Introducción a la Programación

Prof. Agustín Gravano

Primer semestre de 2022

Clase teórica 11: Listas (continuación)

Listas (`List[T]`) · Operaciones básicas

Dadas las variables `a,b` de tipo `List[T]`; `x,y,z` de tipo `T`; `i` de tipo `int`:

- `a = list()` Crea una lista de `T` vacía.
- `a = []` Crea una lista de `T` vacía.
- `a = [x,y,z]` Crea una lista de `T` no vacía.
- `len(a)` Devuelve la longitud de la lista.
- `x in a` Consulta pertenencia de un elemento en la lista.
- `a.append(x)` Agrega un elemento al final de la lista.
- `a.pop()` Elimina el elemento de la última posición y lo devuelve.
- `a[i]` Lee el valor en la `i`-ésima posición.
- `a[i] = x` Sobreescribe la `i`-ésima posición.
- `a.insert(i,x)` Inserta un elemento en la `i`-ésima posición.
- `a.pop(i)` Elimina la `i`-ésima posición y devuelve su valor.
- `a == b` Compara dos listas.
- `a + b` Concatena dos listas.
- `a * i` Concatena `i` veces seguidas la lista a.

¿Qué imprimen los siguientes programas?

Pensar las respuestas en papel, sin ayuda de la computadora.

```python
# Programa 1
n:int = 99
m:int = n
n = n + 1
print(n, m)
```

```python
# Programa 2
s:str = 'qwe'
t:str = s
s = s + 'rty'
print(s, t)
```

```python
# Programa 3
a:List[int] = [1, 2]
b:List[int] = a
a.append(3)
print(a, b)
```

Tratemos de entender a qué se deben estos comportamientos distintos.

Tipos inmutables

Veamos el siguiente código:

```
1  n:int = 5
2  n = n + 1
```

En la línea 1, la variable `n` toma el valor 5 (decimos que "refiere al objeto entero 5").

En la línea 2, evaluar la expresión `n + 1` resulta en 6, un nuevo valor que se asigna a `n`. La variable `n` ahora refiere a un nuevo objeto. El 6 no tiene nada que ver con el 5. No es "5 incrementado", ni nada parecido.

No hay forma de modificar un entero. Sí podemos combinarlo con otros enteros mediante operaciones que resultan en nuevos enteros: 5 * 2 resulta en el nuevo entero 10.

Por eso, decimos que el tipo `int` es inmutable.

Tipos inmutables

Con strings pasa lo mismo:

```
1  s:str = 'qwe'
2  s = s + 'rty'
```

`'qwerty'` es un nuevo valor (un nuevo objeto) de tipo `str`. No es una modificación de `'qwe'`.

`int`, `str`, `float` y `bool` son tipos inmutables. Sus valores (objetos) no pueden modificarse.

Esto explica el comportamiento de los dos primeros programas:

```
1  # Programa 1
2  n:int = 99
3  m:int = n
4  n = n + 1
5  print(n, m)
```

```
1  # Programa 2
2  s:str = 'qwe'
3  t:str = s
4  s = s + 'rty'
5  print(s, t)
```

Programa 1: Luego de la línea 3, las variables `n` y `m` refieren al mismo objeto (99). En la línea 4, `n` pasa a referir a un nuevo objeto (100), sin relación con el objeto referido por `m`. Ídem en el programa 2.

Las listas son **mutables**

¿Qué sucede entonces con las listas?

```
1  a:List[int] = [1, 2, 3]
2  a.append(4)
```

En la línea 2, la función (o método) append modifica a la lista referida por la variable a. Lo mismo ocurre en la siguiente asignación:

```
3  a[1] = 8
```

La lista referida por a sigue siendo el mismo objeto después de las líneas 2 y 3, pero modificado.

Como es posible modificar listas, decimos que son mutables.

Esto explica el comportamiento del programa 3. Luego de la línea 3, las variables a y b refieren a la misma lista, que se modifica en la línea 4. En este caso, a y b siguen refiriendo al mismo objeto.

```
1  # Programa 3
2  a:List[int] = [1, 2]
3  b:List[int] = a
4  a.append(3)
5  print(a, b)
```

Tipos mutables e inmutables · Ejercicio

Determinar qué imprime el siguiente programa (pensarlo en papel antes de ejecutarlo en Python):

```
1  p:bool = True
2  q:bool = p
3  p = False
4  print(p, q)
5
6  bs1:List[bool] = [True, False]
7  bs2:List[bool] = bs1
8  bs2.pop()
9  bs2.pop()
10 print(bs1, bs2)
```

¿Qué imprimen los siguientes programas?

Pensar las respuestas en papel, sin ayuda de la computadora.

```python
# Programa 4
def f(m:int):
    m = m + 1
n:int = 1000
f(n)
print(n)
```

```python
# Programa 5
def g(t:str):
    t = t + 'ches'
s:str = 'no'
g(s)
print(s)
```

```python
# Programa 6
def h(b:List[bool]):
    b.append(False)
a:List[bool] = [True, True]
h(a)
print(a)
```

Pasaje de argumentos

Al pasar argumentos a una función, vale el mismo razonamiento que en las asignaciones a variables.

Programa 4: En la invocación `f(n)` de la línea 5, el parámetro `m` se inicializa con el mismo valor de la variable `n` (1000). La línea 3 asigna a `m` un nuevo valor (1001), lo cual no modifica al valor 1000. Por ese motivo, `n` no ve alterado su valor, que sigue siendo 1000.

Programa 5: En la invocación `g(s)` de la línea 5, el parámetro `t` se inicializa con el mismo valor de la variable `s` (`'no'`). La línea 3 asigna a `t` un nuevo valor (`'noches'`), lo cual no modifica al valor `'no'`. Por ese motivo, `s` no ve alterado su valor, que sigue siendo `'no'`.

Programa 6: En la invocación `h(a)` de la línea 5, el parámetro `b` se inicializa con el mismo valor de la variable `a` (`[True, True]`). La línea 3 agrega `False` al final de la lista. Esto modifica a la lista, que también es referida por `a`. Por ese motivo, `a` ve alterado su valor, que pasa a ser `[True, True, False]`.

Pasaje de argumentos · Ejercicio

Determinar qué imprime por pantalla este programa (pensarlo en papel, sin la ayuda de la computadora):

```
1  def cuenta_regresiva(n:int) -> str:
2      vr:str = ''
3      while n > 0:
4          vr = vr + str(n)
5          n = n - 1
6      return vr
7
8  def evanesco(cosas:List[str]):
9      while len(cosas)>0:
10         cosas.pop()
11
12 num:int = 5
13 print(num, cuenta_regresiva(num), num)
14
15 items:List[str] = ['pergamino', 'poción', 'serpiente']
16 evanesco(items)
17 print(items)
```

Funciones con efectos colaterales

Los tipos mutables permiten que las funciones tengan efectos colaterales.

Además de (opcionalmente) retornar un valor, las funciones pueden modificar los argumentos mutables, alterando así el estado del programa.

¡Hay que tener mucho cuidado con esto!

Ejemplo (posiblemente catastrófico):

```
1  def contar_ocurrencias(x:str, xs:List[str]) -> int:
2      ''' Requiere: nada.
3          Devuelve: cantidad de ocurrencias de x en xs.'''
4      vr:int = 0
5      while len(xs)>0:
6          if xs.pop()==x:
7              vr = vr + 1
8      return vr
9
10 mi_lista = ['a', 'b', 'c', 'a', 'x']
11 print(contar_ocurrencias('a', mi_lista))
12 print(contar_ocurrencias('a', mi_lista))
```

Funciones con efectos colaterales

Importante: Los efectos colaterales también deben describirse en la especificación de la función.

```
1  def contar_ocurrencias(x:str, xs:List[str]) -> int:
2      '''
3      Requiere: nada.
4      Devuelve: cantidad de ocurrencias de x en xs.
5      Modifica: xs queda vacía.
6      '''
7      vr:int = 0
8      while len(xs)>0:
9          if xs.pop()==x:
10             vr = vr + 1
11     return vr
12
13 mi_lista = ['a', 'b', 'c', 'a', 'x']
14 print(contar_ocurrencias('a', mi_lista))
15 print(contar_ocurrencias('a', mi_lista))
```

Cuando una función no tenga efectos colaterales, puede omitirse la cláusula `Modifica` en su especificación (como veníamos haciendo).

Repaso de la clase de hoy

- Tipos inmutables: `bool`, `int`, `float`, `str`.
- Tipos mutables: listas (por ahora).
- Asignación y pasaje de argumentos de tipos mutables e inmutables.
- Funciones con efectos colaterales. Cláusula `Modifica`.

Bibliografía complementaria:

- APPP2, secciones 7.6, 8.7 a 8.16.
- HTCSP3, secciones 8.8, 11.6 a 11.20 (no prestar atención todavía al uso de `for`).

Con lo visto, ya pueden resolver hasta el Ejercicio 8 (inclusive) de la Guía de Ejercicios 5.