UD02 – Estructuras de control

Introducción

Todo programa informático está formado por instrucciones que se ejecutan en forma secuencial de «arriba» a «abajo», de igual manera que leeríamos un libro. Este orden constituye el llamado flujo del programa. Es posible modificar este flujo secuencial para que tome bifurcaciones o repita ciertas instrucciones. Las sentencias que nos permiten hacer estas modificaciones se engloban en el control de flujo.

Python no utiliza llaves para definir los bloques de código, los bloques de código se definen a través de espacios en blanco, preferiblemente 4.

Operadores de comparación

Operador	Símbolo
Igualdad	==
Desigualdad	!=
Menor que	<
Menor o igual que	<=
Mayor que	>
Mayor o igual que	>=

Operadores lógicos

Podemos escribir condiciones más complejas usando los operadores lógicos:

- and
- or
- not

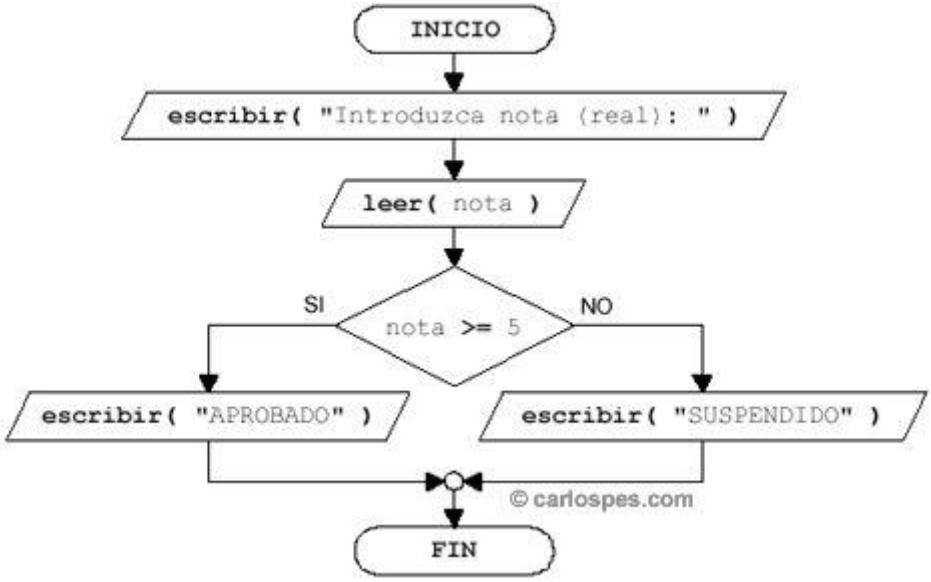
Algoritmo

Un algoritmo es una secuencia ordenada de pasos que conducen a la solución de un problema. Tienen tres características:

- Son precisos en el orden de realización de los pasos.
- Están bien definidos de forma que usando un algoritmo varias veces con los mismos datos, dé la misma solución.
- Son finitos, deben acabarse en algún momento.

Los algoritmos deben representarse de forma independiente del lenguaje de programación que luego usaremos.

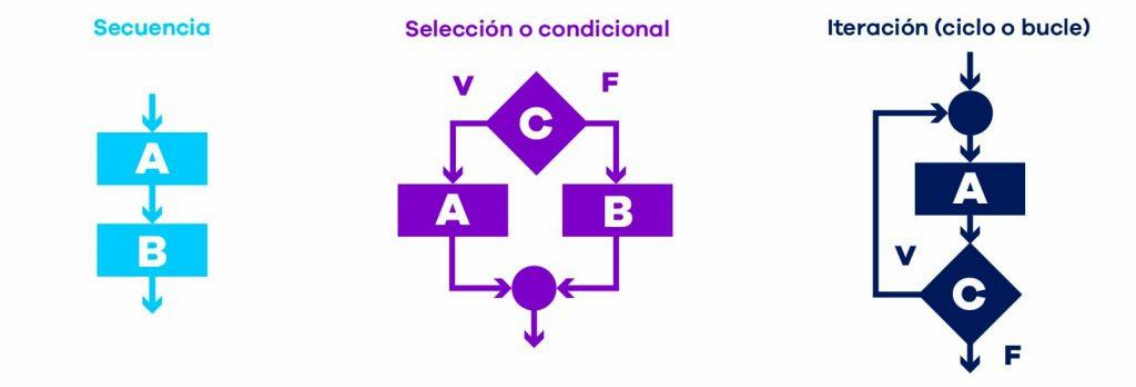
SÍMBOLO	NOMBRE	FUNCIÓN
	Inicio / Fin	Es el inicio y el final de un proceso
	Línea de flujo	Es el orden que llevan las actividades u operaciones
	Entrada / Salida	Son las lectura de los datos de la entrada y la impresión de datos en la salida
	Proceso	Representa las operaciones de cualquier tipo
	Decisión	Se analiza una situación con verdadero o falso



Programación Estructurada

La programación estructurada es un paradigma de programación que se basa en la estructuración de los programas en tres elementos básicos:

- Secuencia: Los programas se ejecutan secuencialmente, es decir, una instrucción tras otra.
- Condicionales: Los programas pueden ejecutar una parte de código u otra dependiendo de una condición.
- Bucles: Los programas pueden ejecutar una parte de código varias veces.



La sentencia IF

La sentencia condicional en Python (al igual que en muchos otros lenguajes de programación) es if. En su escritura debemos añadir una expresión de comparación terminando con dos puntos al final de la línea. Veamos un ejemplo:

```
temperature = 40

if temperature > 35:
    print('Aviso por alta temperatura')
```

Nótese que en Python no es necesario incluir paréntesis (y) al escribir condiciones. Hay veces que es recomendable por claridad o por establecer prioridades.

En el caso anterior se puede ver claramente que la condición se cumple y por tanto se ejecuta la instrucción que tenemos dentro del cuerpo de la condición. Pero podría no ser así. Para controlar ese caso existe la sentencia `else`. Veamos el mismo ejemplo anterior pero añadiendo esta variante:

```
temperature = 20

if temperature > 35:
    print('Aviso por alta temperatura')
else:
    print('Parámetros normales')
```

También podemos hacer uso de la sentencia `elif`:

```
temperature = 28

if temperature < 20:
    if temperature < 10:
        print('Nivel azul')
    else:
        print('Nivel verde')
elif temperature < 30:
    print('Nivel naranja')
else:
    print('Nivel rojo')
```

Ejecución paso a paso… Debug

Asignaciones condicionales

Supongamos que queremos asignar un nivel de riesgo de incendio en función de la temperatura. En su versión clásica escribiríamos:

```
temperature = 35

if temperature < 30:
    fire_risk = 'LOW'
else:
    fire_risk = 'HIGH'
```

Sin embargo, esto lo podríamos abreviar con una asignación condicional de una única línea:

```
fire_risk = 'LOW' if temperature < 30 else 'HIGH'
```

Otro ejemplo. Para poder hacer una llamada VoIP necesitamos tener al menos un 40% de batería o al menos un 30% de cobertura:

```
power = 50
signal_4g = 20

if power > 40 or signal_4g > 30:
    print("Puedes realizar la llamada")
```

«Booleanos» en condiciones

Cuando queremos preguntar por la veracidad de una determinada variable «booleana» en una condición, la primera aproximación que parece razonable es la siguiente:

```
is_cold = True

if is_cold == True:                    ### equivalente if is_cold
    print('Coge chaqueta')
else:
    print('Usa camiseta')
```

Ejercicio:

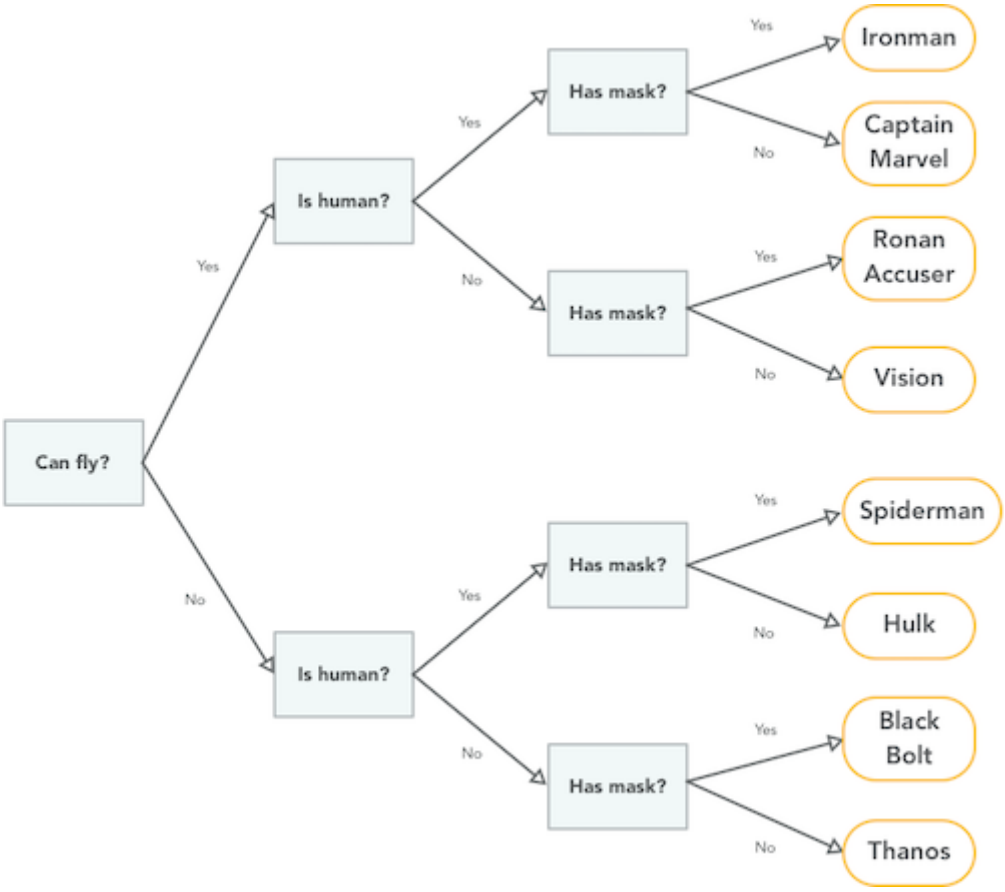
Conoces la página: https://es.akinator.com/

Vamos a construir un programa muy simple, que haga algo parecido:

Implementa un «clon» de Akinator que permita adivinar un personaje de Marvel en base a las tres preguntas siguientes:

1.- ¿Puede volar?
2.- ¿Es humano?
3.- ¿Tiene máscara?

Generando:



Ejemplo:

Entrada: can_fly = True, is_human = True y has_mask = True

Salida: Ironman

Sentencia match-case

Una de las novedades más esperadas (y quizás controvertidas) de Python 3.10 fue el llamado Structural Pattern Matching que introdujo en el lenguaje una nueva sentencia condicional. Ésta se podría asemejar a la sentencia «switch» que ya existe en otros lenguajes de programación

Comparando valores

En su versión más simple, el «pattern matching» permite comparar un valor de entrada con una serie de literales. Algo así como un conjunto de sentencias «if» encadenadas. Veamos esta aproximación mediante un ejemplo:

```
color = '#FF0000'

match color:

    case '#FF0000':
        print('rojo')
    case '#00FF00':
        print('verde')
    case '#0000FF':
```

```
        print('azul')
```

¿Qué ocurre si el valor que comparamos no existe entre las opciones disponibles? Pues en principio, nada, ya que este caso no está cubierto. Si lo queremos controlar, hay que añadir una nueva regla utilizando el guión bajo _ como patrón:

```
color = '#AF549B'

match color:
    case '#FF0000':
        print('Rojo')
    case '#00FF00':
        print('Verde')
    case '#0000FF':
        print('Azul')
    case _:
        print('Unknown color!')
```

Ejercicio:

Escriba un programa en Python que pida (por separado) dos valores numéricos y un operando (suma, resta, multiplicación, división) y calcule el resultado de la operación, usando para ello la sentencia match-case.

Controlar que la operación no sea una de las cuatro predefinidas. En este caso dar un mensaje de error y no mostrar resultado final.

Ejemplo

Entrada: 4, 3, +

Salida: 4+3=7

Bucles:

Cuando queremos hacer algo más de una vez, necesitamos recurrir a un bucle. En esta sección veremos las distintas sentencias en Python que nos permiten repetir un bloque de código.

La sentencia `while`

El primer mecanismo que existe en Python para repetir instrucciones es usar la sentencia while. La semántica tras esta sentencia es: «Mientras se cumpla la condición haz algo».

```
# Ejemplo while
i = 0
while i < 10:
    print(i)
    i += 1
```

Ejecución paso a paso

Veamos otro sencillo bucle que repite un saludo mientras así se desee:

```
want_greet = 'S'  # importante dar un valor inicial

while want_greet == 'S':
    print('Hola qué tal!')
    want_greet = input('¿Quiere otro saludo? [S/N] ')
print('Que tenga un buen día')
```

Romper un bucle while

Python ofrece la posibilidad de romper o finalizar un bucle antes de que se cumpla la condición de parada.

Supongamos que en el ejemplo anterior, establecemos un máximo de 4 saludos:

```
MAX_GREETS = 4

num_greets = 0
want_greet = 'S'

while want_greet == 'S':
    print('Hola qué tal!')
    num_greets += 1
    if num_greets == MAX_GREETS:
        print('Máximo número de saludos alcanzado')
        break
    want_greet = input('¿Quiere otro saludo? [S/N] ')
print('Que tenga un buen día')
```

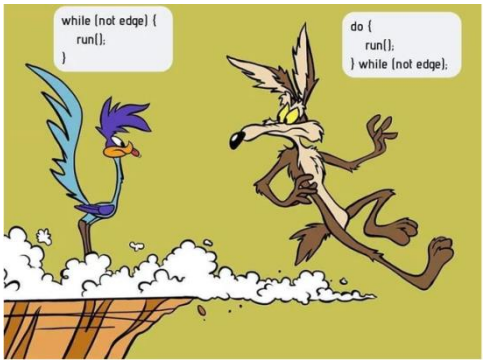
Bucle infinito

Se produce si no establecemos correctamente la condición de parada…

Bucle do …. While

Python no ofrece el bucle do … while, pero lo comentamos ya que lo veremos en otros lenguajes:

```
// Ejemplo do-while en Kotlin
var i = 0
do {
    println(i)
    i++
} while (i < 10)
```



Bucles anidados

Como todas las estructuras de control, podemos incluir un bucle dentro de otro.

Veamos un ejemplo de 2 bucles anidados en el que generamos todas las tablas de multiplicar:

```
factor = num_table = 1
while factor <= 10:
    while num_table <= 10:
        result = num_table * factor
        print(f'{num_table} * {factor} = {result}')
        num_table += 1
    num_table = 1
    factor +=1
```

Cuidado, la complejidad computacional de un algoritmo crece rápidamente al anidar bucles.

La sentencia for

Python permite recorrer aquellos tipos de datos que sean iterables Algunos ejemplos de tipos y estructuras de datos que permiten ser iteradas (recorridas) son: cadenas de texto, listas, diccionarios, ficheros, etc.

```
word =  Python
for letter in word:
    print(letter)
```

También podremos romper un bucle for:

```
word =  Python
for letter in word:
    if letter ==    t   :
        break
    print(letter)
```

Secuencias de números

Es muy habitual hacer uso de secuencias de números en bucles. Python no tiene una instrucción específica para ello. Lo que sí aporta es una función range() que devuelve un flujo de números en el rango especificado. Una de las grandes ventajas es que la «lista» generada no se construye explícitamente, sino que cada valor se genera bajo demanda. Esta técnica mejora el consumo de recursos, especialmente en términos de memoria.

La técnica para la generación de secuencias de números es muy similar a la utilizada en los «slices» de cadenas de texto. En este caso disponemos de la función `range(start, stop, step)`:

- start: Es opcional y tiene valor por defecto 0.
- stop: es obligatorio (siempre se llega a 1 menos que este valor).
- step: es opcional y tiene valor por defecto 1.

range() devuelve un objeto iterable, así que iremos obteniendo los valores paso a paso con una sentencia for ... in 3. Veamos diferentes ejemplos de uso:

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


for i in range(3):  # No hace falta indicar el inicio si es 0
    print(i)

for i in range(1, 6, 2):
    print(i)

for i in range(2, -1, -1):
    print(i)
```

Usando el guión bajo

Hay situaciones en las que no necesitamos usar la variable que toma valores en el rango, sino que únicamente queremos repetir una acción un número determinado de veces.

Para estos casos se suele recomendar usar el guión bajo _ como nombre de variable, que da a entender que no estamos usando esta variable de forma explícita:

```
for _ in range(10):
    print('Repeat me 10 times!')
```