Object-Oriented Programming

Sareh Taebi

FAU
FLORIDA ATLANTIC UNIVERSITY

Terminology

- Each **object** created in a program is an **instance** of a **class**.
- Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition typically specifies **instance variables**, also known as **data members**, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute.

member functions = methods

object oriented programming will help us achieve all 3
2 big failures b/c software was not adaptable:
- therac 25- too much radiation- could not use software on new machine
- arian 5 shuttle- exploded after take off- software- type conversion error

Goals

- Robustness
  - We want software to be capable of handling unexpected inputs that are not explicitly defined for its application. raise exceptions
- Adaptability
  - Software needs to be able to evolve over time in response to changing conditions in its environment.
- Reusability
  - The same code should be usable as a component of different systems in various applications.

we know how to use data types
- don't need to know how to use
- black box

Abstract Data Types

- **Abstraction** is to distill a system to its most fundamental parts.
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).
- An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
- An ADT specifies what each operation does, but not how it does it.
- The collective set of behaviors supported by an ADT is its **public interface**.

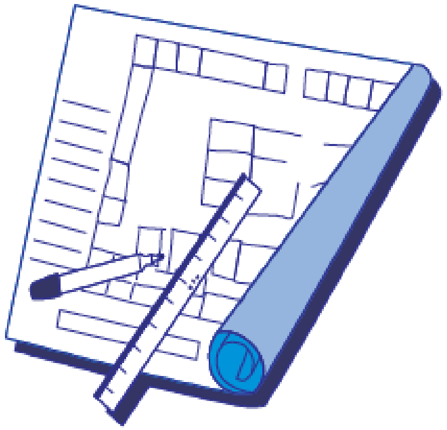
Object-Oriented Design Principles

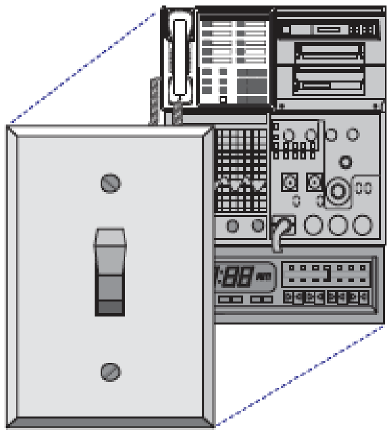
Python modules
- library of functions
- each function has a separate task
- address one set of problems

Abstraction
- simplify to user

- ❑ Modularity
- ❑ Abstraction
- ❑ Encapsulation
  - keeping local variables & info hidden

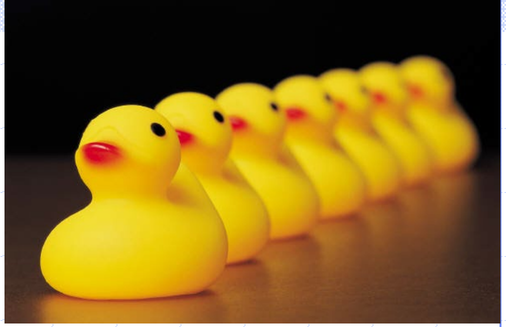
Modularity


Abstraction


Encapsulation

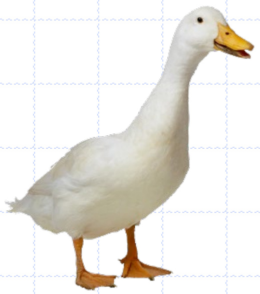
Duck Typing

only python
- not declaring variables
- automatic type



- Python treats abstractions implicitly using a mechanism known as **duck typing**.
  - A program can treat objects as having certain functionality and they will behave correctly provided those objects provide this expected functionality.
- As an interpreted and dynamically typed language, there is no "compile time" checking of data types in Python, and no formal requirement for declarations of abstract base classes.
- The term "duck typing" comes from an adage attributed to poet James Whitcomb Riley, stating that "when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

Abstract Base Classes



- Python supports abstract data types using a mechanism known as an **abstract base class (ABC)**.
- An abstract base class cannot be instantiated, but it defines one or more common methods that all implementations of the abstraction must have.
- An ABC is realized by one or more concrete classes that inherit from the abstract base class while providing implementations for those method declared by the ABC.
- We can make use of several existing abstract base classes coming from Python's collections module, which includes definitions for several common data structure ADTs, and concrete implementations of some of these.

Encapsulation

some elements are private
- public- only what user needs
- based on constructor

- Another important principle of object-oriented design is **encapsulation**.
  - Different components of a software system should not reveal the internal details of their respective implementations.
- Some aspects of a data structure are assumed to be public and some others are intended to be internal details.
- Python provides only loose support for encapsulation.
  - By convention, names of members of a class (both data members and member functions) that start with a single underscore character (e.g., _secret) are assumed to be nonpublic and should not be relied upon.

- not requires to be nonpublic

Design Patterns

- **Algorithmic patterns:**
- Recursion - function repeats
- Amortization
- Divide-and-conquer
- Prune-and-search
- Brute force
- Dynamic programming
- The greedy method

- **Software design patterns:**
- Iterator
- Adapter
- Position
- Composition
- Template method
- Locator
- Factory method

Object-Oriented Software Design

- **Responsibilities**: Divide the work into different actors, each with a different responsibility.
- **Independence**: Define the work for each class to be as independent from other classes as possible. each function has 1 purpose
- **Behaviors**: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.
  - look for possible issues

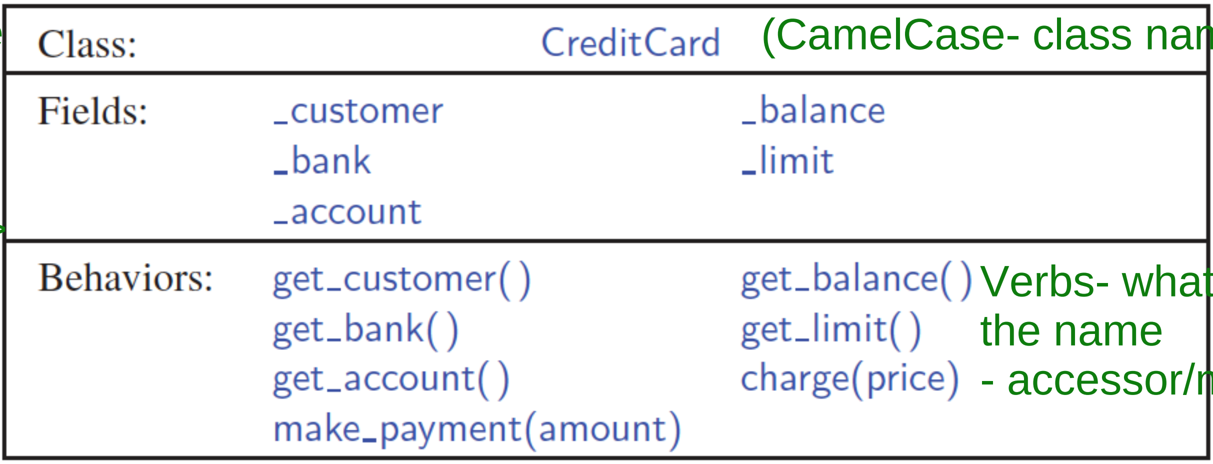
Unified Modeling Language (UML)

A **class diagram** has three portions.

1. The name of the class
2. The recommended instance variables
3. The recommended methods of the class.

- class should have behaviors
- every member

- premature class->
- can create and add to it



Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit

Behaviors:	get_customer()
	get_bank()
	get_account()
	make_payment(amount)
		get_balance()
		get_limit()
		charge(price)

(CamelCase- class names)

Verbs- what it does in the name
- accessor/mutator

memorize terminology

Class Definitions

- A class serves as the primary means for abstraction in object-oriented programming.
- In Python, every piece of data is represented as an instance of some class.
- A class provides a set of behaviors in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, or **data members**).

The **self** Identifier

- In Python, the **self** identifier plays a key role.

  - object coordinates
  --- brings info into the class

- In any class, there can possibly be many different instances, and each must maintain its own instance variables.
- Therefore, each instance stores its own instance variables to reflect its current state. Syntactically, **self** identifies the instance upon which a method is invoked.

Example

overloading operators
- use equal instead of set/get

first create object of the class
- then use them

```
class CreditCard:
  """A consumer credit card."""

  def __init__(self, customer, bank, acnt, limit):
    """Create a new credit card instance.

    The initial balance is zero.

    customer  the name of the customer (e.g., 'John Bowman')
    bank      the name of the bank (e.g., 'California Savings')
    acnt      the acount identifier (e.g., '5391 0375 9387 5309')
    limit     credit limit (measured in dollars)
    """
    self._customer = customer
    self._bank = bank
    self._account = acnt
    self._limit = limit
    self._balance = 0

```

Example, Part 2

```
def get_customer(self):
  """Return name of the customer."""
  return self._customer

def get_bank(self):
  """Return the bank's name."""
  return self._bank

def get_account(self):
  """Return the card identifying number (typically stored as a string)."""
  return self._account

def get_limit(self):
  """Return current credit limit."""
  return self._limit

def get_balance(self):
  """Return current balance."""
  return self._balance
```

Example, Part 3

```
def charge(self, price):
  """Charge given price to the card, assuming sufficient credit limit.

  Return True if charge was processed; False if charge was denied.
  """
  if price + self._balance > self._limit:      # if charge would exceed limit,
    return False                                # cannot accept charge
  else:
    self._balance += price
    return True

def make_payment(self, amount):
  """Process customer payment that reduces balance."""
  self._balance −= amount
```

Constructors

- A user can create an instance of the CreditCard class using a syntax as:

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

- Internally, this results in a call to the specially named `__init__` method that serves as the constructor of the class.
- Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables.