ACIT2515

Object oriented programming

Definitions

- Class: Defines a general category (i.e., book, bank account)
  - Blueprint (or template) for creating an object
  - A custom data type
- Attributes: values for a specific object
- Methods: behaviors (or capabilities) of a class
- Object or Instance: a specific instance of a class
- State: the current values of the attributes in an object

Object oriented programming

Objectives

- Modular design, and reusable code
- Data protection
- Driven by objects and behaviours rather than programming logic

Important concepts

- Modelization
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Modelization

The objective of a model is to "describe" an object.

- How can we describe a person?
  - For the government: name, address, SIN number
  - At BCIT: name, student number, program
  - At the hairdresser: first name, phone number, appointment date
- We represent the objects by creating a *model*, and implement it using a class

Class syntax in Python

- `class MyClassName:` to define a new class "block"
- Indent the code below
- Except for class variables, a class definition only has **methods**
- Methods are defined inside the class block, and receive `self` as first parameter
- A special method is `__init__`, always called upon *initialization* of the instance

Python example

```
class Student:
  def __init__(self, name, student_number):
    self.name = name
    self.student_number = student_number
    self.program = "CIT"

john = Student("John Doe", "A01234567")
```

Encapsulation (and visibility)

- The state of an object should only be changed by the object itself.
- Use behaviours (= methods) to alter the state, instead of changing the attributes.
- Some object oriented programming languages have a visibility feature - where you can make attributes *private*. Only the object itself can change them.
- Does not exist in Python, but there is a convention: use `_` in the beginning of the attribute to mean it is private.
- Using private attributes and using methods to change the state is called **encapsulation**.

Abstraction

- The implementation of the behaviours of the object is the responsibility of the class.
- Other objects should not depend upon private attributes or methods.
- They should only use the *public interface* of the class: the public methods and attributes.

The `self` parameter

- All functions defined within in a class are called "methods"
- They receive an implicit parameter: `self`, which represents the current **instance**
- This allows encapsulation: each instance has its own version of the class attributes, and they are accessed with `self`

Parameter validation

- Imagine the `Student` class should not allow students with "invalid names".
- We will make it raise an exception if that is not the case.

```
class Student:
  def __init__(self, name, student_number):
    if not name or type(name) is not str:
      raise AttributeError("Name cannot be empty!")
    self.name = name
    self.student_number = student_number
    self.program_name = "CIT"

john = Student("", "A01234567") # Will raise an Exception!
john = Student(42, "A01234567") # Will raise an Exception!
```

Using Python `property`

- Python has a decorator which makes encapsulation and abstraction easier.
- You can use the `@property` decorator on a method, and then it will behave as *an attribute*. This allows you to implement getters very easily.

```
class Student:
  [...]
  @property
  def info(self):
    return f"{self.name} ({self.student_number})"

john = Student("John Doe", "A01234567")

# Note how we use info as if it were a variable!
john.info == "John Doe (A01234567)"
```

Python properties: the setter

```
class Student:
  [...]
  @property
  def program(self):
    return self.program_name

  @program.setter
  def program(self, value):
    if value not in ("CIT", "CST", "BTech"):
      raise ValueError("Invalid program name!")

    self.program_name = value

john = Student("John Doe", "A01234567")
john.program == "CIT"       # Runs the getter method
john.program = "Btech"      # Runs the setter method
john.program = "something"  # Raises an Exception!
```

You can only create a setter when you have a getter defined (property).
The getter and setter methods are called the same name. You also need to use this name in the `@[name].setter` decorator.

Static methods and class methods

- You can define methods that are common to all objects of the same class.
- Static methods do not get any reference to the class or object
- Class methods receive a reference to the **class** when called directly from the class name
- This reference is typically called `cls`
- Use the `@staticmethod` decorator for static methods (no reference)
- Use the `@classmethod` decorator for class methods (reference to the class)

Static method example

```
class Student:
  [...]
  @staticmethod
  def check_program(value):
    # This is a static method. It does not receive an implicit argument.
    # It does not have access or know about the class / instance
    return value in ("CIT", "CST", "BTech")

Student.check_program("CIT")  # will return True

john = Student("John Doe", "A01234567")
john.check_program("CIT")     # will also return True
```

Class method example

```
class Student:
  PROGRAMS = ("CIT", "CST", "BTech")
  [...]
  @staticmethod
  def check_program(cls, value):
    # This is a static method. It does not receive an implicit argument.
    # It does not have access or know about the class / instance
    return value in cls.PROGRAMS

Student.check_program("CIT")  # will return True

john = Student("John Doe", "A01234567")
john.check_program("CIT")     # will also return True
```

Class variables

- Class variables are attributes that are common to all objects of the same class.
- They are defined in class block, but outside any methods.
- The previous example uses a class variable `PROGRAMS`
- Class methods have access to class variables (but not instance variables)
- Static methods do not have access to class variables, nor instance variables