Unidad 1: Introducción a Estructuras de Datos

Estructuras de Datos

Facultad de Ciencias de la Administración - UNER

In [1]:
```
from datetime import datetime
print(f"{'PRIMER' if datetime.now().month < 7 else 'SEGUNDO'} CUATRIMESTRE DE {datetime.now()
```

```
PRIMER CUATRIMESTRE DE 2024 😎
```

Objetivos

- Identificar las distintas alternativas de las que se dispone para controlar el flujo de ejecución de programas.
- Entender como hacer combinaciones de las mismas.

Temas a desarrollar:

- Ejecución Secuencial de sentencias.
- Estructuras de control condicionales: `if`, `elif`, `else`, `match`.
- Estructuras de control iterativas: `while`, `for`, `for in range`, `break`, `continue`.

Estructuras de control

- Una **estructura de control** es una sentencia de control y las sentencias cuya ejecución ésta controla.
- Es un bloque de código que permite **agrupar** instrucciones de manera controlada.
- Todos los lenguajes de programación tienen un conjunto mínimo de instrucciones que permiten especificar el control propio del algoritmo que se requiere implementar.
  - Este conjunto mínimo debe contener estructuras para la **Secuencia, Decisión, Selección e Iteración**.

Secuencia

- La estructura de control más simple está representada por una sucesión de instrucciones (por ejemplo asignaciones), en la que el **orden de ejecución coincide con el orden físico de aparición de las instrucciones**.

- La implementación de la estructura de control secuencia se logra simplemente ubicando **una instrucción debajo de la anterior**.

Decisión - Ejecución condicional

- Cuando programamos, casi siempre necesitamos controlar condiciones y cambiar el comportamiento del programa de acuerdo a si se cumplen o no. Las estructuras de control condicionales nos dan esta habilidad.

In [ ]:
```
x = int(input("Ingrese un número: "))
if x > 0:
        print("x es positivo")
```

- La **expresión booleana** después de la palabra reservada `if` es llamada **condición**. Si es `True` (verdadero), la sentencia identada que le sigue se ejecuta. Caso contrario, nada sucede.
- El `if` tiene un encabezado seguido por un cuerpo identado. Las sentencias como esta se llaman **declaraciones compuestas**.
  - No hay límite para el número de sentencias que pueden aparecer en el cuerpo de un bloque `if`, pero tiene que ser al menos una.
  - Ocasionalmente, es útil tener un cuerpo sin sentencias (para guardar espacio para algo que vamos a agregar más tarde).
  - En ese caso, podemos escribir la palabra reservada `pass` que no hace nada.

In [ ]:
```
if x < 0:
        pass # Aquí falta manejar qué pasa cuando x es menor que 0.
```

Ejecución alternativa

- Una segunda forma de la sentencia `if` es la ejecución alternativa, donde hay dos posibilidades y una condición que determina cuál se ejecuta.
- La sintaxis es así:

In [ ]:
```
x = int(input("Ingrese un número: "))
if x % 2 == 0:
    print ("x es par")
else:
    print("x es impar")
```

- El resto de la división entera de `x` por `2` es `0` cuando `x` es par.
- En caso que la condición sea `False` (falso), el segundo conjunto de sentencias se ejecuta.
- Dado que la condición debe ser `True` o `False`, exactamente una de las alternativas se ejecuta. Las alternativas son llamadas ramas, porque como si fueran ramas donde se bifurca el flujo de ejecución.

Condiciones encadenadas

- A veces hay más de dos posibilidades y necesitamos **más de dos ramas**. Una forma de expresar un cómputo de este tipo es encadenar condiciones.

In [ ]:
```
x = int(input("Ingrese un número para asignar a x: "))
y = int(input("Ingrese un número para asignar a y: "))
if x < y:
    print ("x es menor que y")
elif x > y:
    print ("x es mayor que y")
else:
    print("x e y son iguales")
```

- `elif` es una abreviación de **"else if"**.
- Otra vez, exactamente sólo una de las ramas se va a ejecutar.
- No hay límite para la cantidad de sentencias que pueden aparecer dentro de un bloque `elif`.
- Si hay una cláusula `else`, tiene que estar al final, no es obligatorio que exista.

Condiciones encadenadas (2)

In [ ]:
```
def dibujar_a():
    print("dibujar_a()")
def dibujar_b():
    print("dibujar_b()")
def dibujar_c():
    print("dibujar_c()")
```

In [ ]:
```
opcion = input("Seleccione una opción: a, b, c: ")
if opcion == "a":
    dibujar_a()
elif opcion == "b":
    dibujar_b()
elif opcion == "c":
    dibujar_c()
```

Condiciones anidadas

- Un condicional puede ser también anidado dentro de otro. Podemos escribir el ejemplo anterior de esta forma:

In [ ]:
```
if x == y:
    print("x e y son iguales")
else:
    if x < y:
        print ("x es menor que y")
    else:
        print("x es mayor que y")
```

- El condicional exterior contiene dos ramas. La primera contiene una única sentencia simple. La segunda contiene otra sentencia `if` que tiene sus propias dos ramas.
- Esas dos ramas son también sentencias simples, sin embargo, podrían haber sido sentencias condicionales también.
- Puede parecernos que la identanción de las sentencias hace fácil de leer la estructura, pero los condicionales anidados se vuelven difíciles de leer muy rápido. Es una buena idea evitarlos lo más que podamos.

Condiciones anidadas (2)

- Los operadores lógicos pueden ser una solución para evitar los condicionales anidados. Por ejemplo, podemos reescribir el siguiente código usando un condicional simple:

In [ ]:
```
x = int(input("Ingrese un valor para x: "))
if 0 < x:
        if x < 10:
                print("x es un número positivo de un sólo dígito")
```

- La instrucción `print` se ejecuta sólo si pasamos las dos condiciones, así que podemos obtener el mismo efecto utilizando un operador `and`:

In [ ]:
```
if 0 < x and x < 10:
        print ("x es un número positivo de un sólo dígito")
```

- Para este tipo de condiciones, **Python** provee una opción más concisa:

In [ ]:
```
if 0 < x < 10:
        print ("x es un número positivo de un sólo dígito")
```

Operador ternario

- El **operador ternario** es una herramienta muy potente que muchos lenguajes de programación tienen.
- En **Python** es un poco distinto a lo que sería en los lenguajes de la familia de **C**, pero el concepto es el mismo.
- Se trata de una cláusula `if`, `else` que se define en una sola línea utilizada que retorna un valor si se cumple una condición y otro en caso que no se cumpla.
- En vez de escribir:

In [ ]:
```
a = 10
b = 10
# Muchas líneas...
# if b != 0:
#     c = a / b
# else:
#     c = -1
# Con operador ternario:
c = a/b if b != 0 else -1
```

Selección - Múltiples alternativas con match

- En este tipo de **estructuras de control** se nos permite ejecutar diferentes secciones de código dependiendo de una condición.
- La sentencia de control `match/case` toma un objeto y verifica si coincide con uno o más patrones. Por último lleva a cabo una acción si se encontraron coincidencias.
- Por ejemplo:

In [ ]:
```
x = int(input("Ingrese un valor para x: "))
match x:
    case 1:
        print("Seleccionó 1")
    case 2:
        print("Seleccionó 2")
    case _:
        print("No entró en ninguno")
```

- Donde si el valor de `x` coincide con el de uno de los valores que siguen a la palabra reservada `case`, entonces, las sentencias dentro del bloque `case` se ejecutan. El símbolo `_` al final, es un comodín, se seleccionará en caso que no existan coincidencias con los valores expresados en las entradas `case` previas.

Repetición / Iteración

- Se llama **iteración** a la habilitad de ejecutar un bloque de sentencias de manera repetida.
- Hay dos tipos de iteraciones:
  - **Iteración indefinida**: el bloque de código se ejecuta hasta que se cumple alguna condición. En **Python** con un bucle `while`.
  - **Iteración definida**: el número de repeticiones se especifica explícitamente de antemano. En **Python** con un bucle `for`.

Bloque while

- Usando un `while` podemos hacer una cuenta descendente de números:

In [ ]:
```
def cuenta_descendente(n):
    while n > 0:
        print (n)
        n = n - 1
    print ("It's the final countdown!!! Tarata taaaaa!!!")

cuenta_descendente(10)
```

- Casi que se puede leer la instrucción `while` como que si fuese en inglés:
  - *"Mientras n sea mayor que 0 mostrar el valor de n y luego decrementar n. Cuando se llegue a 0 mostrar It's the final countdown"*.

Bloque while (2)

- Más formalmente, aquí el flujo de ejecución del bloque while:
  1. Determinar si la condición es verdadera o falsa.
  2. Si es falsa, salir del `while` y continuar con la ejecución de la siguiente sentencia.
  3. Si la condición es verdadera, ejecutar el cuerpo y luego volver al paso 1.
- El tipo de flujo es llamado bucle porque el tercer paso vuelve hacia atrás al primer paso.
- El cuerpo del bucle **tiene que cambiar el valor de una o más variables** para que la **condición** se vuelva **falsa** en algún momento y **termine el bucle**.
- En caso que no fuese así, el bucle se repetirá por siempre, lo que se llama **bucle infinito**.

Rangos o range

- Los **rangos** son secuencias de números enteros no modificables predefinidas. Se crean utilizando la función `range`, que también determina el tipo de dato.
- Así `range (4)` genera la siguiente salida:

In [ ]:
```
print(list(range(4))) # después vemos que es list()
```

- Un rango genera una secuencia de números que van desde `0` por defecto hasta el número que se pasa como parámetro menos `1`.
  - En realidad, se pueden pasar **hasta tres parámetros** separados por coma, donde el primero es el inicio de la secuencia, el segundo el final y el tercero el salto que se desea entre números.
  - Por defecto se empieza en `0` y el salto es de `1`. Cabe destacar que los índices pueden ser negativos y tan grandes como se desee.
- Por lo tanto, si hacemos `range(5,20,2)`, se generarán números de `5` a `19` de dos en dos.

Rangos o range (2)

- Los rangos son inmutables y presentan la gran ventaja de que son iteradores (concepto que veremos más adelante en profundidad), por lo que, cuando se crean, no guardan la información que representan, sino el procedimiento necesario para generar la secuencia de números.
  - Esto hace que ocupen muy poco espacio y permitan controlar la memoria utilizada en los programas, ya que van generando los valores de uno en uno y no todos a la vez.

Rangos o range (3)

- Se puede acceder a un elemento en concreto del rango utilizando un índice y usar operaciones como `len()` o `in` e incluso usar selecciones de subsecuencias.

In [ ]:
```
rango = range (0, 24, 2)
print(rango)
print("8 in rango: ", 8 in rango)
print("7 in rango: ", 7 in rango)
print(rango[3])
```

Bucle for

- El bucle for de **Python** tiene algunas particularidades comparado con otros lenguajes de comparación.
- El `for` es un tipo de bucle, similar al `while` pero con ciertas diferencias.
- La principal es que el número de iteraciones de un `for` esta definido de antemano, mientras que en un `while` no.
- En un bucle `while` la condición es evaluada en cada iteración para decidir si volver a ejecutar o no el código, en el `for` no existe tal condición, sino un iterable que define las veces que se ejecutará el código.
- En el siguiente ejemplo vemos un bucle `for` que se ejecuta `5` veces, y donde la variable `i` incrementa en cada iteración su valor en `1`.

In [ ]:
```
for i in range(0, 5):
    print(i)
```

Bucle for (2)

- Es preferible utilizar un bucle `for` (en vez de `while`) cuando la cantidad de iteraciones se conoce de antemano o está definida por las dimensiones de un tipo de datos que soporte operaciones de iteración.
- En **Python** se puede iterar prácticamente todo, como por ejemplo un str. En el siguiente ejemplo vemos como `i` va tomando los valores de cada letra:

In [ ]:
```
for i in "Python":
    print(i)
```

Break

- A veces se ejecutan algunas iteraciones de un bucle y nos damos cuenta que no es necesario seguir con las siguientes. En estas situaciones la sentencia `break` hace que la ejecución continúe inmediatamente después del bucle.
- Por ejemplo, queremos capturar los datos ingresados por el usuario hasta que escriba **listo**. Podríamos escribir:

In [ ]:
```
while True:
    linea = input('Ingrese un texto o listo para terminar: ')
    if linea == 'listo':
        break
    print(linea)
    print('Listo!')
```

- La condición del bucle es `True`, por tanto siempre se va a cumplir. El bucle se ejecuta hasta que llega la sentencia `break`.
- Esta forma de programar los bucles `while` es común porque podemos controlar si una condición se cumple en cualquier lugar dentro del bucle (no solo al principio) y podemos expresar que se detenga afirmativamente ("*Frená si pasa esto!*") en vez de negativamente ("*Hacé esto hasta que pase aquello*").

Continue

- La palabra reservada `continue` nos permite terminar la iteración actual y continuar con la siguiente.
- Por ejemplo:

In [ ]:
```
for i in "Python":
    if i == "t":
        continue
    print(i)
```

Bibliografía

- Óscar Ramírez Jiménez: "*Python a fondo*" 1era Edición. Editorial Marcombo S.L.. 2021.
- Allen Downey. "*Think Python*". 2da Edición. Editorial Green Tea Press. 2015.
- Bill Lubanovic. *"Introducing Python". 2da Edición. O' Reilly. 2020.
- Eirc Matthes: "*Python Crash Course*". 1era Edición. Editorial No Starch Press. 2016.
- Zed A. Shaw: "*Learn Python 3 the Hard Way*". 1era Edición. Editorial Addison-Wesley. 2017.
- Web - John Sturtz: *Python "for" Loops (Definite Iteration)*. Enlace.