Ayudantía 04

Repaso I1

Jaime Castro Felipe Garrido

Departmento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

IIC2233, 2015-1

Tabla de Contenidos

OOP

Clases

Definición

Objeto: es una colección de datos que además tiene asociado comportamientos.

Definición

Clase: Describe a un objeto en OOP.

Pregunta

¿Recuerdan la idea de **encapsulamiento**?

OOP

Formas de encapsulamiento

Convención de uso interno

Anteponer un guión bajo al nombre de un método o variable.
Ejemplo: `_spam`.

Name Mangling

Cualquier variable o método con dos guiones bajos y con hasta un guión bajo al final no se puede llamar directamente desde afuera de la clase.
Ejemplo: `__spam`

OOP

Herencia

```
1 class Vehiculo:
2     # ...
3
4 class Auto(Vehiculo):
5     # ...
6
7 class Camion(Vehiculo):
8     # ...
```

Preguntas conceptuales

- ¿Cuáles son las clases Padre? ¿Y las hijas? ¿A qué nos referimos con *super clase*?
- ¿Cuál es la super clase de Vehículo?
- ¿Qué gano con esto? Las clases hijas reciben datos y comportamiento de su padre

OOP

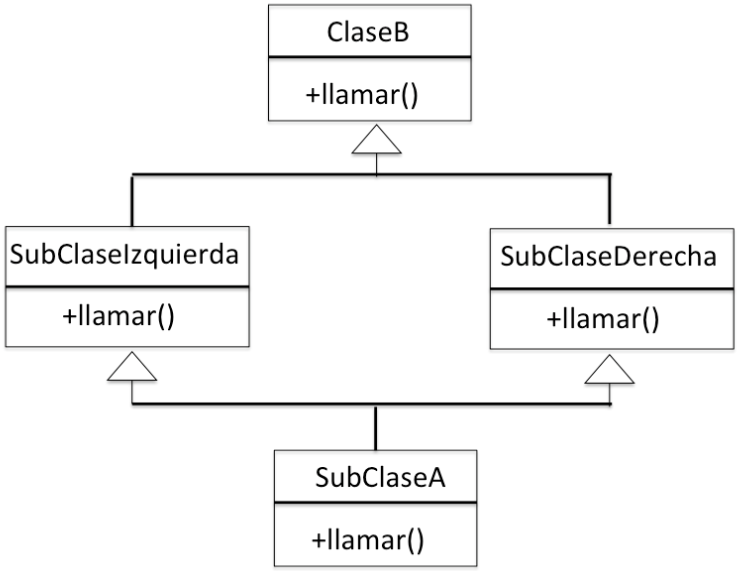
Herencia Múltiple

En Python es posible heredar de más de una clase a la vez.

```
1 class Investigador:
2     # ...
3
4 class Docente:
5     # ...
6
7 class Academico(Docente, Investigador):
8     # ...
```

OOP

Problema del diamante



- Este problema surge cuándo dos clases heredan de otra tercera y, además una cuarta clase tiene cómo padre a las dos últimas.
- ¿De dónde heredo mi método?
- Python establece un orden lineal entre las clases (MRO) con el algoritmo C3

OOP

Problema del diamante

Pregunta

¿Se producía este problema para el ejemplo del Académico, Docente e Investigador? Si

Pregunta

- ¿Qué implicancia hay para la inicialización de los objetos? Podríamos inicializar varias veces el objeto
- ¿Cómo lo solucionamos? Cada clase debe preocuparse de llamar a inicializar a la clase que la "precede"

OOP

Polimorfismo

- Se llama Polimorfismo a la posibilidad de obtener distintos comportamientos mediante la invocación a métodos de un mismo nombre, pero de clases distintas.
- **¿Qué hay que hacer?** Basta con redefinir la función que tenía el padre. Esto se conoce como overriding

OOP

Overriding de operadores

- Cuando sumamos dos listas o dos números, en realidad se llama al método `__add__(self, otroObjeto)` de la clase `int` o `list`. Sucede algo análogo cuando comparamos, por ejemplo, con <.
- Nosotros podemos personalizar cualquier clase con estos métodos para usar estos operadores.

Algunos métodos

- `__add__(self, obj)`: +
- `__lt__(self, obj)`: <
- `__gt__(self, obj)`: >
- `__repr__(self)`: Permite personalizar la forma en que se imprime el objeto.

OOP

Duck Typing

- "If it walks like a duck and quacks like a duck then it is a duck"
- La idea es que no importa qué tipo de objeto se tenga, sino que tenga las mismas acciones
- Esta característica está presente sólo en algunos lenguajes.

OOP

Properties

- Son atributos a los que se les asigna los métodos getter, setter y deleter de manera manual (en código).
- **¿Para qué pueden servir?** Para agregar comportamiento al setear, obtener o eliminar datos. **Ejemplos:**
  - Al setear una variable, no quiero que se haga con ciertos valores.
  - Un valor es calculado. Quiero que se calcule la primera vez que se pide, y el resto de las veces ocupar ese valor ya calculado.
  - Se requiere hacer varias operaciones al eliminar un valor.
  - En general, no está mal usarlas si no se dan estos casos
- Podemos establecerlas usando la keyword `property` o usando decoradores

OOP

Propierties

Ejemplo del material del curso con decoradores:

```
1  class Color:
2      def __init__(self, rgb_code, nombre):
3          self.rgb_code = rgb_code
4          self._nombre = nombre
5      @property
6      def nombre(self):
7          return self._nombre
8      @nombre.setter
9      def nombre(self, valor):
10         self._nombre = valor
11     @nombre.deleter
12     def nombre(self):
13         del self._nombre
```

¿Cómo era con la palabra property?

Estructuras de datos

Listas y tuplas

Las listas y tuplas soportan indexación de la forma `secuencia[indice]`, permiten colocar datos de distinto tipo, y se pueden iterar en bucles `for`.

Listas

- Mutable
- Datos ordenados según ingreso

Tuplas

- Inmutable
- Orden según el ingreso
- Existen las **Named Tuples**. **¿Para qué sirven?**

Estructuras de datos

Diccionarios

- Los diccionarios permiten asociar elementos pares de elementos (key y value), pero no está garantizado el orden interno. La estructura es mutable.
- La implementación es a través de una Tabla de Hash. Esto permite que la búsqueda de un valor dada su key sea muy rápida, en $\mathcal{O}(1)$
- Las keys pueden ser de tipos diferentes, pero hashables
- Se pueden iterar las keys (por defecto), los values, o los ítems en un bucle `for`
- **Default Dics:** Permiten asignar un valor por defecto para los casos cuando se accesa el diccionario usando una key que no existe. Aceptan una función para ser asignada como valor por default, la cual puede realizar cualquier acción y retornar cualquier objeto.

Estructuras de datos

Sets

Funcionan tal y como los conjuntos matemáticos

```
conjunto_a = set()
conjunto_b = {'IIC1253', 'IIC1103', 'IIC2233', 'IIC1253'}
conjunto_c = set(lista_con_hashables)
```

- Permite iteración en `for`
- No está garantizado el orden, pero si la unicidad. Tampoco soportan indexación y permiten guardar solo tipos hashables

Algunas operaciones:

```
conj.add(value)
conj.remove(value)
conj.union(conj2)
conj.intersection(conj2)
conj.symmetric_difference(conj2)
```

Estructuras de datos

Stacks

Stacks:

- Estructura que cumple con **LIFO**
- Nos basamos en las listas que ya conocemos.

Métodos básicos en la pila	Implementación en Python	Descripción
Pila.push(elemento)	Lista.append(elemento)	Agrega secuencialmente elementos a la pila
Pila.pop()	Lista.pop()	Retorna y extrae el último elemento agregado a la pila
Pila.top()	Lista[-1]	Retorne el último elemento agregado sin extraerlo
len(Pila)	len(Lista)	Retorna la cantidad de elementos en la pila
Pila.is_empty()	len(Lista) == 0	Verifica si la pila está vacía

Estructuras de datos

Queue

Queues:

- Estructura que cumple con **FIFO**
- Ocupamos `from collections import deque`
- Tenemos las Colas normales y las Colas de doble extremo. Si nos basamos en deque, la diferencia está en los métodos que utilizamos

Métodos básicos en la pila	Implementación en Python	Descripción
Cola.enqueue(elemento)	deque.append(elemento)	Agrega un objeto a la cola
Cola.dequeue()	deque.popleft()	Retorna y extrae el primer objeto de la cola
Cola.first()	deque[1]	Retorna el primer elemento de la cola sin extraerlo
len(Cola)	len(deque)	Retorna el número de elementos en la cola
Cola.is_empty()	len(deque) == 0	Verifica si la cola está vacía

Funciones

Especiales

- `__len__(self)`: Retorna el número de elementos del objeto.
- `__getitem__(self, index)`: Retorna el objeto que esta en la posición `index` del objeto y permite la llamada de `objeto[index]`
- `__reversed__(self)`: Retorna una secuencia en orden inverso (itera `__len__` veces sobre el objeto usando `__getitem__`, hacia atrás)
- `__lt__` , `__gt__`, `__le__`, `__ge__`, `__eq__(self, otra_clase)`: permite comparar clases con los símbolos <=, >=, <, >, =, respectivamente.

Funciones

Especiales

- `__getattr__(self, nombre_atributo)`: Retorna el atributo del nombre que se le entrega.
- `__setattr__(self, nombre_atributo, nuevo_valor)`: Cambia el valor del atributo por el valor que se le entrega
- `__next__(self)`: Retorna los elemento de una secuencia de manera ordenada
- `__iter__(self)`: Permite iterar sobre una clase.

Funciones

Especiales

- `zip(secuencia1, secuencia2, ...)`: Toma dos o más secuencias y retorna un conjunto de tuplas de los ítems en la misma posición.

- `enumerate(secuencia)`: Rotorna tuplas con el íncide y cada elemento según corresponda.

Comprehensions

¿Comó crearlos?

Una manera de crear secuencias de rápidamente:

- Listas

```
1 L = [item for item in item if condicion]
```

- Sets

```
1 L = {item for item in item if condicion}
```

- Diccionarios

```
1 L = {item.key: item.value for item in item if condicion}
```

Iteradores

Objeto sobre el que se puede iterar.

- Posee el método `__iter__`.
- Permite usar ciclos `for` sobre el objeto.
- Posee el método `__next__`
- Permite usar el metodo `next()` sobre el objeto.

Generadores

- Son un caso especial de los iteradores.
- Al terminar su uso, desaparecen.

```
1 G = (item for item in item if condicion)
```

Funciones Generadoras

Contiene al menos un yield.

```
1   def function(number):
2     while True:
3         number *= 2
4         i = yield number
5         number += i
```

Funciones Lambda

- Funciones en una linea.
- Son creadas en el aire.

```
lambda arg1, arg2, ...: retorno if condicion else retorno_si_no
```

Funciones

Map - Reduce - Filter

- **Map** Retorna un generador que resulta de aplicar una función a todos los elementos de un iterable. La función debería retornar un elemento "transformado".
- **Reduce** Retorna lo que resulta de aplicar una función a la secuencia dada, tomando cada par de elementos. La función debería retornar algo dado dos elementos.
- **Filter** Retorna un generador de todos los elementos de un iterable que cumplen cierta condición, dada en una función. La función debería ser booleana.

```
map(funcion, iterable)
reduce(funcion, iterable)
filter(funcion, iterable)
```

Decoradores

- Son funciones que modifican funciones o clases.
- Pueden agregar agregar comportamientos o datos adicionales.

```
1  def funcion(n):
2    return n*10
3
4  def decorador(f):
5    def nueva_funcion(n):
6      return f(n)**2 + 20
7    return nueva_funcion
8
9  funcion = decorador(funcion)
```

Decoradores

Otra manera de aplicar los decoradores.

```
1     def decorador(f):
2       def nueva_funcion(n):
3         return f(n)**2 + 20
4       return nueva_funcion
5
6     @decorador
7     def funcion(n):
8       return n*10
```

Metaclases

- Son clases que instancian clases.
- La metaclase por defecto de Python es `type`.

```
1
2   Fruta = type("Fruta", (), {})
3
4   Manzana = type("Manzana", (Fruta,), {"ser_comida":
        lambda self: print("Fui comida")})
```

Metaclases

Definir una metaclase

```
1  class MetaClase(type):
2
3      def __new__(meta, nombre_clase, bases, diccionario):
4          diccionario["saludar"] = lambda self: print("Hola a
               Todos!")
5
6          return super().__new__(meta, nombre_clase, bases,
               diccionario)
```