INFO5002: Intro to Python for Info Sys

Encapsulation

Northeastern University

Slides created by: Zachary Doucet

I want to protect my data!

- It is generally good practice to expose only the **minimum** amount of data necessary.

```
class BankAccount:
    id = 0
    name = ""
    balance = 0
    credit = 0
    interest_rate = 0
    overdrafted = False
    login_dates = []
```

Maybe don't need to have all of these attributes **publicly** exposed.

You can hide with encapsulation

- Encapsulation is the process of selectively hiding data between components.
  - Protect from unauthorized or accidental mutations.
  - Add validation to getting and mutating.
  - Hiding business logic.

There are three access modifiers

- Public: any component can access (default).
- Protected: class and subclasses.
- Private: only the class.

Only private access modifer enforced (through name mangling).
It is however good practice to use protected.

The mode is defined through frontal underscores

- Public: no underscore.
- Protected: single underscore.
- Private: double underscore.

```
class BankAccount:
    id = 7312835182 # Public
    _name = "John Doe" # Protected
    __balance = 0   # Private
```

Getter

- Private attributes cannot be accessed outside the class and thus must be shared to a method known as a getter.

```
class BankAccount:
    def __init__(self, initial_balance):
        self.__balance = initial_balance
    def get_balance():
        return self.__balance
```

We can get attribute's value without being able to mutate.

- Can add logic to the getter.

Setter

- Private attributes cannot be mutated outside the class and thus must be updated through a method known as a setter.

```
class BankAccount:
    def __init__(self, initial_balance):
        self.__balance = initial_balance
    def set_balance(new_balance):
        if new_balance < 0:
            self.overdraft = True
            return
        self.__balance = new_balance
```

- Can add logic to the setter.

Let’s Practice

- Create a class **BankAccount** which has a constructor that takes in an initial balance. The balance should be updated only through two methods **deposit** and **withdraw** which both take in a number representing deposit or withdrawal amount. Add logic to guard against overdraft. Trying to withdraw more than currently in balance should prevent transaction.