

Programar (en Python)

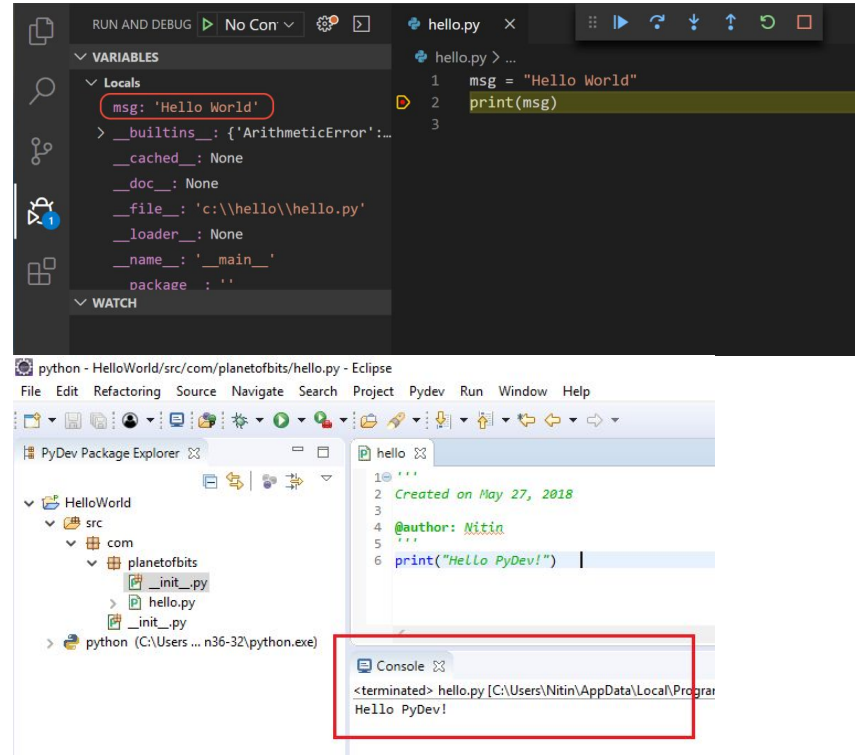
Módulo de Programación.
CFGS Desarrollo de Aplicaciones Web
IES Gran Capitán



python™

¿Cómo hacer programas en Python?

- Editor de textos
 - Visual Studio Code
 - Geany
 - Sublime Text
- IDE
 - Eclipse
 - Extensión PyDev
 - PyCharm
 - NetBeans





Convenciones y estilo

- Los ficheros con los programas tienen extensión **py**.
- Usar comentarios.
 - Al principio para indicar qué hace el programa, autor/a, versión.
 - Es interesante, sobre todo al principio, poner el algoritmo.
 - Donde haya más complejidad.
- Separar partes del programa con líneas en blanco.
- Python recomienda una guía de estilo recogida en documentos PEPs (*Python Enhancement Proposals*).
 - El actual se llama [PEP8](#).



Primer programa

perimetro.py

```
1 from math import pi
2 radio = 1
3 perímetro = 2 * pi * radio
4 perímetro
```

¿Qué hace? ¿funciona?

¿qué le falta? ¿cómo lo podemos mejorar?



Lectura de datos desde teclado: **input()**

La función **input()** detiene la ejecución del programa y espera a que el usuario escriba un texto y pulse la tecla de retorno de carro; en ese momento prosigue la ejecución y la función **devuelve una cadena** con el texto que **tecleó** el usuario.

Admite como parámetro una cadena de caracteres que se mostrará antes del *cursor*.



input()



Ejemplos

`input([prompt])`

Si el argumento *prompt* está presente, se escribe a la salida estándar sin una nueva línea a continuación. La función lee entonces una línea de la entrada, la convierte en una cadena (eliminando la nueva línea), y retorna eso. Cuando se lee EOF, se lanza una excepción `EEOFError`. Ejemplo:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

Si el módulo `readline` estaba cargado, entonces `input()` lo usará para proporcionar características más elaboradas de edición de líneas e historiales.

Lanza un evento de auditoría `builtins.input` con el argumento `prompt` antes de leer entrada

Lanza un evento de auditoría `builtins.input/result` con el resultado justo después de haber leído con éxito la entrada.



print()



Ejemplos

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Imprime *objects* al flujo de texto *file*, separándolos por *sep* y seguidos por *end*. *sep*, *end*, *file* y *flush*, si están presentes, deben ser dados como argumentos por palabra clave.

Todos los argumentos que no son por palabra clave se convierten a cadenas tal y como `str()` hace y se escriben al flujo, separados por *sep* y seguidos por *end*. Tanto *sep* como *end* deben ser cadenas; también pueden ser `None`, lo cual significa que se empleen los valores por defecto. Si no se indica *objects*, `print()` escribirá *end*.

El argumento *file* debe ser un objeto que implemente un método `write(string)`; si éste no está presente o es `None`, se usará `sys.stdout`. Dado que los argumentos mostrados son convertidos a cadenas de texto, `print()` no puede ser utilizada con objetos fichero en modo binario. Para esos, utiliza en cambio `file.write(...)`.

Que la salida sea en búfer o no suele estar determinado por *file*, pero si el argumento por palabra clave *flush* se emplea, el flujo se descarga forzosamente.

Legibilidad de los programas

`ilegible.py`

```
1 h = float(input('Dame h: '))
2 v = float(input('y v: '))
3 z = h * v
4 print('Resultado 1 {0:6.2f}'.format(z))
5 v = 2 * h + v + v
6 print('Resultado 2 {0:6.2f}'.format(v))
```




Legibilidad de los programas

legible.py

```
1 print('Programa para el cálculo del perímetro y el área de un rectángulo.')
2
3 altura = float(input('Dame la altura (en metros): '))
4 anchura = float(input('Dame la anchura (en metros): '))
5
6 área = altura * anchura
7 perímetro = 2 * altura + 2 * anchura
8
9 print('El perímetro es de {0:6.2f} metros.'.format(perímetro))
10 print('El área es de {0:6.2f} metros cuadrados.'.format(área))
```



Comentarios

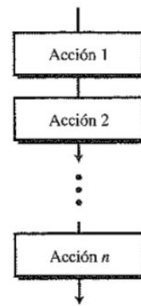
rectangulo.py

```
1 # Programa: rectangulo.py
2 # Propósito: Calcula el perímetro y el área de un rectángulo a partir de su altura y anchura.
3 # Autor: John Cleese
4 # Fecha: 1/1/2010
5
6 # Petición de los datos (en metros)
7 altura = float(input('Dame la altura (en metros): '))
8 anchura = float(input('Dame la anchura (en metros): '))
9
10 # Cálculo del área y del perímetro
11 área = altura * anchura
12 perímetro = 2 * altura + 2 * anchura
13
14 # Impresión de resultados por pantalla
15 print('El perímetro es de {0:6.2f} metros.'.format(perímetro)) # solo dos decimales.
16 print('El área es de {0:6.2f} metros cuadrados.'.format(área))
```

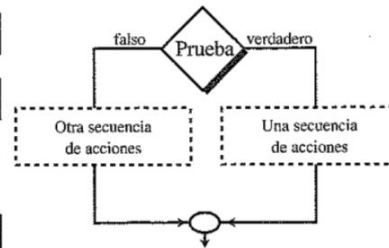
Programación estructurada

La *programación estructurada* es un **paradigma** de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de ordenador o algoritmo, utilizando únicamente **subrutinas** (funciones o procedimientos) y **tres estructuras de control**:

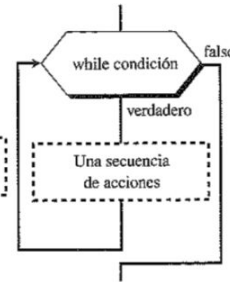
- Secuenciales.
- Alternativas.
- Repetitivas.



a. Secuencia



b. Decisión



c. Repetición

Ejemplo: resolución ecuación 1er. grado

¿Qué falla en este programa? ¿se puede arreglar? ¿cómo?

primer_grado.py

```
1 print('Programa para la resolución de la ecuación  $ax + b = 0$ .')
2
3 a = float(input('Valor de a: '))
4 b = float(input('Valor de b: '))
5
6 x = -b / a
7
8 print('Solución:', x)
```

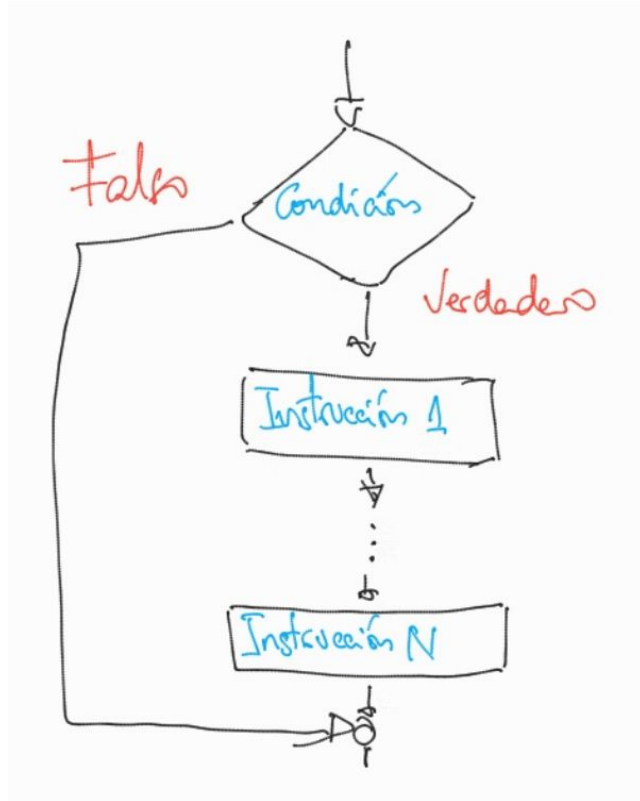


Estructura alternativa SIMPLE

Sentencia que **evalúa** una **expresión lógica**:

- Si es **verdadera** se ejecuta de manera secuencial un bloque de instrucciones.
- Si es **falsa** no se ejecuta nada.
- Una vez terminada se continúa la ejecución de forma secuencial por la siguiente instrucción.

Estructura alternativa SIMPLE



Si condición
Instrucción 1
:
Instrucción n
Fin-Sí

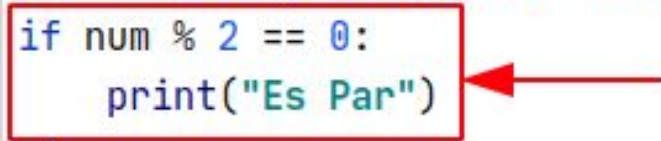


Sentencia **if**

Las acciones, que serán sentencias Python válidas, se escriben con un sangrado mayor que el de la línea que contiene la condición. Estas acciones solo se ejecutan si la condición proporciona como resultado el valor booleano *True*.

```
1 if condición:
2     acción
3     acción
4     ...
5     acción
```

```
num = int(input("Dime el número: "))
if num % 2 == 0:
    print("Es Par")
```





Ejemplo: resolución ecuación 1er. grado.

Sentencia **if**

```
primer_grado.py
1 print('Programa para la resolución de la ecuación  $ax + b = 0$ .')
2
3 a = float(input('Valor de a: '))
4 b = float(input('Valor de b: '))
5
6 if a != 0:
7     x = -b / a
8     print('Solución:', x)
```

¿Algo que mejorar?

Ejemplo: resolución ecuación 1er. grado.

Sentencia **if**

```
primer_grado.py
1 print('Programa para la resolución de la ecuación  $ax + b = 0$ .')
2
3 a = float(input('Valor de a: '))
4 b = float(input('Valor de b: '))
5
6 if a != 0:
7     x = -b / a
8     print('Solución:', x)
9
10 if a == 0:
11     print('La ecuación no tiene solución.')
```

Sentencia **if** *anidada*

Las estructuras de control pueden **anidarse**.

primer_grado.py

```
1 print('Programa para la resolución de la ecuación  $ax + b = 0$ .')
2
3 a = float(input('Valor de a: '))
4 b = float(input('Valor de b: '))
5
6 if a != 0:
7     x = -b / a
8     print('Solución: ', x)
9
10 if a == 0:
11     if b != 0:
12         print('La ecuación no tiene solución.')
13     if b == 0:
14         print('La ecuación tiene infinitas soluciones.')
```

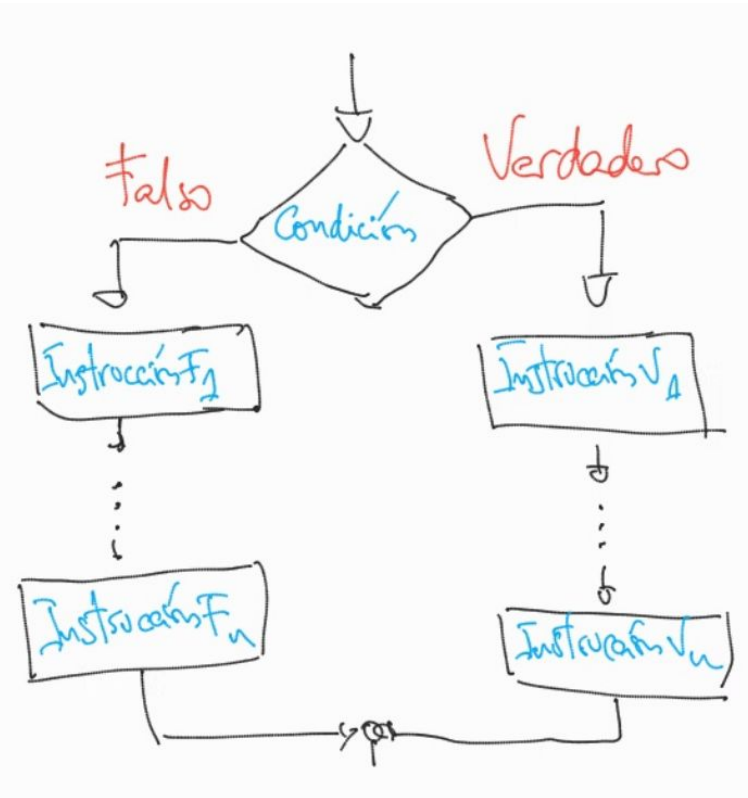


Estructura alternativa DOBLE

Sentencia que **evalúa** una **expresión lógica**:

- Si es **verdadera** se ejecuta de manera secuencial un bloque de instrucciones.
- Si es **falsa** se ejecuta otro bloque de instrucciones.
- Una vez terminada se continúa la ejecución de forma secuencial por la siguiente instrucción.

Estructura alternativa DOBLE



Si condición
Instrucción V₁
⋮
Instrucción V_n

Sino
Instrucción F₁
⋮
Instrucción F_n

FIN-Si

Sentencia **if** (doble)

```
1 if condición:  
2     acciones  
3 if condición contraria:  
4     otras acciones
```

↓

```
1 if condición:  
2     acciones  
3 else:  
4     otras acciones
```

```
num = int(input("Dime el número: "))  
if num % 2 == 0:  
    print("Es Par")  
else:  
    print("Es impar")
```



Ejemplo



Evaluación con **cortocircuitos**

No provoca una *división por cero*:

```
1 if a == 0 or 1/a > 1:  
2     ...
```

Al calcular el resultado de una expresión lógica, normalmente se evalúa (siguiendo las reglas de asociatividad y precedencia oportunas) **lo justo** hasta **conocer** el resultado.

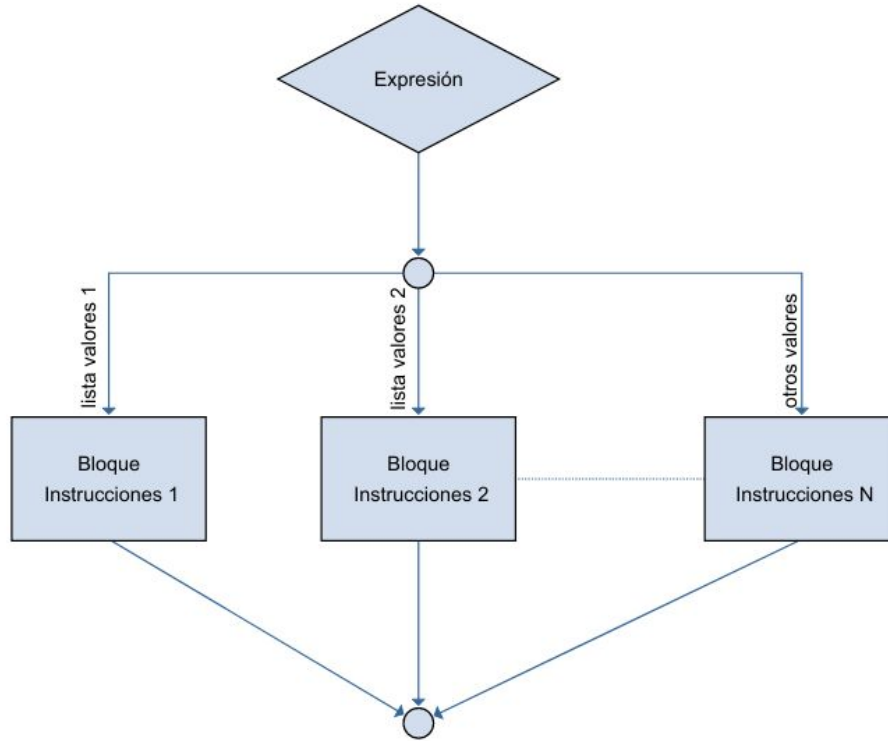


Estructura alternativa MÚLTIPLE

Una sentencia alternativa múltiple permite **seleccionar**, por medio del **valor de una expresión**, el siguiente bloque de instrucciones a ejecutar de entre varios posibles.

Opcionalmente, se puede escribir un bloque de instrucciones a ejecutar en el caso de que el valor obtenido al evaluar la expresión, no se encuentre en ninguna de las listas de valores especificadas.

Estructura alternativa MÚLTIPLE



SEGÚN *expresión*

CASO *valor1*: *instrucciones 1*

CASO *valor2*: *instrucciones 2*

...

OTROS: *instrucciones Otros*

FIN-SEGÚN

Estructura alternativa MÚLTIPLE

- Los lenguajes de alto nivel suelen implementar esta estructura con sentencias **switch**.
- Python ([hasta la versión 3.10](#)) no trae esa sentencia, en su lugar se usa la forma compacta de la sentencia **if**.



Ejemplo

```
1 if condición:
2     ...
3 else:
4     if otra condición:
5         ...
```



```
1 if condición:
2     ...
3 elif otra condición:
4     ...
```



Estructura alternativa MÚLTIPLE

En la versión 3.10 una de las novedades fue la sentencia [match](#) que simplifica la forma compacta de `if`

```
print("Calculadora básica")
print("-----")

# datos de entrada
n1 = float(input("Dame el 1er número: "))
n2 = float(input("Dame el 2º número.: "))

# mostramos las opciones
print("\nMenú de opciones")
print("-----")
print("1. Sumar")
print("2. Restar")
print("3. Multiplicar")
print("4. Dividir")

option = int(input("\nDame una opción: ")) # pedimos la opción
```

```
# ejecutar la opción escogida
match option:
    case 1:
        print("Suma:", n1 + n2)
    case 2:
        print("Resta:", n1 - n2)
    case 3:
        print("Multiplicación:", n1 * n2)
    case 4:
        print("División:", n1 / n2)
    case _:
        print("Opción incorrecta")
```



Estructura repetitiva (o iterativa)

Una estructura repetitiva, **bucle** o **ciclo** es una secuencia de código que se ejecuta repetidas veces por una *sentencia*, hasta que la condición asignada a dicho bucle deja de cumplirse.

Denominamos **iteración** al hecho de repetir un grupo de acciones varias veces. A veces se conoce el número de iteraciones y otras no.



Bucle **while**. Precondición.

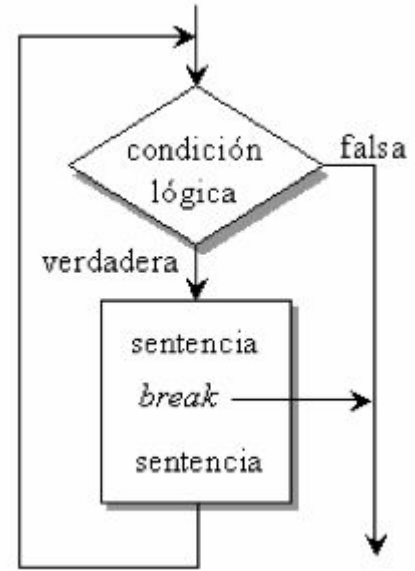
El bucle **while** o bucle **mientras** es un ciclo repetitivo basado en los resultados de una expresión lógica; se encuentra en la mayoría de los lenguajes de programación estructurados.

El propósito es repetir un bloque de código mientras una condición se mantenga verdadera.



Ejemplo

En Python ☐

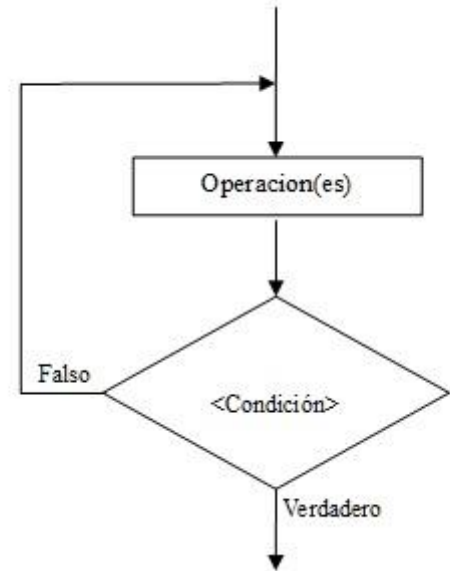


```
1 while condición:  
2     acción  
3     acción  
4     ...  
5     acción
```

Bucle **repetir** / **do-while**. Postcondición.

El bucle **repetir** comprueba la condición de finalización al final del cuerpo del bucle, y si esta es cierta continúa con el resto del programa. Las instrucciones se ejecutarán al menos una vez.

El bucle **do-while** es similar, pero el ciclo terminará si la condición es falsa, al contrario del anterior.





Bucle **repetir** / **do-while**. Postcondición.

El bucle **repetir** puede ser sustituido por un bucle **mientras**.

```
Repetir  
  (Cuerpo del bucle)  
Hasta que (condición)
```



```
(Cuerpo del bucle)  
Mientras NO(condición)  
  (Cuerpo del bucle)  
fmientras
```

Este bucle es útil cuando se desean realizar las acciones que están dentro al menos en una ocasión.

Algunos lenguajes, como Python, prescinden de este bucle.



Bucle **repetir** / **do-while** ¿en Python?

A pesar de que Python no cuenta con el ciclo REPETIR (o con el DO-WHILE) es muy fácil emularlo con un "while True:" y un "if" con un "break" al final del cuerpo del ciclo.

Bucle REPETIR

```
while True:
    instrucciones
    if condicion_de_salida:
        break
```

Bucle DO-WHILE

```
while True:
    instrucciones
    if not condicion_de_seguir:
        break
```



Ejemplo



Bucle **para** (o **for**)

El bucle **for** es una estructura de control en programación en la que se puede indicar de antemano el número máximo de iteraciones.

```
para  $i \leftarrow x$  hasta  $n$  a incrementos de  $s$  hacer  
instrucciones  
fin para
```




Elementos del bucle **for**

- Variable de control.
- Inicialización de la variable de control.
- Condición de control.
- Incremento: se toma por defecto el valor 1.
- Cuerpo: es lo que se hará en cada iteración, pueden ser una o más instrucciones.



Bucle **for** en Python

El [bucle for de Python](#) es una evolución del concepto de visto anteriormente. Se le suele llamar **for-each**.

```
1 for variable in serie de valores:  
2     acción  
3     acción  
4     ...  
5     acción
```

Se lee como «*para todo elemento de una serie, hacer. . .*».



Bucle **for** en Python

potencias.py

```
1 número = int(input('Dame un número: '))
2
3 for potencia in [2, 3, 4, 5]:
4     print('{0} elevado a {1} es {2}'.format(número, potencia, número ** potencia))
```

raices.py

```
1 número = float(input('Dame un número: '))
2
3 for n in range(2, 101):
4     print('La raíz {0}-ésima de {1} es {2}'.format(n, número, número**(1/n)))
```



Esquema iterativo general

Las tres estructuras vistas NO RESUELVEN todos los casos que nos podemos encontrar de la forma más EFICIENTE posible:

- El esquema PARA no nos sirve cuando no conocemos a priori el número de iteraciones.
- Los esquemas REPETIR y MIENTRAS no son la solución más eficiente en algunas circunstancias por el hecho de testear la condición al final o al principio únicamente.



Esquema iterativo general

La solución estaría en un esquema que nos permita controlar el grupo de acciones que se repiten en cualquier punto. A este esquema le vamos a llamar **ITERAR**.

ITERAR

instrucciones

SALIR SI condición

instrucciones

FIN-ITERAR



en Python

while True:

instrucciones

if condición:

break

instrucciones



Ejemplo



Esquemas iterativos anidados

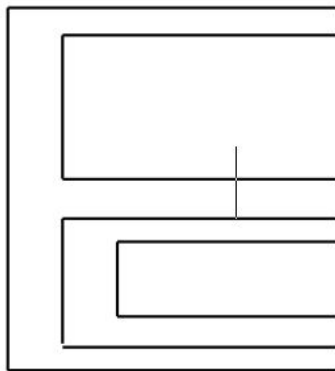
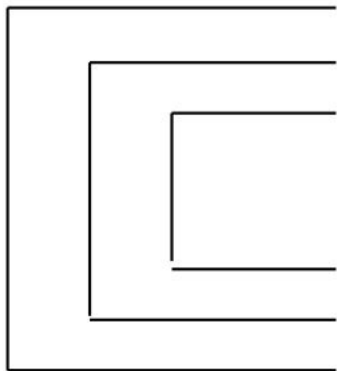
De la misma forma que los esquemas condicionales, los esquemas iterativos también pueden anidarse, las reglas son similares. A tener en cuenta:

- La estructura interna debe estar incluida dentro de la externa.
- No puede haber solapamiento.

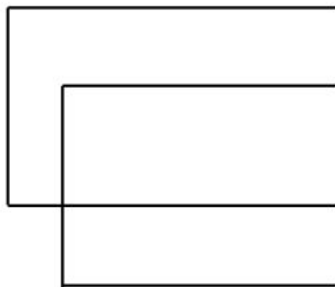
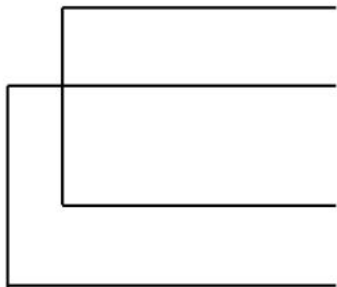


Ejemplo

Esquemas iterativos anidados



CORRECTOS



INCORRECTOS



Elementos auxiliares: **contador**



Ejemplo

Un **contador** es una variable cuyo valor se incrementa en una cantidad fija, positiva o negativa, y generalmente asociado a un bucle.

Se utiliza para contabilizar el número de veces que es necesario repetir una acción o para contar las repeticiones de un suceso particular.

Toma un valor inicial (0 normalmente) y cuando se realiza el suceso a contar incrementa su valor (1 normalmente).



Elementos auxiliares: **acumulador**



Ejemplo

Un **acumulador** es una variable cuyo valor se incrementa sucesivas veces en cantidades variables.

Se utiliza en aquellos casos en que se desea obtener el total acumulado de un conjunto de cantidades, siendo preciso inicializarlo con el valor 0.

También se usa en las situaciones en que hay que obtener un total como producto de distintas cantidades, debiéndose inicializar con el valor 1.



Elementos auxiliares: **interruptor** (switch)

Un **interruptor** es una variable que puede tomar dos valores exclusivamente (verdad o falso, 1 ó 0, etc...). Se utiliza para:

- Recordar en un determinado punto de un programa la ocurrencia o no de un suceso anterior, para salir de un bucle o para decidir en un esquema alternativo que acción realizar.
- Para hacer que dos acciones diferentes se ejecuten alternativamente dentro de un bucle.



Ejemplo



Tipos estructurados: **secuencias**

- Hasta ahora hemos tratado con datos de cinco tipos distintos: enteros, flotantes, lógicos, cadenas y listas. Los tres primeros son tipos de **datos escalares**. Las cadenas y las listas son tipos de datos secuenciales.
- Un dato de tipo escalar es un elemento único, atómico.
- Un dato de tipo secuencial se compone de una sucesión de elementos y una cadena es una sucesión de caracteres. Los datos de tipo secuencial son **datos estructurados**.



Cadenas (strings)

- Una cadena es una sucesión de caracteres.
 - Se encierran entre comillas simples o dobles.
- Operadores: **+**, *****
- Funciones: **int()** **len()** **float()** **str()** **ord()**
chr()
- Escapes (****): **\n** **\r** **\b** **\t** **** **\"** **\'**
- Indexación:

```
>>> cadena = "Hola Mundo!"
>>> cadena[0]
'H'
>>> cadena[3]
'a'
_
```

```
>>> cadena[len(cadena) - 1]
'!'
>>> cadena[-1]
'!'
_
```



Más información



Cadenas (strings)

- Las cadenas en Python (str) son **inmutables**.

```
>>> cadena[0] = "h"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

```
>>> id(cadena)
139958573596848
>>> cadena = 'h' + cadena[1:]
>>> cadena
'holá Mundo!'
>>> id(cadena)
139958573597040
```

Subcadena desde la
posición 1 al final

El objeto almacenado en
la variable es otro



Recorrido de cadenas

```
>>> cadena = 'Hola mundo!!!'
```

```
>>> for c in cadena:  
...     print(c)  
...
```

```
H  
o  
l  
a  
  
m  
u  
n  
d  
o  
!  
!  
!
```

```
-
```

```
>>> for i in range(len(cadena)):  
...     print(cadena[i])  
...
```

```
H  
o  
l  
a  
  
m  
u  
n  
d  
o  
!  
!  
!
```

```
-
```

```
>>> for i in range(len(cadena)):  
...     print(cadena[len(cadena)-i-1])  
...
```

```
!  
!  
!  
o  
d  
n  
u  
m  
  
a  
l  
o  
H
```

```
-
```



Cadenas: el operador de corte (**slicing**)

```
>>> cadena = "0123456789ABCDEF"
>>> cadena[2:10]
'23456789'
>>> cadena[0:5]
'01234'
>>> cadena[:5]
'01234'
>>> cadena[10:15]
'ABCDE'
>>> cadena[10:]
'ABCDEF'
>>> cadena[:]
'0123456789ABCDEF'

>>> cadena[2:10:2]
'2468'
>>> cadena[::-1]
'FEDCBA9876543210'
>>> cadena[::-2]
'FDB97531'
```



Más información



Cadenas: métodos

<code>count()</code>	<code>isalpha()</code>	<code>encode()</code>	<code>split()</code>
<code>find()</code>	<code>capitalize()</code>	<code>center()</code>	<code>splitlines()</code>
<code>index()</code>	<code>title()</code>	<code>ljust()</code>	<code>join()</code>
<code>startswith()</code>	<code>upper()</code>	<code>rjust()</code>	<code>format()</code>
<code>endswith()</code>	<code>isupper()</code>	<code>strip()</code>	<code>translate()</code>
<code>isdigit()</code>	<code>lower()</code>	<code>lstrip()</code>	
<code>isnumeric()</code>	<code>islower()</code>	<code>rstrip()</code>	
<code>isdecimal()</code>	<code>swapcase()</code>	<code>replace()</code>	

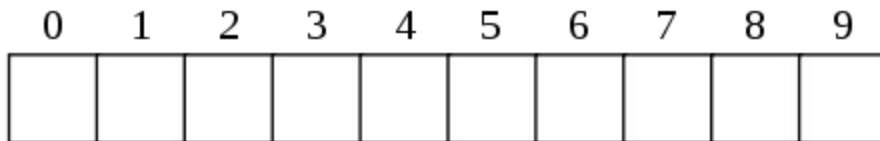


Más información



Arrays

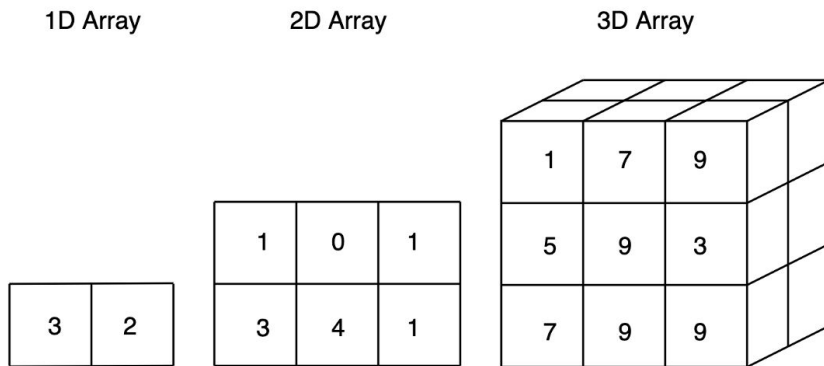
- Un **array** es un tipo de dato capaz de almacenar múltiples valores del **mismo tipo** y de manera contigua a los que accedemos mediante uno o más **índices**.
- Cuando solo tiene una dimensión se llama **vector**.





Arrays multidimensionales

- Un array multidimensional utiliza varios índices para localizar cada elemento.
- En el fondo podemos considerarlo como un array que, a su vez, contiene otros arrays.





Arrays en Python

- Python no tiene soporte integrado (*builtin*) para arrays, pero en su lugar se pueden usar listas de Python.
- Las listas son poco eficientes cuando hay muchos datos, si quisiéramos usar arrays, propiamente dichos, tendríamos que usar librerías como [numpy](#) o el módulo [array](#) de la librería estándar.
- La versatilidad de una lista penaliza a medida que su tamaño va creciendo pero sólo cuando se trata de listas muy grandes.



Creación de una lista

```
lista1 = []
```

```
lista2 = [1, 2, 3, 4, 5]
```

```
lista3 = [0] * 100
```

```
lista4 = list()
```

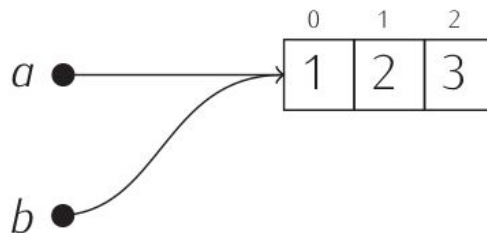
```
lista5 = [n for n in range(1, 11)]
```



Almacenamiento de listas en memoria

- Python almacena las listas del mismo modo que las cadenas: mediante referencias (punteros) a la secuencia de elementos.
- La asignación a una variable del contenido de otra variable que almacena una lista supone la copia de su referencia, así que ambas acaban apuntando a la misma zona de memoria.

```
a = [1, 2, 5]  
b = a  
b[2] = 3
```





Cosas que ya sabemos sobre las listas

Python proporciona operadores y funciones similares para trabajar con tipos de datos similares. Las cadenas y las listas tienen algo en común: ambas son secuencias de datos, así pues, muchos de los operadores y funciones que trabajan sobre cadenas también lo hacen sobre listas.

- Operadores: `+`, `*`, `in`, indexación, *slices*, relacionales, `is`
- Funciones: `len()` `str()`



Modificación y manipulación de listas

- Las listas, a diferencia de las cadenas, son **mutables**.
 - ¡Cuidado con las asignaciones de variables!
- **Modificamos** un elemento accediendo a él con su **índice**.
- **Añadimos** con `append()` `insert()` `extend()`
- **Borramos** con `remove()` `pop()` `clear()` `del`
- Ordenamos con `sort()` `reverse()` `sorted`
- Más métodos: `count()` `index()` `copy()`



Más información

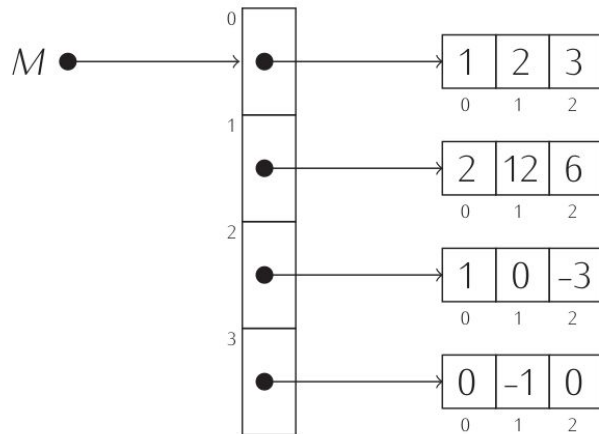


Matrices con listas

Las matrices son disposiciones bidimensionales de valores.
En Python podemos implementarlas con una lista de listas.

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 12 & 6 \\ 1 & 0 & -3 \\ 0 & -1 & 0 \end{pmatrix}$$

```
>>> M = [[1, 2, 3],  
          [2, 12, 3],  
          [1, 0, -3],  
          [0, -1, 0]]  
>>> M[0][0]  
1  
>>> M[2][2]  
-3  
>>> M[3]  
[0, -1, 0]  
>>>
```





Operaciones con matrices

- Creación
 - `m = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]`
 - `m = [[0] * 3, [0] * 3, [0] * 3]`
 - `m = []`
 `for _ in range(3):`
 `m.append([0] * 3)`
 - `m = [[0] * 3 for _ in range(3)]`
 - `m = [[0] * [COLUMNAS] for _ in range(FILAS)]`



Funciones

Una función es un trozo de código que realiza una tarea muy concreta y que se puede incluir en cualquier programa cuando hace falta resolver esa tarea. Opcionalmente, las funciones aceptan una entrada (parámetros de entrada) y devuelven una salida.

```
def nombre_función(param1, param2, ...):  
    instrucciones  
  
    return valor_devuelto
```

Funciones (en **Python**)

```
combinatorio = factorial(n) // (factorial(m) * factorial(n - m))
```

```
def factorial(x):  
    f = 1  
    for i in range(x, 0, -1):  
        f = f * i  
    return f
```

parámetro actual

parámetro formal

devolución



Más información

Funciones (en Java)

```
resultado = factorial(n) / (factorial(m) * factorial(n - m));
```

```
public static long factorial(long num) {  
    long f = 1;  
  
    for (int i = 2; i <= num; i++) {  
        f *= i;  
    }  
    return f;  
}
```

parámetro actual

parámetro formal

tipo devuelto

devolución

Funciones (en **Python** con anotaciones)

```
combinatorio = factorial(n) // (factorial(m) * factorial(n - m))
```

```
def factorial(x: int) -> int:
```

```
    f: int = 1
```

```
    for i in range(x, 0, -1):
```

```
        f = f * i
```

```
    return f
```

parámetro actual

parámetro formal

devolución



Más información



Tipado dinámico / estático

- Habrás observado en la diapositiva anterior que en Java es necesario especificar los tipos de datos de los parámetros de la función, de las variables y del valor devuelto por la función, mientras que en Python no hay que hacerlo.
- Java usa [tipado estático](#) y Python [tipado dinámico](#).
 - Ambos tienen sus [ventajas e inconvenientes](#).
 - Python puede trabajar con tipos estáticos a través de [anotaciones](#).



Más información



Ámbito de las variables

El ámbito de una variable es el contexto donde la variable puede ser usada. Generalizando mucho podríamos hablar de variables:

- Locales.
 - Válidas para una función.
 - Los parámetros formales (por valor) son locales a su función.
- Globales.
 - Válidas en todo el programa.
 - En Python definidas en el programa principal (fuera de las funciones).



Ámbito de las variables en Python

Python distingue tres tipos de variables: las variables locales y dos tipos de variables libres (globales y no locales):

- **Locales:** pertenecen al ámbito de la función y pueden ser accesibles a niveles inferiores.
- **Libres:**
 - **Globales:** pertenecen al ámbito del programa principal.
 - **No locales:** pertenecen a un ámbito superior al de la función, pero no son globales.



Más información



Variables locales en Python

Si no se han declarado como globales o no locales, las variables **a las que se asigna valor en una función** se consideran variables locales, es decir, sólo existen en la propia función, incluso cuando en el programa exista una variable con el mismo nombre.



Más información

```
def factorial(x):  
    f = 1  
    for i in range(x, 0, -1):  
        f = f * i  
    return f
```



Variables libres en Python

- Si a una variable no se le asigna valor en una función, Python la considera **libre** y busca su valor en los niveles superiores de esa función, empezando por el inmediatamente superior y continuando hasta el programa principal.
- Si a la variable se le asigna valor en algún nivel intermedio la variable se considera **no local** y si se le asigna en el programa principal la variable se considera **global**.



Variables libres en Python

- Si a una variable que Python considera libre (porque no se le asigna valor en la función) tampoco se le asigna valor en niveles superiores, Python dará un mensaje de error.
- Si queremos asignar valor a una variable en una función, pero no queremos que Python la considere local, debemos declararla en la función como **global** o **nonlocal**.



Más información



Paso de parámetros por valor y referencia

- Paso por valor:
 - Se pasa a la función una copia del valor del parámetro real, cualquier modificación que se le haga al parámetro formal no tendrá ningún efecto fuera de la función.
- Paso por referencia (o por variable).
 - Se pasa una referencia a la variable que se pasa como parámetro real. Si el parámetro formal cambia dentro de la función también lo hará la variable que se pasó como parámetro real.



Más información

En Python no se hace esta distinción al crear o invocar a la función. El comportamiento depende del tipo de variable (mutable o inmutable) con la que estamos tratando.



Pasando parámetros a funciones

- Por posición.
 - La forma más básica. El 1er parámetro real se corresponde con el 1er parámetro formal, el 2º con el 2º y así sucesivamente.
- Por nombre.
 - Se llama a la función usando el nombre del parámetro formal con = y su valor. El orden no importa.
- Por defecto.
 - Se le asignan valores por defecto a los parámetros formales de la función por si no se le pasan al invocarla.
- Parámetros de longitud variable.
 - El número de parámetros que se le pasan a la función no está limitado.



Más información



Sentencia **return**

El uso de la sentencia **return** permite realizar dos cosas:

- Salir de la función y transferir la ejecución de vuelta a donde se realizó la llamada.
 - Una vez se llama a **return** se finaliza la ejecución de la función y se vuelve al punto donde fue invocada.
- Devolver uno o varios valores, fruto de la ejecución de la función.
 - En caso de devolver más de un valor, deben estar separados por comas.



Más información



Recursividad

Una **función recursiva** es aquella que está definida en función de sí misma, por lo que se llama repetidamente a sí misma hasta llegar a un punto de salida. Tiene dos secciones de código claramente divididas:

- La sección en la que la función se llama a sí misma (caso general).
- Una condición en la que la función termina sin volver a llamarse (caso base o trivial).



Más información