National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2024/2025

Tutorial 9
Object-Oriented Programming

In this tutorial, we will explore two ideas: the simulation of a world in which objects are characterized by collections of state variables, and the use of *object-oriented programming* as a technique for modularizing worlds in which objects interact. These ideas are presented in the context of a simple simulation game like the ones available on many computers.

First let's practice implementing a simple OOP object from scratch. Let's suppose we want to model a MMORPG game world using OOP where players can use weapons to kill animals to gain items. As such it is essential to model a class `Thing`, which will be superclass of items in the game.

Task 1: Implement simple `Thing`

The essential properties of a `Thing` are as follows :

1. The constructor should take in 1 parameter, the name of the `Thing`.

```
>>> stone = Thing('stone')
```

2. `owner` : an attribute that stores the owner object of the `Thing`, usually a `Person` object. In OOP, we set this attribute to `None` during initialization when this `Thing` does not belong to any `Person` yet (to signify the absence of an object value).

```
>>> stone.owner
None
```

3. `is_owned()`: returns a boolean value, `True` if the thing is "owned" and False otherwise.

```
>>> stone.is_owned()
False
```

4. `get_owner()`: returns the `Person` object who owns the `Thing` object.

```
>>> stone.get_owner()
None
```

Implement the class `Thing` such that it satisfies the above properties and methods.

Task 2: Extend your `Thing`

The above `Thing` definition is still not satisfactory; it should support the following methods as well. Modify and extend your `Thing` definition from Task 1.

1. `get_name()` : returns the name (string) of the `Thing`

```
>>> stone.get_name()
'stone'
```

2. `place` : Just like the `owner` attribute, we need to keep state of the `Place` object where the `Thing` is in. Similarly, this attributes should be initialized to `None` when the `Thing` object is created.

```
>>> stone.place
None
```

3. `get_place()` : returns the place associated with the `Thing`.

```
>>> stone.get_place()
None
```

Task 3: Inheritance from `MobileObject`

Now imagine if you were to try to model other objects in the game world. Objects like `Person` and `Animal` will definitely share `get_name()` and `get_place()` with `Thing`. It will be a hassle and clunky to redefine these methods for all other objects in our game world. Fortunately for you, this is where Inheritance (and `hungry_games.py`) comes to the rescue!

Inside `hungry_games.py`, you will find that `get_name()` is captured by the class `NamedObject` while `get_place()` is captured by `MobileObject`.

Many other basic objects in the game world are defined in `hungry_games.py`.

Now Ben Bitdiddle wants to try modifying the original definition of `Thing` from Task 1 such that it inherits from `MobileObject` instead. Here are his class header and constructor (the rest of his definition require no changes):

```
class Thing(MobileObject):

    def __init__(self, name):
        self.name = name
        self.owner = None
```

What is wrong with his code? Try testing with

```
>>> stone = Thing('stone')
>>> stone.get_place()
```

Task 4: OOP Adventure Game

The basic idea of simulation games is that the user plays a character in an imaginary world inhabited by other characters. The user plays the game by issuing commands to the computer, and these commands have the effect of moving the character about and performing acts in the imaginary world, such as picking up objects. The computer simulates the legal moves and rejects illegal ones. For example, it is illegal to move between places that are not connected (unless you have special powers). If a move is legal, the computer updates its model of the world and allows the next move to be considered.

Our game takes place in a strange, imaginary world called NUS, with imaginary places such as the Central Library, LT15, Data Comm Lab 2 and the Arts Canteen. In order to get going, we need to establish the structure of this imaginary world: the objects which exist and the ways in which they relate to each other.

Initially, there are three instantiation operations for creating objects:

```
Thing(name)
Place(name)
Person(name, health, threshold)
```

Each time we make a place, person, or thing, we give it a name. In addition, a person has a health value and a threshold factor, which determines how often the person moves.

The `Place` object has a method, `add_object`, which would add an object to the place. The method accepts a `Person` or a `Thing` object, and puts it in that location.

```
beng_office = Place("beng_office")
bing_office = Place("bing_office")

beng = Person("beng", 100, 3)
bing = Person("bing", 100, 2)

beng_office.add_object(beng)
bing_office.add_object(bing)
```

All objects in the system are implemented as Python classes.

Once you load the system (by running `engine.py`), you will be able to control `beng` and `bing` by sending them appropriate messages. As you enter each command, the computer reports what happens and where it is happening. For instance, imagine we have interconnected a few places so that the following scenario is feasible:

```
>>> beng.objects_around()
[<beng_card : SDCard>]

>>> beng.go(north)
beng moves from beng_office to com1_classrooms

>>> beng.go(north)
beng moves from com1_classrooms to com1_open_area

>>> print(beng.get_place().get_exits())
['WEST', 'EAST', 'NORTH', 'SOUTH']

>>> beng.go(north)
beng moves from com1_open_area to lt15

>>> bing.go(down)
bing moves from bing_office to lt15
```

In principle, you could run the system by issuing specific commands to each of the creatures in the world, but this defeats the intent of the game since that would give you explicit control over all the characters. Instead, we will structure our system so that any character can be manipulated automatically in some fashion by the computer. We do this by creating a list of all the characters to be moved by the computer and by simulating the passage of time by a special function, `clock`, that sends a `move` message to each creature in the list. A `move` message does not automatically imply that the creature receiving it will perform an action. Rather, like all of us, a creature hangs about idly until he or she (or it) gets bored enough to do something. To account for this, the third argument to `Person` specifies the average number of clock intervals that the person will wait before doing something (the threshold factor).

Before we trigger the clock to simulate a game, let's explore the properties of our world a bit more.

First, let's create a `python_docs` and place it in `lt15` (where `beng` and `bing` now are).

```
>>> python_docs = Thing("python_docs")
>>> lt15 = Place("lt15")
>>> lt15.add_object(python_docs)
```

Next, we'll have `beng` look around. He sees the documentation, `bing` and `proffy`. The documentation looks useful, so we have `beng` take it and leave.

```
>>> beng.take(python_docs)
beng took python_docs in lt15

>>> beng.go(north)
beng moves from lt15 to forum
```

`bing` had also noticed the manual; he follows `beng` and attempts to snatch the documentation away but failed. Smugly, `beng` goes off to the Central Library:

```
>>> bing.go(north)
bing moves from lt15 to forum
>>> bing.objects_around()
[<beng : Person>]

>>> bing.take(python_docs)
bing cannot take python_docs.
>>> beng.go(up)
beng moves from forum to central_library
```

Unfortunately for `bing`, the dungeon LT15 is inhabited by a troll named `proffy`. A troll is a kind of person: it can move around, take things, and so on. When a troll gets a `move` message from the clock, it acts just like an ordinary person—unless someone else is in the room. When `proffy` decides to `act`, it's game over for `bing`:

```
>>> bing.go(south)
bing moves from forum to lt15
>>> proffy.act() # proffy decides not to act (yet!)
```

After a few more moves, `proffy` acts again:

```
>>> proffy.act()
bing went to heaven!
```

Implementation The source code for this problem set is found in the files `engine.py` and `hungry_games.py`. The provided code in `hungry_games.py` contains a basic object system, functions to create people, places, things, together with various other useful functions. engine.py contains the skeleton of a simple game by loading `hungry_games.py` and creating all the objects in the game world like Places. You will be asked to extend the game world in `hungry_games.py` by writing appropriate functions and extensions to the existing functions in your missions.

1. Draw a simple inheritance diagram showing all the kinds of objects (classes) defined in the adventure game system, the inheritance relations between them, and the methods defined for each class. This is critical in helping you to understand the OOP system in `hungry_games.py` for your missions.

2. Suppose we evaluate the following statements:

   ```
   ice_cream = Thing("ice_cream")
   ice_cream.owner = beng
   ```

   Come up with statements whose evaluation will reveal all the properties of `ice_cream` and verify that its (new) owner is indeed `beng`.

3. Now suppose we evaluate the following statements:

   ```
   ice_cream = Thing("ice_cream")
   ice_cream.owner = beng
   beng.ice_cream = ice_cream
   ```

   Is there anything wrong with the last two statements? What's the moral of the story?

4. Suppose that, in addition to `ice_cream` we defined above, we define

   ```
   rum_and_raisin = NamedObject("ice_cream")
   ```

   Are `ice_cream` and `rum_and_raisin` the same object (i.e., does `ice_cream is rum_and_raisin` evaluate to `True`)?

5. Now let's make two similar objects in our world.

   ```
   burger1 = Thing("burger")
   burger2 = Thing("burger")
   ```

   Are `burger1` and `burger2` the same object? Would `burger1 == burger2` evaluate to `True`? Let's say we want to a way to compare `Thing` objects, and objects that have the same name and are at the same location should evaluate to `True` when we compare them with `==`. How would you do it? (i.e. Objects like `burger1` and `burger2` should be considered equal when tested with `==`.)