PyLadies

(Vienna) 25.4.2020

Who?

International mentorship group with a focus on helping more women become active participants and leaders in the Python open-source community.

Our mission is to promote, educate and advance a diverse Python community through outreach, education, conferences, events and social gatherings.

Agenda for today (remote)

---

1. Object oriented programming in Python
2. Inheritance
3. Exercises, networking, project

Goals

- Be able to create own classes
- Subclasses (inheritance)
- Understand class attributes and methods
- Create a card game as a project

Resources not only for this lecture

- Cheat sheet for Python: https://github.com/ehmatthes/pcc/releases/download/v1.0.0/beginners_python_cheat_sheet_pcc_all.pdf
- Stack overflow and google :)
- Geeks for geeks:https://www.geeksforgeeks.org/object-oriented-programming-in-python-set-1-class-and-its-members/

Objects

- Python - basically everything you can store into variable is an object
- Objects contain data and have *behaviour* (instructions what to do with data)
- string object
  - contains information (text)
  - has methods like count or lower
- If string would not be object, python would have to implement much more functions specific for data

Objects

- **Attributes** of each object are specific for given object
- But **methods** are same for all objects from a same class

Classes

- to find a class of your object, use function type
  - `type('abc')`
  - `type(True)`
  - `type(12)`
- What is a class? It's a description how every object of the same type behaves

Classes

- Python classes are callable as they were functions:
  - `string_class = type('abc')`
  - `string_class(8)`
  - `-> it is behaving as str()`
- This way we now know that functions str, int, float and others are not function, but actually classes!
  - `type('abcdefgh') == str`
- Classes can create objects (object is instance of class)

Custom classes

- Useful when you need different objects with similar behaviour
- Let’s start with class named Kittie

```
class Kittie:
    def meow(self):
        print("Meow!")
```

Custom classes

- Creation of the object - be careful about uppercase letters (consensus)

```
kittie = Kittie()
```

- And now we can call the previously defined method on our kittie

```
kittie.meow()
```

Attributes

- Custom classes can have attributes
- Attributes - information that is stored by the instance of the class

```
misty = Kittie()
misty.name = 'Misty'
print(misty.name)
```

Attributes

Objects are connecting data and behaviour!

Most of the times behaviour is defined in the class and data are stored in attributes of the objects. We can differentiate Kitties, for example, by their names because of the attributes.

Self parameter

- Each method has access to the specific object (instance of class) on which is called just because of parameter self.
- You can modify the kitties so they will add their name to the meowing!

```
class Kittie:
    def meow(self):
        print("{}: Meow!".format(self.name))
```

Self parameter

```
misty = Kittie()
misty.name = 'Misty'
misty.meow()
```

The command misty.meow called a method which when it's called assigns the object misty as first argument to the function meow.

Other arguments

```
class Kittie:
    def eat(self, food):
        print("{}: I eat {}!".format(self.name, food))

tiger = Kittie()
tiger.name = 'Tiger'
tiger.eat('fish')
```

__init__

- Passing arguments to the class in the moment it is created
- Problem in previous code? You need to add attribute name in order the meow function works

```
smokey = Kittie(name='Smokey')
```

__init__

```
class Kittie:
    def __init__(self, name):
        self.name = name
    def meow(self):
        print("{}: Meow!".format(self.name))
smokey = Kittie('Smokey')
smokey.meow()
```

__init__

- This way, you can’t create Kittie without a name and functions will always work
- There are other important methods like **__str__** which will convert object into string
- Or **__dict__** returns all attributes that an object has

Kitties - complete

```
class Kittie:
    def __init__(self, name):
        self.name = name
    def meow(self):
        print("{}: Meow!".format(self.name))
    def eat(self, food):
        print("{}: Meow meow! I like {} very much!".format(self.name, food))
```

Cat exercise

- The Cat can meow with the meow method.
- The Cat has 9 lives when she's created (she can't have more than 9 and less than 0 lives).
- The Cat can say if she is alive (has more than 0 lives) with the alive method.
- The Cat can lose lives (method lose_life).
- The Cat can be fed with the eat method that takes exactly 1 argument - a specific food (string). If the food is fish, the Cat will gain one life (if she is not already dead or doesn't have maximum lives).

Inheritance

- We now have class for Kitties and we can similarly create class for dogs:

```
class Doggie:
    def __init__(self, name):
        self.name = name
    def woof(self):
        print("{}: Woof!".format(self.name))
    def eat(self, food):
        print("{}: Woof woof! I like {} very much!".format(self.name, food))
```

Inheritance

- Most of the code is the same!
- There is a mechanism to avoid copying things all the time

- Kitties and doggies are animals. So you can create a class for all animals, and specify everything that applies to all animals. In the classes for each animal, you just write the specifics.

Inheritance

```
class Animal:
    def __init__(self, name):
        self.name = name
    def eat(self, food):
        print("{}: I like {} very much!".format(self.name, food))

class Kittie(Animal):
    def meow(self):
        print("{}: Meow!".format(self.name))

class Doggie(Animal):
    def woof(self):
        print("{}: Woof!".format(self.name))
```

Inheritance - summary

- With the command: `class Kittie(Animal)` you are telling Python that the class Kittie inherits behaviour from the class Animal.
- (In other programming languages: Kittie is derived from Animal or it extends Animal. Derived classes are called subclasses and the main one is the superclass)
- When Python searches for a method or attribute, and it can’t find it in the class itself it looks into the superclass. So everything that is defined for Animal applies to Kittie (unless you overwrite it).

Overwriting methods

If you don't like some behaviour of the superclass, you can define a method with the same name in the subclass:

```
class Kittie(Animal):
    def eat(self, food):
        print("{}: I don't like {} at all!".format(self.name, food))

smokey = Kittie('Smokey')
smokey.eat('dry food')
```

super()

- Sometimes you want some behaviour from original method in overwritten method.
- You need to pass everything you want to use into this method called with **super**

```
class Kittie(Animal):
    def eat(self, food):
        print("({} is looking at {})".format(self.name, food))
        super().eat(food)

smokey = Kittie('Smokey')
smokey.eat('dry food')
```

Polymorphism

When we know that Kittie and Doggie and any other similar class are animals, we can create a list of animals, but we don't care which animals they are specifically:

This is quite important behaviour of subclasses: When you have a Kittie, you can use it anywhere where a program expects an Animal, because each kittie is an animal.

Polymorphism

- Each kittie or doggie is an animal, each cabin or house is a building. In those examples, heredity makes sense.
- If heredity does not make clear sense, sometimes it is better to use ‘has’ (tree has leaves) - leave is not a subclass of tree.
- Liskov substitution principle

Generalisation

- Meow can be written in a better way (and named in a better way) so it is possible to be used for all animals:

```
animals = [Kittie('Smokey'), Doggie('Doggo')]

for animal in animals:
    animal.speak()
    animal.eat('meat')
```

Project - simplified text based Blackjack game

- Rules: 52 card deck (JQK=10, A=1/11 - player choice),
- Player against Dealer, player dealt 2 cards, dealer 1
- 21 limit (Blackjack), if 22 or more total - bust=lose
- Player tells if he wants another card or not, one by one
- Dealer played automatically, if total > 17, stops, else draws new cards until bust or total > 17[1]
- After both stop, show cards & score, winner gets bets, new round

[1] **Dealer and A(11/1):** If dealer has A and counting it as 11 would bring the total to 17 or more (but not over 21), the dealer must count the ace as 11 and stand.

Smaller project - text based Blackjack game

- You can start without Classes, update and modify later
- Following guidelines not mandatory for anyone :)
- First think about the structure of task, divide it to separate steps, do not start coding right ahead
- Test each individual component first if possible
- Example Blackjack Python code using classes (however not ideal implementation) https://github.com/UndeadFairy/pyladies_vienna/tree/master/OOP

Blackjack - steps guidelines

- Deck of cards - shuffle it
- Deal initial hand
- Calculate score
- Check win/lose
- Draw a new card
- Add “game” logic - player asked to draw using input()
- Then if player not bust, dealer plays automatically
- Announce winner
- Handle ace 1/11

Resources and materials general

- advent of code - adventofcode.com
- hackerrank - hackerrank.com
- Django Girls django tutorial -> (29th August in Vienna)
- https://www.practicepython.org
- Nice Python exercices at one place
  https://github.com/tystar86/python_exercises
- https://automatetheboringstuff.com
- https://diveintopython3.problemsolving.io

Next topics

Databases

Graphics

Testing

Flask

fill the form regarding your interests please :) → https://forms.gle/UtfgVGe6AhhRwx539

Thank you and see you next time

Coding session - **7.5.2020**

Next workshop - **16.5.2020** Testing in Python! + Webtesting (make your “user” with Selenium)