


6 Control de flujo.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

En Python existen declaraciones que permiten controlar el flujo de un programa para realizar acciones complejas. Entre estas declaraciones tenemos las siguientes:

- `while`
- `for`
- `if`
- `match`

Junto con estas declaraciones generalmente se utilizan las siguientes operaciones lógicas cuyo resultado puede ser `True` o `False`:

Python	Significado
<code>a == b</code>	¿son iguales <code>a</code> y <code>b</code> ?
<code>a != b</code>	¿son diferentes <code>a</code> y <code>b</code> ?
<code>a < b</code>	¿ <code>a</code> es menor que <code>b</code> ?:
<code>a <= b</code>	¿ <code>a</code> es menor o igual que <code>b</code> ?
<code>a > b</code>	¿ <code>a</code> es mayor que <code>b</code> ?
<code>a >= b</code>	¿ <code>a</code> es mayor o igual que <code>b</code> ?
<code>not A</code>	El inverso de la expresión <code>A</code>
<code>A and B</code>	¿La expresión <code>A</code> y la expresión <code>B</code> son verdaderas?
<code>A or B</code>	¿La expresión <code>A</code> o la expresión <code>B</code> es verdadera?:

7 while

Se utiliza para repetir un conjunto de instrucciones mientras una expresión sea verdadera:

```
while expresión:  
    código ...
```

Por ejemplo:

```
a = 0 # Inicializamos a en 0

print('Inicia while') # Instrucción fuera del bloque while

while a < 5: # Mientras a sea menor que 5 realiza lo siguiente:
    print(a) # Imprime el valor de a
    a += 1   # Incrementa el valor de a en 1

print('Finaliza while') # Instrucción fuera del bloque while
```

- Como se observa, el código después de `while` tiene una sangría (*indentation*): las líneas de código están recorridas hacia la derecha.
- Este espacio en blanco debe ser al menos de uno, pero pueden ser más.
- Por omisión, en JupyterLab (y algunos otros editores, se usan 4 espacios en blanco para cada línea de código dentro del bloque.
- El número de espacios en blanco se debe mantener durante todo el bloque de código.
- Cuando termina el sangrado, es decir las líneas de código ya no tienen ningún espacio en blanco al inicio, se cierra el bloque de código, en este caso el `while`.
- El uso de una sangría para organizar los bloques de código lo hace Python para que el código sea más entendible.
- **Ejemplos válidos:**

```
while a < 5:
    print(a)
    a += 1
```

```
while a < 5:
    print(a)
    a += 1
```

- **Ejemplos NO válidos**

```
while a < 5:
    print(a)
a += 1
```

```
while a < 5:
print(a)
a += 1
```

7.1 Ejemplo 1.

Los números de Fibonacci, denotados con F_n forman una secuencia tal que cada número es la suma de dos números precedentes e inicia con el 0 y el 1. Matemáticamente se escribe como:

$$\begin{aligned}F_0 &= 0 \\F_1 &= 1 \\F_n &= F_{n-1} + F_{n-2} \quad \text{para } n > 1.\end{aligned}$$

La secuencia es entonces: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Vamos a calcular esta secuencia usando la instrucción `while`:

```
a, b = 0, 1 # Definimos los primeros dos elementos:

while a < 1000:      # Mientras a sea menor que 1000 realiza lo siguiente:
    print(a, end=', ') # Imprime a y b (separados por una coma)
    a, b = b, a+b     # Calcula los siguientes dos elementos
```

8 if, elif, else

Esta declaración permite ejecutar un código dependiendo del resultado de una o varias expresiones lógicas. La estructura es como sigue:

```
if expresion1:
    codigo1 ...
elif expresion2:
    codigo2 ...
elif expresion3:
    codigo3 ...
else:
    codigo4
```

Si la `expresion1` es verdadera, entonces se ejecuta el `codigo1`, en otro caso se evalúan las siguientes expresiones y dependiendo de cuál es verdadera se ejecuta el código correspondiente. Cuando ninguna de las expresiones es verdadera, entonces se ejecuta el código de la sección `else`, es decir el `codigo4`.

Observa que se siguen las mismas reglas de sangrado que en el `while`.

Veamos un ejemplo:

```
# Modifica los valores de a y b, y observa el resultado
a = 10
b = 20
```

```
if a < b:
    print('a es menor que b')
elif a > b:
    print('a es mayor que b')
elif a == b:
    print('a es igual a b')
else:
    print('Esto nunca pasa')
```

Las expresiones pueden ser más complejas:

```
# Modifica los valores de a y b, y observa el resultado
a = 10
b = 20
if (a < b) or (a > b):
    print(f'a = {a}, b = {b}')
```

9 Operador ternario

Este operador permite evaluar una expresión lógica y generar un valor para un resultado **True** y otro diferente para un resultado **False**; todo esto se logra en una sola línea de código como sigue:

```
resultado = valor1 if expresion else valor2
```

Por ejemplo:

```
# Usa valores para c = 1, 2, 4, 4, 5, 6, 20 y observa el resultado
c = 1
r = c if c > 5 else 0
print(r)
```

10 for

Permite iterar sobre el contenido de cualquier secuencia (cadena, lista, tupla, conjunto, diccionario, archivo, ...). La forma de esta declaración es como sigue:

```
for i in secuencia:
    codigo
```

Las reglas de sangrado se siguen en esta declaración.

Por ejemplo:

```
gatos = ['Persa', 'Sphynx', 'Ragdoll', 'Siamés']
```

```
for i in gatos:
    print(i)
```

10.1 Función zip

La función `zip(s1, s2, ...)` permite combinar dos o más secuencias en una sola; genera tuplas con los elementos de: las secuencias y va iterando sobre ellas.

Por ejemplo

```
# Dos listas de la misma longitud
gatos = ['Persa', 'Sphynx', 'Ragdoll', 'Siamés']
origen = ['Irán', 'Toronto', 'California', 'Tailandia']
print(gatos)
print(origen)

# Combinamos las listas en una sola secuencia
print('\n(Raza, Origen)')
print('-'*20)
for t in zip(gatos, origen):
    print(t)
```

```
['Persa', 'Sphynx', 'Ragdoll', 'Siamés']
['Irán', 'Toronto', 'California', 'Tailandia']
```

```
(Raza, Origen)
```

```
-----
('Persa', 'Irán')
('Sphynx', 'Toronto')
('Ragdoll', 'California')
('Siamés', 'Tailandia')
```

```
# Se puede extraer la información de cada secuencia:
for g, o in zip(gatos, origen):
    print('La raza {} proviene de {}'.format(g, o))
```

```
La raza Persa proviene de Irán
La raza Sphynx proviene de Toronto
La raza Ragdoll proviene de California
La raza Siamés proviene de Tailandia
```

10.2 Conversión de zip a list, tuple, set, dict

Estrictamente `zip` es una clase que define un tipo dentro de Python, por lo que es posible convertir del tipo `zip` a alguna otra secuencia básica de datos de Python.

```
z = zip(gatos, origen)

# Verificar el tipo de zip
print(type(z))
print(z)
```

```
<class 'zip'>
<zip object at 0x7f0598954a40>
```

```
lista = list(zip(gatos, origen))
tupla = tuple(zip(gatos, origen))
conj = set(zip(gatos, origen))
dicc = dict(zip(gatos, origen)) # Solo funciona para dos secuencias

print(lista)
print(tupla)
print(conj)
print(dicc)
```

```
[('Persa', 'Irán'), ('Sphynx', 'Toronto'), ('Ragdoll', 'California'), ('Siamés',
'Tailandia')]
(('Persa', 'Irán'), ('Sphynx', 'Toronto'), ('Ragdoll', 'California'), ('Siamés',
'Tailandia'))
{('Persa', 'Irán'), ('Sphynx', 'Toronto'), ('Siamés', 'Tailandia'), ('Ragdoll',
'California')}
```

```
{'Persa': 'Irán', 'Sphynx': 'Toronto', 'Ragdoll': 'California', 'Siamés': 'Tailandia'}
```

10.3 Función **enumerate**

Permite enumerar los elementos de una secuencia. Genera tuplas con el número del elemento y el elemento de la secuencia.

Por ejemplo:

```
print(gatos)

# Enumeramos la secuencia
print('\n(Numero, Raza)')
print('-'*20)

for t in enumerate(gatos):
    print(t)
```

```
['Persa', 'Sphynx', 'Ragdoll', 'Siamés']
```

```
(Numero, Raza)
```

```
-----
```

```
(0, 'Persa')
(1, 'Sphynx')
(2, 'Ragdoll')
(3, 'Siamés')
```

```
for i, g in enumerate(gatos):
    print(i, g)
```

```
0 Persa
1 Sphynx
2 Ragdoll
3 Siamés
```

Lo anterior permite usar el indexado para acceder a los elementos de una secuencia:

```
for i, g in enumerate(gatos):
    print(i, gatos[i])
```

```
0 Persa
1 Sphynx
2 Ragdoll
3 Siamés
```

10.4 Conversión de `enumerate` a `list`, `tuple`, `set`, `dict`

Estrictamente `enumerate` es una clase que define un tipo dentro de Python, por lo que es posible convertir del tipo `enumerate` a alguna otra secuencia básica de datos de Python:

```
e = enumerate(gatos)

# Verificar el tipo de enumerate
print(type(e))
print(e)
```

```
<class 'enumerate'>
<enumerate object at 0x7f0598472d90>
```

```
lista = list(enumerate(gatos))
tupla = tuple(enumerate(gatos))
conj = set(enumerate(gatos))
dicc = dict(enumerate(gatos))

print(lista)
print(tupla)
print(conj)
print(dicc)
```

```
[(0, 'Persa'), (1, 'Sphynx'), (2, 'Ragdoll'), (3, 'Siamés')]
((0, 'Persa'), (1, 'Sphynx'), (2, 'Ragdoll'), (3, 'Siamés'))
{(1, 'Sphynx'), (0, 'Persa'), (3, 'Siamés'), (2, 'Ragdoll')}
```

```
{0: 'Persa', 1: 'Sphynx', 2: 'Ragdoll', 3: 'Siamés'}
```

10.5 Funcion range

Esta función genera una secuencia iterable con un inicio, un final y un salto:

```
range(start, stop, step)
```

La secuencia irá desde **start** hasta **stop-1** en pasos de **step**. Por ejemplo:

```
for i in range(1,20): # Por omisión step = 1
    print(i, end= ', ')
```

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,

```
for i in range(1,20, 2):
    print(i, end= ', ')
```

1, 3, 5, 7, 9, 11, 13, 15, 17, 19,

```
for i in range(20, 1, -2): # El paso puede ser negativo
    print(i, end= ', ')
```

20, 18, 16, 14, 12, 10, 8, 6, 4, 2,

Usando **range()** se puede acceder a una secuencia mediante el indexado:

```
N = len(gatos) # Longitud de la lista gatos

for i in range(0, N):
    print(i, gatos[i])
```

```
0 Persa
1 Sphynx
2 Ragdoll
3 Siamés
```

10.6 Conversión de range a list, tuple, set

Estrictamente **range** es una clase que define un tipo dentro de Python, por lo que es posible convertir del tipo **range** a alguna otra secuencia básica de datos de Python:


```
N = len(gatos) # Longitud de la lista gatos
r = range(0,N)

# Verificar el tipo de range
print(type(r))
print(r)
```

```
<class 'range'>
range(0, 4)
```

```
lista = list(range(0,N))
tupla = tuple(range(0,N))
conj = set(range(0,N))

print(lista)
print(tupla)
print(conj)
```

```
[0, 1, 2, 3]
(0, 1, 2, 3)
{0, 1, 2, 3}
```

10.7 break, continue, else, pass

Estas son palabras clave que se pueden usar en ciclos `while` o `for`: * `break`: terminar el ciclo más interno. * `continue`: saltarse a la siguiente iteración sin terminar de ejecutar el código que sigue. * `else`: **NO** se ejecuta el código de esta cláusula si el ciclo es finalizado por el `break`. * `pass`: no hacer nada y continuar.

Veamos algunos ejemplos:

```
# Se itera por una lista de palabras, cuando se encuentra
# la letra 'h' se termina el ciclo interno y se continua con
# la siguiente palabra.
for palabra in ["Hola", "mundo", "Pythonico"]:
    print('Palabra: ', palabra)
    for letra in palabra:
        print('\t letra: ', letra)
        if letra == "h":
            break
```

```
Palabra:  Hola
    letra:  H
    letra:  o
    letra:  l
    letra:  a
Palabra:  mundo
    letra:  m
```

```

letra: u
letra: n
letra: d
letra: o
Palabra: Pythonico
letra: P
letra: y
letra: t
letra: h

```

```

# Se itera por una lista de palabras, cuando se encuentra
# la letra 'h' se termina el ciclo interno y se continua con
# la siguiente palabra. La cláusula 'else' se ejecuta si no
# se encuentra la letra.
for palabra in ["Hola", "mundo", "Pythonico"]:
    print('Palabra: ', palabra)
    for letra in palabra:
        print('\t letra: ', letra)
        if letra == "h":
            break
    else:
        print('No encontré la letra "h"')

```

```

Palabra: Hola
letra: H
letra: o
letra: l
letra: a
No encontré la letra "h"
Palabra: mundo
letra: m
letra: u
letra: n
letra: d
letra: o
No encontré la letra "h"
Palabra: Pythonico
letra: P
letra: y
letra: t
letra: h

```

```

# Esta declaración pass no hace nada. Se usa principalmente
# para cuestiones de desarrollo de código a un nivel abstracto.
i = 0
while i > 10:
    pass

```

```
# La siguiente función calcula la secuencia de Fibonacci
def fib(n):
    #     print(i, end=" ")
    pass

# En este punto del programa requiero el uso de la función fib(n):

fib(100000) #
```

10.7.1 Más ejemplos.

```
# Calcula números primos usando la
# criba de Eratóstenes
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'igual a ', x, '*', n//x)
            break
    else:
        print(n, 'es un número primo')
```

```
2 es un número primo
3 es un número primo
4 igual a  2 * 2
5 es un número primo
6 igual a  2 * 3
7 es un número primo
8 igual a  2 * 4
9 igual a  3 * 3
```

```
# Determina números pares e impares
for num in range(2, 10):
    if num % 2 == 0:
        print("Número par ", num)
        continue
    print("Número impar", num)
```

```
Número par  2
Número impar 3
Número par  4
Número impar 5
Número par  6
Número impar 7
Número par  8
Número impar 9
```

```
# Checa la clave de un usuario. Después de tres
# intentos fallidos termina. Si se da la clave
```

```
# correcta (despedida) se termina.  
suma = 0  
while suma < 3:  
    entrada = input("Clave:")  
    if entrada == "despedida":  
        break  
    suma = suma + 1  
    print("Intento %d. \n " % suma)  
print("Tuviste {} intentos fallidos.".format(suma))
```

Clave: despedida

Tuviste 0 intentos fallidos.

11 match (desde la versión 3.10)

Similar al switch de lenguajes como C, C++, Java.

```
def http_error(status):  
    match status:  
        case 400:  
            return "Bad request"  
        case 404:  
            return "Not found"  
        case 418:  
            return "I'm a teapot"  
        case _:  
            return "Something's wrong with the internet"
```

```
# Modifica el valor del argumento y observa lo que sucede  
http_error(500)
```

"Something's wrong with the internet"

```
# Modifica los valores de la siguiente tupla y observa el resultado  
point = (0,0)  
  
match point:  
    case (0, 0):  
        print("Origin")  
    case (0, y):  
        print(f"Y={y}")  
    case (x, 0):  
        print(f"X={x}")  
    case (x, y):  
        print(f"X={x}, Y={y}")
```

```
case _:  
    raise ValueError("Not a point")
```

Origin

Para más detalles véase [match Statements](#).