INTRO2CS

Tirgul 8
What We’ll Be Seeing Today

- Introduction to Object-Oriented Programming (OOP).
- Using Objects
- Special methods
What is OOP?

- So far our programs were made of different variables and functions operating on them.
  - This programming paradigm is called **Procedural Programming**

- **Object Oriented Programming** is a different programming paradigm!
  - In **OOP**, we encapsulate our data inside objects.
  - Objects operate on/between themselves.
  - Don’t worry – it will become clearer.. 😊
Let’s play an RPG!
We have a hero!

name = 'Mark'
health = 50
magic_points = 80
inventory = {'gold': 40, 'healing potion': 2 , 'sword': 1}
weapon = 'sword'
location = [4, 6]
We add a goblin..
We add a goblin..

We’ll have to change the variables accordingly:

```python
hero_name = 'Mark'
hero_health = 50
hero_magic_points = 80
hero_inventory = {'gold': 40, 'healing potion': 2, 'sword': 1}
hero_weapon = 'sword'
hero_location = [4, 6]

monster_name = 'Goblin'
monster_health = 20
monster_magic_points = 0
monster_inventory = {'gold': 12, 'dagger': 1}
monster_weapon = 'dagger'
monster_location = [2, 3]
```
We want more monsters!
What now?

- monster1_name, monster2_name… - Bad coding!
- Instead, Maybe:
  
  ```python
  monster_names = ['Goblin', 'Dragon', 'Goblin']
  monster_healths = [20, 300, 18]
  monster_magic_points = [0, 200, 0]
  monster_inventories = [{'gold': 12, 'dagger': 1}, {'gold': 890, 'magic amulet': 1}, {'gold': 15, 'dagger': 1}]
  monster_weapons = ['dagger', 'fire', 'dagger']
  monster_locations = [[2, 3], [1, 5], [4, 2]]
  ```

- Gets confusing..
- A single creature is “spread” over many lists
What now?

- How about:

```python
monsters = [{
    'name': 'Goblin',
    'health': 20,
    'magic points': 0,
    'inventory': {
        'gold': 12,
        'dagger': 1
    },
    'weapon': 'dagger',
    'location': [2, 3],
},
{
    'name': 'Dragon',
    'health': 300,
    'magic points': 200,
    'inventory': {
        'gold': 890,
        'magic_amulet': 1
    },
    'weapon': 'fire',
    'location': [1, 5],
},
{
    'name': 'Goblin',
    'health': 18,
    'magic points': 0,
    'inventory': {
        'gold': 15,
        'dagger': 1
    },
    'weapon': 'dagger',
    'location': [4, 2]
}]
```

- But the inventory of a monster is a dictionary-in-a-dictionary-in-a-list!!

- Using **OOP** this is becomes simple!
The OOP Way - Classes

- Define a **Class** for similar objects
- The hero and monsters are all living things:
  ```python
class LivingThing():
    pass

hero = LivingThing()  # Creates an object of the class
hero.name = 'Mark'     # Set Member Variables for the object
hero.health = 50
hero.magic_points = 80
hero.inventory = {'gold': 40, 'healing potion': 2, 'sword': 1}
hero.weapon = 'sword'
hero.location = [4, 6]
...
```
... monsters = []
temp_monster = LivingThing()  # Creates a new object of the class
temp_monster.name = 'Goblin'
temp_monster.health = 20
temp_monster.magic_points = 0
temp_monster.inventory = {'gold': 12, 'dagger': 1}
temp_monster.weapon = 'dagger'
temp_monster.location = [2, 3]
monsters.append(temp_monster)

... temp_monster = LivingThing()  # Creates a new object of the class
temp_monster.name = 'Dragon'
temp_monster.health = 300
temp_monster.magic_points = 200
temp_monster.inventory = {'gold': 890, 'magic amulet': 1}
temp_monster.weapon = 'fire'
temp_monster.location = [1, 5]
monsters.append(temp_monster)
...
Seems a bit inefficient - All LivingThing are similar!

- We can define functions for all objects of the same class:

```python
class LivingThing:
    def set_attributes(self, name, health, magic_points, inventory, weapon, location):
        self.name = name
        self.health = health
        self.magic_points = magic_points
        self.inventory = inventory
        self.weapon = weapon
        self.location = location

hero = LivingThing()
hero.set_attributes('Mark', 50, 80, {'gold': 40, 'healing potion': 2, 'sword': 1}, 'sword', [4, 6])
monsters = []
temp_monster = LivingThing()
temp_monster.set_attributes('Goblin', 20, 0, {'gold': 12, 'dagger': 1}, 'dagger', [2, 3])
monster.append(temp_monster)
temp_monster = LivingThing()
temp_monster.set_attributes('Dragon', 300, 200, {'gold': 890, 'magic amulet': 1}, 'fire', [1, 5])
...
The special word ‘self’

- A **Class** is **not** an **Object**!
- Objects are **instances** of a class
- A class “function” is called a **method**
- **Methods** are called by **specific** **Objects**
- The first parameter all **Methods** get is ‘**self**’
- ‘**self**’ is the **specific** object that called the method!
The special method `__init__()`:

- A special method that is called after we create a new object:
  ```python
class LivingThing():
    def __init__(self, name, health, magic_points, inventory, weapon, location, armor=None):
        self.name = name
        self.health = health
        self.magic_points = magic_points
        self.inventory = inventory
        self.weapon = weapon
        self.location = location
        self.armor = armor

hero = LivingThing('Mark', 50, 80, {'gold': 40, 'healing potion': 2, 'sword': 1}, 'sword', [4, 6])
monsters = []
temp_monster = LivingThing('Goblin', 20, 0, {'gold': 12, 'dagger': 1}, 'dagger', [2, 3])
monster.append(temp_monster)
temp_monster = LivingThing('Dragon', 300, 200, {'gold': 890, 'magic amulet': 1}, 'fire', [1, 5])
```
The special method \texttt{\_\_init\_\_}():

- Called a “constructor”
- If no explicit constructor is defined - default empty constructor
- All instances will have the members defined in the constructor
- The constructor is always called \texttt{\_\_init\_\_}!
What else can we do?

- Assume our hero takes damage:
  ```python
  hero.health -= 10
  if hero.health < 0:
      print(hero.name + ' has died!')
  ```
- Can lead to code duplication..
- The non-OOP solution – use a function:
  ```python
  def take_damage(living_object, damage):
      living_object.health -= damage
      if living_object.health < 0:
          print(living_object.name + ' has died!')
  ```
- Much better!
- Still, difficult to maintain. Can have thousands of functions!
Using methods:

- The OOP way: use a Method!

```python
class LivingThing:
    #...other code in the class...

    def take_damage(self, damage):
        self.health -= damage
        if self.health <= 0:
            print(self.name + ' is dead!')

    #...other code in the class...

hero = LivingThing('Mark', 50, 80, {'gold': 40, 'healing potion': 2, 'sword': 1}, ['sword', [4, 6]])
hero.take_damage(10)
```
class LivingThing():
    #...other code in the class...

    def move(self, direction):
        if direction == UP:
            self.location[0] -= 1
        elif direction == DOWN:
            self.location[0] += 1
        elif direction == LEFT:
            self.location[1] -= 1
        elif direction == RIGHT:
            self.location[1] += 1

    #...other code in the class...
Let’s try moving.. (cont.)

- But what about avoiding collisions?

```python
class LivingThing():
    UP, DOWN, LEFT, RIGHT = range(4)
    #...other code in the class...

def move(self, direction, monsters):
    new_location = self.location[:]
    if direction == self.UP:
        new_location[0] -= 1
    elif direction == self.DOWN:
        new_location[0] += 1
    elif direction == self.LEFT:
        new_location[1] -= 1
    elif direction == self.RIGHT:
        new_location[1] += 1
    for monster in monsters:
        if monster.location == new_location:
            return
    self.location = new_location
    #...other code in the class...
```
Let’s try moving.. (cont.)

- Do we only care about collisions?
- Let’s define another class:

```python
class GameBoard():
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.monsters = []
        self.hero = None
        self.trees = []
        self.game_over = False

...
Let’s try moving.. (cont.)

...  

def is_free(self, location):
    if not 0 <= location[0] < self.height or not 0 <= location[1] < self.width:
        return False
    for monster in self.monsters:
        if location == monster.location:
            return False
    for tree in self.trees:
        if location == tree:
            return False
    if location == self.hero.location:
        return False
    return True

...
def put_tree(self, location):
    if self.is_free(location):
        self.trees.append(location)
        return True
    return False

def put_monster(self, monster):
    if self.is_free(monster.location):
        self.monsters.append(monster)
        return True
    return False

def put_hero(self, hero):
    if self.is_free(hero.location):
        self.hero = hero
        return True
    return False
class LivingThing:
    #...other code in the class...

    def move(self, direction, board):
        new_location = self.location[:]
        if direction == UP:
            new_location[0] -= 1
        elif direction == DOWN:
            new_location[0] += 1
        elif direction == LEFT:
            new_location[1] -= 1
        elif direction == RIGHT:
            new_location[1] += 1
        if board.is_free(new_location):
            self.location = new_location
            return True
        else:
            return False

    #...other code in the class...
Do we have to pass ‘board’ each time?

Instead, save as a member!

```python
class LivingThing():
    def __init__(self, name, health, magic_points, inventory, weapon, location, board):
        self.name = name
        self.health = health
        self.magic_points = magic_points
        self.inventory = inventory
        self.weapon = weapon
        self.location = location
        self.board = board

...
Let’s try moving.. (cont.)

...  

def move(self, direction):
    new_location = self.location[:]
    if direction == UP:
        new_location[0] -= 1
    elif direction == DOWN:
        new_location[0] += 1
    elif direction == LEFT:
        new_location[1] -= 1
    elif direction == RIGHT:
        new_location[1] += 1
    if self.board.is_free(new_location):
        self.location = new_location
        return True
    else:
        return False

#…other code in the class…
Are we safe?

- What if we change location directly?

- In Python, every method and member can be accessed directly!

- We use a naming convention to declare “private” members/methods:
  
  ```python
  self._location
  def _get_item():
    self._monsters
    self._width
  ```

- Private members/methods should only be used by their parent object!

- Part of a bigger concept - Encapsulation
Encapsulation

- A driver doesn’t need to know what’s inside his car!
- Easier to code
- Specific implementation doesn’t matter
- Prevents bugs (like changing “location”)
- If direct access is need:
  - Getters – return an inner value

```python
def get_location():
    return self.location
```

- Setters – set an inner value

```python
def set_location(location):
    self.location = location
```

- Still implementation invariant!
Application Programming Interface

- The API defines the functionality an object allows
- Explains how to interact with an instance
- “Design by Contract”
- Not committed to internal implementation!
Everything in Python is an Object!

```python
a = []
a.append(4)
' '.join(a)

b = 5
dir(b)
```

```
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__', '__div__', '__divmod__', '__doc__', '__float__', '__floordiv__', '__format__', '__getattribute__', '__getnewargs__', '__hash__', '__hex__', '__index__', '__init__', '__int__', '__invert__', '__long__', '__lshift__', '__mod__', '__mul__', '__neg__', '__new__', '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']
```
___str___(self)

- returns a String to be printed
- print(obj) ⇔ print(obj.__str__()) ⇔ print(str(obj))

```python
class LivingThing():
    PRINTABLE = {'goblin': 'G', 'dragon': 'D'}  # Better ways to do this – in the future!
    #...other code in the class...

    def __str__(self):
        if self.name in self.PRINTABLE:
            return self.PRINTABLE[self.name]
        else:
            return 'H'

    #...other code in the class...
```
```
class GameBoard():
    #...other code in the class...

def __str__(self):
    board_lists = [[' '] * self.width for rows in range(self.height)]
    for tree_row, tree_column in self.trees:
        board_lists[tree_row][tree_column] = '*'
    for monster in self.monsters:
        monster_row, monster_column = monster.get_location()
        board_lists[monster_row][monster_column] = str(monster)
    hero_row, hero_column = self.hero.get_location()
    board_lists[hero_row][hero_column] = str(self.hero)
    board_strings = ['#' * (self.width + 1)]
    for row in board_lists:
        board_strings.append(''.join(row))
    return '#

#...other code in the class...
```
\_\_repr\_\_(self)

- returns a String to represent the object
- calling the obj in interpreter $\Leftrightarrow$ print(obj.__repr__())

```python
class LivingThing():
    #...other code in the class...

def __repr__(self):
    return 'A LivingThing in Location: ' + str(self.location) + ' By the name: ' + self.name

#...other code in the class...
```
__contains__(self, element)

- Checks if an element is in our object.
- element in obj ⇔ obj.__contains__(element)

```python
class GameBoard():
    #...other code in the class...

    def __contains__(self, living_thing):
        return living_thing in self.heroes or living_thing in self.monsters

    #...other code in the class...
```
### Other special operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td><code>__add__</code>(self, other)</td>
</tr>
<tr>
<td>-</td>
<td><code>__sub__</code>(self, other)</td>
</tr>
<tr>
<td>*</td>
<td><code>__mul__</code>(self, other)</td>
</tr>
<tr>
<td><code>//</code></td>
<td><code>__floordiv__</code>(self, other)</td>
</tr>
<tr>
<td><code>/</code></td>
<td><code>__div__</code>(self, other)</td>
</tr>
<tr>
<td><code>%</code></td>
<td><code>__mod__</code>(self, other)</td>
</tr>
<tr>
<td><code>**</code></td>
<td><code>__pow__</code>(self, other[, modulo])</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td><code>__lshift__</code>(self, other)</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td><code>__rshift__</code>(self, other)</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td><code>__and__</code>(self, other)</td>
</tr>
<tr>
<td><code>^</code></td>
<td><code>__xor__</code>(self, other)</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td><code>__lt__</code>(self, other)</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td><code>__le__</code>(self, other)</td>
</tr>
<tr>
<td><code>==</code></td>
<td><code>__eq__</code>(self, other)</td>
</tr>
<tr>
<td><code>!=</code></td>
<td><code>__ne__</code>(self, other)</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td><code>__ge__</code>(self, other)</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>__gt__</code>(self, other)</td>
</tr>
</tbody>
</table>
class Weapon:
    def __init__(self, name, damage):
        self.name = name
        self.damge = damage

    def get_damage(self):
        return self.damage

    def __str__(self):
        return self.name
What else can we do?

class Armor():
    def __init__(self, name, protection):
        self.name = name
        self.protection = protection

    def get_protection(self):
        return self.protection

    def __str__(self):
        return self.name
class GameBoard:
    #...other code in the class...
    def reset(self):
        self.__init__(self.height, self.width)
        self.put_tree((1,1))
        self.put_tree((1,2))
        self.put_tree((1,3))
        self.put_tree((2,3))
        weapon = Weapon('sword', 6)
        armor = Armor('wooden shield', 1)
        hero = LivingThing('Mark', 50, 80, {'gold': 40, 'potion': 2, 'sword':1}, weapon, [4, 6], armor)
        self.put_hero(hero)
        weapon = Weapon('dagger', 2)
        monster = LivingThing('Goblin', 20, 0, {'gold': 12, 'dagger': 1}, weapon, [3, 3])
        self.put_monster(monster)
What else can we do?

... 

```python
weapon = Weapon('fire', 30)
monster = LivingThing('Dragon', 300, 200, {'gold': 890, 'amulet': 1}, weapon, [1, 5])
self.put_monster(monster)
weapon = Weapon('dagger', 2)
monster = LivingThing('Goblin', 18, 0, {'gold': 15, 'dagger': 1}, weapon, [4, 2])
special.put_monster(monster)
```

#...other code in the class...
class LivingThing():
    #…other code in the class…
    def attack(self, enemy):
        if not enemy.take_damage(self.weapon.get_damage()):
            self.inventory[gold] += enemy.get_inventory[gold]
            return false
        return true

def take_damage(self, damage):
    damage_adjustment = 0
    if self.armor:
        if self.armor:
            damage_adjustment += self.armor.get_protection()
    self.health -= max((damage – damage_adjustment), 0)
    if self.heatl > 0:
        return true
    return false
... 

```python
def random_move(self):
    choice = randint(0, 10)
    if choice in range(4):
        self.move(choice)
    else:
        # attack everybody around us
        self.board.attack_neighbors(self)  # we leave implementing this
                                        # to GameBoard!

    #...other code in the class...
```
class GameBoard:
    def play_turn(self, action):
        if action in self.hero.UP, self.hero.DOWN, self.hero.LEFT, self.hero.RIGHT:
            self.hero.move(action)
        elif action == self.hero.ATTACK:
            self.attack_neighbors(self.hero)
        for monster in self.monsters:
            monster.random_move()
        return not self.game_over

    def attack_neighbors(self, attacker):
        for neighbor in attacker.get_neighbors():
            killed = attacker.attack(neighbor)
            if killed:
                self._kill(neighbor)
...
What else can we do?

...  

```python
def _kill(self, living_thing):
    if living_thing is self.hero:
        self.game_over = True
        return True
    if living_thing in self.monsters:
        self.monsters.remove(living_thing)
        return True
    return False

#...other code in class
```
What else do we need?

class LivingThis():
    #...other code in class

    def get_neighbors(self):
        neighbors = []
        if self is not self.board.get_hero() and self.near(self.board.get_hero()):
            neighbors.append(self.board.get_hero())
        for monster in self.board.get_monsters():
            if self is not monster and self.near(monster):
                neighbors.append(monster)
        return neighbors

    def near(self, living_thing):
        self_row, self_column = self.location
        other_row, other_column = living_thing.get_location
        return abs(self_row-other_row) < 2 and abs(self_column-other_column) < 2

    #...other code in class
What else do we need?

```python
from os import system

class GameRunner:
    def __init__(self):
        self.board = GameBoard()
        self.board.reset()
```

...
What else do we need?

... 

```python
def run(self):
    while True:
        system('cls')
        print(self.board)
        action = self.get_action_from_usr()
        if action == ABORT_GAME:
            break
        alive = self.board.play_turn(action)
        if not alive:
            play_again = input(GAME_OVER_MSG)
            if play_again == PLAY_AGAIN:
                self.board.reset()
            else:
                break
```