Algunos objetos nativos de Python

Hemos visto que es conveniente separar el **manejo de los datos** de la **complejidad de los procesos** requeridos para resolver un problema del mundo real. En un segundo paso, nos preocupamos de implementar los tipos de datos que necesitemos.

Pero esto no es una idea tan reciente, por lo que en todos los lenguajes de programación existen **bibliotecas** (*libraries*, *modules*, etc.) con **estructuras de datos** complejas creadas por otros programadores en un sinfín de ámbitos. Incluso, las estructuras más clásicas tienden a estar disponibles de forma nativa.

Veamos algunas de las estructuras nativas de Python, que nos permitirán resolver muchos problemas directamente y también nos facilitarán la construcción de nuestras propias estructuras de datos. Recordemos que en Python, las estructuras de datos se llaman **clases de objetos**, y que aunque **desconocemos cómo está organizado internamente** el contenido de estos objetos, tenemos disponibles **métodos** que nos permiten acceder a ese contenido, y modificarlo en objetos que son de naturaleza **mutable**.

Filas

Las **filas** (*queues*) son **secuencias de valores** que cuando reciben un nuevo elemento lo **agregan al final**, y cuando se retira un elemento lo **sacan desde el frente**. Es decir, esto emula él comportamiento típico de las filas (o colas) que hacemos los seres humanos, como las filas de los bancos o los supermercados.

Las filas son una de las estructuras de datos clásicas, ya que las tareas que uno o más usuarios quieren ejecutar en un computador son normalmente **encoladas** en el orden en que se solicitan. Cuando la CPU se desocupa, atiende primero a aquella tarea que lleva más tiempo en la fila, es decir, la tarea que se encuentra al frente de la cola.

En Python, las filas están implementadas como objetos de clase `list`, como se muestra en el Ejemplo 1.

El método `.append(x)` agrega el elemento `x` al final de la lista; el método `.pop(i)` saca y devuelve el elemento de la lista en la posición `i`. De esta forma, es simple hacer que las listas de Python se comporten como filas.

Ejemplo 1

```
>>> fila = list()
>>> fila.append("promedio")
>>> fila.append(3)
>>> fila.append(5.5)
>>> fila.append("notas")
>>> fila.append(1)
>>> fila.append(3.5)
>>> print fila
['promedio', 3, 5.5, 'notas', 1, 3.5]
>>>
>>> atender = fila.pop(0)
>>> print atender
Promedio
>>>
>>> print fila
[3, 5.5, 'notas', 1, 3.5]
>>>
```

Pilas

Otra de las estructuras de datos clásicas son las **pilas** (*stacks*). En las pilas, los elementos nuevos se colocan **al frente** de la secuencia y también se retiran desde **el frente**. Es decir, tienen el comportamiento de las pilas de platos al lavar la loza: primero enjabonamos los platos, uno a uno, y los vamos **apilando**; luego abrimos el agua y los enjuagamos, desde el que está encima hasta el que está en la base. Las pilas aparecieron hace muchos años en la computación porque es la estructura que se requiere para **administrar llamadas a subrutinas**. Por ejemplo, supongamos que una subrutina A, invoca una subrutina B, la que invoca una subrutina C, la que invoca una subrutina D; cuando la subrutina D termina su ejecución, el control de ejecución debe volver a la subrutina que la llamó, es decir a la subrutina C; cuando la subrutina C termina, el control ha de devolverse a la subrutina B; sólo cuando la subrutina B termina, el control vuelve a la primera subrutina A. Note que en la recursión A = B = C = D, pero el mecanismo es el mismo y permite devolver el control al estado de ejecución pendiente que corresponde.

En Python, las pilas también están implementadas como objetos de clase `list`, como se muestra en el Ejemplo 2.

Ejemplo 2

```
>>> pila = list()
>>> pila.append("fact(5)")
>>> pila.append("fact(4)")
>>> pila.append("fact(3)")
>>> pila.append("fact(2)")
>>> pila.append("fact(1)")
>>> pila.append("fact(0)")
>>> print pila
['fact(5)', 'fact(4)', 'fact(3)', 'fact(2)', 'fact(1)', 'fact(0)']
>>>
>>> finalizado = pila.pop()
>>> print finalizado
fact(0)
>>>
>>> finalizado = pila.pop()
>>> print finalizado
fact(1)
>>>
>>> print pila
['fact(5)', 'fact(4)', 'fact(3)', 'fact(2)']
>>>
```

Conjuntos

Python también dispone de **conjuntos** en forma nativa. Aquí "conjunto" se ha de entender como en su **definición matemática**: un grupo finito y sin orden de elementos **únicos** (no pueden existir dos elementos iguales en un conjunto). En Python se agrega otra restricción: los elementos deben ser además **inmutables**.

Los conjuntos en Python están implementados como la clase `set`. Se puede especificar una secuencia de valores como parámetro, desde donde se toman los elementos del nuevo conjunto.

Ejemplo 3

```
>>> cjto = set()
>>> print cjto
set([])
>>>
>>> cjto = set("Mississipi")
>>> print cjto
set(['i', 'p', 's', 'M'])
>>>
>>> cjto = set((1, 5, 0, 9, 3, 0, 2, 1))
>>> print cjto
set([0, 1, 2, 3, 5, 9])
>>>
```

Notemos, en el Ejemplo 3, que un conjunto vacío es representado como `set([])` por el intérprete de Python. También que cuando un conjunto se crea a partir de un string, éste es descompuesto en los caracteres que contiene. En todos los casos, elementos repetidos son ignorados. Debemos recordar que los elementos en un conjunto **no tienen un orden** preestablecido, por lo que no se puede acceder a sus elementos usando indexación, como en el caso de listas y tuplas, pero sí se puede iterar por los sus elementos con un ciclo `for-in` (Ejemplo 4).

El Ejemplo 4 también muestra que podemos agregar elementos a un conjunto mediante el método `.add(elem)` y eliminar elementos a través del método `.discard(elem)`. Notemos que pedir al conjunto que elimine un elemento que no existe, no tiene efecto alguno.

Ejemplo 4

```
>>> cjto = set((1, 5, 0, 9, 3, 0, 2, 1))
>>> cjto
set([0, 1, 2, 3, 5, 9])
>>>
>>> cjto.add(4)
>>> cjto.add(5)
>>> cjto.add(6)
>>> cjto
set([0, 1, 2, 3, 4, 5, 6, 9])
>>>
>>> cjto.discard(0)
>>> cjto.discard(1)
>>> cjto.discard(0)
>>>
>>> for elem in cjto:
      print elem,


2 3 4 5 6 9
>>>
```

La clase `set` además provee métodos para las operaciones propias de la teoría de conjuntos como son la **unión**, **intersección**, **diferencia asimétrica** (considera a todos los elementos que pertenecen al primer conjunto, excepto a los que pertenecen al segundo), **diferencia simétrica** (considera a todos los elementos que pertenecen exclusivamente a cada uno de los conjuntos). Estos métodos entregan un nuevo conjunto, como se puede observar en el Ejemplo 5. Es importante destacar que las operaciones usadas en el Ejemplo 5 son conmutativas, con la excepción de la diferencia asimétrica, que como su nombre lo indica, entrega resultados distintos dependiendo del orden de operandos.

Ejemplo 5

```
>>> cjto1 = set([1,2,3,4])
>>> cjto2 = set([3,4,5,6])
>>>
>>> cjto1.union(cjto2)
set([1, 2, 3, 4, 5, 6])
>>> cjto1.intersection(cjto2)
set([3, 4])
>>> cjto1.difference(cjto2)
set([1, 2])
>>> cjto2.difference(cjto1)
set([5, 6])
>>> cjto1.symmetric_difference(cjto2)
set([1, 2, 5, 6])
>>>
```

Pregunta 1

Resuelve con tu grupo la pregunta 1 de la actividad.

Diccionarios

Otra estructura nativa de Python son los **diccionarios**. Estas estructuras contienen elementos **atómicos o compuestos** (como tuplas, listas, conjuntos u otros diccionarios), que están asociados a un valor **inmutable** que los **indexa**, denominado **llave** (*key*). Esta característica es la que da origen a su nombre: en un diccionario (o enciclopedia, o libreta de direcciones) todas las palabras que comienzan con la letra A están incluidas en una sección dedicada a la letra A; lo mismo para la letra B, la letra C, etc. La gran ventaja de esta organización es que si buscamos, por ejemplo, una palabra que comienza con la letra T, **no debemos buscar secuencialmente**, palabra por palabra, sino que nos podemos ir directamente a la sección con las palabras que comienzan con T. Esto hace el proceso de **búsqueda más eficiente**.

Los diccionarios están disponibles en Python como la clase `dict`. Existen varias formas de especificar un elemento y su índice a un diccionario, pero por claridad de código sólo utilizaremos la alternativa en que **se especifica el par (llave, valor)**, como se hace en el Ejemplo 6. Cuando se crea un diccionario, podemos especificar una secuencia de ítems iniciales.

Ejemplo 6

```
>>> notas = dict()
>>> notas
{}
>>>
>>> notas = dict([('Juan',5.0), ['Pedro', 2.5], ('Maria',6.5)])
>>> notas
{'Juan': 5.0, 'Pedro': 2.5, 'Maria': 6.5}
>>>
>>> notas['Sofia'] = 4.0
>>> notas['Juan'] = 3.0
>>> print notas
{'Juan': 3.0, 'Pedro': 2.5, 'Sofia': 4.0, 'Maria': 6.5}
>>>
```

En el Ejemplo 6, podemos ver que un ítem es mostrado como `<llave>: <valor>` por el intérprete de Python, mientras que los diccionarios se muestran como una secuencia de ítems entre paréntesis de llave ({ }). También vemos que para **agregar un elemento**, e utiliza la sintaxis de **indexación** en que el índice es una nueva llave. Cuando se especifica un elemento con una llave que ya existe, el valor asociado a esa llave se sobre escribe (las llaves son únicas y no pueden repetirse).

Ejemplo 7

```
>>> >>> notas = dict()
>>> notas['Juan'] = 5.0
>>> notas['Pedro'] = 2.5
>>> notas['Maria'] = 6.5
>>> notas['Sofia'] = 4.0
>>>
>>> for llave in notas.keys():
      print llave, "obtuvo un", notas[llave]


Juan obtuvo un 5.0
Pedro obtuvo un 2.5
Sofia obtuvo un 4.0
Maria obtuvo un 6.5
>>>
>>> notas['German']

Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    notas['German']
KeyError: 'German'
>>>
>>> notas.has_key('German')
False
>>> notas.has_key('Maria')
True
>>>
```

La indexación también nos permite **acceder a un elemento** del diccionario. Pero, al igual que en los conjuntos, los elementos **no tienen un orden determinado** y no podemos usar posiciones, como lo hacemos con las listas o tuplas. El índice que debemos usar es **la llave del elemento** buscado. La operación de indexación genera una **excepción** si la llave especificada no está incluida en el diccionario. Existe el método `.has_key(llave)` para poder consultar si una llave está o no está presente en un diccionario. Podemos iterar sobre las llaves de un diccionario por medio de método `.keys()`, que devuelve una copia de la lista de llaves del diccionario. Estas características pueden verse en el Ejemplo 7. Notemos que las llaves tampoco tienen un orden definido. Si se requiere recorrer los elementos de un diccionario en un orden determinado, debe procurarse dar ese orden a la lista de llaves previo a la iteración.

Pregunta 4

Resuelve la pregunta 2 de la actividad con la ayuda de tu grupo de trabajo.