Clase 15: Mutación y Aliasing

(Capítulo 11 del apunte)

Hasta ahora, todas nuestras funciones reciben parámetros y retornan un valor de acuerdo a éstos

- Para los mismos parámetros se devuelve el mismo valor
- Esto facilita muchísimo hacer testing

Sin embargo, hay programas que requieren que una función pueda tener una cierta **memoria** sobre las acciones sobre los datos.

Por qué las funciones "con memoria" son útiles

Disponer de memoria puede ser muy útil

- Servicios al usuario para administrar información

Pero, la programación es **más compleja y requiere ser cuidadoso**

- Ya no es fácil hacer testing (no conocemos a priori la respuesta correcta)
- Un cambio realizado a la memoria puede tener impacto en distintas partes del programa

¿Cómo implementamos una función con memoria?

Usando **variables de estado**. Una variable de estado es una variable que almacena (recuerda) valores manteniéndolos entre distintas llamadas a la función.

- En Python, una variable de estado se implementa como una variable **global**
- Se puede acceder desde cualquier función y desde el programa principal
- Su valor puede **mutar** (actualizarse)

Ambiente de una variable

El ambiente de una variable (o *scope*) es la parte del código en donde una variable puede ser accesada

- **Variables locales**:
  - Se definen dentro de una función
  - Su scope es la función: una vez que termina la función, desaparece la variable
- **Variables de estado** (o **globales**):
  - Se definen a nivel de la línea de comando

- Su **scope** es el programa completo

Ejemplo: Definición y uso de variable de estado:

```
# Variable de estado
contador = 0

# procedimiento: None -> None
# efecto: modifica variable contador incrementando su valor en 1
def procedimiento():
    global contador
    uno = 1
    contador = contador + uno

# Programa principal
print(str(contador))


procedimiento()
print(str(contador))
procedimiento()
procedimiento()
procedimiento()
print(str(contador))


print(str(contador))
procedimiento()
procedimiento()
procedimiento()
print(str(contador))
```

Ejemplo: Implementar una agenda AI2E con memoria con las siguientes operaciones

En nuestro ejemplo de abreviaciones y significados (la aplicación AI2E de la clase pasada), cada vez que necesitamos consultar o modificar la agenda debemos pasarla como parámetro o recibirla de vuelta.

1. Siempre debemos poner como segundo parámetro la variable agenda. Por ejemplo:

   ```
   print("El significado de uwu es", buscar("uwu", agenda))
   ```

2. Además, cuando cambiamos algo en la agenda, la función produce *otra agenda* nueva que *debemos guardar en una nueva variable*. Por ejemplo:

```
agenda = cambiar("uwu","lindo",agenda)
```

Fijarse que sobre-escribimos la variable `agenda` con lo que retorna la función cambiar.

Agenda AI2C con Memoria

Dado que tenemos siempre una sola agenda con las abreviaciones, nos gustaría poder consultarla y cambiarla sin estar incluyendo en parámetro agenda todas las veces.

- Nos gustaría, por ejemplo, consultarla sin indicar qué agenda estamos usando:

```
print("El significado de uwu es", buscar("uwu"))
```

- Y si ahora cambiamos el significado de "uwu" a "lindo",

```
cambiar("uwu","lindo")
```

- Pero, al volver a consultarla, debiera "acordarse" del cambio:

```
print("El nuevo significado de uwu es", buscar("uwu"))
```

debiera mostrar

```
El nuevo significado de uwu es lindo
```

```
# registro: abrev(str) sig(str)
estructura.crear("registro", "abrev sig")

# buscar: str -> str
# devuelve significado de una abreviacion en agenda (o None si no está)

# agregar: str str -> None
# efecto: agrega registro con abreviacion y significado a la agenda

# borrar: str -> None
# efecto: borra de la agenda el registro con la abreviacion indicada
```

Debemos decidir qué estructura utilizaremos como memoria para nuestra agenda

- Usaremos una **lista de registros**

Nota: por simplicidad, la lista no estará ordenada.

```
from lista import *

# registro: abrev(str) sig(str)
estructura.crear("registro", "abrev sig")

# Variable de estado
agendaAbreviaciones = listaVacia



# buscar: str -> str
# devuelve significado de una abreviacion en agenda (o None si no está)
def buscar(abr):
    global agendaAbreviaciones
    return buscarRec(abr, agendaAbreviaciones)

# buscarRec: str lista(registro) -> str
# devuelve significado de una abreviacion en agenda global (o None si no está)
def buscarRec(abr, unaLista):
    if vacia(unaLista):
        return None
    elif cabeza(unaLista).abrev == abr:
        return cabeza(unaLista).sig
    else:
        return buscarRec(abr, cola(unaLista))


# Ejemplo de uso
significado = buscar("uwu")
if significado == None:
  print("uwu no está")
else:
  print("uwu significa",significado)


# agregar: str str -> None
# efecto: agrega un nuevo registro con abreviacion y significado
#         al comienzo de la (variable glogal) lista agendaAbreviaciones
def agregar(abr,sig):
  global agendaAbreviaciones
  agendaAbreviaciones = lista(registro(abr,sig), agendaAbreviaciones)



agregar("uwu","lindo")
print(buscar("uwu"))
```

```
print(buscar("bff"))
agregar("bff","mejor amigx")
print(buscar("bff"))
```

Propuesto: escribir la función `borrar()`

¿Cómo hacer testing?

- Definir un estado conocido a la(s) variable(s) de estado
- Invocar las funciones que se están probando y comprobar que, para el estado definido, devuelven la respuesta correcta y/o modifican la variable de estado en la forma esperada

Testing para funciones buscar y agregar:

```
# Variable de estado
agendaAbreviaciones = listaVacia

# Test de todas las funciones, inicialmente agendaAbreviaciones == listaVacia

agregar("uwu", "tierno")
agregar("bff", "mejor amigx")

assert buscar("uwu") == "tierno"
assert buscar("bff") == "mejor amigx"
assert buscar("afk") == None

agregar("afk", "lejos del teclado")
agregar("rofl", "riendome")

assert buscar("afk") == "lejos del teclado"
assert buscar("rofl") == "riendome"
```

Estructuras de datos mutables

Las estructuras de datos (datos compuestos) que hemos usado hasta ahora **no son mutables**

- Los valores de los campos de la estructura no se pueden modificar

Ahora veremos como definir estructuras mutables con el modulo estructura

```
import estructura
```

```
estructura.crear("deportista", "nombre deporte") # estructura NO mutable

p = deportista("Michael Jordan", "basquetbol")
print(p.deporte)



p.deporte = "beisbol"


estructura.mutable("deportista", "nombre deporte") # estructura mutable

p = deportista("Michael Jordan", "basquetbol")
print(p.deporte)



p.deporte = "beisbol"
print(p.deporte)
```

Aliasing

```
import estructura
estructura.mutable("cliente", "nombre saldo")
juan = cliente("Juan", 1000)
pedro = juan
print("El saldo de Juan es", juan.saldo)

# Modificamos el nombre y saldo de Pedro
pedro.nombre = "Pedro"
pedro.saldo = 0

print("El saldo de Juan es", juan.saldo)
```

Aliasing: Por qué ocurre y cómo prevenirlo

Datos Básicos

- Los tipos de datos básicos (`int, float, bool, str`) se pasan por **valor**
- Esto significa que el valor es "copiado" al guardarse en otra variable

```
numero=1
otroNumero=numero
numero=2


numero
```

```
numero


otroNumero
```

Datos Compuestos

- Los tipos de datos compuestos **NO se copian**, se pasan por **referencia**
- Esto significa que un dato compuesto es una estructura almacenada memoria
- La variable asociada a la estructura es solo una referencia (identificador o dirección) de donde se encuentra el dato en memoria (permite accederlo)
- Al asignar una variable que contiene una referencia sólo se está haciendo una copia de la referencia y no del dato mismo (es como copiar la dirección de una casa, y no la casa misma)
- Esto es denominado **ALIASING**

En general NO es problema cuando usamos estructuras NO mutables. De hecho, ya lo hemos usado así.

El problema ocurre al usar estructuras MUTABLES.

```
import estructura
estructura.mutable("cliente", "nombre saldo")
juan = cliente("Juan", 1000)
alias = juan
juan.saldo = 0


print(juan.saldo)


print(alias.saldo)
```

A partir de ahora llamaremos **referencias** a los identificadores de datos compuestos

- La referencia nos permite acceder a los atributos del dato compuesto
- Intuitivamente: la referencia es una flecha que apunta al dato compuesto

Podemos tener dos referencias al mismo dato compuesto

- La segunda referencia es un alias de la primera

Un ejemplo no-trivial de mutación:

Poder usar estructuras mutables es muy útil, pero es fuente potencial de diversos errores.

```
from lista import *
```

```
estructura.mutable("deportista", "nombre deporte")

q = lista(deportista("Michael Jordan", "basquetbol"), None)

r = lista(q.valor, lista(q.valor.deporte, None))

q.valor.deporte = "beisbol"

print(r.valor.deporte) # que debe mostrar?

print(r.siguiente.valor)
```

¿Copiar sin Aliasing?

Si queremos copiar estructuras en forma efectiva debemos poner esfuerzo en crear una nueva estructura. A esto se le llama *copia efectiva*.

```
juan=cliente("Juan",1000)
juan2=cliente(juan.nombre,juan.saldo) #nuevo cliente

print("Nombre:", juan.nombre)
print("Saldo:", juan.saldo)

juan.saldo = 0

print("Nombre:", juan2.nombre)
print("Saldo:", juan2.saldo)
```