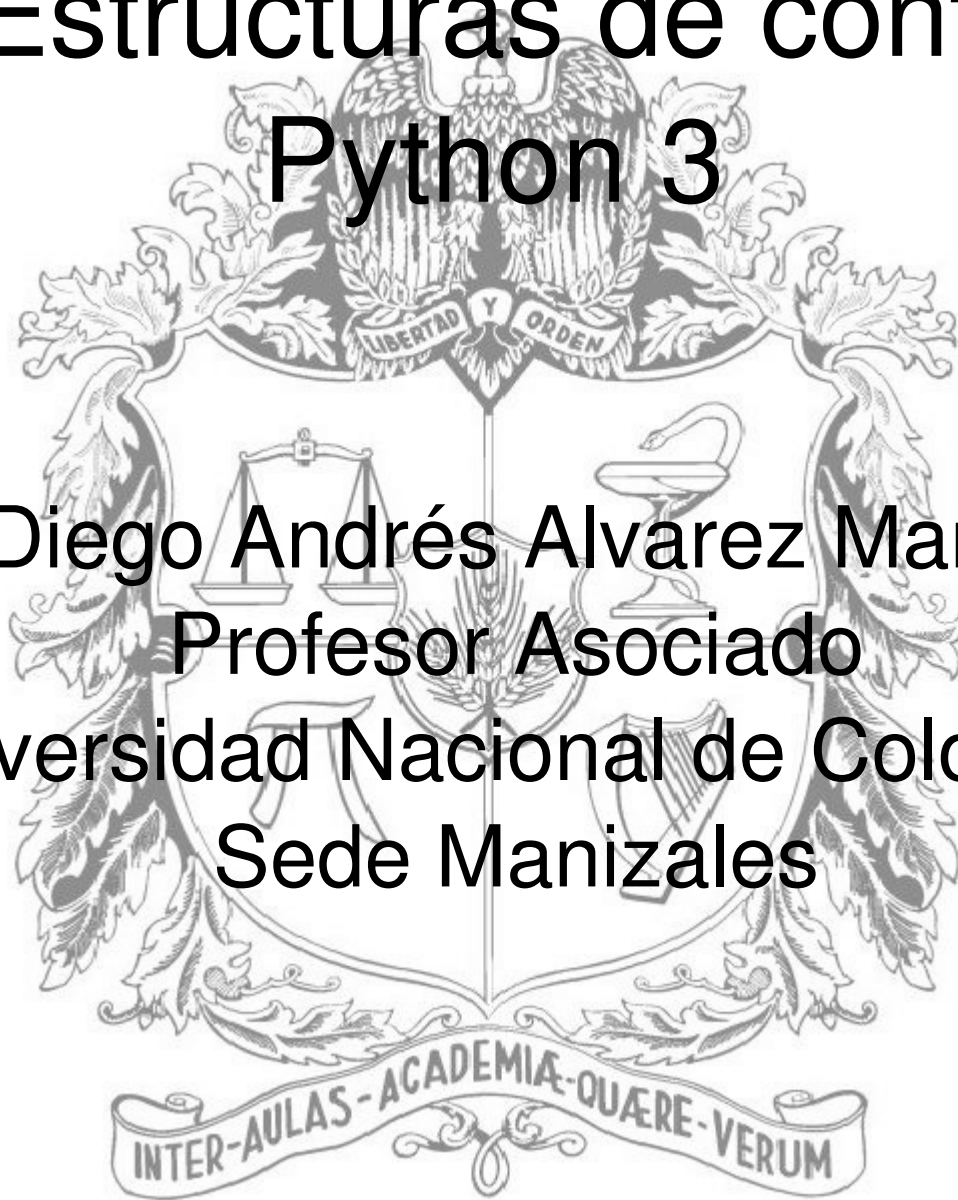


05 – Estructuras de control en Python 3

Diego Andrés Álvarez Marín
Profesor Asociado
Universidad Nacional de Colombia
Sede Manizales

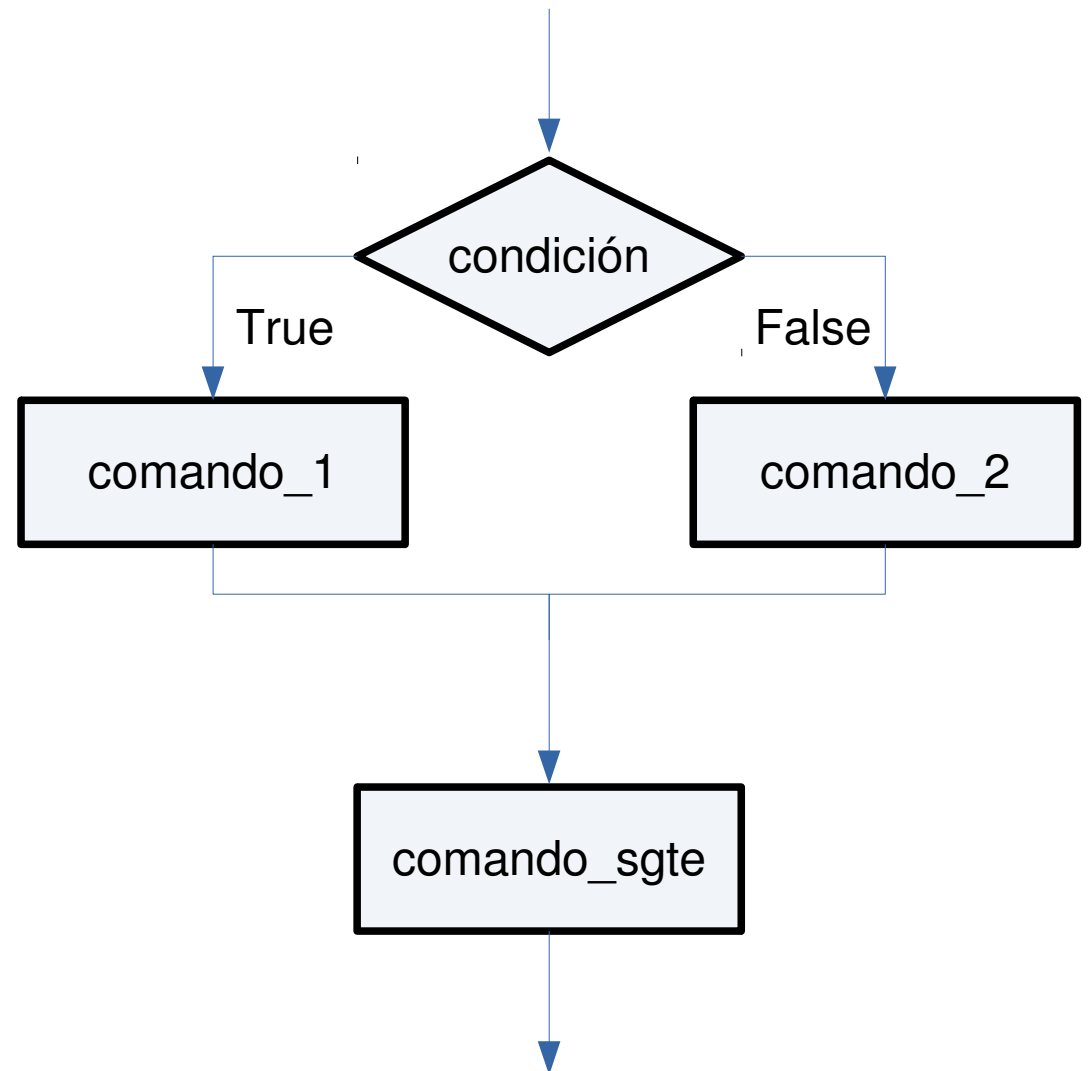


Estructuras de control

- Hay varios tipos de estructuras de control:
 - Estructuras condicionales o de selección: if
 - Estructuras iterativas o de repetición: for, while
 - Estructuras de emisión y captura de excepciones

if else

```
if condición:  
    comando_1  
else:  
    comando_2  
  
comando_sgte
```

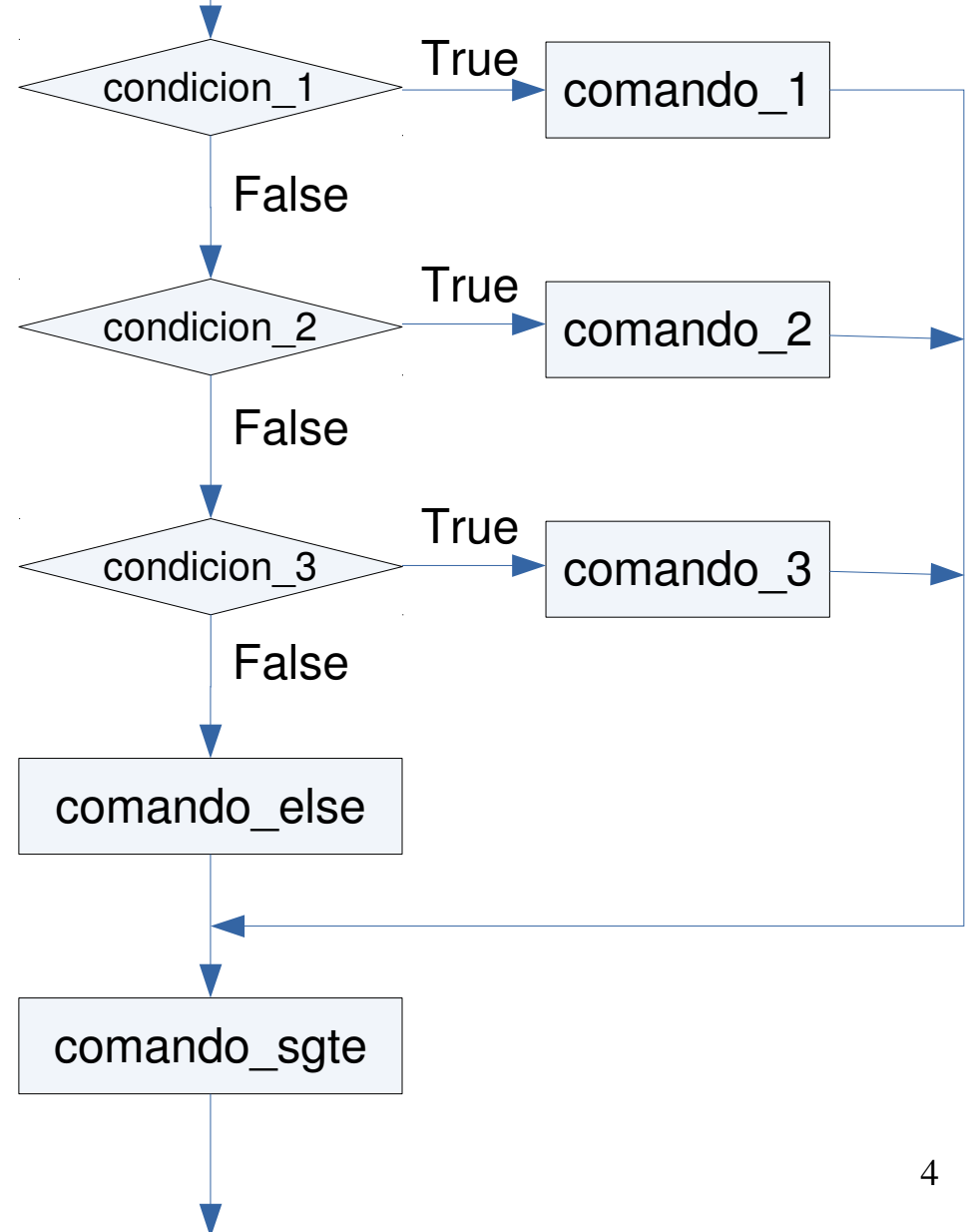


```
if condicion_1:  
    comando_1  
elif condicion_2:  
    comando_2  
...  
elif condicion_N:  
    comando_N  
else:  
    comando_else  
  
comando_sgte
```

Pueden haber una o más partes **elif**. La parte **else** es opcional.

En Python no existe la instrucción **switch-case** de otros lenguajes
http://en.wikipedia.org/wiki/Switch_statement

if – elif – else



if - elif - else

```
if x == 10:
    print('x vale 10')

if x == 10:
    print('x vale 10')
else:
    print('x no es igual a 10')

if x == 1:
    print('x vale 1')
elif x == 2:
    print('x vale 2')
elif x == 3:
    print('x vale 3')
else:
    print('x tiene otro valor')
```

```
1 a=3; b=2; c=4
2
3 ▼ if a < b:
4 ▼     if a < c:
5         r = a
6 ▼     else:
7         r = c
8 ▼ else:
9 ▼     if b < c:
10         r = b
11 ▼     else:
12         r = c
13
14 print('El menor vale', r)
```

Observe
que los ifs
se pueden
anidar

Line: 16 of 16 Col: 1 LINE INS
daalvarez@eredron:~ > python3 05_if_else.py
El menor vale 2

```

1  x = int(input("Entre un número: "))
2
3  ▼ if x<0:
4      print("El número es negativo")
5
6      # Aquí cambio el número a cero
7      x = 0
8  ▼ elif x == 0:
9      print("El número es cero")
10 ▼ elif 0 < x <= 10:
11     print("El número está en el intervalo (0,10]")
12 ▼ else:
13     print("El número es mayor que 10")

```

Observe que se ejecutó la línea 7 y que la línea 8 y la 9 no se ejecutaron

Line: 15 of 15 Col: 10 LINE INS

```

daalvarez@eredron:~ > python3 04_if.py
Entre un número: -100
El número es negativo
daalvarez@eredron:~ > python3 04_if.py
Entre un número: 0
El número es cero
daalvarez@eredron:~ > python3 04_if.py
Entre un número: 5
El número está en el intervalo (0,10]
daalvarez@eredron:~ > python3 04_if.py
Entre un número: 15
El número es mayor que 10
daalvarez@eredron:~ >

```

```
if catName == 'Fuzzball':  
    print('Your cat is fuzzy.')  
else:  
    if catName == 'Spots':  
        print('Your cat is spotted.')  
    else:  
        if catName == 'FattyKitty':  
            print('Your cat is fat.')  
        else:  
            if catName == 'Puff':  
                print('Your cat is puffy.')  
            else:  
                print('Your cat is neither fuzzy nor spotted nor fat nor puffy.')
```

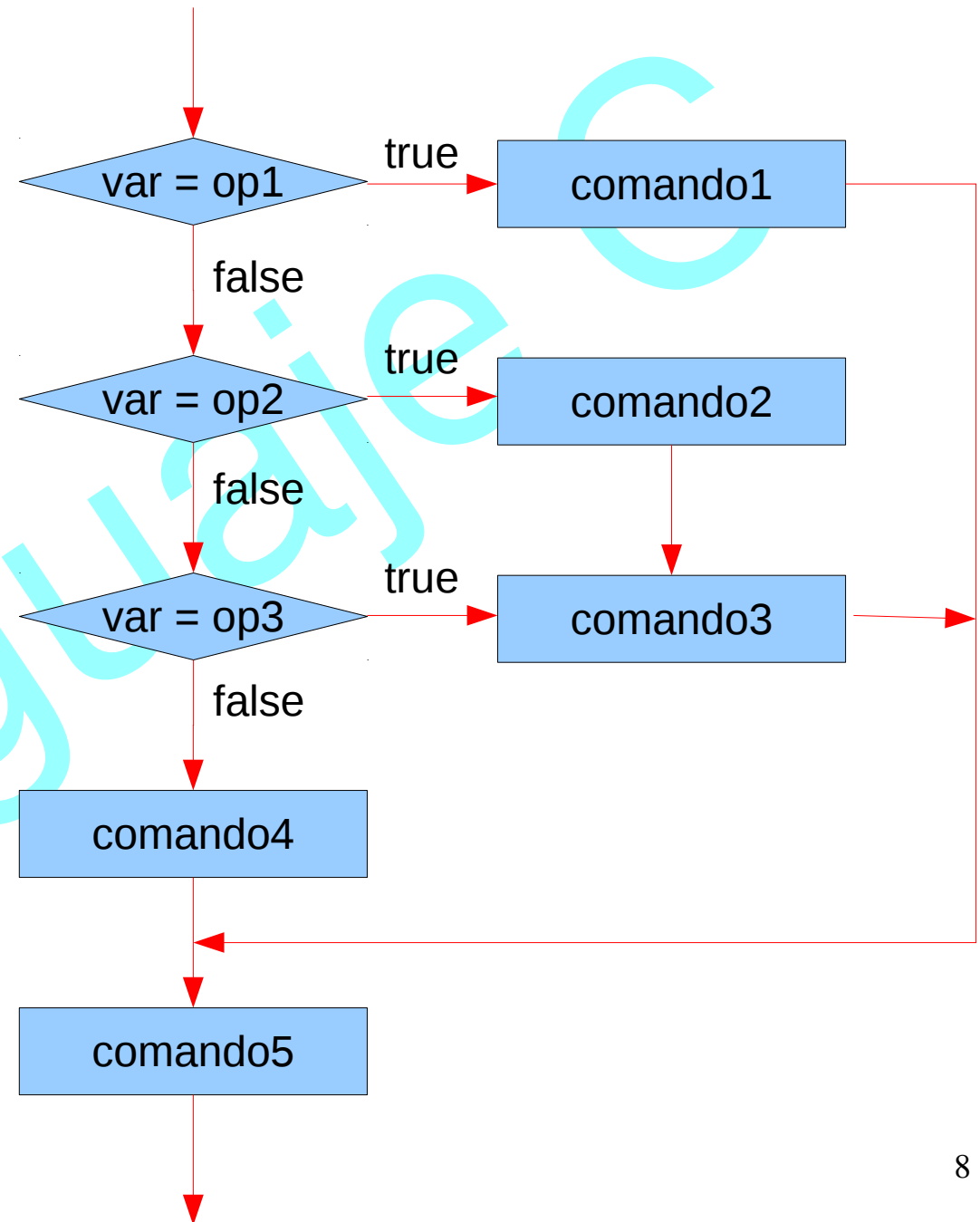
Estos ifs anidados son equivalentes a los mostrados en el cuadro inferior; sin embargo en el cuadro inferior es mucho más fácil identificar la idea del algoritmo.

Typing all those spaces means you have more chances of making a mistake with the indentation. So Python has the **elif** keyword. Using **elif**, the above code looks like this:

```
if catName == 'Fuzzball':  
    print('Your cat is fuzzy.')  
elif catName == 'Spots':  
    print('Your cat is spotted.')  
elif catName == 'FattyKitty':  
    print('Your cat is fat.')  
elif catName == 'Puff':  
    print('Your cat is puffy.')  
else:  
    print('Your cat is neither fuzzy nor spotted nor fat nor puffy.')
```

switch case default

```
switch(var)
{
case op1:
    comando1;
    break;
case op2:
    comando2;
case op3:
    comando3;
    break;
default:
    comando4;
}
comando5;
```

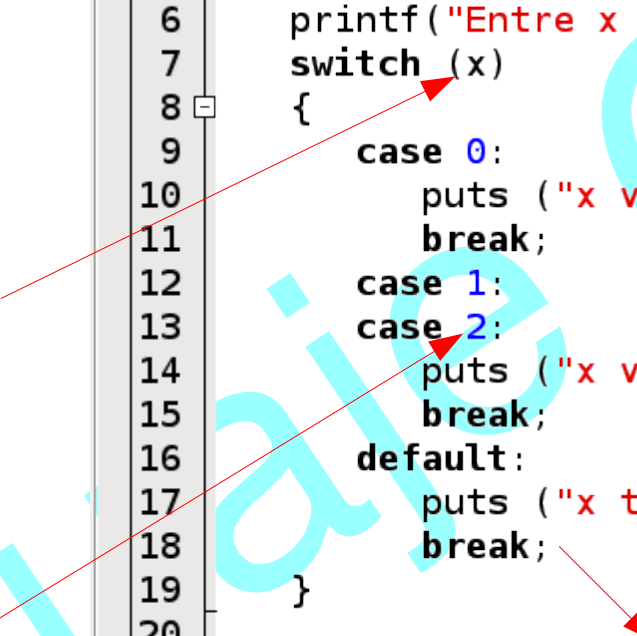


switch case default

Las expresiones comparadas deben ser de tipo entero, carácter o enum.

Las expresiones contra las que se comparan deben ser constantes enteras o tipo carácter o los valores que puede tomar un enum.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x;
6      printf("Entre x = "); scanf("%d", &x);
7      switch (x)
8      {
9          case 0:
10             puts ("x vale 0");
11             break;
12          case 1:
13          case 2:
14             puts ("x vale 1 o 2");
15             break;
16          default:
17             puts ("x tiene otro valor");
18             break;
19      }
20
21      return 0;
22 }
```



Este último break es innecesario

Line: 24 Col: 15 INS NORM file:///home/diego/programas/04

```
diego@earendil:~/programas$ ./04_switch_case_default
Entre x = 0
x vale 0
diego@earendil:~/programas$ ./04_switch_case_default
Entre x = 1
x vale 1 o 2
diego@earendil:~/programas$ ./04_switch_case_default
Entre x = 2
x vale 1 o 2
diego@earendil:~/programas$ ./04_switch_case_default
Entre x = 3
x tiene otro valor
diego@earendil:~/programas$
```

GNU C extension

switch case default

Se puede especificar un rango en la etiqueta, algo así como:

case bajo ... alto:

Cuidado con los espacios:

bajo ... alto	correcto
bajo...alto	incorrecto

GNU C extension quiere decir que esta es una característica soportada por el compilador gcc, pero que no está definida en el estándar ANSI C

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char c;
6     printf("Entre una letra = "); scanf("%c", &c);
7     switch (c)
8     {
9         case 'A' ... 'Z':
10             printf("%c es una letra mayúscula\n", c);
11             break;
12         case 'a' ... 'z':
13             printf("%c es una letra minúscula\n", c);
14             break;
15         case '0' ... '9':
16             printf("%c es un número\n", c);
17             break;
18         default:
19             printf("%c es un espacio o un símbolo\n", c);
20             break;
21     }
22
23     return 0;
24 }
```

Este último break es innecesario

Line: 30 Col: 35 INS NORM file:///home/diego/programas/04_switch_case_de

```
diego@earendil:~/programas$ ./04_switch_case_default
Entre una letra = R
R es una letra mayúscula
diego@earendil:~/programas$ ./04_switch_case_default
Entre una letra = r
r es una letra minúscula
diego@earendil:~/programas$ ./04_switch_case_default
Entre una letra = 5
5 es un número
diego@earendil:~/programas$ ./04_switch_case_default
Entre una letra = (
( es un espacio o un símbolo
diego@earendil:~/programas$
```

Expresiones condicionales

expresion_1 if condicion else expresion_2

```
>>> x = 20
>>> y = 10 if x>10 else -10
>>> y
10
>>> y = 200 + 10 if x==0 else -10
>>> y
-10
>>> y = 200 + (10 if x==0 else -10)
>>> y
190
>>> y = (200 + 10) if x==0 else -10
>>> y
-10
```

Es preferible
colocar
paréntesis
para evitar
confusiones

Expresiones condicionales

```
>>> x = 0
>>> print("{0} archivo{1}".format((x if x!=0 else "no hay"),
                                   ("s" if x!=1 else "")))

no hay archivos
>>> x = 1
>>> print("{0} archivo{1}".format((x if x!=0 else "no hay"),
                                   ("s" if x!=1 else "")))

1 archivo
>>> x = 10
>>> print("{0} archivo{1}".format((x if x!=0 else "no hay"),
                                   ("s" if x!=1 else "")))

10 archivos
>>> |
```

Evaluación con cortocircuitos

```
>>> a = 0
>>> 1/a
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    1/a
ZeroDivisionError: division by zero
>>> if a == 0 or 1/a > 1: print('Se ejecutó el if')

Se ejecutó el if
>>> if 1/a > 1 or a == 0: print('Se ejecutó el if')

Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    if 1/a > 1 or a == 0: print('Se ejecutó el if')
ZeroDivisionError: division by zero
>>> if a != 0 and 1/a > 1: print('Se ejecutó el if')

>>> |
```

Cuando el primer término de un **or** es **True**, Python devuelve **True** y no evalúa el segundo término.

Cuando el primer término de un **and** es **False**, Python devuelve **False** y no evalúa el segundo término.

Ciclo while-else

La parte del **else** es opcional.

El **bloque_1** se ejecuta mientras la condición sea verdadera.

```
while condición:  
    bloque_1  
else:  
    bloque_2  
bloque_3
```

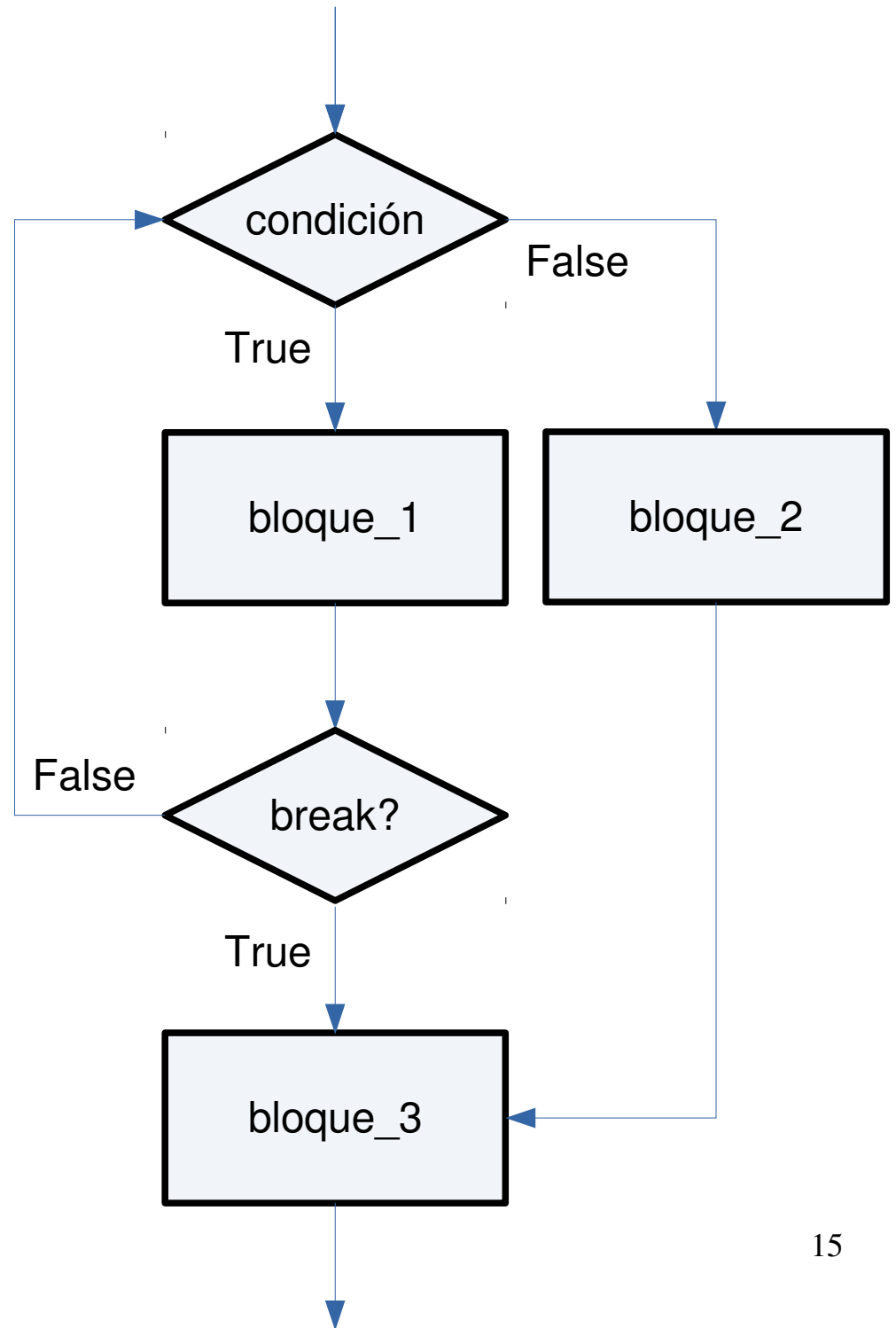
Si **condición** es o se vuelve falsa, el ciclo termina normalmente y si la parte del **else** existe, el **bloque_2** se ejecuta.

Si el ciclo no termina normalmente (se salió con un **break**, o con un **return**, o se lanzó una excepción), la parte del **else**, es decir el **bloque_2**, no se ejecuta.

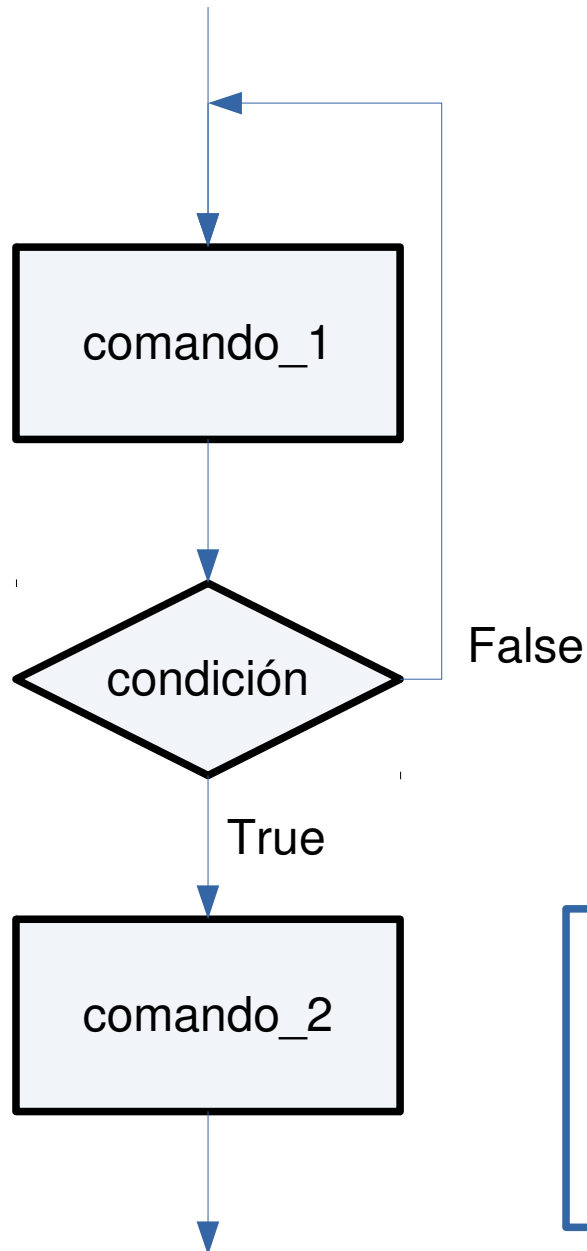
El ciclo mientras: while-else

```
while condición:  
    bloque_1  
else:  
    bloque_2  
    bloque_3
```

El ciclo while se ejecuta mientras la condición sea verdadera. Si la condición es falsa, o aparece un break, se sale del ciclo.



El ciclo “repita hasta que”

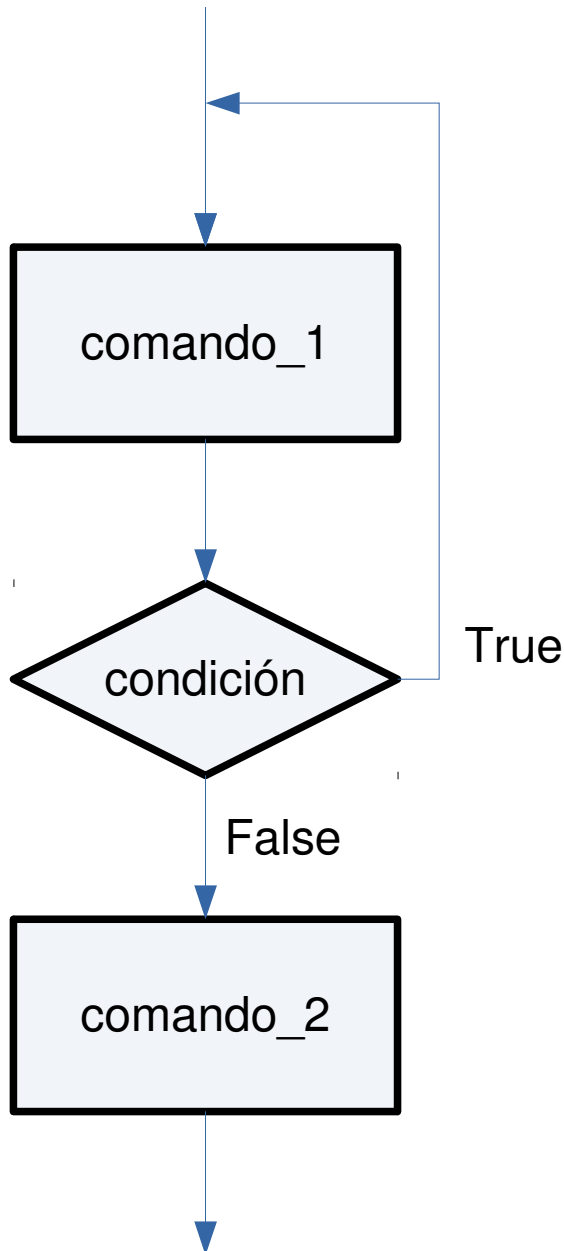


```
while True:  
    comando_1  
    if condición:  
        break
```

comando_2

El ciclo “repita hasta que” se ejecuta cuando la condición es FALSA y se sale cuando la condición se vuelve VERDADERA

El ciclo “haga mientras”



```
# FORMA 1
while True:
    comando_1
    if condición:
        pass
    else:
        break
```

comando_2

Forma recomendada:

```
# FORMA 2
while True:
    comando_1
    if not condición:
        break
comando_2
```

El ciclo se “haga mientras” se repite mientras la condición sea VERDADERA y se sale cuando la condición se vuelve FALSA

Implementando el ciclo

“repita hasta que” con un while

Uno puede leer este bucle infinito con la salida utilizando el if como un:
“repita hasta que (ch = 'Z')”

Se puede utilizar un **break** para salirse del ciclo

```
1 c = ord('A')
2 while True:
3     ch = chr(c)
4     print(ch, end=' ')
5     if ch == 'Z':
6         break
7     c += 1
8 print()
```

Line: 11 of 11 Col: 1 LINE INS

```
daa@heimdall ~ $ python3 05_repita_hasta_que.py
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
daa@heimdall ~ $
```

```
1 ▼ def encontrar_en_lista(lst, target):
2     index = 0
3 ▼   while index < len(lst):
4 ▼       if lst[index] == target:
5           break
6           index += 1
7 ▼   else:
8       index = None
9       return index
10
11 L = [1, 2, 'x', 4, 'yyy', [1, 4]]
12
13 print(encontrar_en_lista(L, 'yyy'))
14 print(encontrar_en_lista(L, 'zzz'))
```

< Line: 16 of 16 Col: 1

LINE INS

daa@heimdall ~ \$ python3 04_else_en_ciclo.py

4

None

daa@heimdall ~ \$ █

for – in

```
for variable in iterable:  
    comando_1  
else:  
    comando_2
```

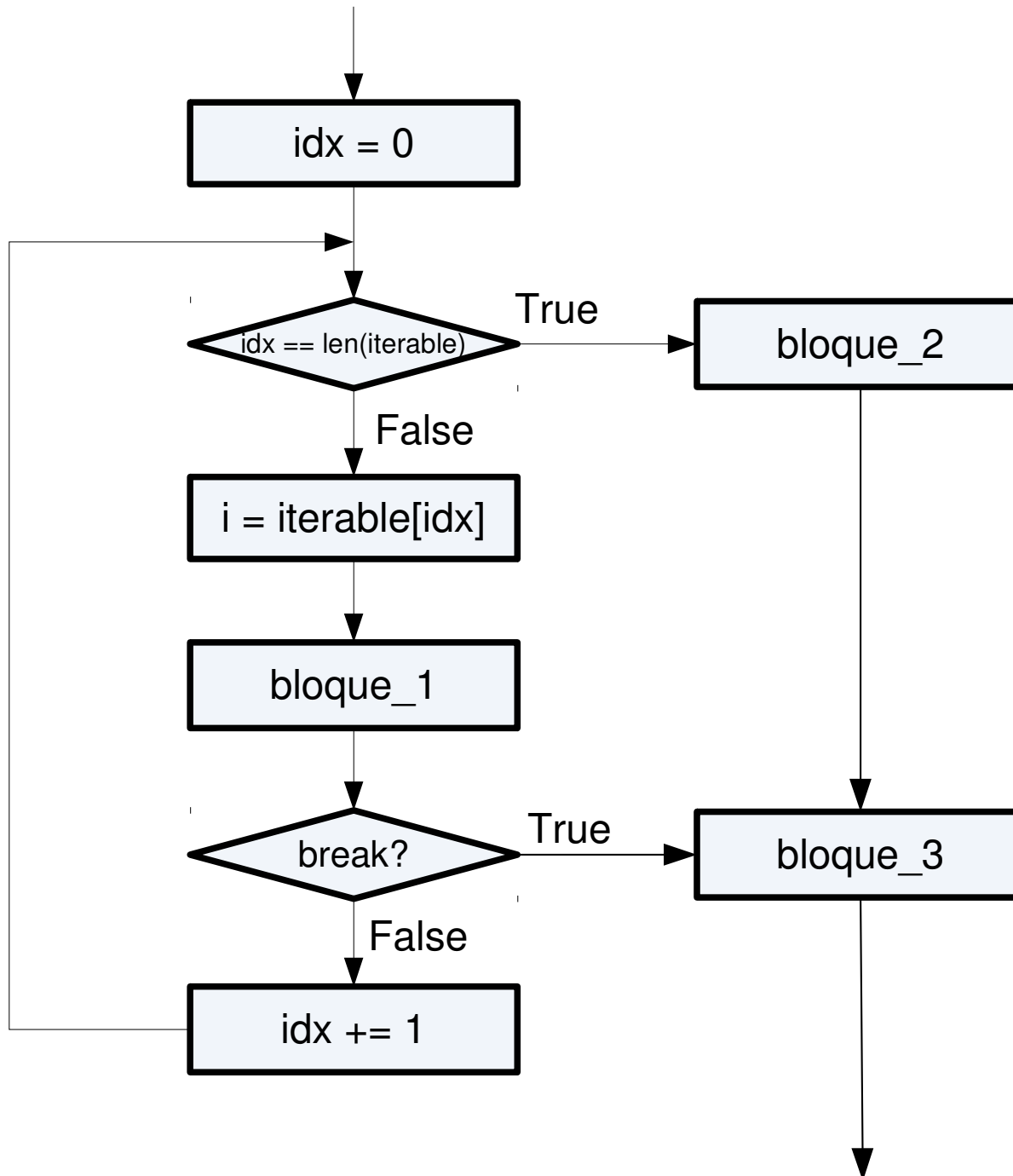
Un iterable es cualquier tipo de datos sobre el cual se puede iterar: cadenas (se itera de letra en letra), listas, tuplas, diccionarios, lo que retorna la función `range()`, etc.

Aquí “**in**” no funciona como el operador “**in**”.

El for-in de Python se asimila mucho al ciclo “for each” de otros lenguajes de programación

http://en.wikipedia.org/wiki/Foreach_loop

for – in - else



```
for i in iterable:  
    bloque_1  
else:  
    bloque_2  
    bloque_3
```

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

0
1
2
3
4
>>> s = 'María tenía una ovejita'.split()
>>> for i in range(len(s)):
    print(i, s[i])
```

```
0 María
1 tenía
2 una
3 ovejita
>>> for c in 'Buen día!': print(c)
```

```
B
u
e
n
```

```
d
í
a
!
```

```

1 ▼ for letra in "ABCDEF":
2     if letra in "AEIOU":
3         print(letra, "es una vocal")
4     else:
5         print(letra, "es una consonante")
6
7 print()
8
9 ▼ for pais in ["Perú", "Panamá", "Chile"]:
10     print(pais)
11
12 print()
13
14 # Usualmente el la lista se asigna
15 # primero a una variable:
16 paises = ["Perú", "Panamá", "Chile"]
17 ▼ for pais in paises:
18     if len(pais) == 6:
19         paises.append("Colombia")
20     print(pais)
21
22 print()
23
24 ▼ for pais in paises[:]:
25     if len(pais) == 6:
26         paises.append("México")
27     print(pais)

```

```

daa@heimdall ~ $ python3 -i 04_for.py
A es una vocal
B es una consonante
C es una consonante
D es una consonante
E es una vocal
F es una consonante

```

-i obliga a que se empiece el modo interactivo después de ejecutar el programa. Así pues, la consola de Python queda abierta después de finalizar la ejecución del programa.

Si en vez de "Colombia" hubiera sido "México", el ciclo hubiera sido infinito.

```

>>> paises
['Perú', 'Panamá', 'Chile', 'Colombia', 'México']
>>>

```

Aquí se creó una copia de la secuencia sobre la cual se está iterando. Si no se hubiera creado la copia, el bucle sería infinito.

Error en el uso de “del”

```
1 a = [1, 2, -1, -4, 5, -2]
2
3 ▼ for i in range(len(a)):
4     print(i, '->', a)
5 ▼     if a[i] < 0:
6         del a[i]
```

Line: 8 of 8 Col: 1 LINE INS

```
daa@heimdall ~ $ python3 03_del.py
0 -> [1, 2, -1, -4, 5, -2]
1 -> [1, 2, -1, -4, 5, -2]
2 -> [1, 2, -1, -4, 5, -2]
3 -> [1, 2, -4, 5, -2]
4 -> [1, 2, -4, 5, -2]
5 -> [1, 2, -4, 5]
Traceback (most recent call last):
  File "03_del.py", line 5, in <module>
    if a[i] < 0:
IndexError: list index out of range
daa@heimdall ~ $
```

Es una mala práctica de programación eliminar los elementos de una lista sobre la cual se está iterando, ya que se presta para confusiones y errores.

Índice de bucle **for-in**: ¡prohibido asignar!

Hemos aprendido que el bucle **for-in** utiliza una variable índice a la que se van asignando los diferentes valores del rango. En muchos ejemplos se utiliza la variable *i*, pero solo porque también en matemáticas los sumatorios y productorios suelen utilizar la letra *i* para indicar el nombre de su variable índice. Puedes usar cualquier nombre de variable válido.

Pero que el índice sea una variable cualquiera no te da libertad absoluta para hacer con ella lo que quieras. En un bucle **for-in**, las variables de índice solo deben usarse para consultar su valor, nunca para asignarles uno nuevo. Por ejemplo, este fragmento de programa es incorrecto:

```
1 for i in range(0, 5):  
2     i += 2
```

Y ahora que sabes que los bucles pueden anidarse, también has de tener mucho cuidado con sus índices. Un error frecuente entre primerizos de la programación es utilizar el mismo índice para dos bucles anidados. Por ejemplo, estos bucles anidados están mal:

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

En el fondo, este problema es una variante del anterior, pues de algún modo se está asignando nuevos valores a la variable *i* en el bucle interior, pero *i* es la variable del bucle exterior y asignarle cualquier valor está prohibido.

Recuerda: *nunca debes asignar un valor a un índice de bucle **for-in** ni usar la misma variable índice en bucles anidados.*

Algunos objetos “iterables”

```
>>> enumerate(['a', 'b', 'c'])
<enumerate object at 0x7fea3ef02ca8>
>>> list(_)
[(0, 'a'), (1, 'b'), (2, 'c')]
>>>
>>> campo = ['nombre', 'apellido', 'edad']
>>> dato = ['Ana', 'Vélez', 20]
>>> zip(campo, dato)
<zip object at 0x7fea3ee8c7c8>
>>> list(_)
[('nombre', 'Ana'), ('apellido', 'Vélez'), ('edad', 20)]
>>>
>>> range(10)
range(0, 10)
>>> list(_)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> reversed(_)
<list_reverseiterator object at 0x7fea4055ccf8>
>>> list(_)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>>
>>> L = [5, 2, 5, 1, -2, 5]
>>> sorted(L)
[-2, 1, 2, 5, 5, 5]
```

Un objeto **iterable** es uno que retorna elementos sucesivos de una secuencia dada cuando se itera sobre esta; el objeto iterable no crea una lista para ahorrar de este modo espacio de memoria.

El comando **list()** crea una lista a partir de un iterable. Se dice que el comando **for** es un “**iterador**”.

for en listas

```
1 # enumerate retorna el índice y el valor
2 ▼ for i, v in enumerate(['tic', 'tac', 'toe']):
3     print(i, v)
4
5 # zip() acopla dos secuencias al mismo tiempo
6 questions = ['name', 'quest', 'favorite color']
7 answers = ['lancelot', 'the holy grail', 'blue']
8 ▼ for q, a in zip(questions, answers):
9     print('What is your {0}? It is {1}.'.format(q, a))
10
11 # reversed() permite iterar sobre una secuencia en orden inverso
12 ▼ for i in reversed(range(1, 10, 2)):
13     print(i)
14
15 # sorted() permite iterar sobre una secuencia en forma ordenada,
16 # sin cambiar los contenidos de la lista
17 basket = ['aa', 'cc', 'bb', 'rr', 'aa']
18 ▼ for f in sorted(basket):
19     print(f)
```

Line: 21 of 40 Col: 43 LINE INS
daalvarez@eredron:~ > python3 04_for_en_listas.py

```
0 tic
1 tac
2 toe
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

```
9
7
5
3
1
aa
aa
bb
cc
rr
```

Alterando listas dentro de un ciclo for

```
21 # antes de alterar una lista con un for haga una copia de esta,
22 # ya que el for no hace una copia de esta:
23 print('Con copia = ')
24 palabras = ['gato', 'ventana', 'arrojar']
25 ▼ for w in palabras[:]: #[:] hace la copia
26 ▼     if len(w) > 6:
27         palabras.insert(0, w)
28
29 print(palabras)
30
31 print('Sin copia = ')
32 palabras = ['gato', 'ventana', 'arrojar']
33 ▼ for w in palabras:
34 ▼     if len(w) > 6:
35         palabras.insert(0, w)
36 ▼     if len(palabras) > 10: # sin esta condición el ciclo hubiera sido infinito
37         break
38
39 print(palabras)
```

Line: 44 of 45 Col: 1

LINE INS

05_for_en_listas.py UTF-8

```
daa@heimdall ~ $ python3 05_for_en_listas.py
```

```
Con copia =
```

```
['arrojar', 'ventana', 'gato', 'ventana', 'arrojar']
```

```
Sin copia =
```

```
['ventana', 'ventana', 'ventana', 'ventana', 'ventana', 'ventana', 'ventana', 'ventana', 'ventana',  
, 'gato', 'ventana', 'arrojar']
```

```
daa@heimdall ~ $
```

```

>>> lineah = ' '
>>> for i in range(1,6):
        lineah += (' '*9) + str(i)

>>> print(lineah)
          1          2          3          4          5
>>> print('0123456789' * 6)
012345678901234567890123456789012345678901234567890123456789

```

Asignación múltiple en un ciclo for

```
>>> # Recuerde que:
>>> x,y = 10,15      # con tuplas
>>> x
10
>>> y
15
>>> x,y = (11,52)    # con tuplas
>>> x
11
>>> y
52
>>> (x,y) = (21,12)  # con tuplas
>>> x
21
>>> y
12
>>> x,y = [10,15]    # con listas
>>> x
10
>>> y
15
>>> [x,y] = [24,78]  # con listas
>>> x
24
>>> y
78
```

```
>>> L = [[1, 'x'], [2, 'y'], [3, 'z']]
>>> for i,j in L:
>>>     print(i,j)
```

```
1 x
2 y
3 z
>>> |
```

for en diccionarios

Aquí se utilizó `sorted()` para darle cierta estructura a la presentación de los datos del diccionario.

```
1 d = {'xxx': 123, 'yyy': 456, 'zzz': 789}
2
3 ▼ for k in d:
4     print(k)
5     ...
6 ▼ for k in d:
7     print(d[k])
8
9 d['rrr'] = -1
10 print(d)
11
12 ▼ for k, v in d.items():
13     print(k, v)
14
15 ▼ for k, v in sorted(d.items()):
16     print(k, v)
```

Line 17, Column 1

```
daalvarez@eredron ~ $ python3 11_for_dictionaries.py
```

```
xxx
yyy
zzz
123
456
789
{'xxx': 123, 'yyy': 456, 'zzz': 789, 'rrr': -1}
xxx 123
yyy 456
zzz 789
rrr -1
rrr -1
xxx 123
yyy 456
zzz 789
```

```
daalvarez@eredron ~ $ █
```

La función range()

`range(start, stop, step)` Returns an integer iterator. With one argument (*stop*), the iterator goes from 0 to *stop* - 1; with two arguments (*start*, *stop*) the iterator goes from *start* to *stop* - 1; with three arguments it goes from *start* to *stop* - 1 in steps of *step*.

```
range(5, 10)
5 through 9
```

```
range(0, 10, 3)
0, 3, 6, 9
```

```
range(-10, -100, -30)
-10, -40, -70
```

→ El “step” puede ser incluso negativo

```
>>> print(range(2,10))
range(2, 10)
>>> |
```

→ Esto es lo que pasa cuando se intenta imprimir un `range()`:

break

El **break** se utiliza para salirse de la iteración actual de un ciclo (**for**, **while**).

El **break** funciona igual que en lenguaje C: se sale inmediatamente del ciclo más anidado, sin verificar de nuevo la condición.

```
1 ▼ for n in range(10, 0, -1):
2     print(n, end=', ')
3 ▼     if n == 3:
4         print('Conteo abortado!!!')
5         break
6     print('FIN!')
```

Line: 8 of 8 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 05_break.py
10, 9, 8, 7, 6, 5, 4, 3, Conteo abortado!!!
FIN!
daalvarez@eredron:~ > █
```

break y else en un ciclo

```
>>> for n in range(2, 10):  
...     for x in range(2, n):  
...         if n % x == 0:  
...             print(n, 'equals', x, '*', n//x)  
...             break  
...     else:  
...         # loop fell through without finding a factor  
...         print(n, 'is a prime number')
```

```
...  
2 is a prime number  
3 is a prime number  
4 equals 2 * 2  
5 is a prime number  
6 equals 2 * 3  
7 is a prime number  
8 equals 2 * 4  
9 equals 3 * 3
```

Si se pone un **break** dentro de un conjunto de ciclos anidados, el **break** solo se sale del ciclo más interior, no del resto de los otros ciclos.

El **else** se ejecuta en un ciclo for después de iterar sobre toda la lista y en un ciclo **while** cuando la condición se vuelve falsa; no se ejecuta cuando se sale del ciclo con un **break**.

```

1 ▼ def encontrar_en_lista(lista, objetivo):
2 ▼     for index, x in enumerate(lista):
3 ▼         if x == objetivo:
4             break
5 ▼     else:
6         index = None
7     return index
8
9 ▼ def encontrar_en_lista2(lista, objetivo):
10     index = 0
11 ▼     while index < len(lista):
12 ▼         if lista[index] == objetivo:
13             break
14             index += 1
15 ▼     else:
16         index = None
17     return index
18
19 L = [1, 2, 'x', 4, 'yyy', [1, 4]]
20
21 print(encontrar_en_lista(L, 'yyy'))
22 print(encontrar_en_lista(L, 'zzz'))
23
24 print(encontrar_en_lista2(L, 'yyy'))
25 print(encontrar_en_lista2(L, 'zzz'))

```

Line: 27 of 27 Col: 1

LINE INS

daa@heimdall ~ \$ python3 05_else_en_ciclo.py

4

None

4

None

```

1  # Programa para adivinar un número secreto entre 1 y 20
2
3  import random
4
5  num_secreto = random.randint(1, 20)
6
7  ▼ for intentos in range(1,7):
8      # Leer el número desde el teclado
9      ▼ while True:
10         num = int(input("Entre un número entre 1 y 20 = "))
11         if 1 <= num <= 20: break
12
13     ▼ if num < num_secreto:
14         print('El número entrado es muy pequeño.')
15     ▼ elif num > num_secreto:
16         print('El número entrado es muy grande.')
17     ▼ else: # if num == num_secreto:
18         print('Adivinaste el número secreto en', intentos, 'intentos!')
19         break
20     ▼ else:
21         print('Perdiste. El número secreto era el', num_secreto)
22
23     print('*** FIN DEL JUEGO ***')

```

Line 25, Column 1

INSERT Soft Tabs: 4 ▼ UTF-8 ▼ Python ▼

```

El número entrado es muy pequeño.
Entre un número entre 1 y 20 = 18
El número entrado es muy grande.
Entre un número entre 1 y 20 = 14
El número entrado es muy pequeño.
Entre un número entre 1 y 20 = 16
Adivinaste el número secreto en 4 intentos!
*** FIN DEL JUEGO ***

```

continue

```
1 ▼ for n in range(10, 0, -1):
2     if n == 5: continue
3     print(n, end=', ')
4     print('CERO!!!')
```

Line: 6 of 6 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 05_continue2.py
10, 9, 8, 7, 6, 4, 3, 2, 1, CERO!!!
daalvarez@eredron:~ >
```

¿Y dónde quedó el 5?

continue usa en ciclos para saltarse el resto del bloque actual y salta de nuevo a la condición del ciclo (**for**, **while**). Desde este punto de vista, si es un ciclo “haga hasta que” hecho con un “**while True: ... if condicion: break**”, el **continue** regresa a evaluar el True del ciclo **while**.

Si se pone un **continue** dentro de un ciclo anidado, este solo afecta el ciclo más interior.

continue

```
1 ▼ for n in range(2,10):
2 ▼     if n%2 == 0:
3         print(n, "es un número par")
4         continue
5     print(n, "es un número impar")
6
```

Line: 9 of 9 Col: 7 LINE OVR

```
daa@heimdall ~ $ python3 04_continue.py
2 es un número par
3 es un número impar
4 es un número par
5 es un número impar
6 es un número par
7 es un número impar
8 es un número par
9 es un número impar
daa@heimdall ~ $
```

- El **continue** se usa en ciclos (**for**, **while**) para saltarse el resto del bloque actual y salta de nuevo a la condición del ciclo.
- Si se pone un **continue** dentro de un ciclo anidado, este solo afecta el ciclo interior.
- En un ciclo **while** obliga a que se ejecute de nuevo la condición y vuelve al principio del ciclo para la nueva iteración
- En un ciclo **for** obliga a que se realice la siguiente iteración.

Ciclos infinitos

Los bucles son muy útiles a la hora de confeccionar programas, pero también son peligrosos si no andas con cuidado: es posible que no finalicen nunca. Estudia este programa y verás qué queremos decir:

```
⚡ bucle_infinito.py
1 i = 0
2 while i < 10:
3     print(i)
```

La condición del bucle siempre se satisface: dentro del bucle nunca se modifica el valor de *i*, y si *i* no se modifica, jamás llegará a valer 10 o más. El ordenador empieza a mostrar el número 0 una y otra vez, sin finalizar nunca. Es lo que denominamos un *bucle sin fin* o *bucle infinito*.

Cuando se ejecuta un bucle sin fin, el ordenador se queda como «colgado» y nunca nos devuelve el control. Si estás ejecutando un programa desde la línea de órdenes Unix o una consola de Windows, puedes abortarlo pulsando **C-c**. Si la ejecución tiene lugar en Eclipse/Pydev puedes abortar la ejecución del programa pulsando en el cuadrado rojo que aparece en la barra superior de la consola.



Ctrl C

Ciclos anidados

- Se pueden utilizar los comandos **break** y **continue** para salir de ellos.
- Recuerde que cuando hay ciclos anidados, el comando **break** solo se sale del ciclo que contiene dicha palabra, no de los otros ciclos.

Una excepción a la regla de sangrado

Cada vez que una sentencia acaba con dos puntos (:), Python espera que la sentencia o sentencias que le siguen aparezcan con un mayor sangrado. Es la forma de marcar el inicio y el fin de una serie de sentencias que «dependen» de otra.

Hay una excepción: si solo hay *una* sentencia que «depende» de otra, puedes escribir ambas en la misma línea. Este programa:

```
1 a = int(input('Dame_un_entero_positivo:'))
2 while a < 0:
3     a = int(input('Te_he_dicho_positivo:'))
4 if a % 2 == 0:
5     print('El_número_es_par')
6 else:
7     print('El_número_es_impar')
```

y este otro:

```
1 a = int(input('Dame_un_entero_positivo:'))
2 while a < 0: a = int(input('Te_he_dicho_positivo:'))
3 if a % 2 == 0: print('El_número_es_par')
4 else: print('El_número_es_impar')
```

son equivalentes, aunque el primero resulta más legible.

Instrucciones en un solo reglón

```
1 L = [10, 'x', -123, [1, 2, 'r'], None, 30, (2,3)]
2
3 if L[3] == [1, 2, 'r']: print("L[3] contiene [1, 2, 'r']")
4
5 if L[4] == None: print("L[4] contiene None")
6 else: print("L[4] no contiene None")
7
8 for i in L: print(i)
```

Line: 10 of 10 Col: 1 LINE INS
daalvarez@eredron:~ > python3 04_sola_linea.py

L[3] contiene [1, 2, 'r']

L[4] contiene None

10

x

-123

[1, 2, 'r']

None

30

(2, 3)

daalvarez@eredron:~ > █

Recuerde que es mucho
más eficiente escribir:

`if L[4] is None:`

pass

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
... 
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
... 
```

Another place `pass` can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
...     pass  # Remember to implement this!
... 
```

List comprehensions

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
File "<stdin>", line 1, in ?
    [x, x**2 for x in range(6)]
        ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions

```
1 # El siguiente procedimiento:
2 squares = []
3 ▼ for x in range(10):
4     squares.append(x**2)
5 print(squares)
6
7 # Se puede escribir como:
8 squares = list(map(lambda x: x**2, range(10)))
9 print(squares)
10
11 # O como:
12 squares = [x**2 for x in range(10)] # list comprehension
13 print(squares)
14
15 # El siguiente procedimiento:
16 combinaciones = []
17 ▼ for x in [1,2,3]:
18     ▼ for y in [3,1,4]:
19         ▼ if x != y:
20             combinaciones.append((x, y))
21 print(combinaciones)
22
23 # Se puede escribir utilizando una "list comprehension":
24 combinaciones = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
25 print(combinaciones)
```

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
>>> |
```

Line: 31 of 34 Col: 1 LINE INS

daa@heimdall ~ \$ python3 04_comprehension.py

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

Nested list comprehensions

```
28 matriz = [  
29     [1, 2, 3, 4],  
30     [5, 6, 7, 8],  
31     [9, 10, 11, 12]  
32 ]  
33  
34 # El siguiente código transpone la matriz  
35 transpuesta = [[fila[i] for fila in matriz] for i in range(4)]  
36 print(transpuesta)  
37  
38 # El siguiente código es equivalente:  
39 transpuesta = []  
40 for i in range(4):  
41     transpuesta.append([fila[i] for fila in matriz])  
42 print(transpuesta)  
43  
44 # Y este también:  
45 transpuesta = []  
46 for i in range(4):  
47     fila_transpuesta = []  
48     for fila in matriz:  
49         fila_transpuesta.append(fila[i])  
50     transpuesta.append(fila_transpuesta)  
51 print(transpuesta)
```

Line: 52 of 52 Col: 1

LINE INS

```
daa@heimdall ~ $ python3 04_comprehension.py  
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]  
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]  
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```

1 # Para crear una matriz, primero se debe inicializar esta
2
3 # Crea una matriz de 4 filas y 6 columnas
4 Matriz = [[0 for c in range(6)] for f in range(4)]
5
6 ▼ for fila in Matriz:
7     print(fila)
8
9     print()
10    Matriz[0][0] = 1
11    Matriz[3][0] = 5
12    Matriz[2][4] = -8
13
14 ▼ for fila in Matriz:
15     print(fila)
16
17    Matriz[4][6] = 15

```

“nested list
comprehension”

Implementando matrices con listas de listas

Line: 19 of 19 Col: 1 LINE INS

daalvarez@eredron:~ > python3 04_listas_de_listas.py

```

[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]

```

```

[1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, -8, 0]
[5, 0, 0, 0, 0, 0]

```

Traceback (most recent call last):

File "04_listas_de_listas.py", line 17, in <module>

Matriz[4][6] = 15

IndexError: list index out of range

daalvarez@eredron:~ > █

Set and dict comprehensions

```
>>> # list comprehension
>>> [x for x in 'abracadabra' if x not in 'abc']
['r', 'd', 'r']
>>> # set comprehension
>>> {x for x in 'abracadabra' if x not in 'abc'}
{'r', 'd'}
>>> # dict comprehension
>>> {x : x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
>>> |
```


Errores de sintaxis vs. Excepciones

```
>>> 10 + 20)
File "<stdin>", line 1
    10 + 20)
        ^
```

```
SyntaxError: invalid syntax
```

```
>>> labc
File "<stdin>", line 1
    labc
        ^
```

```
SyntaxError: invalid syntax
```

```
>>> 10+20+30
File "<stdin>", line 1
    10+20+30
        ^
```

```
IndentationError: unexpected indent
```

```
>>> 1/'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: division by zero
```

```
>>> █
```

Los errores de sintaxis (**SyntaxError**) aparecen cuando no se respetó la gramática propia del lenguaje de programación. Se caracterizan por la flecha (^) que indica la posición del error.

Las excepciones aparecen cada vez que el intérprete no sabe como analizar o proceder ante una orden dada (así sea sintácticamente correcta).

En <https://docs.python.org/3/library/exceptions.html> puede encontrar la lista de posibles excepciones

Las excepciones

Sucedan por condiciones "inesperadas" en el programa, como por ejemplo, el disco estaba lleno, no había memoria disponible, etc. o porque se escribieron órdenes que aunque cumplen la gramática del lenguaje, no son correctas.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    10 * (1/0)
```

```
ZeroDivisionError: division by zero
```

```
>>> 3 + 2*x
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    3 + 2*x
```

```
NameError: name 'x' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    '2' + 2
```

```
TypeError: Can't convert 'int' object to str implicitly
```

```
>>> |
```

Aquí
ZeroDivisionError,
NameError y
TypeError son los
tipos de excepciones.

Lanzando excepciones genéricas

```
In [1]: x = 5
```

```
In [2]: if x > 5:
...:     raise Exception(f'x should not exceed 5. The value of x was: {x}')
...:
```

```
In [3]: x = 6
```

```
In [4]: if x > 5:
...:     raise Exception(f'x should not exceed 5. The value of x was: {x}')
...:
```

```
-----
Exception                                Traceback (most recent call last)
<ipython-input-4-27e710637659> in <module>()
      1 if x > 5:
----> 2     raise Exception(f'x should not exceed 5. The value of x was: {x}')
      3
```

```
Exception: x should not exceed 5. The value of x was: 6
```

Python

The `AssertionError` Exception

```
import sys
assert ('linux' in sys.platform), "This code runs on Linux only."
```

If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be `False` and the result would be the following:

Python

```
Traceback (most recent call last):
  File "<input>", line 2, in <module>
AssertionError: This code runs on Linux only.
```

In this example, throwing an `AssertionError` exception is the last thing that the program will do. The program will come to halt and will not continue. What if that is not what you want?

```
def linux_interaction():  
    assert ('linux' in sys.platform), "Function can only run on Linux systems."  
    print('Doing something.')
```



```
try:  
    linux_interaction()  
except AssertionError as error:  
    print(error)  
    print('The linux_interaction() function was not executed')
```

Running this function on a Windows machine outputs the following:

Shell

```
Function can only run on Linux systems.  
The linux_interaction() function was not executed
```

Manejo de excepciones

Se ejecuta bloque_try.

```
try:
    bloque_try
except exception_group1 as variable1:
    excepcion_1
...
except exception_groupN as variableN:
    excepcion_N
else:
    bloque_else
finally:
    bloque_finally
```

Si no ocurrieron excepciones, se termina el bloque_try y se ejecuta el bloque_else (el cual es opcional). Finalmente, se ejecuta el bloque_finally.

Si ocurrieron excepciones, se salta el resto del bloque_try y se ejecuta la excepción (excepcion_i) correspondiente.

Debe haber al menos un bloque except; los bloques else y finally son opcionales. La parte "as variable" es opcional.

```

1 ▼ while True:
2 ▼     try:
3         num = int(input('Entre un número entero entre 0 y 100 = '))
4         if 0 <= num <= 100: break
5 ▼     except ValueError:
6         print('Por favor entre un entero. Intente de nuevo ...')
7
8 print('El número entrado es el', num)

```

Line: 10 of 10 Col: 1

LINE INS

daa@heimdall ~ \$ python3 05_try_except_basico.py

Entre un número entero entre 0 y 100 = Hola

Por favor entre un entero. Intente de nuevo ...

Entre un número entero entre 0 y 100 =)(/!"#

Por favor entre un entero. Intente de nuevo ...

Entre un número entero entre 0 y 100 = 12.3

Por favor entre un entero. Intente de nuevo ...

Entre un número entero entre 0 y 100 = -200

Entre un número entero entre 0 y 100 = 300

Entre un número entero entre 0 y 100 = _____

► ENTER

Por favor entre un entero. Intente de nuevo ...

Entre un número entero entre 0 y 100 = _____

► ENTER

Por favor entre un entero. Intente de nuevo ...

Entre un número entero entre 0 y 100 = 23

El número entrado es el 23

daa@heimdall ~ \$ █

NOTA: en el bloque **try-except** solo debe ir el código que se sabe que puede fallar. **No se coloca sobre todo el programa.**


```
>>> 1 / 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
>>> from math import sqrt
>>> sqrt(-1)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: math domain error
```

NOTA: en el bloque **try-except** solo debe ir el código que se sabe que puede fallar. **No se coloca sobre todo el programa.**

Es posible usar varias cláusulas **except**, una por cada tipo de error a tratar:

```
segundo_grado.py
1 from math import sqrt
2
3 a = float(input('Valor de a: '))
4 b = float(input('Valor de b: '))
5 c = float(input('Valor de c: '))
6
7 try:
8     x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
9     x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
10    if x1 == x2:
11        print('Solución: x={0:.3f}'.format(x1))
12    else:
13        print('Soluciones: x1={0:.3f} y x2={1:.3f}'.format(x1, x2))
14 except ZeroDivisionError:
15     if b != 0:
16         print('La ecuación no tiene solución.')
17     else:
18         print('La ecuación tiene infinitas soluciones.')
19 except ValueError:
20     print('No hay soluciones reales')
```

En el libro de Marzal et. al encuentro dos errores:

1. Línea 10: No se pueden comparar floats con **==** sino que se deben comparar con **math.isclose()**
2. Línea 10 a 13: estas líneas deben estar fuera del **try-except**, ya que ellas no generarán errores. Estas líneas deben ponerse en la parte **else** del **try-except**, o simplemente fuera y a continuación del **try-except**.

Funcionamiento del try-except

- First, the **try** clause (the statement(s) between the **try** and **except** keywords) is executed.
- If no exception occurs, the **except** clause is skipped and execution of the **try** statement is finished.
- If an exception occurs during execution of the **try** clause, the rest of the clause is skipped. Then if its type matches the exception named after the **except** keyword, the except clause is executed, and then execution continues after the **try** statement.
- If an exception occurs which does not match the exception named in the **except** clause, it is passed on to outer **try** statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above

En caso de bloques **try** anidados, si una excepción no tiene un correspondiente bloque **except**, entonces, la excepción se pasa al **try** más exterior. Si no se encuentra su **except** asociado, entonces se dice que "no se pudo manejar la excepción" y la ejecución del programa termina después del bloque **try**.

Cuando una excepción dada (excepcion_i) corresponde a varias excepciones, se pueden poner todas esas excepciones en una tupla, de forma similar a:

```
except (RuntimeError, TypeError, NameError):  
    pass
```

El **pass** en este caso se utiliza, para no hacer nada.

El bloque **else** se ejecuta cuando el bloque **try** ha finalizado correctamente; no se ejecuta si ocurre una excepción. Es conveniente utilizar el bloque **else** ya que esto es mejor que agregar código adicional al bloque **try** ya que evita capturar accidentalmente una excepción que no se quería.

```

1 import sys
2
3 try:
4     f = open('miarchivo.txt')
5     den = int(f.readline())           # lee el primer reglón de miarchivo.txt
6     x = 1/den
7 except OSError as err:
8     print("Error OS: {0}".format(err))
9 except ValueError:
10    print("No pude convertir el dato a un entero.")
11 except:
12    print("Error inesperado:", sys.exc_info()[0])
13    raise
14 else:
15    print('1/{0} = {1}'.format(den,x))
16
17 f.close()

```

`sys.exc_info()` retorna información sobre la excepción más reciente que fue capturada en un bloque `try-except`

Line 18, Column 1

INSERT Soft Tabs: 4 UTF-8 Python

```

daalvarez@eredron ~ $ echo $'100' > miarchivo.txt;      cat miarchivo.txt
100

```

```

daalvarez@eredron ~ $ python3 05_try_except_raise.py
1/100 = 0.01

```

```

daalvarez@eredron ~ $ echo $'Hola' > miarchivo.txt;      cat miarchivo.txt
Hola

```

```

daalvarez@eredron ~ $ python3 05_try_except_raise.py
No pude convertir el dato a un entero.

```

```

daalvarez@eredron ~ $ echo $'0' > miarchivo.txt;      cat miarchivo.txt
0

```

```

daalvarez@eredron ~ $ python3 05_try_except_raise.py
Error inesperado: <class 'ZeroDivisionError'>

```

```

Traceback (most recent call last):
  File "05_try_except_raise.py", line 6, in <module>
    x = 1/den

```

```

ZeroDivisionError: division by zero

```

```

daalvarez@eredron ~ $ rm miarchivo.txt

```

```

daalvarez@eredron ~ $ python3 05_try_except_raise.py
Error OS: [Errno 2] No such file or directory: 'miarchivo.txt'

```

`echo` con el `>` graba el texto al archivo
`cat` muestra el contenido del archivo
`rm` borra el archivo

```

try:
    raise Exception('argumento 1', 'argumento 2')
except Exception as xyz:
    print(type(xyz)) # tipo de la excepción
    print(xyz.args) # los argumentos se almacenan en .args
    print(xyz)      # pero se pueden imprimir directamente
    x, y = xyz.args # desempaquetamiento de los argumentos
    print('x =', x)
    print('y =', y)

```

```

<class 'Exception'>
('argumento 1', 'argumento 2')
('argumento 1', 'argumento 2')
x = argumento 1
y = argumento 2

```

```

>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: int division or modulo by zero

```

La parte **as** sirve para asociar la excepción a una variable que se puede utilizar posteriormente para referirse a la excepción. El valor en esa variable depende de la excepción.

Lanzando excepciones (raising exceptions)

```
In [4]: try:
...:     print('Se lanza una excepción')
...:     raise ZeroDivisionError('Este es el texto explicativo')
...: except ZeroDivisionError:
...:     print('Aquí se agarró la excepción')
...:     raise # y este comando la vuelve a lanzar
...:
```

Se lanza una excepción
Aquí se agarró la excepción

```
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-4-c0e0d957de73> in <module>()
      1 try:
      2     print('Se lanza una excepción')
----> 3     raise ZeroDivisionError('Este es el texto explicativo')
      4 except ZeroDivisionError:
      5     print('Aquí se agarró la excepción')
```

ZeroDivisionError: Este es el texto explicativo

```
In [1]: def dividir(num, den):
...:     try:
...:         resultado = num/den
...:         return resultado
...:     except ZeroDivisionError:
...:         raise ZeroDivisionError("El denominador no puede ser cero")
...:
```

```
In [2]: dividir(2,3)
Out[2]: 0.6666666666666666
```

```
In [3]: dividir(1,0)
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-1-38006117b4de> in dividir(num, den)
      2     try:
----> 3         resultado = num/den
      4         return resultado
```

ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

```
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-3-b17f97058930> in <module>()
----> 1 dividir(1,0)

<ipython-input-1-38006117b4de> in dividir(num, den)
      4         return resultado
      5     except ZeroDivisionError:
----> 6         raise ZeroDivisionError("El denominador no puede ser cero")
      7
```

ZeroDivisionError: El denominador no puede ser cero

finally clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e. a "**finally**" clause is always executed regardless if an exception occurred in a try block or not.

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print("You should have given either an int or a float")
except ZeroDivisionError:
    print("Infinity")
finally:
    print("There may or may not have been an exception.")
```

The output of the previous script, if saved as "finally2.py", for various values looks like this:

```
bernd@venus:~/tmp$ python finally2.py
Your number: 37
There may or may not have been an exception.
bernd@venus:~/tmp$ python finally2.py
Your number: seven
You should have given either an int or a float
There may or may not have been an exception.
bernd@venus:~/tmp$ python finally2.py
Your number: 0
Infinity
There may or may not have been an exception.
bernd@venus:~/tmp$
```



```
In [10]: def divide(x, y):
...:     try:
...:         result = x / y
...:     except ZeroDivisionError:
...:         print("division by zero!")
...:     else:
...:         print("result is", result)
...:     finally:
...:         print("executing finally clause")
...:
```

```
In [11]: divide(2, 1)
result is 2.0
executing finally clause
```

```
In [12]: divide(2, 0)
division by zero!
executing finally clause
```

```
In [13]: divide("2", "1")
executing finally clause
```

In real world applications, the **finally** clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-13-62cca4a1982f> in <module>()
----> 1 divide("2", "1")

<ipython-input-10-da329b5e3343> in divide(x, y)
      1 def divide(x, y):
      2     try:
----> 3         result = x / y
      4     except ZeroDivisionError:
      5         print("division by zero!")
```

```
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

```

1 ▼ def func1():
2 ▼     try:
3         print("try statement in func1(). After this return 1")
4         return 1
5 ▼     finally:
6         print("After the try statement in func1(), return 2")
7         return 2
8
9 ▼ def func2():
10 ▼     try:
11         print("Raise a value error")
12         raise ValueError()
13 ▼     except:
14         print("An error has been raised! return 1!")
15         return 1
16 ▼     finally:
17         print("Okay after all that let's return 3")
18         return 3
19
20 print(func1())
21 print(func2())

```

Line 22, Column 1

```

daalvarez@eredron ~ $ python3 05_try_return.py
try statement in func1(). After this return 1
After the try statement in func1(), return 2
2
Raise a value error
An error has been raised! return 1!
Okay after all that let's return 3
3
daalvarez@eredron ~ $ █

```

A **finally** clause is always executed before leaving the **try** statement, whether an exception has occurred or not. When an exception has occurred in the **try** clause and has not been handled by an **except** clause (or it has occurred in an **except** or **else** clause), it is re-raised after the **finally** clause has been executed. The **finally** clause is also executed “on the way out” when any other clause of the **try** statement is left via a **break**, **continue** or **return** statement.

```
In [2]: def esta_funcion_falla():
...:     x = 1/0
...:
...:     try:
...:         esta_funcion_falla()
...:     except ZeroDivisionError as mierror:
...:         print('El error es:', mierror)
...:
```

El error es: division by zero

```
In [4]: while True:
...:     try:
...:         x = int(input("Por favor entre un número: "))
...:         break
...:     except ValueError:
...:         print("Oops! Debe entrar un número. Intente de nuevo ...")
...:     except KeyboardInterrupt:
...:         print("\n *** Acaba de presionar Ctrl+C ***")
...:
```

Por favor entre un número: Hola

Oops! Debe entrar un número. Intente de nuevo ...

Por favor entre un número: ^C
*** Acaba de presionar Ctrl+C ***

Por favor entre un número: 123

Aquí se presionó
Ctrl+C

Referencias

- Wikipedia
- <http://www.inventwithpython.com/>
- <http://www.diveintopython3.net/>
- Documentación de Python:
 - <https://docs.python.org/3/tutorial/index.html>
 - <https://docs.python.org/3/>
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: <http://dx.doi.org/10.6035/Sapientia93>
- <https://realpython.com/python-exceptions/>