Introducción a la Programación

Prof. Agustín Gravano

Primer semestre de 2022

Clase teórica 20: Clases y objetos

Problema: Frutería Online

Queremos programar una frutería.

De cada fruta nos interesa guardar:

- Nombre (un string)
- Precio por kilo (un float)
- Estaciones en las cuales está disponible (un conjunto de strings)

Ejemplo:

- Banana de Ecuador
- $ 150.0 por kilo
- disponible en primavera, otoño e invierno

```python
class Fruta:    ## Definición de la clase Fruta.
    def __init__(self, n:str, p:float, es:Set[str]):
        ''' Inicializa una fruta con nombre n, precio p, estaciones es. '''
        self.nombre:str = n
        self.precio:float = p
        self.estaciones:Set[str] = es

    def disponible_en(self, estacion:str) -> bool:
        ''' Determina si esta fruta suele estar disponible en la estación dada.
        Requiere: estacion es 'primavera', 'verano', 'otoño' o 'invierno'.
        Devuelve: True si estacion está en las estaciones de la fruta;
                  False en caso contrario.'''
        return estacion in self.estaciones

    def __repr__(self) -> str:
        ''' Devuelve una representación string de la fruta. '''
        return self.nombre + ' ($' + str(self.precio) + '/kg)'

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PRI:str = 'primavera'
VER:str = 'verano'
OTO:str = 'otoño'
INV:str = 'invierno'

## Creación de 3 objetos de la clase Fruta.
p:Fruta = Fruta('Pera Williams', 70.0, {VER})
u:Fruta = Fruta('Uva verde', 60.0, {VER})
b:Fruta = Fruta('Banana de Ecuador', 150.0, {PRI, OTO, INV})

frutas:List[Fruta] = [b,p,u]
```

Programación Orientada a Objetos (OOP)

OOP es un paradigma de programación que permite definir tipos usando clases para modelar problemas con objetos que funcionan como representaciones de entidades del mundo real.

Lenguajes: Java, Smalltalk, C++, Python, Ruby, etc.

Una clase es un "plano" que define las operaciones y la representación interna de los objetos del nuevo tipo.

Los objetos son instancias del nuevo tipo que se manipulan con las operaciones definidas en la clase.

Crédito de los dibujos: Klinko

Método de inicialización y atributos

Al crear un objeto de una clase, se ejecuta un método especial llamado `__init__`, que inicializa a dicho objeto:

```
def __init__(self, n:str, p:float, es:Set[str]):
    '''Inicializa una fruta con nombre n, precio p, estaciones es.'''
    self.nombre:str = n
    self.precio:float = p
    self.estaciones:Set[str] = es
```

Al invocar al constructor `Fruta`, los argumentos se pasan a `__init__`:

```
p:Fruta = Fruta('Pera Williams', 70.0, {'verano'})
print(p.nombre)                 # imprime 'Pera Williams'
print(p.precio)                 # imprime 70.0
print(p.estaciones)             # imprime {'verano'}
```

- El parámetro `self` refiere al objeto que se está inicializando.
- `nombre`, `precio` y `estaciones` son atributos del objeto.
- Se accede a los atributos con la sintaxis *`objeto.atributo`*, para lectura y escritura. Ejemplos: `self.nombre`, `p.precio`.

Mutabilidad

Los objetos de las clases que definimos son mutables.

```
p:Fruta = Fruta('Pera Williams', 70.0, {'verano'})
q:Fruta = p
print(p.precio, q.precio)           # imprime 70.0 70.0

p.precio = 69.99
print(p.precio, q.precio)           # imprime 69.99 69.99
```

Métodos

Un método es una función definida dentro de una clase y que se invoca sobre objetos de esa clase.

```
def disponible_en(self, estacion:str) -> bool:
    ''' Determina si esta fruta suele estar disponible en la estación dada.
    Requiere: estacion es 'primavera', 'verano', 'otoño' o 'invierno'.
    Devuelve: True si estacion está en las estaciones de la fruta;
              False en caso contrario.'''
    return estacion in self.estaciones
```

Los métodos se invocan siempre sobre un objeto:

```
p:Fruta = Fruta('Pera Williams', 70.0, {'verano'})
print(p.disponible_en('verano'))      # imprime True
print(p.disponible_en('invierno'))    # imprime False
```

- El parámetro `self` refiere al objeto sobre el cual se invocó el método.
- Los métodos son funciones. Entonces, la invocación a un método siempre va con paréntesis, aunque el método no reciba argumentos.

Impresión de un objeto

Por defecto, `print(f)` imprime algo inservible como `<__main__.Fruta object at 0x7f0027401130>`. Eso es el nombre de la clase, seguido de la dirección de memoria en la que se encuentra. (!?)

Para indicar cómo se debe imprimir un objeto, implementamos el método `__repr__` en la clase correspondiente:

```
def __repr__(self) -> str:
    ''' Devuelve una representación string de la fruta. '''
    return self.nombre + ' ($' + str(self.precio) + '/kg)'
```

Ahora podemos imprimir una fruta, y también convertirla a string:

```
p:Fruta = Fruta('Pera Williams', 70.0, {'verano'})
print(p)          # imprime 'Pera Williams ($70.00/kg)'
s:str = str(p)
print(s[:4])      # imprime 'Pera'
```

Observación: También existe el método __str__ para convertir a strings. Aconsejamos trabajar con __repr__, que es más general. Acá hay una buena explicación de sus diferencias: https://www.analyticslane.com/2020/07/03/diferencias-entre-str-y-repr-en-python/ En cualquier caso, este es un tema menor, muy específico de Python.

Ordenar una lista de objetos

Si queremos ordenar una lista de objetos con `sort`, necesitamos definir la comparación por menor entre esos objetos. En nuestro ejemplo, elegimos basar la comparación en el precio de las frutas:

```
class Fruta:
    # ...

    def __lt__(self, other) -> bool:
        ''' Devuelve True si self < other; False en caso contrario.'''
        return self.precio < other.precio

b:Fruta = Fruta('Banana de Ecuador', 150.0, {PRI, OTO, INV})
p:Fruta = Fruta('Pera Williams', 70.0, {VER})
u:Fruta = Fruta('Uva verde', 60.0, {VER})
print(p < u)      # imprime False
print(u < p)      # imprime True

frutas:List[Fruta] = [b, p, u]
frutas.sort()
print(frutas)
```

La última línea imprime la lista de frutas ordenada por precio:

```
[Uva verde ($60.0/kg), Pera Williams ($70.00/kg), Banana de Ecuador ($150.00/kg)]
```

Otros métodos

Hay otros métodos que pueden implementarse para darle semántica a la comparación entre objetos.

- `__eq__(self, other)`: `objeto_1 == objeto_2`
- `__ne__(self, other)`: `objeto_1 != objeto_2`
- `__lt__(self, other)`: `objeto_1 < objeto_2`
- `__gt__(self, other)`: `objeto_1 > objeto_2`
- `__le__(self, other)`: `objeto_1 <= objeto_2`
- `__ge__(self, other)`: `objeto_1 >= objeto_2`

Observación muy molesta: Dentro de la definición de una clase `MiClase`, por defecto no se puede usar `MiClase` en un type hint. **Ejemplo:**

```
def MiClase:
    def f(self, other:MiClase) -> MiClase:
        # Esto arroja NameError: name 'MiClase' is not defined
```

Entonces, omitimos los type hints dentro de la clase que se está definiendo.

Repaso de la clase de hoy

- Clases y objetos
- Atributos y métodos
- Los objetos de las clases que definimos son mutables
- Método de inicialización `__init__`
- Métodos especiales `__repr__`, `__lt__`, `__eq__`, etc.

Bibliografía complementaria:

- APPP2, capítulos 12, 13 y 14 (excepto 14.9).
- HTCSP3, capítulos 15 y 16.

Con lo visto, ya pueden resolver hasta el Ejercicio 2 (inclusive) de la Guía de Ejercicios 9.