Introduction à la Programmation Orientée Objet (POO)

1. Le concept de classe

La notion la plus importante en programmation orientée objet est le concept de classe.

Les classes sont des **moules**, des **patrons qui permettent de créer des objets en série sur le même modèle**.

On peut se représenter une classe comme le schéma de construction ainsi que la liste des fonctionnalités d'un ensemble d'objets.

Définir une classe permet de créer des objets identiques, à la chaîne

En programmation orientée objet, on n'a affaire qu'à des classes et des objets (ou instance de classe). Tous les **éléments manipulés en programmation objet sont des objets** (d'où le nom) **dont la construction repose sur la définition d'une classe**.

1.1 Comment créer une classe avec Python ?

En utilisant ce modèle, nous pourrons alors **créer des objets**, appelées **instances**. Pour **créer des classes en Python**, on utilise le mot-clef **class**:

```
1  class Animal:
2      pass
```

Dans cet exemple, on crée une classe nommée **Animal**.

On choisit ainsi le modèle qu'auront les différentes instances de la classe Animal.

Pour l'instant cette classe est très simple. Elle va être complexifiée.

Pour **instancier cette classe**, nous allons **appeler** la **classe** Animal **comme si elle était une fonction**:

```
16  animal1 = Animal()
17  animal2 = Animal()
```

On a ainsi créé **deux objets, deux instances** de la classe **Animal**: animal1 et animal2.

Si on affiche le type de ces deux objets grâce à la fonction **type**, on voit que ces deux objets appartiennent bien à la classe **Animal**.

```
30  type_classe = type(animal1)
31  print(type_classe)
```

```
<class '__main__.Animal'>
```

1.2 Les constructeurs

Les **constructeurs** peuvent être considérés comme des fonctions **dans un premier temps** (pas exactement mais gardons ce mot pour l'instant).

Très importantes: ce sont les fonctions qui sont **appelées lorsqu'un objet est créé**.

Dans la plupart des cas, il faut le définir.

Par exemple, si nous voulons **ajouter un affichage qui nous prévient de la création d'un objet** de la classe **Animal**, nous allons définir la fonction **__init__** comme suit:

```python
class Animal:
    def __init__(self):
        print("un animal vient d\'etre cree")
        pass
```

Cette fonction **__init__** prend un argument très important: **self**.

Ce mot-clef **désigne l'objet lui-même**. Ce mot est **très important**.

1.3 Les attributs

Il faut **ajouter des caractéristiques** à nos animaux pour qu'ils soient intéressants.
Ces **caractéristiques** sont ce qu'on appelle les **attributs**.
Par exemple, nous allons donner un âge à tous nos objets de la classe **Animal**:

```python
class Animal:
    def __init__(self):
        print("un animal vient d\'etre cree")
        pass
        self.age = 0
```

Dans la **fonction __init__**, on a ajouté la définition d'une **variable self.age** qui vaut 0.
Lorsqu'on définit self.<quelquechose>, on définit un attribut des objets de la classe.
Ainsi toutes les instances de la classe Animal auront un **attribut** age qui lors de la création de l'objet vaudra 0.

Pour accéder aux attributs d'un objet, on utilise un point.

Il faut donc voir les **attributs** comme les **caractéristiques des objets d'une classe:** tous les objets qui seront créés à partir de cette classe (**qui instancieront cette classe**), posséderont ces mêmes caractéristiques.

En les définissant dans la fonction **__init__**, toutes les instances de cette classe auront ces caractéristiques.

On peut modifier ces valeurs, leur donner une toute autre valeur objet par objet sans problème mais elles seront initialisées de la même façon:

```
17    animal2.age = 25
```

1.4 Synthèse sur le concept de classe

- Les **classes** sont des **modèles** pour des **objets**
- Ces **modèles** peuvent avoir des **caractéristiques** représentées par des **attributs**
- Ces **objets** ou **instances de classe**, sont créés grâce à une "fonction" spéciale, **le constructeur**.

2. Les méthodes

Rappel par un exemple : si la **classe** prise est **Citoyen**, les **attributs** des instances de cette classe peuvent être nom, prénom, sexe, date de naissance, lieu de naissance, l'identifiant, la signature et la taille.

Définir une classe Citoyen serait donc comme définir un modèle de carte d'identité vide

2.1 Méthodes vs Fonctions

Si fonctions et méthodes peuvent prendre des **entrées** et renvoyer des **sorties**, les fonctions ne sont pas relatives à un objet.

En **programmation orientée objet**, les fonctions n'existent pas puisque tout est objet, c'est-à-dire instance de classe.

Grâce à une méthode, on va pouvoir réaliser des **opérations** qui sont **spécifiques à un objet**: modifier ses attributs, les afficher, les retourner (ou les initialiser dans le cas de la méthode __init__).

2.2 Comment créer une méthode

Si nous revenons à notre exemple de la **classe** Animal pour laquelle on a défini un attribut age. La **méthode** __init__ permet d'initialiser l'attribut age à 0.

Nous allons pouvoir créer une **méthode vieillir** qui va **ajouter 1 à l'attribut age**:

```python
class Animal:
    def __init__(self):
        print("un animal vient d'etre cree")
        pass
        self.age = 0

    def vieillir(self):
        self.age = self.age + 1
        pass
```

Si on s'intéresse à la **définition de cette méthode** plus en détail, on remarque l'utilisation du **mot-clef *def*** qui introduit normalement la définition d'une fonction. Mais cette déclaration de méthode est indentée de manière à se retrouver **"à l'intérieur" de la définition de la classe**.

Grâce à cette définition, tous les objets qui sont des **instances de cette classe** Animal auront la possibilité d'**appeler cette méthode**:

Pour **appeler une méthode d'instance**, on utilise un **point** et puisqu'il s'agit d'une sorte de fonction, on utilise des parenthèses.

La définition d'une méthode fonctionne de la même façon que celle d'une fonction pour les **arguments en entrée, les sorties** …

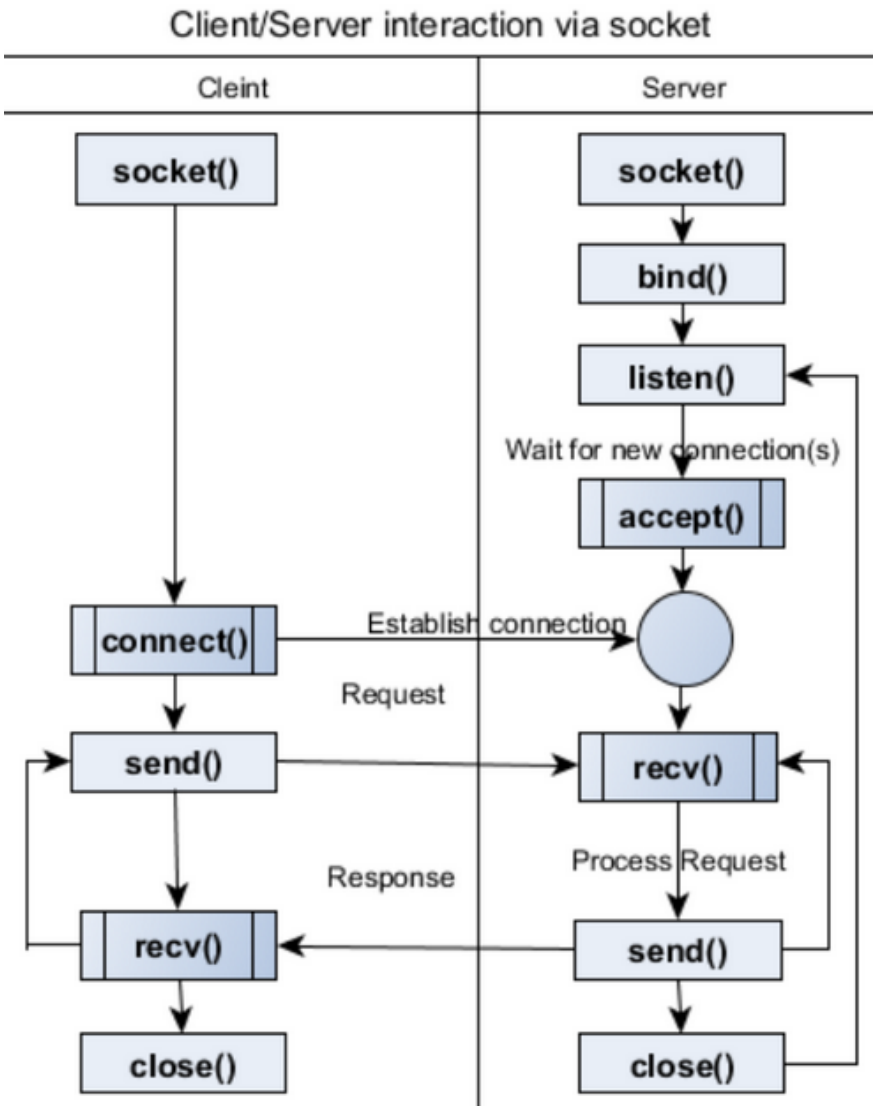
On pourra ainsi utiliser des arguments supplémentaires à **self**.

```python
class Animal:
    def __init__(self):
        print("un animal vient d\'etre cree")
        pass
        self.age = 0

    def vieillir(self):
        self.age = self.age + 1
        pass

    def nommer(self, nom):
        self.nom = nom
```

```python
animal1 = Animal()
animal2 = Animal()

animal2.age = 25

animal2.vieillir()

print(animal1.age)
print(animal2.age)

animal1.nommer('Milou')

print(animal1.nom)
```

Par exemple, si on veut donner un nom à nos animaux, on peut introduire un attribut nom dans la **méthode __init__** et construire une méthode nommer qui permettra de **changer la valeur de l'attribut** nom:

```python
class Animal:
    def __init__(self):
        print("un animal vient d\'etre cree")
        pass
        self.age = 0
        self.nom = 'toutou'
```


Client/Server interaction via socket
Cleint
Server
socket()
socket()
bind()
listen()
Wait for new connection(s)
accept()
connect()
Establish connection
Request
send()
recv()
Process Request
Response
recv()
send()
close()
close()

server.py

```python
#!/usr/bin/env python

# skeleton from http://kmkeen.com/socketserver/2009-04-03-13-45-57-003.html

import socketserver, subprocess, sys
from threading import Thread
from pprint import pprint
import json

my_unix_command = ['bc']
HOST = 'localhost'
PORT = 2000

class SingleTCPHandler(socketserver.BaseRequestHandler):
    "One instance per connection.  Override handle(self) to customize action."
    def handle(self):
        # self.request is the client connection
        data = self.request.recv(1024)  # clip input at 1Kb
        text = data.decode('utf-8')
        pprint(json.loads(text))
        self.request.send('OK'.encode('utf-8'))
        self.request.close()
```

```
23
24  class SimpleServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
25      # Ctrl-C will cleanly kill all spawned threads
26      daemon_threads = True
27      # much faster rebinding
28      allow_reuse_address = True
29
30      def __init__(self, server_address, RequestHandlerClass):
31          socketserver.TCPServer.__init__(self, server_address, RequestHandlerClass)
32
33  if __name__ == "__main__":
34      server = SimpleServer((HOST, PORT), SingleTCPHandler)
35      # terminate with Ctrl-C
36      try:
37          server.serve_forever()
38      except KeyboardInterrupt:
39          sys.exit(0)
```

https://gist.github.com/pklaus/c4c37152e261a9e9331fn