Estructura Cíclicas

En este nivel se presentan los conceptos necesarios para que un conjunto de instrucciones se ejecute varias veces dependiendo de alguna condición o de los datos que proporcione el usuario.

Hasta el momento los programas que hemos construido ejecutan cada instrucción una única vez, a menos que se encuentren dentro de funciones que se llamen varias veces. El problema es que desde que se escriba el programa también va a quedar establecida la cantidad de veces que se llame cada función.

Este problema puede solucionarse a través del uso de instrucciones iterativas, las cuales nos permiten expresar cuántas veces tiene que ejecutarse una instrucción sin tener que escribirla muchas veces. Más aún, la cantidad no tendría que estar definida a priori sino que se le podría preguntar al usuario o podría depender de una condición que se revise dentro del programa.

En esta sección vamos a introducir el concepto de instrucciones iterativas y vamos a explicar cómo se implementan en Python usaremos las instrucciones while y for.

Actualizando las variables

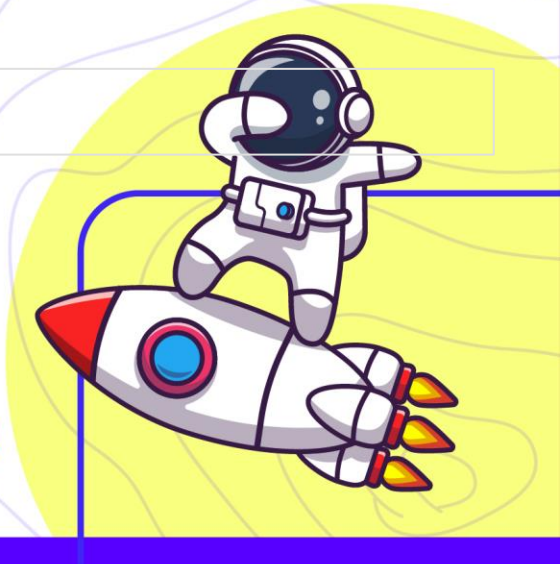
Un patrón común en las declaraciones de asignación es una declaración de asignación que actualiza una variable, donde el nuevo valor de la variable depende del antiguo.

In [ ]:
```
x=0
x = x + 1
print(x)
```

Esto significa "obtener el valor actual de x, añadir 1, y luego actualizar x con el nuevo valor".

Si intentas actualizar una variable que no existe, obtienes un error, porque Python evalúa el lado derecho antes de asignar un valor a x:

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



Antes de que puedas actualizar una variable, tienes que inicializarla, normalmente con una simple asignación:

In [ ]:
```
x = 0
x = x + 1
print(x)
```

Actualizar una variable añadiendo 1 se llama un incremento; restar 1 se llama un decremento.

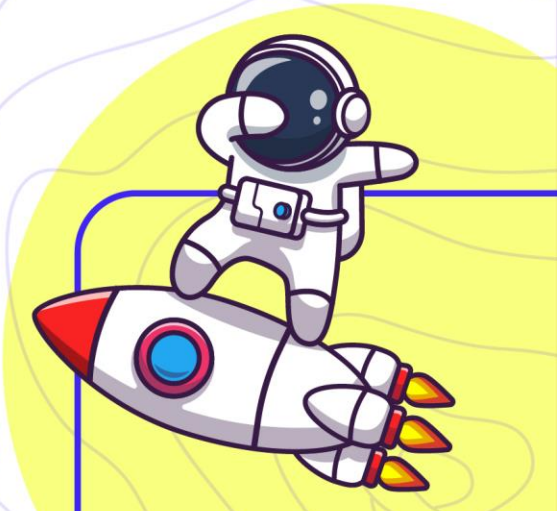
In [ ]:
```
y = 10
y = y-1
print(y)
```

El ciclo While

Las computadoras se utilizan a menudo para realizar tareas repetitivas. Repetir tareas idénticas o similares sin cometer errores es algo que los ordenadores hacen bien y la gente lo hace mal. Debido a que la iteración es tan común, Python proporciona varias características de lenguaje para hacerlo más fácil.

En Python, así como en muchos otros lenguajes, la base de las instrucciones iterativas es una expresión como la siguiente: “mientras que X sea cierto, haga Y”. En inglés, esto se traduciría como “while X, do Y”. Por esto, la instrucción fundamental para expresar iteraciones en Python y en muchos lenguajes se llama while.

Así como un if define una estructura en la cual se tiene que tener como mínimo una condición y un cuerpo, en el caso del while ocurre algo similar: se requiere una condición (la X en la expresión anterior) y un cuerpo (la Y). La principal diferencia con un if es que la condición se va a evaluar muchas veces y que el cuerpo se va a ejecutar cada vez que la condición sea verdadera. Observemos esto en un ejemplo:



In [ ]:

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Despegue!')
```

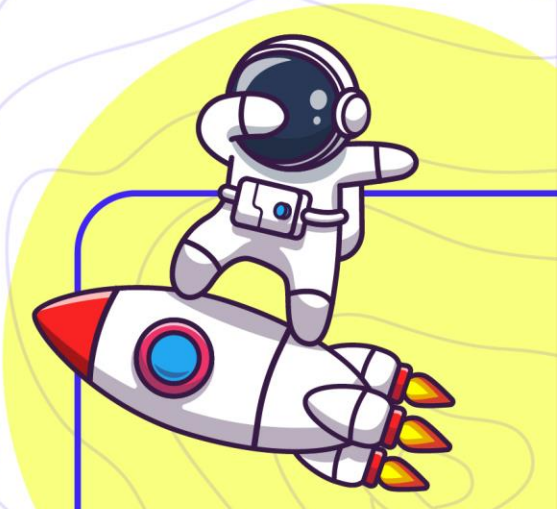
Lo primero que encontramos en este código es que vamos a crear una nueva variable llamada n y la vamos a inicializar en 5. A continuación inicia un bloque while, donde la condición es n > 0. Esto quiere decir que el cuerpo del while (todo lo que está indentado después de :) se va a ejecutar varias veces, hasta que la condición deje de ser verdadera. Esta condición se evaluará antes de la primera vez que se ejecute el cuerpo y se volverá a evaluar después de cada ejecución del cuerpo. El cuerpo de este while tiene sólo dos instrucciones: la primer imprime en la consola el valor actual de la variable n mientras que la segunda reduce el valor de n en uno. La última instrucción del ejemplo imprime la cadena '¡Despegue!' en la consola.

Más formalmente, aquí está el flujo de ejecución de la declaración while:

- Evalúa la condición, dando Verdadero o Falso.
- Si la condición es falsa, salga de la declaración while y continúe la ejecución en la siguiente declaración.
- Si la condición es verdadera, ejecute el cuerpo y luego vuelva al paso 1.

Este tipo de flujo se llama bucle porque el tercer paso vuelve hacia la parte superior. Llamamos cada vez que ejecutamos el cuerpo del bucle una iteración. Para el bucle anterior, diríamos: "Tenía cinco iteraciones", lo que significa que el cuerpo del bucle se ejecutó cinco veces.

El cuerpo del bucle debe cambiar el valor de una o más variables para que eventualmente la condición se vuelva falsa y el bucle termine. Llamamos variable de iteración a la variable que cambia cada vez que el bucle se ejecuta y controla cuando el bucle termina. Si no hay una variable de iteración, el bucle se repetirá para siempre, resultando en un bucle infinito.



Elementos de un while

Acabamos de mostrar con un ejemplo cómo es la ejecución de un while. Ahora estudiaremos con un poco más de cuidado los diferentes elementos que se tienen que considerar cuando se construya una instrucción iterativa. Para esto usaremos la siguiente función que sirve para calcular el factorial de un número recordando que:

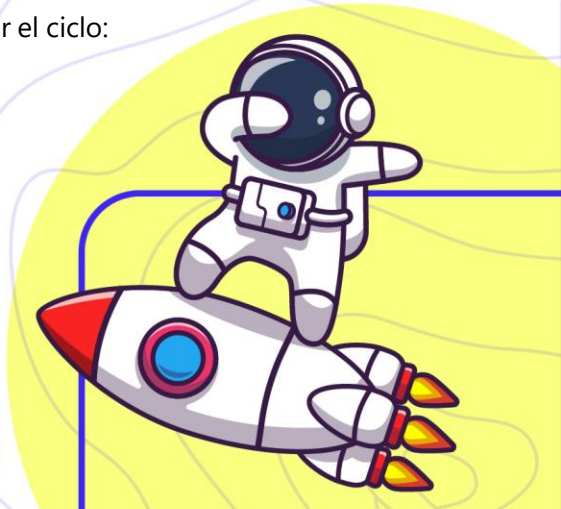
In [ ]:
```
def factorial(n: int) -> int:
    resultado = 1
    numero_actual = 2

    while numero_actual <= n:
        resultado = resultado *
        numero_actual numero_actual += 1
    return resultado
```

Esta función calcula el factorial del número n multiplicando entre ellos todos los números anteriores. El punto importante es que esto lo vamos a hacer número por número, partiendo desde el número 1 y llegando hasta el número n.

Inicialización

Aunque no es parte explícita de un while, las instrucciones que se encuentran antes son importantísimas porque realizan la inicialización del ciclo. Es decir, dejan las variables que nos interesen en el estado necesario para que se pueda ejecutar el ciclo y se obtenga el resultado esperado.

En el caso de nuestra función, hay dos instrucciones que sirven para inicializar el ciclo:



In [ ]:
```
resultado = 1
numero_actual = 2
```

La primera instrucción sirve para crear una variable donde dejaremos el resultado de nuestra función. En este caso, ese resultado será el factorial del parámetro n. La variable resultado la hemos inicializado en 1 por varios motivos que discutiremos más adelante.

La segunda instrucción sirve para crear una variable que nos permitirá saber cuál es el siguiente número que tenemos que multiplicar para seguir calculando el factorial. En este caso, la variable numero_actual la inicializamos en 2 porque el resultado ya estaba inicializado en 1. Si la variable la hubiéramos inicializado en 1 no habría cambiado nuestro programa, pero habría hecho una iteración más en el que habría multiplicado 1*1.

Condición del ciclo

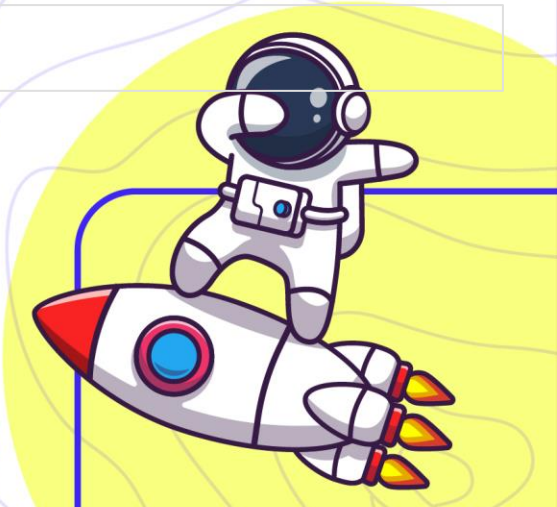
La siguiente parte del análisis de nuestra función se concentra en la condición del while. Debemos recordar que cuando la condición de un while sea verdadera, el cuerpo del ciclo deberá ejecutarse. Si lo vemos desde el punto de vista opuesto, el cuerpo del ciclo tendrá que ejecutarse hasta que la condición sea falsa.

Las dos perspectivas son equivalentes pero, dependiendo del problema, es posible que una de las dos perspectiva sea más fácil de entender. La recomendación que podemos hacer es utilizar nombres de variables que sean muy claros, de tal forma que la condición sea fácil de leer. Por ejemplo, en el caso de nuestra función, la condición puede leerse fácilmente como “Mientras que el número actual sea menor o igual que n, se debe hacer …”.

In [ ]:
```
while numero_actual <= n:
```

La misma condición también se podría haber escrito de la siguiente forma, en la cual la lectura natural sería “Mientras que el número actual no sea mayor a n, se debe hacer …”.

In [ ]:
```
while not numero_actual > n:
```



Cuerpo del ciclo

Después de evaluada la condición de un while, se ejecuta una o varias veces el cuerpo del ciclo. El caso de nuestra función, el cuerpo tiene dos instrucciones:

In [ ]:
```
resultado = resultado * numero_actual
numero_actual = numero_actual + 1
```

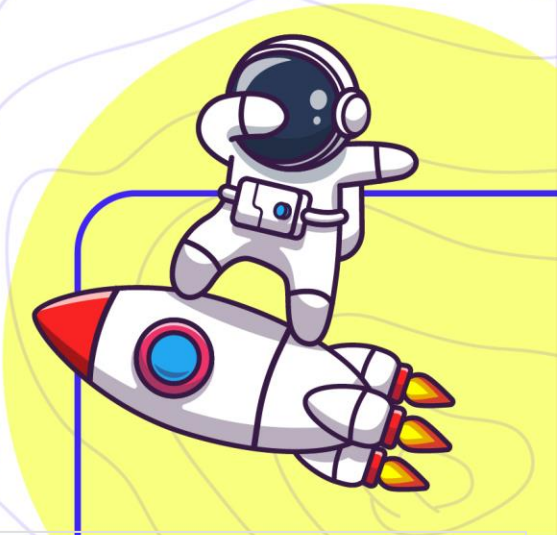
La primera instrucción es la que se encarga de ir acumulando en la variable resultado el valor del factorial del número. Para esto, la instrucción toma el valor que se había acumulado hasta el momento, lo multiplica por el número actual y vuelve a guardarlo en resultado. La segunda instrucción se encarga de ir incrementando de uno en uno el valor de numero_actual.

Antes de la primera ejecución, el valor de resultado es 1 y el valor de numero_actual es 2, como se determinó en la inicialización. Eso quiere decir que el valor de resultado es equivalente al valor de 1! y que el siguiente número por el que debería multiplicarse es 2.

Después de la primera ejecución del ciclo, el valor de resultado se modifica para que sea 2, y el valor de numero_actual se incrementa en uno. Esto quiere decir que ahora el valor de resultado es equivalente al valor de 2!.

En la siguiente iteración el valor de resultado se multiplica por 3 y y el valor de numero_actual llega a 4. Como ahora el valor de resultado es 6, quiere decir que es equivalente a 3!.

El proceso continua hasta que numero_actual es mayor a n. Por ejemplo, si n fuera 6, entonces en la última iteración resultado se multiplicaría por 6 quedando con un valor equivalente al de 6! y numero_actual llegaría a 7. La siguiente vez que se revisara la condición ya no sería verdadera y el ciclo terminaría.



Avance

La segunda instrucción del cuerpo, numero_actual += 1, tiene el rol de avanzar el ciclo hacia su terminación. Este rol es muy importante dentro de cualquier ciclo: siempre tiene que haber una o varias instrucciones que hagan que con cada iteración el ciclo esté más cerca de terminar. Si no se cumpliera esto, el ciclo nunca terminaría.

En nuestra función para el cálculo del factorial, con cada iteración aumentamos en uno el valor de numero_actual, con lo cual nos aseguramos que eventualmente este número sea mayor a n y el ciclo termine. En el caso del contador para el despegue, vamos reduciendo el valor de contador haciendo que eventualmente se vuelva falsa la condición del ciclo: contador > 0.

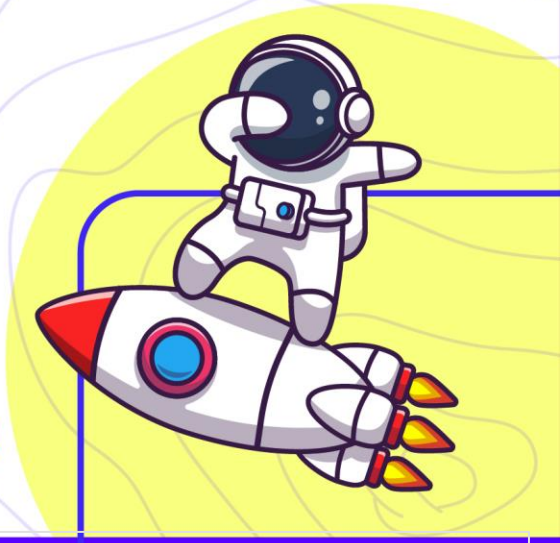
Problema con el avance

En los dos ejemplos que hemos estudiado identificar el avance fue relativamente fácil. Más adelante en esta sección estudiaremos algunos programas donde no es tan fácil ver que el ciclo se está acerca acercando a la terminación. Por ahora veamos unos ejemplos de programas con problemas y que resultarán en ciclos infinitos.

Alejarse de terminación

In [ ]:
```
i = 1
while i > 0:
    print(i)
    i = i+ 1
print("Terminé")
```

Este primer programa nunca termina porque el valor de i siempre es mayor a 0.



Brincarse la meta

In [ ]:
```
# le toca oprimir el boton de pausa sino el sigue dandole

i = 1
while i != 10:
    print(i)
    i = i+ 2
print("Terminé")
```

Este programa tampoco termina porque i siempre va a ser diferente que 10. El problema acá es que estamos incrementando i de dos en dos, pero empezando en 1. Esto quiere decir que i sólo va a asumir valores impares. Una forma fácil de solucionar este problema habría sido cambiar la condición para que fuera i < 10.

Problemas de indentación

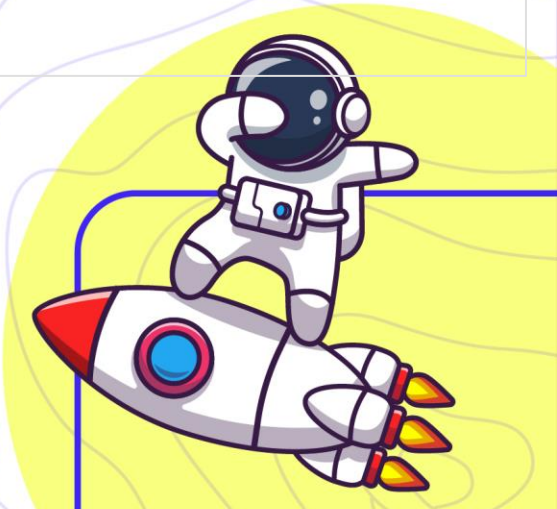
In [ ]:
```
i = 1
while i < 10:
    print(i)
i = i + 1
print("Terminé") #aqui tambien le toca hacer lo mismo
```

Aunque este programa tiene todas las instrucciones que se esperarían, tampoco terminará nunca su ejecución. El problema acá es que el avance no está dentro del ciclo: la variable i siempre va a tener el valor 1 porque el único lugar donde se cambia es inmediatamente después del ciclo.

Olvidar el avance

In [ ]:
```
i = 1
while i < 10:
    print(i)
print("Terminé")
```

Aunque es muy sencillo, este programa ilustra el problema que se presenta más frecuentemente cuando se trabaja con ciclos: olvidar el avance. Como en este caso i nunca cambia de valor, el ciclo no se acerca a terminación a medida que se ejecuta.



Bucle 'while' controlado por Evento

A continuación, se presenta un ejemplo del uso del bucle while controlado por Evento:

In [ ]:
```
promedio, total, contar = 0.0, 0, 0


print("Introduzca la nota de un estudiante (-1 para salir):
") grado = int(input())
while grado != -1:
    total = total + grado
    contar = contar + 1
    print("Introduzca la nota de un estudiante (-1 para salir):
    ") grado = int(input())
promedio = total / contar
print("Promedio de notas del grado escolar es: " + str(promedio))
```

En este casi el evento que se dispara cuando el usuario ingresa el valor -1, causando que el bucle while se interrumpo o no se inicie.

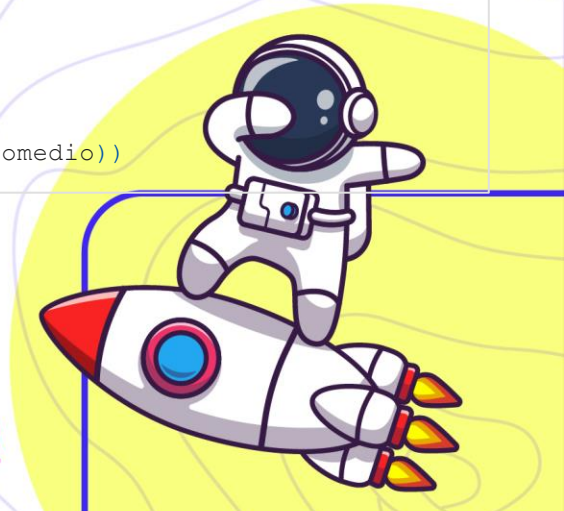
Bucle 'while' con 'else'

Al igual que la sentencia if, la estructura while también puede combinarse con una sentencia else).

El nombre de la sentencia else es equivocada, ya que el bloque else se ejecutará en todos los casos, es decir, cuando la expresión condicional del while sea False, (a comparación de la sentencia if).

In [ ]:
```
promedio, total, contar = 0.0, 0, 0
mensaje = "Introduzca la nota de un estudiante (-1 para salir): "

grado = int(input(mensaje))
while grado != -1:
    total = total + grado
    contar += 1
    grado = int(input(mensaje))
else:
    promedio = total / contar
    print("Promedio de notas del grado escolar: " + str(promedio))
```

La sentencia else tiene la ventaja de mantener el mismo nombre y la misma sintaxis que en las demás estructuras de control.



Sentencias utilitarias

A continuación, se presentan algunos ejemplos del uso de sentencias utilitarias usadas en el bucle while:

Sentencia break

A continuación, se presenta un ejemplo del uso del bucle while controlado la sentencia break:

In [ ]:
```
variable = 10

while variable > 0:
    print('Actual valor de variable:',
    variable) variable = variable -1
    if variable == 5:
        break
```

Adicionalmente existe una forma alternativa de interrumpir o cortar los ciclos utilizando la palabra reservada break.

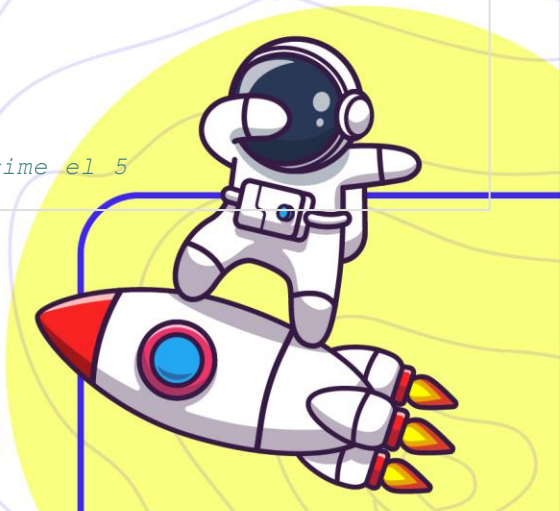
Esta nos permite salir del ciclo incluso si la expresión evaluada en while (o en otro ciclo como for) permanece siendo True. Para comprender mejor use el mismo ejemplo anterior pero se interrumpe el ciclo usando la sentencia break.

Sentencia continue

A continuación, se presenta un ejemplo del uso del bucle while controlado la sentencia continue:

In [ ]:
```
variable = 10

while variable > 0:
    variable = variable -1
    if variable == 5:
        continue
    print('Actual valor de variable:', variable) # no imprime el 5
```

La sentencia continue hace que pase de nuevo al principio del bucle aunque no se haya terminado de ejecutar el ciclo anterior.



Ejemplos

Sucesión de Fibonacci

Ejemplo de la Sucesión de Fibonacci con bucle while:

In [ ]:
```
a, b = 0, 1
while b < 100:
    print b,
    a, b = b, a + b
```

Los ciclos for

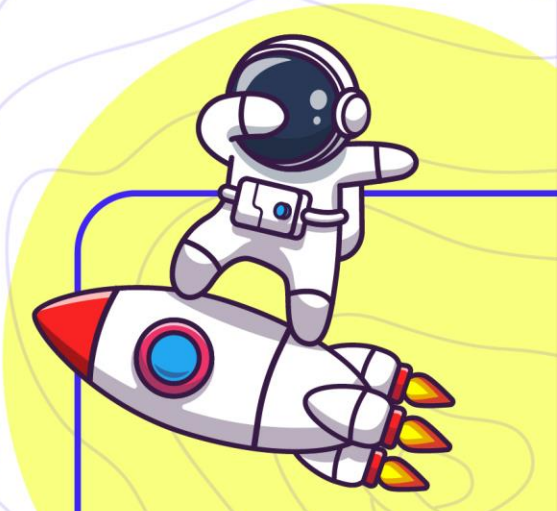
La sentencia for en Python difiere un poco de lo que uno puede estar acostumbrado en lenguajes como C o Pascal. En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal) o darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia for de Python itera sobre los ítems de cualquier secuencia (una lista o una cadenas de caracteres), en el orden que aparecen en la secuencia.

Cada uno de los siguientes elementos solitarios del bucle también debe tener una sangría de 4 o más espacios. Si una línea no tiene sangría, se considera que está fuera del bucle y también terminará cualquier línea adicional considerada en el bucle. Un error común es eliminar los espacios y, por lo tanto, finalizar el ciclo prematuramente.

En bucle for se utiliza para ejecutar un bloque de instrucciones un número determinado de veces.

Para los bucles for se utiliza una variable de contador cuyo valor aumente o disminuya con cada repetición del bucle.

El siguiente ejemplo hace que un procedimiento se ejecute 4 veces. La instrucción for especifica la variable de contador xy sus valores inicial y final. Python incrementará automáticamente la variable contador (x) en 1



después de llegar al final del bloque de ejecución.

In [ ]:
```
for x in range(0, 3):
    print("Estamos en la iteración " + str(x))
```

Python puede usar cualquier método iterable como contador de bucle for. En el caso anterior estamos usando range(). Otros objetos iterables pueden ser listas o string. También puede crear sus propios objetos iterables si es necesario.

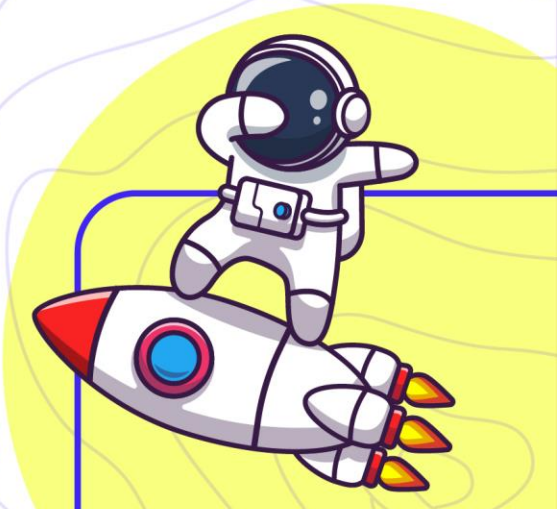
A veces es necesario aumentar o disminuir la variable de contador según el valor que especifique. En el siguiente ejemplo, la variable del contador j se incrementa en 2 cada vez que se repite el ciclo. Cuando finaliza el ciclo, el total es la suma de 0, 2, 4, 6 y 8.

In [ ]:
```
for j in range(0, 10, 2):
    print("Estamos en la iteración " + str(j))
```

Para disminuir la variable del contador, use un rangevalor negativo . Debe especificar un valor final que sea menor que el valor inicial. En el siguiente ejemplo, la variable del contador jse reduce en 2 cada vez que se repite el ciclo. Cuando finaliza el ciclo, el total es la suma de 10, 8, 6, 4 y 2.

In [ ]:
```
for j in range(10, 0, -2):
    print("Estamos en la iteración " + str(j))
```

Puede salir de cualquier instrucción for antes de que el contador alcance su valor final mediante la break instrucción igual que como en el while. Dado que normalmente solo desea salir en determinadas situaciones, como cuando se produce un error, también puede utilizar la instrucción if en el bloque de instrucción True . Si la condición es Falsa , el ciclo se ejecuta como de costumbre.



In [ ]:
```
oracion = 'Mary entiende muy bien Python'
frases = oracion.split() # convierte a una lista cada palabra
print("La oración analizada es:", oracion, ".\n")

for palabra in range(len(frases)):
    print("Palabra: {0}, en la frase su posición es:
        {1}".format( frases[palabra], palabra))
```

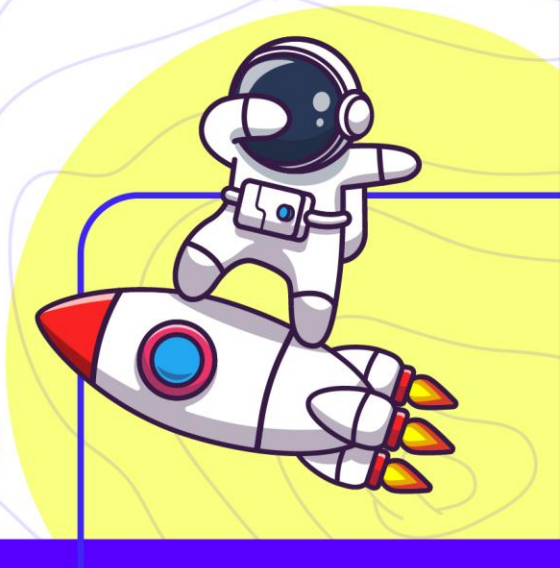
Si se necesita iterar sobre una secuencia de números. Genera una lista conteniendo progresiones aritméticos, por ejemplo, como se hace en el fragmento de código fuente anterior.

Bucle 'for' con Diccionarios

A continuación, se presenta un ejemplo del uso del bucle for con tipos de estructuras de datos diccionarios:

In [ ]:
```
datos_basicos = {
    "nombres":"Leonardo Jose",
    "apellidos":"Caballero Garcia",
    "cedula":"26938401",
    "fecha_nacimiento":"03/12/1980",
    "lugar_nacimiento":"Maracaibo, Zulia,
    Venezuela", "nacionalidad":"Venezolana",
    "estado_civil":"Soltero"
}
```

In [ ]:
```
clave = datos_basicos.keys()
valor = datos_basicos.values()

cantidad_datos = datos_basicos.items()

for clave, valor in
    cantidad_datos: print(clave +
    ": " + valor)
```



In [ ]:
```
frutas = {'Fresa':'roja', 'Limon':'verde', 'Papaya':'naranja', 'Manzana':'amarilla', 'Gua
for nombre, color in frutas.items():
    print(nombre, "es de color", color)
```

Ejemplo dos con diccionarios

In [ ]:
```
frutas = {'Fresa':'roja', 'Limon':'verde', 'Papaya':'naranja', 'Manzana':'amarilla', 'Gua
for llave in frutas:
    print(llave, 'es de color', frutas[llave])
```

El ejemplo anterior itera un diccionario con datos básicos de una persona.

Bucle 'for' con 'else'

Al igual que la sentencia if y el bucle while, la estructura for también puede combinarse con una sentencia else.

El nombre de la sentencia else es equivocada, ya que el bloque else se ejecutará en todos los casos, es decir, cuando la expresión condicional del bucle for sea False, (a comparación de la sentencia if).

In [ ]:
```
db_connection = "127.0.0.1","5432","root","nomina"

for parametro in db_connection:
    print(parametro)
else:
    print("""El comando PostgreSQL es:
$ psql -h {server} -p {port} -U {user} -d
        {db_name}""".format( server=db_connection[0],
        port=db_connection[1],  user=db_connection[2],
        db_name=db_connection[3]))
```

La sentencia else tiene la ventaja de mantener el mismo nombre y la misma sintaxis que en las demás estructuras de control.