Abstracción de datos

Uno de los aspectos importantes que debe decidirse al resolver un problema del mundo real es **cómo representar los datos** de entrada y salida, lo que influye directamente en la **calidad de la solución** que se puede obtener. Consideremos por ejemplo, la simple tarea de calcular el promedio de los controles para un curso de matemáticas que originalmente considera cinco notas:

Solución 1

Objetivo: Calcular el promedio de controles de un estudiante

1. Obtener nota1, nota2, nota3, nota4 y nota5 del estudiante
2. Calcular la suma de estas notas:

$$suma = nota1 + nota2 + nota3 + nota4 + nota5$$

3. Calcular el promedio dividiendo por cinco:

$$promedio = \frac{suma}{5}$$

4. Mostrar el valor promedio obtenido al usuario

La Solución 1 obtiene las cinco notas de un estudiante, calcula el promedio de ellas y entrega el valor obtenido al usuario. Pero esta forma de resolver el problema es muy rígida y **fuerza** al usuario a ingresar siempre cinco notas. La solución no serviría si algún estudiante tuvo una inasistencia justificada y registra sólo cuatro notas, o si el profesor premió algunos estudiantes con un 7.0 extra, o si por un paro estudiantil se lograron hacer sólo tres controles. En este sentido la solución es de **baja calidad**.

Esto ocurre porque quien diseñó la solución estaba pensando más en la implementación de la solución que en el problema mismo. Si pensamos en el problema con mayor generalidad, se podría llegar a la Solución 2.

Solución 2

Objetivo: Calcular el promedio de notas de un estudiante

1. Obtener la lista de notas del estudiante: $l = \{n_1, n_2, n_3, \cdots, n_m\}$
2. Calcular el promedio de las notas como:

$$promedio = \frac{\sum_{i=1}^{m} n_i}{m}$$

3. Mostrar el valor promedio obtenido al usuario

Notemos que la Solución 2 utiliza una **"lista de notas"** del estudiante. Sabemos que esta lista ha de contener valores flotantes con un decimal de precisión y que sólo acepte valores entre 1.0 y 7.0. Deberá ser capaz de obtener estos valores desde teclado o archivos y podrá aceptar que se agreguen y eliminen elementos.

Pero este tipo de lista **no existe en Python** y en ningún otro lenguaje de programación. Será entonces nuestra tarea **implementar** este **nuevo tipo de dato** para poder usarlo en nuestros programas. En Python, usaremos el **tipo** `list` y definiremos **funciones** que permitan agregar y quitar notas, además de subrutinas para leer notas y mostrar las notas contenidas.

Pero todo este **detalle de cómo** manejar las notas, lo hemos **separado del problema** de calcular el promedio, lo que permite **concentrarnos en resolver el problema**, facilitando enormemente nuestra tarea. Además, nuestras soluciones serán **más claras** y de **mayor calidad** (más generales).

Esta idea de usar **estructuras de datos apropiadas** para resolver un problema **sin preocuparse** mucho de cómo vamos a implementarlas, se conoce como **abstracción de datos**.

Importante

Abstracción de datos (Definición): Técnica que tiene por finalidad describir una **estructura de datos** que es **definida indirectamente** por medio de las **operaciones** y las **restricciones** que le son aplicables.

Discutamos por qué la abstracción de datos hace nuestras soluciones más generales. Supongamos que se nos pide construir un programa que nos permita jugar *al gato*. Cualquier estudiante que haya intentado programar este juego sabrá que no es sencillo y tiene muchos detalles en qué fijarse: no se puede colocar una cruz encima de un círculo, el juego se termina cuando se completa una fila, una columna o una diagonal con el mismo símbolo, cómo hacemos para que los usuarios nos indiquen qué jugada quieren hacer, etc. Pero el problema se torna mucho más simple cuando **suponemos que existe** el tipo de dato `TableroGato`, que **conoce del estado** el tablero de juego y que permite **realizar cambios** sobre él. La Solución 3 muestra un modelo de los pasos que permiten simular el juego con esta estructura de datos.

La Solución 3 necesita que la estructura de datos `TableroGato` permita crear un tablero vacío, permita saber si hay una jugada ganadora, permita saber si hay un empate, dé a conocer la jugada ganadora, permita conocer las jugadas válidas, permita registrar una jugada válida sobre el tablero y pueda mostrar al usuario el estado actual del juego.

Solución 3

Objetivo: Permitir que dos usuarios jueguen al gato

```
1.      tablero = nuevo TableroGato vacío
2.      jugada = X
3.      Mientras tablero no indique que hay un ganador ni un empate:
3.1.        Mostrar al usuario el estado actual del tablero
3.2.        Obtener del usuario una posición válida para la jugada según el estado actual del tablero
3.3.        Marcar la jugada en el tablero
3.4.        Si jugada == X:
3.4.1.          jugada = O
            Sino
3.4.2.          jugada = X
3.5.    Si tablero indica que hay un empate:
3.5.1.      Mostrar mensaje "Fue un empate"
        Sino
3.5.2.      jugadaGanadora = jugada ganadora en el estado actual del tablero
3.5.3.      Mostrar mensaje "Ganó el jugador que marcó:" + jugadaGanadora
```

En **programación imperativa**, debemos suponer que **existen funciones** que manejan la estructura de datos:

Función	Descripción
`creaTableroGato()`	**Salida**: una estructura `TableroGato` vacía.
`hayJugadaGanadora(tablero)`	**Entrada**: una estructura `TableroGato`. **Salida**: `True` si hay una jugada ganadora en `tablero`; `False` en caso contrario.
`hayEmpate(tablero)`	**Entrada**: una estructura `TableroGato`. **Salida**: `True` si hay en `tablero` se da un empate; `False` en caso contrario.
`dimeJugadaGanadora(tablero)`	**Entrada**: una estructura `TableroGato`. **Salida**: `X` si las X's forman una jugada ganadora en `tablero`; `O` si las O's forman una jugada ganadora en `tablero`; mensaje de error y finalización del programa si no existe una jugada ganadora.
`eligeJugadaValida(tablero)`	**Entrada**: una estructura `TableroGato`. **Salida**: jugada válida en `tablero` escogida por el usuario.
`registraJugada(tablero, jugada)`	**Entrada**: una estructura `TableroGato` y una jugada válida `jugada`. **Salida**: la estructura `TableroGato` que resulta de realizar la jugada válida indicada.
`muestraTablero(tablero)`	**Entrada**: una estructura `TableroGato`. **Asegura**: se ha mostrado en pantalla las jugadas contenidas en `tablero`.

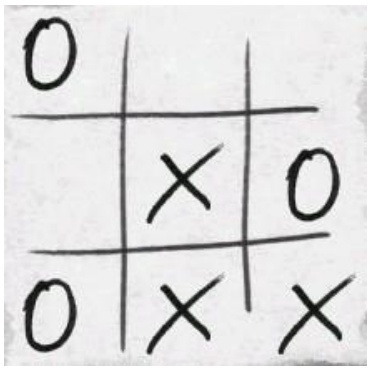
Con esta estructura, la Solución 3 puede convertirse en el bloque principal de nuestro programa en Python, como se muestra en Solución 4.

Solución 4

```
# Bloque principal del programa para jugar al gato

tablero = creaTableroGato()
jugada = X
While not hayJugadaGanadora(tablero) and not hayEmpate(tablero):
   muestraTablero(tablero)
   jugadaValida = eligeJugadaValida(tablero)
   tablero = registraJugada(tablero, jugada)
   if jugada == X:
      jugada = O
   else:
      jugada = X
if hayEmpate(tablero):
   print "Fue un empate"
else:
   jugadaGanadora = dimeJugadaGanadora(tablero)
   print "Ganó el jugador que marcó: " + jugadaGanadora
```

La gran ventaja de esta solución, que asume la existencia de la estructura de datos `TableroGato`, es que funciona **independientemente de la forma** en que se implemente `TableroGato`. Por ejemplo, alguien podría usar una lista con nueve strings representando las jugadas en cada posición del tablero (figura 1A); otra persona podría usar una matriz de enteros con las filas del tablero (figura 1B); otro programador podría inclinarse por una matriz con las columnas del tablero (figura 1C); otro podría usar dos listas, una para las posiciones con X's y otra para las posiciones con O's (figura 1D); otra persona podría inclinarse por una lista de pares (posición, jugada) (figura 1E); etc.



A. `["O", " ", " ", " ", "X", "O", "O", "X", "X"]`

B. `[[1, 0, 0], [0, 2, 1], [1, 2, 2]]`

C. `[[1, 0, 1], [0, 2, 2], [0, 1, 2]]`

D. `[[5, 8, 9], [1, 6, 7]]`

E. `[[9, "X"], [6, "O"], [5, "X"], [1, "O"], [8, "X"], [7, "O"]]`

Figura 1. Diferentes formas de representar un tablero para jugar al gato.

Lo importante es que en **todos estos casos**, y con **cualquier otra implementación** que se nos pueda ocurrir, el programa en Solución 3 permite a dos usuarios jugar al gato, es decir, resuelve el problema del mundo real original.

Pregunta 1

Trabaja con tu grupo en la solución de la primera pregunta de la actividad.

Estructuras de datos en Python

Python provee estructuras de datos en la forma de **objetos**. Los objetos involucran conceptos importantes en programación, que incluso dan origen a un **nuevo paradigma** para construir programas. Sin embargo, estos conceptos están fuera del alcance de este curso inicial, por lo que nos limitaremos a la siguiente definición de objeto:

Importante

Objeto (Definición): Una instancia en la memoria del computador de un **tipo de dato** que tiene un **contenido**, cuya estructura interna muchas veces no es sabida, y que provee **métodos** que permiten conocerlo y/o modificarlo.

Esto significa que si bien los objetos **guardan** uno o más datos internamente, no se tiene un **acceso directo** a ellos. Para conocerlos o modificarlos, el objeto provee **una interfaz** a través de un conjunto de **métodos** que podemos invocar.

¡Ya hemos estado usando objetos en Python! Los strings, las listas, los archivos, son todos objetos, como se muestra en el Ejemplo 1.

Existen muchos módulos en Python que proveen objetos con diferentes finalidades y que nos pueden ahorrar bastante trabajo. Por ejemplo hay módulos que definen objetos para construir interfaces gráficas con ventanas y botones (`wx`), para manejar mensajes de texto (`msnp`), para manipular imágenes (`PIL`), para realizar cómputos científicos y análisis estadísticos (`numpy`, `pandas`), para crear gráficos 2D (`matplotlib`) y 3D (`visual`), para manejar archivos de audio (`pymedia`), para construir juegos (`pygame`), etc.

Python provee en forma nativa de varias de las estructuras de datos más usadas en programación (además de los mencionados strings, archivos y listas), ya sea directamente o como base para implementar estructuras de datos más específicas.

Ejemplo 1

```
>>> str = "Hola MUNDO!"
>>> print str.lower()
hola mundo!
>>>
>>> archivo = open("salida.txt", "w")
>>> archive.write("salida\n")
>>> archive.close()
>>> archive = open("salida.txt", "r")
>>> str = archive.readline()
>>> archive.close()
>>> print str.strip().upper()
SALIDA
>>>
>>> lista = list()
>>> lista.append(5)
>>> lista.insert(0, 2)
>>> lista
[2, 5]
>>>
```

Tuplas

En Python existen estructuras llamadas **tuplas**, las cuáles funcionan de manera muy similar a las listas, con la salvedad de ser **inmutables** (como los strings), es decir sus elementos no pueden ser cambiados después de ser creados. Estas se crean de forma análoga a las listas, con la diferencia que para declarar una tupla constante no es necesario usar paréntesis cuadrados, y el acceso a sus elementos tiene la misma sintaxis y semántica de las listas.

Ejemplo 2

```
>>> tupla = 9, 8, 7
>>> tupla[0]
9
>>> tupla[0:2]
(9, 8)
>>> tupla[0] = 4

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    tupla[0] = 4
TypeError: 'tuple' object does not support item assignment
```

Las listas, al estar delimitadas por paréntesis cuadrados, entregan una sintaxis clara para la creación de listas vacías o con un sólo elemento. Esto no es tan así con las tuplas, y para crear estructuras de tamaño 0 ó 1 se debe recurrir a ciertas notaciones particulares.

Ejemplo 3

```
>>> tuplaLargo0 = ()
>>> len(tuplaLargo0)
0
>>>
>>> tuplaLargo1 = 1,
>>> len(tuplaLargo1)
1
```

Podemos notar que para hacer una tupla vacía, tendremos que definirla como un par de paréntesis redondos vacíos, mientras que para crear una tupla de largo 1, se debe escribir el primer (y único) elemento seguido por una coma.

Las tuplas pueden utilizarse para crear funciones que retornan **múltiples valores**. Por ejemplo, en la función entrega una tupla con el valor mínimo y máximo de una lista de números. Es importante destacar que en realidad se retorna un valor único, correspondiente a una tupla de valor 2 que contiene el mínimo y máximo requerido.

Ejemplo 4

```
def calculaMinimoYMaximo(numeros):
   minimo = numeros[0]
   maximo = numeros[0]
   for numero in numeros:
      if numero < minimo:
         minimo = numero
      if numero > maximo:
         maximo = numero
   resultado = minimo, maximo
   return resultado
```

Pregunta 2

Con tu grupo de trabajo, resuelvan la pregunta 2 de la guía.

Pregunta 3

Resuelve ahora, con la ayuda de tu grupo, la pregunta 3 de la actividad.