CSc 110, Spring 2017

Lecture 34: Encapsulation

Adapted from slides by Marty Stepp and Stuart Reges

Abstraction

Don't need to know this

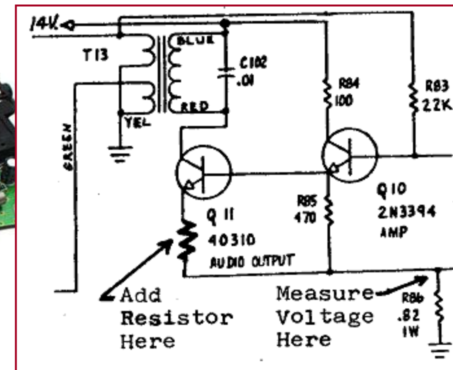
Can focus on this!!

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        sqrt(x ** 2 + y ** 2)

    def translate(dx, dy):
        self.x += dx
        self.y += dy

    def __str__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"

    def draw(self, panel, color):
        panel.canvas.create_oval(self.x, self.y, self.x + 3,
                self.y + 3, outline=color)
        panel.canvas.create_text(self.x, self.y - 10,
                text="(" + str(self.x) + ", " + str(self.y) + ")",
                fill=color)
```

Question: Are there errors in this class definition?

Encapsulation

- **encapsulation**: Hiding implementation details of an object from the users of the class.
    - Encapsulation provides *abstraction*.
        - separates external view (behavior) from internal view (state)
    - Encapsulation protects the integrity of an object's data.



Accessing attributes

- The current implementation of `Point` allows the attributes to be modified.

  - Example:

```
p = Point(3,10)
p.x = 20
```

- We want to specify that the attributes cannot be modified.

Private attributes

- An attribute can be made invisible to outsiders
  - No code outside the class can access or change it easily.
  - Syntax for private attributes:

    __name

  - Examples:

```
self.__id
self.__name
```

- Python prevents the private attributes from being accessed outside the class.

Accessing private attributes

- We can provide methods to get and/or set an attribute's value:

```
# A "read-only" access method to the __x field
("accessor")
def get_x(self):
    return self.__x


# A "write" access method to the __x field ("mutator")
def set_x(self, new_x):
    self.__x = new_x
```

- Methods would be used to access `Point`'s attributes:

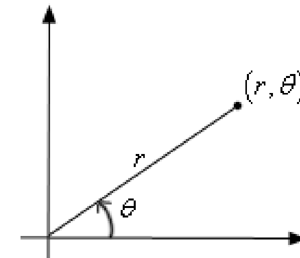
```
if (p1.get_x() < p2.get_x())
    print("p1 is left of p2")
```

Benefits of encapsulation

- Provides abstraction between an object and users of the object.

- Protects an object from unwanted access by code outside the class.
  - A bank app forbids a client to change an `Account`'s balance.

- Allows you to change the class implementation.
  - `Point` could be rewritten to use polar coordinates (radius r, angle ϑ), but with the same methods.



- Allows you to constrain objects' state (**invariants**).
  - Example: Only allow `Point`s with non-negative coordinates.

Point class with private attributes

```
# A Point object represents an (x, y) location.
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def get_x(self):
        return self.__x

    def get_y(self):
        return self.__y

    def distance_from_origin(self):
        return sqrt(self.__x **2 + self.__y **2)

    def translate(self, dx, dy):
        self.__x += dx
        self.__y += dy

    def __str__(self):
        return "(" + str(self.__x) + "," + str(self.__y) + ")"
...
```

Using Point with private attributes

```
def main():
    # create two Point objects
    p1 = Point(5, 2)
    p2 = Point(4, 3)

    # print each point
    print("p1: (" + str(p1.get_x()) + ", " + str(p1.get_y()) + ")")
    print("p2: (" + str(p2.get_x()) + ", " + str(p2.get_y()) + ")")

    # move p2 and then print it again
    p2.translate(2, 4)
    print("p2: (" + str(p2.get_x()) + ", " + str(p2.get_y()) + ")")
```

OUTPUT:

```
p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)
```

Bank account – Version 1

- Write a `BankAccount` class with the following attributes:

```
account_number
name
balance
```

- Implement these methods:

```
deposit(amount)
withdraw(amount)
```

BankAccount – Version 1

```
class BankAccount:
    def  __init__(self, account_number, name, amount):
        self.balance = amount
        self.account_number = account_number
        self.name = name

    def withdraw(self, amount):
        if(self.balance - amount < 0):
            print("transaction rejected: not enough money")
        else:
            self.balance -= amount

    def deposit(self, amount):
        self.balance += amount
```

Bank account – Version 2

- Modify the `BankAccount` class to make the attributes private:

```
account_number
name
balance
```

Implement a method that returns the number of transactions on the account:

```
transaction_count()
```

Note: only the initial account creation, deposits and withdraws are transactions.

BankAccount – Version 2

```
class BankAccount:
    def  __init__(self, account_number, name, amount):
        self.__balance = amount
        self.__account_number = account_number
        self.__name = name
        self.__tcount = 1

    def withdraw(self, amount):
        self.__tcount += 1
        if(self.__balance - amount < 0):
            print("transaction rejected: not enough money")
        else:
            self.__balance -= amount

    def deposit(self, amount):
        self.__tcount += 1
        self.__balance += amount

    def transaction_count(self):
        return self.__tcount
```

Bank account – Version 3

- Modify the `BankAccount` class to keep track of the transactions in a list of tuples.

- Each tuple consists of the string `"d"` ,`"w"`, or `"r"` and the associated amount.

- Modify the `transaction_count` method to use the list.

BankAccount – Version 3

```
class BankAccount:
    def  __init__(self, account_number, name, amount):
        self.__balance = amount
        self.__account_number = account_number
        self.__name = name
        self.__tlist =  [("d", amount)]

    def withdraw(self, amount):
        if(self.__balance - amount < 0):
            print("transaction rejected: not enough money")
            self.__tlist.append(("r",amount))
        else:
            self.__balance -= amount
            self.__tlist.append(("w",amount))

    def deposit(self, amount):
        self.__tlist.append(("d",amount))
        self.__balance += amount

    def transaction_count(self):
        return len(self.__tlist)
```