Programación Orientada a Objetos (OOP)

Terminología

- Cada **objeto** creado en un programa es una **instancia** de una **clase**
- Cada clase presenta al mundo exterior una vista concisa y coherente de los objetos que son instancias de esta clase, sin entrar en demasiados detalles innecesarios, ni dar acceso a otros al funcionamiento interno de los objetos.
- La definición de clase normalmente especifica **las variables de instancia** , también conocidas como **miembros de datos** , que contiene el objeto, así como los **métodos** , también conocidos como **funciones miembro ,** que el objeto puede ejecutar.

Principios y Metas

- Robustez
  - Queremos que el software sea capaz de manejar entradas inesperadas que no están explícitamente definidas para su aplicación.
- Adaptabilidad
  - El software debe ser capaz de evolucionar con el tiempo en respuesta a las condiciones cambiantes de su entorno.
- Reutilización
  - El mismo código debe poder utilizarse como componente de diferentes sistemas en varios aplicaciones

Tipos Abstractos de Datos

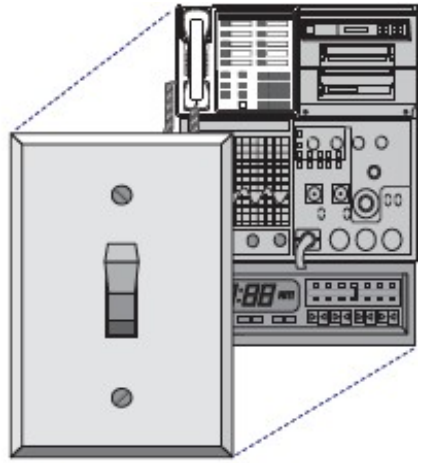
- **La abstracción** es sintetizar un sistema hasta sus partes más fundamentales.
- La aplicación del paradigma de abstracción al diseño de estructuras de datos da lugar a los **tipos abstractos de datos** (TDA's).
- Un TDA es un modelo de una estructura de datos que especifica el **tipo** de datos almacenados, las **operaciones** admitidas en ellos y los tipos de parámetros de las operaciones.
- Un TDA especifica lo que hace cada operación, pero no cómo lo hace.
- El conjunto colectivo de comportamientos soportados por un TDA es su **interfaz pública**

Principios del Diseño Orientado a Objetos

- Modularidad
- Abstracción
- Encapsulación

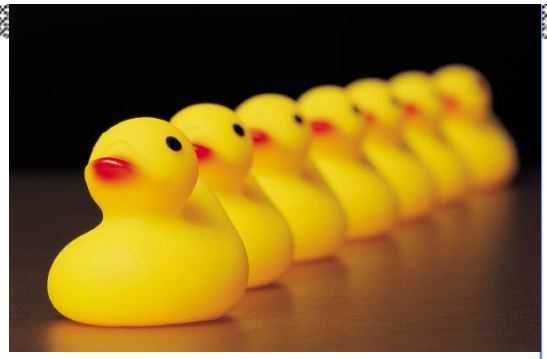
Modularity

Abstraction



Encapsulation

Duck Typing



- Python trata las abstracciones implícitamente mediante un mecanismo conocido como **duck typing** (“tipado a lo pato”)
  - Un programa puede tratar los objetos como si tuvieran cierta funcionalidad y se comportarán correctamente siempre que esos objetos proporcionen la funcionalidad esperada.
- Como un lenguaje interpretado y tipificado dinámicamente, no hay verificación en "tiempo de compilación" de los tipos de datos en Python, y no hay un requisito formal para las declaraciones de clase de base abstractas.
- El término *"duck typing"* proviene de un adagio atribuido al poeta James Whitcomb Riley, que afirma que *"cuando veo un pájaro que camina como un pato, nada como un pato y grazna como un pato, llamo a ese pájaro pato."*

Encapsulación

- Otro principio importante del diseño orientado a objetos es la **encapsulación**.
  - Los diferentes componentes de un sistema de software no deben revelar los detalles internos de sus respectivas implementaciones.
- Algunos aspectos de una estructura de datos son públicos y otros están destinados a ser detalles internos.
- Python solo proporciona escaso soporte para la encapsulación
  - Por convención, se supone que los nombres de los miembros de una clase (tanto los datos como funciones) que comienzan con un guión bajo (por ejemplo, **_secret**, **_privado**) no son públicos y no se debe confiar en ellos.

Patrones de Diseño

- **Diseño de Algorítmos:**
- Recursividad
- Amortización
- Divide y Conquistaras
- Podar y Buscar
- Fuerza Bruta
- Programación Dinámica
- El Método "*Greedy*"

- **Diseño de Software:**
- Iterador
- Adaptador / Wrapper
- Posición
- Composición
- Métodos Plantilla
- Factory method

Lenguaje de Modelado Unificado (UML)

Un **diagrama de clases** tiene tres porciones

1. El **nombre** de la clase.
2. Las **variables de instancia** recomendadas.
3. Los **métodos** recomendados de la clase.

Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit
Behaviors:	get_customer( ) get_bank( ) get_account( ) make_payment(amount)	get_balance( ) get_limit( ) charge(price)

Definiciones de Clase

- Una ***clase*** sirve como medio principal para la abstracción en la orientación a objetos. programación.
- En Python, cada dato se representa como una ***instancia*** de algún clase.
- Una clase proporciona un conjunto de comportamientos en forma de funciones (también conocidas como ***métodos***), con implementaciones que pertenecen a todas sus instancias.
- Una clase también sirve como modelo para sus instancias, determinando efectivamente la forma en que la información de estado para cada instancia se representa en forma de ***atributos*** (también conocidos como ***campos*** , ***variables de instancia*** o simplemente ***datos***).

El identificador *self*

- En Python, el "*autoidentificador*" ***self*** juega un rol fundamental identificando al objeto.
- Para cualquier *clase*, posiblemente puede haber muchas instancias diferentes, y cada una debe mantener sus propias variables de instancia.
- Por lo tanto, cada instancia almacena sus propias variables de instancia para reflejar su estado actual. Sintácticamente, ***self*** identifica la instancia sobre la cual se aplica un método invocado.

Ejemplo

```
1  class CreditCard:
2    """A consumer credit card."""
3
4    def __init__(self, customer, bank, acnt, limit):
5      """Create a new credit card instance.
6
7      The initial balance is zero.
8
9      customer  the name of the customer (e.g., 'John Bowman')
10     bank      the name of the bank (e.g., 'California Savings')
11     acnt      the acount identifier (e.g., '5391 0375 9387 5309')
12     limit     credit limit (measured in dollars)
13     """
14     self._customer = customer
15     self._bank = bank
16     self._account = acnt
17     self._limit = limit
18     self._balance = 0
19
```

Ejemplo, Parte 2

```
20    def get_customer(self):
21      """Return name of the customer."""
22      return self._customer
23
24    def get_bank(self):
25      """Return the bank's name."""
26      return self._bank
27
28    def get_account(self):
29      """Return the card identifying number (typically stored as a string)."""
30      return self._account
31
32    def get_limit(self):
33      """Return current credit limit."""
34      return self._limit
35
36    def get_balance(self):
37      """Return current balance."""
38      return self._balance
```

Ejemplo, Parte 3

```
39    def charge(self, price):
40      """Charge given price to the card, assuming sufficient credit limit.
41
42      Return True if charge was processed; False if charge was denied.
43      """
44      if price + self._balance > self._limit:        # if charge would exceed limit,
45        return False                                 # cannot accept charge
46      else:
47        self._balance += price
48        return True
49
50    def make_payment(self, amount):
51      """Process customer payment that reduces balance."""
52      self._balance -= amount
```

Constructores

- Un usuario puede crear una instancia de la clase *CreditCard* usando:

```
cc = CreditCard('John Doe', '1st Bank', '5391 0375 9387 5309', 1000)
```

- Internamente, esto da como resultado una llamada al método de **`__init__`** con nombre que sirve como el ***constructor*** del la clase.
- Su principal responsabilidad es establecer el estado de un objeto recién creado con variables de instancia

Sobrecarga de Operadores

- Las Clases integradas de Python proporcionan la semántica natural para muchos operadores.
- Por ejemplo, la sintaxis ***a + b*** invoca la *suma* para tipos numéricos, pero la *concatenación* para tipos contenedores.
- Al definir una nueva clase, debemos considerar si se debe definir una semántica (operacional) diferente para ***a + b*** cuando ***a*** o ***b*** es una instancia de esa clase.