacen lo mismo, pero el estilo anterior es más claro.
- **Sangría incorrecta:** Una sangría incorrecta resultará en error.

Los comentarios son muy importantes al escribir un programa. Describen lo que sucede dentro de un programa, de modo que una persona que mira el código fuente no tenga dificultades para descifrarlo. Los comentarios son para que los programadores entiendan mejor un programa.

- **Comentario por línea:** usamos el símbolo hash (#) para comenzar a escribir un comentario. Se extiende hasta que haga un salto de línea.
- **Comentarios de varias líneas:** Podemos tener comentarios que se extiendan hasta varias líneas. Una forma es usar el símbolo hash (#) al principio de cada línea.
- **Comentarios de bloques de líneas:** Se utiliza entre comillas triples simples (''') , generalmente se usan para comentar un bloque de varias líneas.
- **Ignorados:** El intérprete de Python ignora los comentarios.

Tipos de datos simples

La información que se procesa en los programas informáticos se representan de diversas formas.



Los **datos simples o elementales**, son llamados también escalares por ser objetos indivisibles. Los datos simples se caracterizan por tener asociado un solo valor y son de tipo entero (**int**), real o de coma/punto flotante (**float**), booleanos (**bool**) y carácter (**string**).

Tipos de datos simples. Numéricos

Para saber a qué clase pertenece una variable o un valor, usamos la función **type()**. De manera similar, la función **isinstance()** se usa para verificar si un objeto pertenece a una clase en particular.

Enteros (int)

```
a = 5
print(a, "es de tipo", type(a))
b = 2 - 8
print(b, "es de tipo", type(b))
c = 9 + 1
print(c, "es de tipo", type(c))
5 es de tipo <class 'int'>
-6 es de tipo <class 'int'>
10 es de tipo <class 'int'>
```

Reales o Coma Flotante (float)

```
a = 2.
print(a, "es de tipo", type(a))
b = 3.14
print(b, "es de tipo", type(b))
c = -0.325
print(c, "es de tipo", type(c))
2.0 es de tipo <class 'float'>
3.14 es de tipo <class 'float'>
-0.325 es de tipo <class 'float'>
```

Complejos (complex)

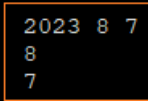
```
a = (4j-5) + (10+1j)
print(a, "es de tipo", type(a))
b = 1+2j
print(b, "is complex number?", isinstance(1+2j,complex))
(5+5j) es de tipo <class 'complex'>
(1+2j) is complex number? True
```

Ejercitación:

Abrimos la consola y usamos siempre **print** para imprimir por pantalla

Imprimir por pantalla utilizando la instrucción **print()**. Imprima el año, mes y día de hoy en tres líneas separadas.

```
anio = 2023
mes = 8
dia = 7
print(anio,mes,dia)
print(mes )
print(dia )
```



Tipos de datos simples. Cadenas

El tipo de dato **string** es la estructura básica para manejar texto, es una secuencia que incluye caracteres alfanuméricos y demás caracteres Unicode.

Cadena simple

```
a = "Esta es una cadena simple"
print(a, "del tipo", type(a))
b = "Otra C@dena de 1 línea"
print(b, "del tipo", type(b))
Esta es una cadena simple del tipo <class 'str'>
Otra C@dena de 1 línea del tipo <class 'str'>
```

Cadenas Múltiples

```
a = '''Esta es una
cadena multiple'''
print(a, "del tipo", type(a))
b = '''Otra C@dena de
2 líneas'''
print(b, "del tipo", type(b))
Esta es una
cadena multiple del tipo <class 'str'>
Otra C@dena de
2 líneas del tipo <class 'str'>
```

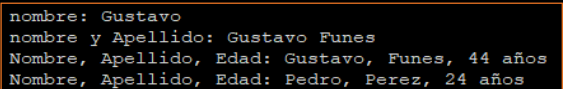
Cadenas o números?

```
a = '15'
print(a, "es del tipo", type(a))
b = '0.152'
print(b, "es del tipo", type(b))
15 es del tipo <class 'str'>
0.152 es del tipo <class 'str'>
```

Ejercitación: Muestre por pantalla

- Su primer nombre utilizando comillas simples
- Su nombre y apellido utilizando comillas dobles
- Su nombre, apellido y edad en una línea y el de su compañero/a en la segunda línea utilizando un solo **print()**

```
print('nombre: Gustavo')
print("nombre y Apellido: Gustavo Funes")
print('''Nombre, Apellido, Edad: Gustavo, Funes, 44 años
Nombre, Apellido, Edad: Pedro, Perez, 24 años''')
```



Tipos de datos simples. Lógicos o booleanos

El tipo de dato para representar valores lógicos o booleanos en Python es bool. Los datos booleanos toman el valor **True** (1 lógico) o **False** (0 lógico)

```
a = True
print(a, "es del tipo", type(a))
b = False
print(b, "es del tipo", type(b))
True es del tipo <class 'bool'>
False es del tipo <class 'bool'>
```

```
a = 3 > 2
print(a, "es del tipo", type(a))
b = 10 < 0.25
print(b, "es del tipo", type(b))
True es del tipo <class 'bool'>
False es del tipo <class 'bool'>
```

```
a = True == 1
print(a, "es del tipo", type(a))
b = False == 0
print(b, "es del tipo", type(b))
c = True and False
print(c, "es del tipo", type(c))
True es del tipo <class 'bool'>
True es del tipo <class 'bool'>
False es del tipo <class 'bool'>
```

El nombre booleano se usa luego que George Boole, matemático inglés, propusiera en el siglo XIX un sistema algebraico basado en estos dos valores lógicos y tres operaciones lógicas: “**y lógico (and)**”, “**o lógico (or)**” y la **negación (not)**.

Conversión implícita de tipos

El proceso de convertir el valor de un tipo de datos (entero, cadena, flotante, etc.) a otro tipo de datos se denomina conversión de tipos.

- Conversión implícita (automática)

```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo

print("El tipo de dato de num_int es ", type(num_int))
print("El tipo de dato de num_flo es ", type(num_flo))
print("El nuevo valor de num_new es ", num_new)
print("El tipo de dato de num_new es", type(num_new))

El tipo de dato de num_int es <class 'int'>
El tipo de dato de num_int es <class 'float'>
El nuevo valor de num_new es 124.23
El tipo de dato de num_new es <class 'float'>
```

- Errores de conversión

```
num = 45
print("numero es de tipo: ", type(num))
num_texto = "10"
print("numero_texto es de tipo: ", type(num_texto))
print(num + num_texto)
numero es de tipo: <class 'int'>
numero_texto es de tipo: <class 'str'>
print(num + num_texto)
TypeError: unsupported operand type(s) for +:
'int' and 'str'
```

Conversión de tipos implícita: Python convierte automáticamente un tipo de datos en otro tipo de datos. Este proceso no necesita ninguna participación del usuario.

Pero no todo es automático si queremos realizar operaciones aritméticas sobre un dato **int** y un dato numérico de tipo de **string** dará error.

Para solventar estos errores existen las conversiones de tipo explícita (manual)

Conversión explícita de tipos

Podemos convertir entre diferentes tipos de datos usando diferentes funciones de conversión de tipos como **int()**, **float()**, **str()**, etc

- Conversión explícita(manual)
- sintaxis : <tipo_de_datos_requeridos>(expresión)
- int(), float(), str(), bool() etc

```
num_texto = "2"
num2 = 3
print("El tipo de dato de num_texto es ", type(num_texto))
print("El tipo de dato de num2 es ", type(num2))
num_texto = int(num_texto)
print("Convertimos num_texto, ahora es ", type(num_texto))
print("Sumamos num_texto y num2, el resultado es ", num_texto+num2)
El tipo de dato de num_texto es <class 'str'>
El tipo de dato de num2 es <class 'int'>
Convertimos num_texto, ahora es <class 'int'>
Sumamos num_texto y num2, el resultado es 5
```

Conversión de tipos Explícita: En la conversión explícita los usuarios convierten el tipo de datos de un objeto al tipo de datos requerido. Se utilizan las funciones predefinidas como **int()**, **float()**, **str()**, etc. para realizar una conversión de tipo explícita. Este tipo de conversión también se denomina encasillamiento porque el usuario convierte (cambia) el tipo de datos de los objetos.

Salida.

print()

Usamos la función **print()** para enviar datos al dispositivo de salida estándar (pantalla).

- Salida simple

```
print('Esto se mostrara por pantalla')  
Esto se mostrara por pantalla
```

- Salida compuesta

```
numero = 10  
print('El valor de numero es:', numero)  
El valor de numero es: 10
```

- Salida con formato

```
dia = 16; mes = 8  
print('Hoy es dia {} del mes {}'.format(dia,mes))  
Hoy es dia 16 del mes 8
```

- Salida con operaciones

```
semanas = 4  
print("En", semanas , "semanas hay", 7 * semanas, "días.")  
En 4 semanas hay 28 días.
```

- Salida con formato y operaciones, sin concatenar

```
edad = 44  
print(f"Dentro de 20 años, tendra {edad + 20} años" )  
Dentro de 20 años, tendra 64 años
```

- **Salida simple:** Este es el tipo de salida que estuvimos utilizando hasta ahora, podemos escribir un texto alfanumérico, con caracteres especiales y mostrado por pantalla, recordemos podemos usar comillas simples, dobles o triples. También podemos imprimir el contenido de las variables y constantes.
- **Salida compuesta:** Podemos mezclar o concatenar texto con variables, para concatenar se utiliza la coma (,)
- **Salida con formato:** A veces nos gustaría formatear nuestra salida para que se vea atractiva. Esto se puede hacer usando el método **str.format()**. Podemos posicionar el contenido de una variable en la posición que deseamos dentro de un texto, utilizando {} llaves, cuando termina el texto luego de la o las comillas debemos separar con **.format(x,y)** donde **x** e **y** con las variables que contienen el valor que deseamos se incluyan en la salida.
- **Salida con operaciones:** Dentro de la salida estamos concatenando con, comas las variables y también podemos realizar operaciones aritméticas
- **Salida con formato y operaciones:** Podemos embeber una variable dentro del texto sin necesidad de concatenarlo con comas, para lograr esto se debe ingresar una **f** antes de las comillas y donde queremos que se responda con el valor de la variable y ejecutar una operación, abrimos las llaves y escribimos la variable y la operación cerramos la llave y continuamos con el texto luego cerramos las comillas y el paréntesis.

Entrada.

input()

Hasta ahora, nuestros programas eran estáticos. El valor de las variables se definió o codificó en el código fuente.

Para permitir flexibilidad, es posible que deseemos tomar la entrada del usuario por medio del teclado. En Python, tenemos la función `input()`.

```
variable = input("texto solicitando ingreso del dato:")
print(variable)
texto solicitando ingreso del dato: Año 2022
Año 2022
```

`input()` siempre guarda los datos como **string**

Analizamos el ejemplo: utilizamos la función `input()`. Esta contiene un texto que solicita el ingreso de un dato, esto es opcional, podremos usar `input()` sin necesidad de agregar un texto. El dato introducido por teclado se asignará a la variable denominada `variable`, luego pedimos que se muestre el contenido de la variable por pantalla.

Ejercitación:

Escriba un programa que permita el Ingreso por teclado de su nombre y apellido, guarde ese dato en una variable llamada **nom_ape** luego utilice la función que reconoce a qué tipos de dato pertenece la variable, y muestre por pantalla el tipo de datos.

Ingrese su edad, guarde el dato en una variable y luego muestre por pantalla el tipo de datos.

Resolución:

```
nom_ape =input("Ingrese nombre y apellido por favor: ")
print("Nombre y Apellido es del tipo:",type(nom_ape))
edad=int(input("Ingrese su edad: "))
print("Edad es del tipo:",type(edad))
```

```
Ingrese nombre y apellido por favor: Facundo Arceri
Nombre y Apellido es del tipo: <class 'str'>
Ingrese su edad: 13
Edad es del tipo: <class 'int'>
```

Entrada. input() y conversión explícita

- La función **input()** siempre nos devuelve un objeto de tipo cadena de texto o **str**.
- Para operar con otro tipo de datos debemos realizar una conversión explícita. **int()**, **float()**, **bool()**, etc.

- Error de tipos

```
edad = input("Ingrese su edad: ")
print("Dentro de 20 años, tendrá", edad + 20, " años" )
Ingrese su edad: 44
print("Dentro de 20 años, tendrá", edad + 20, " años" )
TypeError: can only concatenate str (not "int") to str
```

- Solución, conversión explícita de **str()** a **int()**

```
edad = int(input("Ingrese su edad: "))
print("Dentro de 20 años, tendrá", edad + 20, " años" )
Ingrese su edad: 44
Dentro de 20 años, tendrá 64 años
```

La función **input()** siempre nos devuelve un objeto de tipo cadena de texto o **str**. Hay que tenerlo muy en cuenta a la hora de trabajar con números, ya que debemos realizar una conversión explícita.

Salida. Dando formato

- Valores flotantes:

```
numero = 36.231321654888
print(f'{numero:2.3f}') 36.231
```

- Primera mayúscula (capitalizar)

```
texto = "hola"
print(texto.capitalize()) Hola
```

- Pasar a mayúsculas

```
texto = "hola"
print(texto.upper()) HOLA
```

- Pasar a minúsculas

```
texto = "HOLA"
print(texto.lower()) hola
```

- Limpiar

```
texto = " \n\t Hola!!!! "
print(texto) Hola!!!!
print(texto.strip()) Hola!!!!
```

Limpiar **texto.strip()**, elimina espacios y caracteres no imprimibles al comienzo y final de la cadena. Se utiliza mucho cuando el usuario ingresa un texto y queremos limpiarlo antes de guardarlo en una variable.

Operadores

Operadores aritméticos
+
-
*
/
//
%
**
** (1/2)

Operadores de asignación
=
+=
-=
*=
/=
%=
//=
**=

Operadores de comparación
>
<
==
!=
>=
<=

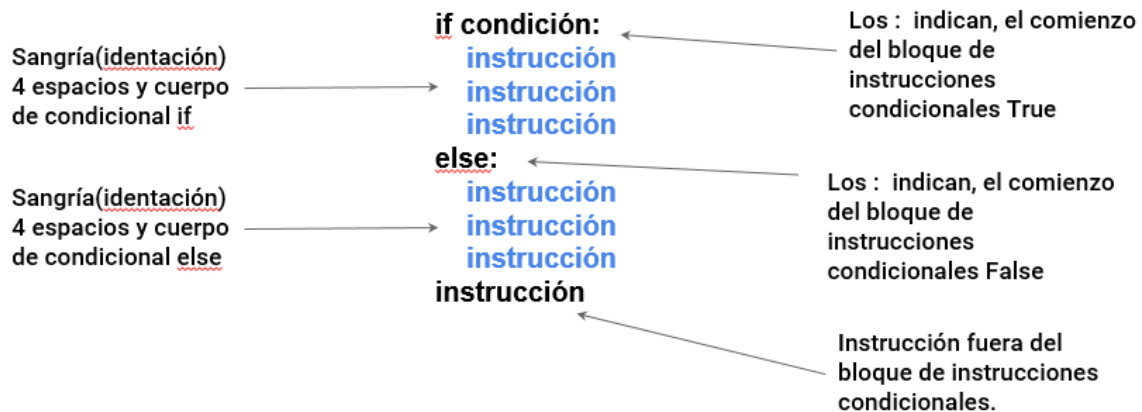
Operador	Significado
and	Verdadero si ambos operandos son verdaderos
or	Verdadero si cualquiera de los operandos es verdadero
not	Verdadero si el operando es falso (complementa el operando)

Operador	Significado
in	Verdadero si el valor/variable se encuentra en la secuencia
not in	Verdadero si el valor/variable NO se encuentra en la secuencia

Estructuras de decisión.

Condicionales dobles if else

La declaración **if: else:** se usa en para la toma de decisiones por **True** y **False**



La declaración **if..else**, ejecutará el cuerpo de **if** solo cuando la condición sea **True** y si la condición es **False** se ejecuta el cuerpo de **else**, tanto **if** como **else** llevan : ,Fijense que **if** y **else** están al mismo nivel de sangría, mientras que los cuerpos del **if** y del **else** si deben tener sangría.

La condición termina con la próxima instrucción sin sangría.

Condicionales dobles if else

Ejercitación. Ingrese un valor numérico entero y muestre por pantalla si es positivo o negativo

Entrada : Número entero
Proceso : Verificar si es mayor que 0
Salida : Mostrar si el número es positivo o negativo

pseudocódigo:

INICIO
ingreso por teclado un *valor*

SI el *valor* es mayor que 0 **ENTONCES**
 mostrar "el *valor* X es positivo"

SINO
 mostrar "el *valor* X es negativo"

FIN

```
print("Positivos Negativos")
valor = int(input("Ingrese un valor: "))
if(valor > 0):
    print(f"El valor {valor} es positivo")
else:
    print(f"El valor {valor} es negativo")
```

```
Positivos Negativos
Ingrese un valor: 10
El valor 10 es positivo

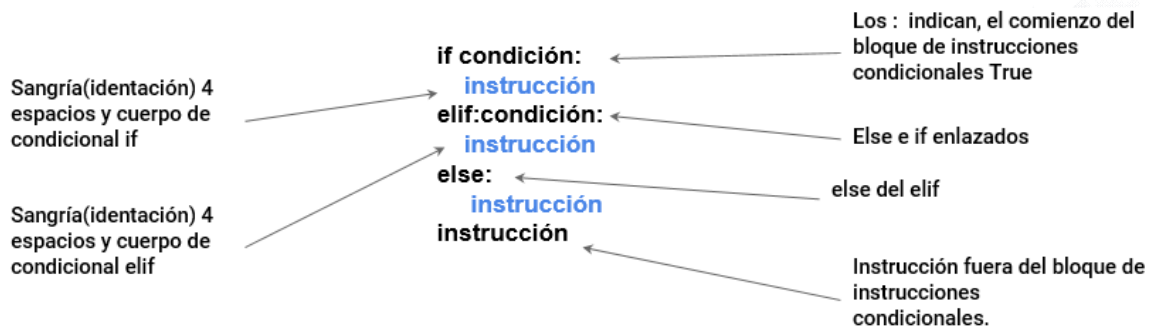
Positivos Negativos
Ingrese un valor: -35
El valor -35 es negativo
```

La estructura de decisión **if**, se compone de la palabra **if**, la condición o condiciones que deben cumplirse y luego dos puntos: Todo lo que deba ejecutarse por esta condición debe llevar sangría, a esto lo denominamos bloque de instrucción condicional y se termina cuando se rompe la sangría.

Python interpreta los valores distintos de cero como **True**. Pero **None** y **0** se interpretan como **False**.

Condicionales anidadas elif

El condicional **elif** sirve para enlazar varios "else if".



Ejercitación: Escribir un programa para un cine que tiene entradas para todas las edades y quiere calcular de forma automática el precio que debe cobrar a sus clientes por entrar. El programa debe preguntar al usuario la edad del cliente y mostrar el precio de la entrada. Si el cliente es menor de 2 años puede entrar gratis, si tiene entre 3 y 12 años debe pagar \$ 250, si es mayor de 18 años \$500 y si es mayor de 65 años es gratis.

```
edad = int(input("Ingrese su edad: "))
if(edad <=2 or edad >=65):
    print("Precio de la entrada es $0")
else:
    if(edad >=3 and edad <= 12):
        print("Precio de la entrada es $250")
    else:
        print("Precio de la entrada es $500")

Ingrese su edad: 45
Precio de la entrada es $500
```

```
edad = int(input("Ingrese su edad: "))
if(edad <=2 or edad >=65):
    print("Precio de la entrada es $0")
elif(edad >=3 and edad <= 12):
    print("Precio de la entrada es $250")
else:
    print("Precio de la entrada es $500")

Ingrese su edad: 45
Precio de la entrada es $500
```

Estructuras iterativas.

Las estructuras iterativas, también son conocidas como ciclos, bucles o estructuras de control repetitivas o iterativas.

Un ciclo o bucle te permite repetir una o varias instrucciones cuantas veces lo necesitemos.

Tipos de ciclos

while (Mientras)

for (Para)

En Python no existe el ciclo **do-while**

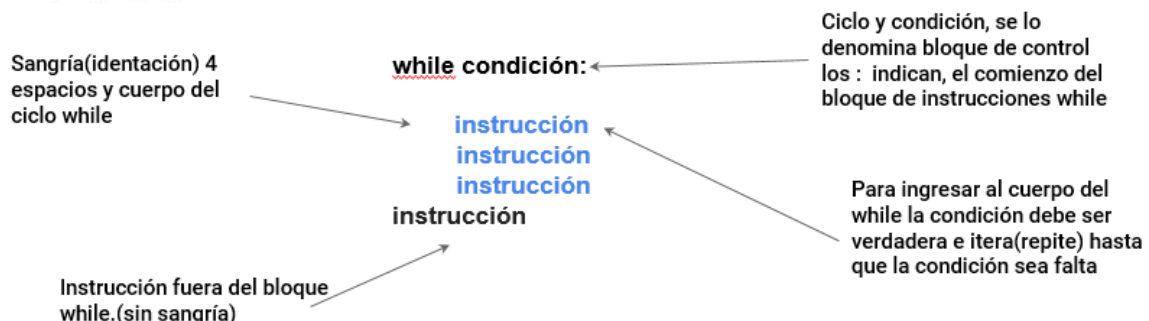
Por ejemplo, si quisiéramos escribir los números del uno al cien no tendría sentido escribir cien líneas de código mostrando un número en cada una, para eso y para varias cosas más (que veremos enseguida), es útil un ciclo. Un ciclo nos ayuda a llevar a cabo una tarea repetitiva en una cantidad de líneas muy pequeña y de forma prácticamente automática.

Existen diferentes tipos de ciclos o bucles en Python, cada uno tiene una utilidad para casos específicos y depende de nuestra habilidad y conocimientos el poder determinar en qué momento es bueno usar alguno de ellos.

En Python no existe el ciclo **do-while**. De hecho, no se necesita.

Estructuras iterativas. **while** (mientras)

En el bucle **while**, primero se comprueba la condición (**bloque de control**). Solo ingresa al cuerpo del ciclo si la condición es **True (verdadera)**. Entonces ejecuta las instrucciones dentro del cuerpo del ciclo y vuelve a verificar la condición. Se repite el ciclo hasta que la condición sea **False (falsa)**



Como ven, en esencia, es lo mismo que un condicional **if()**. Sin embargo, el funcionamiento es completamente diferente, pues esas instrucciones en su interior se ejecutarán tantas veces como sea necesario mientras se siga cumpliendo la condición dada.

Estructuras iterativas. **while (mientras) else, continue.**

La estructura iterativa **while** soporta las siguientes instrucciones:

- **else**, While también posee un bloque cuando la condición no se cumple y es opcional, **pero este bloque no es repetitivo.**
- **while true**, el ciclo siempre se ejecutará porque es una condición infinita siempre será verdadera.
- **break**, la declaración **break**, rompe los ciclos.
- **Continue**, detiene el ciclo donde aparece esta instrucción, salta todas las demás que están por debajo y vuelve a iterar.
- Otra forma de romper un ciclo es presionando las teclas **control + c**, aunque esta última es una salida de emergencia y directamente sale del programa.

Estructuras iterativas. **while (mientras) True, break.**

```
print("Programa iterativo, ingrese numeros, para terminar ingrese 0 (cero)\n")

while True:
    numero = int(input("Ingrese un numero: "))
    print(f"Numero: {numero}")
    if (numero == 0):
        break
    print("Fin del programa")
```

while true, es una condición infinita siempre será verdadera.

Se rompe el ciclo solo si en el código existe la declaración **break**

o si presionamos las teclas **control + c**, aunque esta última es una salida de emergencia.

Entendemos que se ingresa al **while** si la condición es **True (verdadera)**, bueno existe una mejor manera de realizar el programa anterior sin la necesidad de solicitar o inicializar una variable de antemano para que la condición sea verdadera.

Al usar **while(True)**, nos aseguramos que el ciclo se ejecute por siempre. Sin embargo, al poner un llamado a **break** en su interior, el cual se ejecuta SOLO cuando el número ingresado es cero, nos aseguramos de que se cierre el ciclo en ese preciso momento. Así nos hemos ahorrado de repetir el llamado a **input**.

Estructuras iterativas. while (mientras) continue

Veamos un ejercicio parecido al anterior, pero este imprimirá todos los números ingresados que sean positivos, pero no los negativos, utilizando **continue**.

```
print("Programa iterativo, ingrese numeros, para terminar ingrese 0 (cero)\n")

while True:
    numero = int(input("Ingrese un numero: "))
    if (numero < 0):
        continue
    print(f"Numero: {numero}")

print("Fin del programa")
```

Programa iterativo, ingrese numeros, para terminar ingrese 0 (cero)

Ingrese un numero: 10
Numero: 10
Ingrese un numero: -20
Ingrese un numero: 30
Numero: 30
Ingrese un numero: 0
Numero: 0

Analizando el código vemos que si el número es negativo (< 0) no se imprime el valor ingresado, porque se ejecuta un **continue**. Este hace pasar por alto las instrucciones que vienen a continuación y vuelve a iterar.

Estructuras iterativas. While (mientras)

Ejercitación:

- Escribir un programa que muestre automáticamente los números del 1 al 10.
- Escribir un programa que muestre la tabla del 3, sin multiplicar (utilice un acumulador)

```
cont=1
while cont <=10:
    print(f"Numero:{cont}")
    cont= cont+1
```

Numero:1
Numero:2
Numero:3
Numero:4
Numero:5
Numero:6
Numero:7
Numero:8
Numero:9
Numero:10
Finalizado

```
cont=1
tabla=3
while cont <=10:
    print(f"3 x {cont} = {tabla}")
    cont = cont + 1
    tabla = tabla + 3
```

3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30

Estructuras iterativas. while (mientras) anidadas

En el bucle **while** anidado encontramos un bucle dentro de otro.

Sangría(identación) 4
espacios y cuerpo del
ciclo while

```
while condición:
    instrucción
    while condición:
        instrucción
        instrucción
    instrucción
```

Instrucción fuera del bloque
while.(sin sangría)

Ciclo y condición, se lo
denomina bloque de control
los : indican, el comienzo del
bloque de instrucciones while

Para ingresar al cuerpo del
while la condición debe ser
verdadera e itera(repite) hasta
que la condición sea falsa

Para ingresar al cuerpo del
while anidado la condición
debe ser verdadera e
itera(repite) hasta que la
condición sea falsa

Ejercitación: Escribir un programa que muestre las tablas del 1 al 3

```
i = 1
j = 1
resul = 0

while i <= 3:
    print(f"Tabla de {i}")
    while j < 10:
        resul = i * j
        print(f"{i} x {j} = {resul}")
        j = j + 1

    i = i + 1
    j = 1
    resul = 0
    print("\n")
```

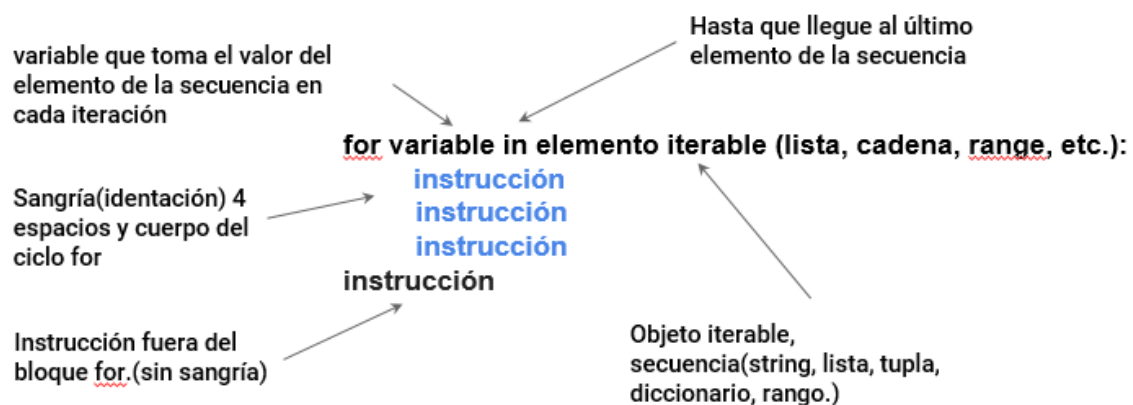
Tabla de 1
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10

Tabla de 2
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8

Estructuras iterativas. for (para)

El ciclo **for** se usa para iterar sobre una secuencia (**lista** , **tupla** , **cadena**) u otros objetos iterables. El bloque de instrucciones que se repite se suele llamar cuerpo del bucle y la iteración sobre una secuencia se llama recorrido.

La instrucción **for (para)** ejecuta una secuencia de instrucciones un número determinado de veces. Al ingresar al bloque, la variable <variable> recibe el valor <inicial> y se ejecuta la secuencia de instrucciones que forma el cuerpo del ciclo.



Los ciclos **for** (o ciclos para) son una estructura de control cíclica. Nos permiten ejecutar una o varias líneas de código de forma iterativa (o repetitiva), pero teniendo cierto control y conocimiento sobre las iteraciones. Pues necesitamos conocer previamente un valor de inicio, un tamaño de paso y un valor final para el ciclo. Se los conoce como ciclos determinados.

Un bucle **for** es un bucle que repite el bloque de instrucciones un número predeterminado de veces mientras no se alcance la longitud de la secuencia, iterará sobre esa secuencia.

En el bucle **for** tenemos que mencionar el **valor** y la secuencia a **iterar**.

La condición es **!=** al último elemento, **índice** ,**valor**

Estructuras iterativas. for (para)

Recorriendo secuencias de cadena.

Recordemos que las cadenas o string son un tipo de dato iterable porque contiene elementos. El cuerpo del bucle se ejecuta tantas veces como elementos tenga la secuencia.

1. Escribir un programa muestre por pantalla los elementos de la palabra (secuencia de cadena) "UNPAZ"

```
palabra = "UNPAZ"
for x in palabra:
    print(x)
```

Para evitar el salto de línea cuando utilizamos print se agrega al final de la palabra o variable una coma y end = " ", y se cierra los paréntesis del print.

```
palabra = "UNPAZ"
for x in palabra:
    print(x, end = ' ')
```

UNPAZ

2. Escribir un programa que muestre por pantalla los elementos numéricos de la siguiente secuencia (1,2,3,4,5)

```
numericos = (1,2,3,4,5)
for i in numericos:
    print(i)
```

3. Escribir un programa que muestre e identifique los números pares e impares del ejercicio anterior

```
numericos = (1,2,3,4,5)
for i in numericos:
    if i % 2 == 0:
        print(f"{i} Par")
    else:
        print(f"{i} Impar")
```

Estructuras iterativas. for y range()

El **range()** genera una secuencia de números que van desde 0 por defecto hasta el número que se pasa como parámetro menos 1.

Sintaxis: `range(parar)`

Si ejecutamos `range(10)` contendrá desde el 0 al 9

Range() acepta tres parámetros separados por coma, donde el primer es el inicio de la secuencia, el segundo el final o parar y el tercero el salto que se desea entre números. Por defecto se empieza en 0 y el salto es de 1.

`range(inicio, parar, paso)`

```
valores_1 = range(10)
valores_2 = range(1,11)
valores_3 = range(1,11,2)

print(valores_1)      range(0, 10)
print(valores_2)      range(1, 11)
print(valores_3)      range(1, 11, 2)

print(list(valores_1)) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(valores_2)) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(list(valores_3)) [1, 3, 5, 7, 9]
```

- **Range** no almacena todos los valores en la memoria, almacena los parámetros.
- Para generar y almacenar en memoria los valores debemos convertir al tipo de dato `list()`, esto genera una lista que podemos almacenar en una variable.

Esta función **range()** no almacena todos los valores en la memoria ; porque sería ineficiente. Por lo tanto, recuerda el inicio, la parada, el tamaño del paso.

Para obligar a esta función a generar todos los elementos, podemos usar la función **list()**.

Otros ejemplos:

```
for i in range(5):
    print(i)
```

0
1
2
3
4

```
for i in range(1,5):
    print(i)
```

1
2
3
4

```
for i in range(-3,3):
    print(i)
```

-3
-2
-1
0
1
2

```
for i in range(1,10,3):
    print(i)
```

1
4
7

Estructuras iterativas. for (para) range - Ejercicios

Ejercitación.

1. Escribir un programa que solicite un número positivo entero y a partir de ese número, muestre los próximos 20 números consecutivos.

```
valor_inicio = int(input("Ingrese un numero entero positivo: "))
for i in range(valor_inicio, valor_inicio + 20):
    print(i)
```

Ingrese un numero entero positivo: 5
5
6
7
...
23
24

2. Escribir un programa que muestre por pantalla 15 números negativos consecutivos

```
for i in range(-15, 0):
    print(i)
```

-15
-14
-13
...
-1

Tipos de datos.

Lista []

La lista es una secuencia ordenada de elementos. Es uno de los tipos de datos más utilizados en Python y es muy flexible. No es necesario que todos los elementos de una lista sean del mismo tipo.

Declarar una lista es bastante sencillo. Los elementos separados por comas se encierran entre paréntesis [].

```
variable = [45, 'texto abc ## ? 123', 45.23]
```

Podemos usar el operador [] para extraer un elemento o un rango de elementos de una lista. El índice comienza desde 0 en Python.

```
variable = [45, 'texto abc ## ? 123', 45.23]
print(variable[0])
print(variable[1])
print(variable[2])
```

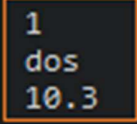
45
texto abc ## ? 123
45.23

Tipos de datos. for y listas []

Ejercitación:

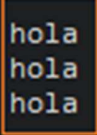
1. Escribir un programa muestre por pantalla los elementos de la lista (secuencia de lista) [1,"dos",10.3]

```
secuencia = [1,'dos',10.3]
for i in secuencia:
    print(i)
```

The output of the first program is displayed in a box, showing the elements of the list: 1, dos, and 10.3, each on a new line.

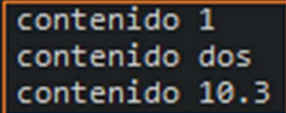
2. Escribir un programa muestre por pantalla la palabra "hola" según la secuencia de lista [1,"dos",10.3]

```
secuencia = [1,'dos',10.3]
for _ in secuencia:
    print("hola")
```

The output of the second program is displayed in a box, showing the word "hola" repeated three times, each on a new line.

3. Escribir un programa muestre por pantalla la palabra "contenido" y cada uno de los elementos de la secuencia según la secuencia de lista [1,"dos",10.3]

```
secuencia = [1,'dos',10.3]
for elemento in secuencia:
    print(f"contenido {elemento} ")
```

The output of the third program is displayed in a box, showing the word "contenido" followed by each element of the list: 1, dos, and 10.3, each on a new line.

Estructuras iterativas. for listas, indexadas len, enumerate

Se puede combinar con la función **len()** con **range()** para iterar una secuencia a través de la indexación.

Se utiliza cuando necesitamos saber la posición del elemento dentro de la secuencia.

```
secuencia = ['pop', 'rock', 'jazz']

for i in range(len(secuencia)):
    print(f"En el indice [{i}] accedemos al elemento {secuencia[i]}")
```

```
En el indice [0] accedemos al elemento pop
En el indice [1] accedemos al elemento rock
En el indice [2] accedemos al elemento jazz
```

Existe una función que nos facilita la enumeración, se llama **enumerate()**.

```
secuencia = ['pop', 'rock', 'jazz']

for indice, i in enumerate(secuencia):
    print(f"En el indice {indice} accedemos al elemento {i}")
```

```
En el indice 0 accedemos al elemento pop
En el indice 1 accedemos al elemento rock
En el indice 2 accedemos al elemento jazz
```

Estructuras iterativas. for anidados

Ejercicio: Escribir un programa que calcule y muestre las tablas del 2 y del 3

```
for i in [2,3]:
    print(f"\n***** Tabla del {i} *****\n")
    for j in range(1,10):
        print(f" {i} x {j} = {i * j:02d}")
```

***** Tabla del 2 *****

2 x 1	=	02
2 x 2	=	04
2 x 3	=	06
2 x 4	=	08
2 x 5	=	10
2 x 6	=	12
2 x 7	=	14
2 x 8	=	16
2 x 9	=	18

***** Tabla del 3 *****

3 x 1	=	03
3 x 2	=	06
3 x 3	=	09
3 x 4	=	12
3 x 5	=	15
3 x 6	=	18
3 x 7	=	21
3 x 8	=	24
3 x 9	=	27

Estructuras de datos. Números aleatorios. Random

Para generar números aleatorios se hace uso del módulo random que contiene diversas opciones o métodos.

Para generar números aleatorios de valor entero, se suele utilizar la función **random.randint(a, b)**

```
import random

nro_aleatorio = random.randint(0,11)
print(nro_aleatorio)
```

3

Para obtener elementos aleatorios a partir de una secuencia, utilizamos la función **random.choice(secuencia)**

```
frutas = ['peras', 'manzanas', 'plátanos', 'ciruelas']
seleccion = random.choice(frutas)
print(seleccion)
```

ciruelas

Para modificar el orden de los elementos de una lista, utilizamos la función `random.shuffle(sequencia)`.

```
frutas = ['peras', 'manzanas', 'plátanos', 'ciruelas']
for i in range(3):
    random.shuffle(frutas)
    print(frutas)
```

```
['ciruelas', 'plátanos', 'peras', 'manzanas']
['manzanas', 'plátanos', 'peras', 'ciruelas']
['plátanos', 'ciruelas', 'peras', 'manzanas']
```

Algoritmos y Estructuras de Datos III

Tecnicatura Superior en Análisis de Sistemas