

INTRODUCCIÓN A LA PROGRAMACIÓN

CON  python

CONTROL DE FLUJO

Flujo de ejecución de un programa

La potencia de la programación no es solo ejecutar una instrucción después de otra como si fuera una lista.

Un programa puede **decidir** saltarse instrucciones, **repetirlas**, o elegir **una de varias** instrucciones para ejecutar.

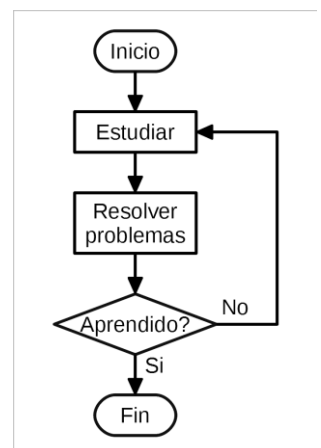
Flujo de ejecución de un programa

El **flujo de ejecución** de un programa es el orden en el que se ejecutan las instrucciones del mismo.

Sentencias de control de flujo

Las **sentencias de control de flujo** pueden decidir que instrucciones ejecutar bajo ciertas condiciones.

Estas sentencias de control de flujo se corresponden directamente con símbolos de un diagrama de flujo. Veremos, para cada porción de código, el diagrama de flujo correspondiente.



Sentencias de control de flujo

Todo programa puede escribirse utilizando únicamente las tres sentencias de control:

- Secuencial (los programas que han hecho hasta ahora)
- Condicionales (también llamadas de selección o alternativas)
- Repetición (también llamados bucles)

Sentencias de control de flujo

condicionales



bucles



Elementos de control de flujo

Las sentencias de control de flujo comprenden una parte llamada **condición**. Estas condiciones siempre se evalúan a un valor booleano, True o False.


Además, las líneas de código Python pueden agruparse en **bloques**. Podemos saber cuando comienza o termina un bloque por la **indentación** (sangrado) de las líneas.



Sentencias de control de flujo: condicionales

Para poder escribir programas útiles, casi siempre vamos a necesitar la capacidad de comprobar condiciones y cambiar el flujo de ejecución del programa de acuerdo a ellas. Las **sentencias condicionales** nos proporcionan esta capacidad.

Los condicionales nos permiten comprobar condiciones y hacer que nuestro programa se comporte de una forma u otra, que ejecute un fragmento de código u otro, dependiendo de esta condición.



Condicional simple: sentencia **if (si)**

La forma más simple de sentencia condicional es la sentencia **if**.

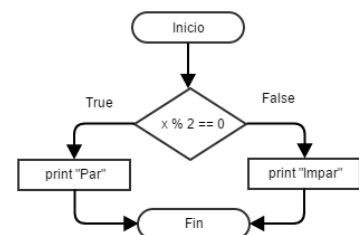
```
x=3
if x > 0:
    print ("Positivo") #se ejecuta si se cumple la condición
print("fin") #se ejecuta siempre
```

La expresión booleana luego del **if** es la **condición**. La sentencia **if** finaliza con dos puntos (:) y la(s) líneas que van detrás de la sentencia **if** van **indentadas**. Si la condición es verdadera, la sentencia indentada será ejecutada. Si la condición es falsa, la sentencia indentada será omitida.

Condicional doble: sentencia **if...else (si – sino)**

La segunda forma de la sentencia **if** es la ejecución alternativa, en la cual existen **dos** posibilidades y la condición determina cual de ellas será ejecutada:

```
x=9
if x % 2 == 0:
    print ("Par") #se ejecuta si se cumple la condición
else:
    print ("Impar") #se ejecuta si no se cumple la condición
print("fin") #se ejecuta siempre
```

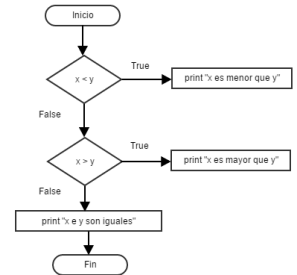


Dado que la condición debe ser obligatoriamente verdadera o falsa, solamente una de las alternativas será ejecutada. Las alternativas reciben el nombre de **ramas**, dado que se trata de ramificaciones en el flujo de ejecución

Condicional múltiple: sentencia **if...elif...(si... sino si)**

Algunas veces hay más de dos posibilidades, de modo que necesitamos más de dos ramas:

```
x=1
y=3
if x < y:
    print ("x es menor que y") #se ejecuta si se cumple la condición
elif x > y:
    print ("x es mayor que y") #se ejecuta si se cumple la condición
else:
    print ("x e y son iguales") #se ejecuta si no se cumple ninguna condición
print("fin") #se ejecuta siempre
```



Nota: **elif** es una abreviatura de **else if**. No hay límite en el número de sentencias elif. Si hay una clausula **else** debe ir al final, pero tampoco es obligatorio que esta exista.

Condicionales



IF
(si)

if **condición**:
 instrucciones

IF...ELSE...
(si...sino...)

if **condición**:
 instrucciones
else:
 instrucciones

IF...ELIF...
(si... sino si...)

if **condición**:
 instrucciones
elif **condición**:
 instrucciones
elif **condición**:
 instrucciones
...
else:
 instrucciones

Actividad

1. Escribe un programa que pida al usuario **dos números** y muestre su **división**. Si el divisor es cero la división no se hace y se muestra el mensaje de "error no se puede dividir por cero"
2. Escribe un programa que pida la **edad** al usuario y si es menor que 18 muestre por pantalla "Es usted menor de edad" si no que muestre "Es usted mayor de edad" y al finalizar que muestre "¡Hasta la próxima!"

Ejemplo:

<code>¿Cuántos años tiene? 17</code>	<code>¿Cuántos años tiene? 25</code>
<code>Es usted menor de edad</code>	<code>Es usted mayor de edad</code>
<code>¡Hasta la próxima!</code>	<code>¡Hasta la próxima!</code>

3. Escribe un programa que haga de **calculadora**: le pide dos números al usuario y la operación a realizar. Finalmente muestra en pantalla el resultado de la operación.

Soluciones

1)

```
num1=int(input("Ingresa un número para hacer una división (dividendo): "))
num2=int(input("Ingresa otro número para hacer una división (divisor): "))
if (num2==0):
    print("No se puede dividir por cero")
else:
    div=num1/num2
    print("El resultado de la división es:", div)
```

2)

```
edad=int(input("¿Cuántos años tiene? "))
if edad<18:
    print("Usted es menor de edad")
else:
    print("Usted mayor de edad")
print("Hasta la próxima")
```

Soluciones

3)

```
num1=float(input("Ingresa un número para hacer un operación: "))
num2=float(input("Ingresa otro número para hacer una operación: "))
op=input("Ingresa la operación (+,-,/,*): ")
if op=="+":
    print("La suma es:",num1+num2)
elif op=="-":
    print("La resta es:",num1-num2)
elif op=="*":
    print("La multiplicación es:",num1*num2)
elif op=="/":
    if num2==0:
        print("No se puede dividir por cero")
    else:
        print("La división es:",num1/num2)
else:
    print("Operación desconocida")
```

Condicional múltiple: la sentencia **match**

Una sentencia **match** recibe una expresión y compara su valor con patrones sucesivos dados en uno o más bloques case. Esto es similar a grandes rasgos con una sentencia switch en Java o JavaScript (y muchos otros lenguajes). Sólo se ejecuta el primer patrón que coincide. Por ejemplo,

```
match estado:
    case 400:
        mensaje = "Bad request"
    case 404:
        mensaje = "Not found"
    case 418:
        mensaje = "I'm a teapot"
    case _:
        mensaje = "Something's wrong with the internet"
```

Observa el último bloque: el «nombre de variable» `_` funciona como un comodín y nunca fracasa la coincidencia. Si ninguno de los casos case coincide, ninguna de las ramas es ejecutada.

Condicionales múltiples

En la **versión 3.10** una de las novedades fue la sentencia **match** que simplifica la forma compacta de if

1. **término** puede ser cualquier literal, dato u objeto de Python.
2. la **sentencia match** evaluará el **término** y lo comparará con cada <patrón-n> de cada **sentencia case** de arriba a abajo.
4. según el patrón de la sentencia case que coincida, se llevará a cabo la <acción-n> correspondiente.
5. si la sentencia case no puede hacer coincidir el término con ningún patrón, se ejecutará la última **acción-default**

```
match term:
    case pattern-1:
        action-1
    case pattern-2:
        action-2
    case pattern-3:
        action-3
    case _:
        action-default
```

Sentencia **elif** vs. sentencia **match**

La declaración **match-case** es una construcción más poderosa y expresiva en comparación con las declaraciones **if-elif-else**.

Mientras que las declaraciones **if-elif-else** se basan en expresiones booleanas, las declaraciones **match-case** pueden coincidir con patrones basados en la estructura y el valor de los datos.

Las declaraciones **match-case** proporcionan una forma más **estructurada y legible de manejar múltiples condiciones** y realizar diferentes acciones en función de esas condiciones.

Lógica booleana

Existen tres operadores lógicos para combinar expresiones booleanas: **and** (y), **or** (o) y **not** (no).

Expresión	Significado
a and b	El resultado es True si ambos (a y b) son True de lo contrario el resultado es False
a or b	El resultado es True si alguno (a o b) es True de lo contrario el resultado es False
not a	El resultado es True si a es False de lo contrario el resultado es False

¿Cuál es el resultado de ejecutar el siguiente código?

```
if 1==1 and 2+2 > 3:
    print("amarillo")
else:
    print("rojo")
```

```
amarillo
-----
(program exited with code: 0)
Presione una tecla para continuar . . .
```

Operador ternario

Los **operadores ternarios**, también conocidos como expresiones condicionales, son operadores que evalúan algo basándose en que una condición sea verdadera o falsa. Simplemente permite probar una condición en una sola línea reemplazando el if-else de varias líneas haciendo el código compacto.

Sintaxis

[si_es_verdadero] if [expresión] else [si_es_falso]

Ejemplo

```
a, b = 10, 20
min = a if a < b else b
print(min)
```

Sentencias de control de flujo: bucles

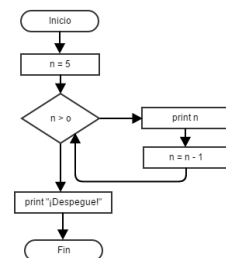
Mientras que las sentencias condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los **bucles** nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

Bucles: sentencia **while**

El bucle **while** (mientras) ejecuta un bloque de instrucciones mientras se cumpla una condición:

```
n = 5
while n > 0:
    print (n)    #se ejecuta mientras se cumpla la condición
    n = n - 1    #se ejecuta mientras se cumpla la condición

print ("¡Despegue!") #se ejecuta siempre.
                    #está fuera del cuerpo del bucle
```



Se puede leer así: mientras n sea mayor que 0, muestra el valor de n y luego reduce el mismo en una unidad. Cuando llegues a 0, sal de la sentencia while y muestra la palabra ¡Despegue!

Este tipo de flujo recibe el nombre de **bucle**. Cada vez que se ejecuta el cuerpo del bucle se dice que realizamos una **iteración**. Para el bucle anterior decimos que ha tenido 5 iteraciones.

Bucles: sentencia **for**

A veces se desea repetir un bucle a través de una colección de cosas, como las líneas de un archivo, una lista de números o una cadena. Cuando se tiene un elemento recorrible, se puede construir un bucle **definido** utilizando una sentencia **for**.

A la sentencia **while** se le llama bucle **indefinido**, porque simplemente se repite hasta que cierta condición se hace falsa. Mientras que el bucle **for** se repite a través de un conjunto de elementos, de modo que ejecuta tantas iteraciones como elementos hay en el conjunto.

Bucles: sentencia **for...in**

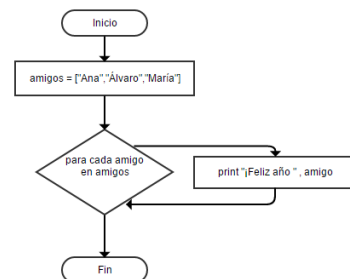
```
amigos = ["Ana", "Pepe", "Juana"]
for amigo in amigos:
    print ("¡Feliz año " + amigo + "!")
```

La forma general del bucle **for..in** es:

```
for <variable> in <secuencia de valores>:
    <cuerpo>
```

La variable `amigos` es una lista (examinaremos las listas con más detalle posteriormente) de tres cadenas y el bucle `for` se mueve recorriendo la lista y ejecuta el cuerpo para cada una de las tres cadenas en la lista.

For e **in** son palabras reservadas, mientras que *amigo* y *amigos* son variables. En concreto, *amigo* es la **variable de la iteración**, cambia en cada iteración del bucle.



Bucles

while (mientras se cumpla la condición)	for (para cada valor en la secuencia)
while condición: instrucción 1 instrucción 2 . . . instrucción n	for variable in secuencia de valores: instrucción 1 instrucción 2 . . . instrucción n

Nota: la **secuencia e valores** puede ser una lista, una cadena o un rango (range)

Función **range()**

Si se necesita iterar sobre una secuencia de números, es apropiado utilizar la función **range()**.

range() toma un entero como parámetro y regresa una **secuencia de números enteros** en sucesión aritmética

```
range(5)
```

El valor final dado nunca es parte de la secuencia, **range(5)** genera 5 valores: 0,1,2,3,4

Función range()

Para **ver** los valores creados con range(), es **necesario convertir el resultado a una lista mediante la función list()**.

```
print(list(range(5)))
```

```
[0, 1, 2, 3, 4]
```

```
-----  
(program exited with code: 0)
```

```
Presione una tecla para continuar . . .
```

Función range()

La función range() puede tener **uno, dos o tres argumentos numéricos enteros**.

Es posible hacer que el rango comience con otro número, o especificar un incremento diferente (incluso negativo; algunas veces se lo llama 'paso'):

range(5, 10)

Devolverá los valores: 5, 6, 7, 8, 9

range(0, 10, 3)

Devolverá los valores: 0, 3, 6, 9

range(-10, -100, -30)

Devolverá los valores: -10, -40, -70

Función range()


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

Salida por pantalla:

0
1
2
3
4




Actividad

4. Escribe el código para un bucle tipo **while** el cual imprima el numero 0 hasta el 7.
 5. Escribe el código para un bucle tipo **for** el cual imprime el numero 0 hasta el 7.
 6. Escribe un programa que pida al usuario un número entero positivo y muestre por pantalla todos los números impares desde 1 hasta ese número separados por comas
- 

Control de estructuras repetitivas

Las estructuras de repetición no tienen sentido si no es que la **condición lógica** depende de alguna **variable** que pueda ver **modificado su valor para diferentes ejecuciones**.


En caso contrario, la condición siempre valdrá lo mismo para cualquier ejecución posible y usar una estructura de control no será muy útil.



Control de estructuras repetitivas

En este caso, si la condición siempre es **false**, nunca se ejecuta el bucle, por lo que es código inútil.

Pero, para las estructuras de repetición, si la condición siempre es **true** el problema es mucho más grave. Como absolutamente siempre que se evalúa si es precisa una nueva iteración, la condición se cumple, el bucle no se deja nunca de repetir. **¡El programa nunca acabará!**



Control de estructuras repetitivas

Un **bucle infinito** es una secuencia de instrucciones dentro de un programa que itera indefinidamente, normalmente porque se espera que se alcance una condición que nunca se llega a producir.



Uso específico de variables: contadores, acumuladores e indicadores

Contador

No es más que una variable que cuenta. Normalmente usamos un contador dentro de un bucle y cambiamos su valor sumándole o restándole una constante, es decir, siempre se le suma o resta la misma cantidad.

- Se **inicializa** con un valor

`cont = 0`

- Se **incrementa/decrementa** cuando ocurre el evento que estamos contando

`cont = cont + 1`



Nota: otra forma de incrementar el contador es `cont += 1`

Uso específico de variables: contadores, acumuladores e indicadores

Contador

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

El ejemplo usa una variable *i* para contar las iteraciones y se imprime por la consola el número de cada iteración: 1,2,3,4,5.

Uso específico de variables: contadores, acumuladores e indicadores

Acumulador

Son variables que acumulan valores de una operación continua. Variable que se incrementa o decrementa en una cantidad no contante (variable, no fija). Al igual que un contador se utiliza normalmente dentro de un bucle pero operando con un valor variable (no siempre se opera con la misma cantidad)

- Se **inicializa** en un valor según la operación que se va a acumular (0 si es suma, 1 si es producto)
- Se **acumula** un valor intermedio

```
acum = acum + num
```

Cantidad a acumular



Uso específico de variables: contadores, acumuladores e indicadores

Acumulador

Ejemplo

Introducir 5 número y sumar los números pares.

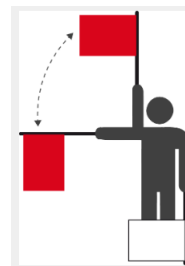
```
suma = 0;
for var in range(1,6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        suma = suma + num
print("La suma de los números pares es ",suma)
```

Uso específico de variables: contadores, acumuladores e indicadores

Indicador, centinela o bandera

Es una variable, normalmente de tipo lógica (boolean), que conserva un estado hasta que un evento requiera cambiarlo y ejecutar otra funcionalidad.

- Se inicializa a un valor lógico
bandera = False
- Cuando ocurre el evento cambiamos el valor
bandera = True



Uso específico de variables: contadores, acumuladores e indicadores

Indicador, centinela o bandera (flag)

```
bandera = True
i = 1
while bandera:
    print(i)
    i += 1
    if i == 11:
        bandera = False
print("fin")
```

Bucles anidados

Al igual que las sentencias condicionales, los bucles pueden anidarse unos dentro de otros. Los bucles anidados le permiten iterar sobre dos o más variables.

¿Qué muestra el siguiente código?

```
for i in range(1,6):
    for j in range(i,0,-1):
        print("*", end=" ")
    print()
```

Bucles anidados

```
for i in range(1,6):
    for j in range(i,0,-1):
        print("*", end=" ")
    print()
```

```
*
**
***
****
*****
```

```
for x in range(1,6):
    print("*"*x)
```

Bucles: sentencias `break` y `continue`

Puedes utilizar la palabra reservada `break` para salir del bucle `while` completamente.

```
for letra in "Python":
    if letra == "h":
        break
    print ("Letra actual : " + letra)
```

Puedes utilizar la palabra reservada `continue` dentro de un bucle, para detener el procesamiento de la iteración actual para ir inmediatamente a la siguiente iteración.

```
for letra in "Python":
    if letra == "h":
        continue
    print ("Letra actual : " + letra)
```

Bucles: sentencias `break` y `continue`

¿Cuál es la salida de este programa?

```
while True:
    nombre = input("Ingresa tu nombre: ")
    if nombre != "Pedro":
        continue
    clave = input("Hola " + nombre + " dime tu clave: ")
    if clave == "pulpo":
        break


print ("Acceso otorgado")
```

Actividad

7. Desarrolla una aplicación que muestre los números del 1 al 20, excepto el 15.
8. Desarrolla una aplicación que muestre los números del 1 al 10, usando un bucle for que vaya del 1 al 20.

¿El uso de break y continue es una mala práctica de programación?

¿Somos malos programadores si utilizamos estas estructuras de salto (break y continue)?
Depende de lo que estés haciendo



¿El uso de break y continue es una mala práctica de programación?

Úsalos con moderación. En líneas generales:

- Cuando se usan al comienzo de un bloque, cuando se realizan las primeras verificaciones, actúan como condiciones previas, por lo que es bueno. Si mejora la legibilidad del código, úsalas.
- Cuando se usan en el medio del bloque, con algo de código, actúan como trampas ocultas, por lo que es malo.

Los buenos programadores usan la solución más simple y limpia posible.



¿El uso de break y continue es una mala práctica de programación?

Tal vez el error esté en comparar las sentencias break y continue con la sentencia **GOTO** (una sentencia propia de los primeros lenguajes de programación como Basic)

El propósito de la instrucción GOTO es transferir el control a un punto determinado del código, donde debe continuar la ejecución.

```
goto etiqueta;

...
...
...

etiqueta: sentencia;
```

Ejemplo

```
template <typename T>
void goto_sort( T array[], size_t n ) {
    size_t i{1}

    first: T current{array[i]};
           size_t j{i};

    second: if ( array[j - 1] <= current ) {
              goto third;
            }

    array[j] = array[j - 1];

    if ( --j ) {
        goto second;
    }

    third: array[j] = current;

    if ( ++i != n ) {
        goto first;
    }
}
```


¿El uso de break y continue es una mala práctica de programación?

La instrucción GOTO ha sido menospreciada en los lenguajes de alto nivel, debido a la dificultad que presenta para poder seguir adecuadamente el flujo del programa.

Esto podía derivar en **código espagueti**

El código espagueti, es un nombre peyorativo utilizado para designar aquellos programas cuyo flujo de ejecución se asemeja a una caótica maraña de espaguetis entrelazados, convirtiéndolo en algo casi imposible de seguir.



¿El uso de break y continue es una mala práctica de programación?

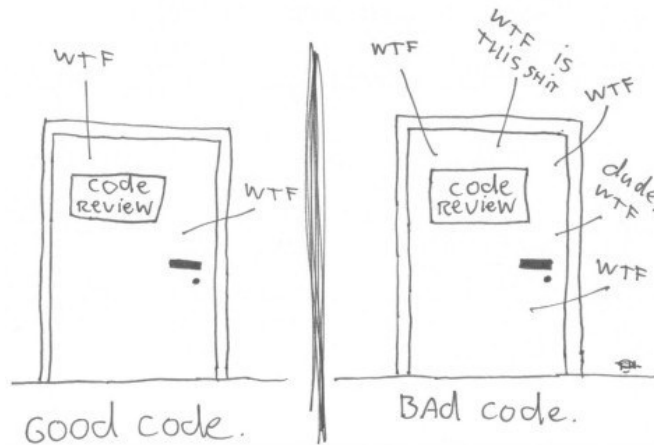
Volviendo a las sentencias break y continue, estas no permiten transferir el control a cualquier parte del código como GOTO.

“Los malos programadores hablan en términos absolutos. Los buenos programadores utilizan la solución más clara posible.”

El abuso en el uso de break y continue hace que el código sea difícil de seguir. Pero si el no usarlos hace que el código sea aún más difícil de leer, entonces es un mal cambio.”



The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>

La sentencia: `pass`

La sentencia `pass` no hace nada.

La utilizaremos cuando queramos crear una clase, método, función, bucle o condicional, pero en el que todavía no queramos definir **ningún comportamiento**. En otros lenguajes sería algo así como declarar por ejemplo una función y entre las llaves no añadir ningún dato. Ejemplos:

```
if 1 == 2:
    pass


while True:
    pass

for num in range(1, 5):
    pass
```

Importando módulos

Todos los programas Python pueden utilizar un conjunto de **funciones predefinidas**, que incluyen por ejemplo **print()**, **len()**, **str()**, **int()**, **input()**, etc.

Python también viene con un conjunto de **módulos** llamados **librerías estándar**. Cada librería es un programa Python que contiene un grupo de funciones relacionadas que pueden ser utilizadas en nuestros programas. Por ejemplo, la librería **math** tiene funciones matemáticas: **abs(x)**, **sqrt(x)**




Módulo de funciones matemáticas: **math**

Pero para poder utilizar las funcionalidades de los módulos debemos importar el módulo con la sentencia **import**.

```
import math
```

```
potencia = math.pow(2,3)  
print(potencia)
```



Módulo de funciones para números aleatorios: **random**

```
import random
for x in range(5):
    aleatorio = random.randint(1,10)
    print (aleatorio)
```


La función **randint(a, b)** genera un número entero entre **a** y **b**, ambos incluidos. **a** debe ser inferior o igual a **b**.



Módulo de funciones para números aleatorios: **random**

Una forma alternativa para la sentencia import es

```
from random import randint
for x in range(5):
    aleatorio = randint(1,10)
    print(aleatorio)
```



Módulo de funciones para números aleatorios: **random**

La función **sample()** devuelve una lista de elementos sin repetición aleatoriamente seleccionados de una secuencia de longitud determinada.

```
from random import sample

lista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(sample(lista,3))

cadena = "¡Hola mundo!"
print(sample(cadena,4))

print(sample(range(100), 5))
```

En cada ejecución da un resultado diferente con elementos no repetidos:

```
[3, 8, 6]
['d', 'u', '!', 'm']
[93, 8, 35, 33, 40]
```

```
[1, 5, 9]
['o', 'l', 'n', 'm']
[76, 0, 87, 77, 49]
```

Módulo de funciones para números aleatorios: **random**

La función **choice()** devuelve un elemento aleatoriamente seleccionado de una secuencia tal como una lista, una tupla, una cadena o un rango.

```
from random import choice

li_lista = ["apple", "banana", "cherry"]
print(choice(li_lista))

cadena = "Hola"
print(choice(cadena))
```

Una salida de este código podría ser

```
banana
l
```

Autoevaluación

1. Explica qué es una **sentencia condicional** y dónde la usarías.
2. Explica qué es una **sentencia repetitiva** y dónde la usarías
3. ¿Cuál es la diferencia entre `range(10)`, `range(0,10)` y `range(0,10,1)`?
4. ¿Cuál es la diferencia entre **break** y **continue**?
5. Busca las funciones **round()** y **abs()** en Internet, mira que hacen y experimenta con ellas en la consola interactiva.