

# NSI Terminale - Données

## Introduction à la programmation objet

qkzk

2019/12/25

## Introduction à la programmation objet

### Programmation modulaire

#### Pourquoi la programmation modulaire ?

- développement logiciel
- modification et maintenance logicielle
- ré-utilisabilité
- création de nouveaux types de données

### Mise en oeuvre

#### Première, données en table, “course au chicon”

- le module `Competitor.py` permet de manipuler des valeurs représentant les compétiteurs de la course. (...)
- Les performances des compétiteurs vont être représentées par leur temps de course (...) Créer un module `Time.py` qui définit le type `Time`

#### Première, “rpg texte”

- expression d’un typage de données à représenter :

*Une correction possible qui utilise les dictionnaires pour modéliser les combattants.*

#### définition d’un type `Competitor`

```
def create(first_name, last_name, sex, birth_date, bib_num):
    """ (...)
    :return: a new record for this competitor
    :rtype: Competitor
    """
    return {
        'bib_num': bib_num,
        'first_name': first_name
    }

def get_firstname(comp):
    """
    :param comp: a competitor
    :type comp: Competitor
    :return: first name of competitor comp
    (...)
    """
    return comp['first_name']
```

## type Competitor : vraiment ?

```
>>> import Competitor_chicon as Competitor
>>> comp = Competitor.create('Alice', 'L', 'F', '2019/12/24', 1)
>>> Competitor.get_firstname(comp)
'Alice'
```

mais :

```
>>> type(comp) == Competitor
False
>>> type(comp) == dict
True
```

## vraiment ?

```
import Competitor_chicon as Competitor
import Time_chicon as Time
...
comp = Competitor.create(...)
Competitor.get_birthdate(comp)
Competitor.to_string(comp)
Competitor.compare(comp, other_comp)
```

```
t = Time.create(...)
Time.to_string(t)
Time.compare(t, other_time)
```

- Si comp est de type Competitor et t est de type Time, alors pourquoi devoir préfixer to\_string par Competitor pour comp et par Time pour t ?

---

```
# dans Competitor
def set_performance(comp, time):
    comp['performance'] = time

# dans wator
def joueur(nom=None, vie=5, force=1):
    if nom is None:
        raise ValueError("Vous devez donner un nom à votre joueur")
    return {"nom": nom,
            "vie": vie,
            "force": force}
>>> robert = Joueur('Robert')
>>> robert
{'nom': 'Robert', 'vie': 5, 'force': 1}
```

Le type n'existe que "dans la tête" du programmeur, les données et les traitements sont séparés.

## Programmation Orientée Objet

Approche de la modélisation du problème à résoudre en terme d'objets :

- on identifie : les "familles" d'objets du problème  
Un objet est une modélisation d'une entité du monde réel ou d'un concept
- on en déduit
  - les abstractions = les classes
  - les fonctionnalités (= traitements/services) dont on a besoin pour chacune

- 
- **421** : Dés/Groupe de 3 dés/Joueur/Partie
  - **combat rpg** : Combattant/Arme/Combat etc.
  - **chicon** : Compétiteur/Temps(performance)/Course(/Palmares)
  - **minimax** : Jeu/Situation/Joueur/Algorithme/Fonction évaluation

## langage à objet

Alan Kay *SmallTalk*

- tout est objet
- chaque objet a un **type**
- chaque objet a sa propre mémoire, constituée d'autres objets
- tous les objets d'un type donné peuvent avoir les mêmes messages
- un programme est un regroupement d'objets qui interagissent par **envois de messages**

## type

c'est quoi un **type** ?

booléen, entier, Competiteur, Temps

## type

un **type** de données définit

- l'ensemble des valeurs possibles pour les données type
- les opérations applicables sur ces données

## classes

une **classe** est un type d'objet

une classe définit

- la liste des **méthodes** et les traitements associés  
-> le comportement des objets
- la liste des **attributs** nécessaires à la réalisation des traitements  
-> l'**état** des objets

les méthodes portent les traitements (comportement, actions) les attributs portent les données

classe = définition d'un modèle pour les objets de la classe

classe = abstraction (on programme des définitions)

## instance

- une classe permet de **créer** des objets
- ces objets sont les valeurs du type de cette classe

## instance

on appelle **instance** un objet créé par une classe

tout objet est instance d'une classe

nécessité d'un **constructeur** dans une classe  
double rôle : construire l'objet et initialiser son état

## méthodes et attributs

### méthode

- une **méthode** est une fonction qui appartient à une classe

*"function member"*

ne peut être utilisée (*appelée, invoquée*) que par les instances de la classe qui la définit

## attribut

- un **attribut** est une donnée qui appartient à un objet  
“*data member*”  
les attributs sont définis par la classe de l'objet

## En Python

### En Python

cf. *mytime.py*

- constructeur : `__init__`
  - initialisation de l'état (attributs)un seul constructeur possible en Python
- une classe est un type (`type()`, `isinstance()`)
- **self** : auto-référence = “l'objet dont on est en train de parler”  
ie. celui que l'on construit ou celui qui invoque (utilise) la méthode
  - > permet d'accéder aux attributs de l'objet : (cf. `__init__`, `get_hours()`)
    - **this** en javascript, java
    - **self** n'est *pas* imposé en Python mais **très** fortement recommandé
    - **self** ne doit jamais être modifié

## méthodes

- **méthode d'objet** vs **méthode de classes**

### méthodes

- **méthodes d'objets** : invoquée par l'objet  
= envoi de messages possibles
  - premier paramètre = **self** (cf `get_hours`, `__init__`)
  - **self** est lié à l'objet utilisé pour invoquer la méthode  
notation pointée : `t1.to_seconds()` -> **self** lié à `t1`
  - permet d'accéder aux attributs de l'objet ou d'invoquer une méthode sur cet objet. cf **compare**

### méthodes

- **méthodes d'objets** : invoquée par l'objet  
= envoi de messages possibles
  - premier paramètre = **self** (cf `get_hours`, `__init__`)
  - **self** est lié à l'objet utilisé pour invoquer la méthode  
notation pointée : `t1.to_seconds()` -> **self** lié à `t1`
  - permet d'accéder aux attributs de l'objet ou d'invoquer une méthode sur cet objet. cf **compare**
- **méthode de classe** : méthode ne dépendant pas d'un objet : **statique** appelée via la classe :  
`Time.from_seconds()` (= fonction, pas OO)  
NB existence des décorateurs `@staticmethod`, `@classmethod`  
les attributs de classe sont également possible `Time.BASE`

## méthodes spéciales

- permet de définir des “opérateurs”
  - `__add__` +, `__mul__` \*, `__sub__` -
  - `__eq__` ==, `__ne__` !=
  - `__lt__` <, `__ge__` <=, `__gt__` >, `__le__` >=

- `__repr__` : dans l'interpréteur : `>>> obj`
- `__str__` : `str(obj)` et `__len__` : `len(obj)`
- `__getitem__` : `obj[i]`
- `__iter__` : `for v in obj`

## encapsulation

\_\_\_\_\_cœur de la POO en NSI

### encapsulation

Les données (attributs) sont regroupées avec les traitements qui les manipulent (méthodes)

- l'encapsulation implique le **masquage des données**
  - l'objet a la maîtrise de ses attributs *via ses méthodes*
  - seules les méthodes sont accessibles

### règle d'or

**les attributs sont déclarés privés** = accessibles uniquement au sein de la classe

en Python, identifiant préfixé de `__`  
on peut aussi définir des méthodes privées.

### exemple

On veut modéliser les nombres complexes.

Quelles sont les fonctionnalités attendues ? *besoin piloté par application*  
càd que souhaite-t-on pouvoir faire avec une donnée de type **Complex** ?

### exemple

On veut modéliser les nombres complexes.

Quelles sont les fonctionnalités attendues ? *besoin piloté par application*  
càd que souhaite-t-on pouvoir faire avec une donnée de type **Complex** ?

- construire (*partie réelle et partie imaginaire / module et argument*)
- accéder aux "données" :
  - partie réelle, partie imaginaire, module, argument
- obtenir (*calculer*) le conjugué
- additionner deux complexes *pour en obtenir un nouveau*
- avoir une représentation textuelle d'un complexe
- etc. (*égalité, multiplication, ...*)

On a donc besoin de **l'interface publique** suivante pour **Complex**

```
class Complex(builtins.object)
|   Complex(x, y)
|
|   Methods defined here:
|
|   __add__(self, other)
|       return Complex résultat de self+other
|
|   __init__(self, x, y)
|       create Complex number [ x+i.y ]
|
|   arg(self)
|       return argument de ce Complex
|
```

---

```

| conjugue(self)
|     return conjugue de ce Complex
|
| from_module_arg(module, arg)
|     build Complex number [ module.e^i(arg) ]
|
| im(self)
|     return partie imaginaire de ce Complex
|
| module(self)
|     return module de ce Complex (of this complex)
|
| re(self)
|     return partie réelle de ce Complex
|

```

## usage

- Ces informations sont suffisantes pour utiliser le type `Complex` dans un module.  
*cf. main\_complex.py*
- Quelle est la représentation des données pour `Complex` ? *cf. complex1.py et complex2.py*  
*pas d'impact sur main\_complex.py*
- Est-ce important ? *cf. mytime2.py*

## séparation de l'interface et de l'implémentation

- **interface publique d'une classe**  
= ensemble des méthodes *publiques* définies par la classe  
= ensemble des services que peuvent rendre les objets

## intérêt ?

- la représentation des données utilisée n'a pas besoin d'être connue elle pourra donc **évoluer** sans perturber l'existant "code client"
  - ce qui compte c'est ce que l'on peut faire, pas comment on le fait  
*en partant du principe que c'est bien fait.*
  - possibilité d'ajouter du contrôle
    - accès en lecture seulement d'un attribut  
`get_hours()` mais pas `set_hours()`
    - contrôle des valeurs classe `Person` avec attribut `__age`  

```

def set_age(self, new_age):
    if new_age < 0:
        new_age = 0
    self.__age = new_age

```
- classe `BankAccount`, accès au solde `get_balance()` contrôlé par code

- 
- lorsque l'on fait l'analyse objet d'un problème, on cherche à déterminer les **services** que doivent rendre les objets  
= les **méthodes**
  - les attributs n'apparaissent que lorsque l'on se pose la question de la mise en oeuvre des méthodes, càd. de leur **implémentation**.  
un attribut existe parce qu'il permet l'implémentation d'une méthode

## exemple

On doit représenter des disques. On a besoin de connaître le rayon, diamètre, aire, périmètre.

- classe `Disc` + méthodes

`get_radius()`, `get_diameter()`, `get_aera()`, `get_perimeter()`

- attributs ?  
dépendent de choix d'implémentation...
- 

### implémentation avec “rayon”

```
def radius(self):  
    return self.__radius  
def diameter(self):  
    return 2 * self.__radius  
def perimeter(self):  
    return 2 * math.pi*self.__radius
```

### implémentation avec “diamètre”

```
def radius(self):  
    return self.__diameter / 2  
def diameter(self):  
    return 2 * self.__diameter  
def perimeter(self):  
    return math.pi*self.__diameter
```

## javascript

cf. *Time.js*

- similarités
  - class
  - constructor
  - this
- différences
  - pas `this/self` en paramètres des méthodes
  - mot-clé `static`
  - pas de notion “privé” mais : voir `getters/setters` (`get/set xxx`) dans version 2 de *Time2.js*

## Polymorphisme / héritage

### Polymorphisme / héritage

- hors programme, donc pas abordé ici
- J'ai des sources si ça vous intéresse.
- permet de créer des objets répondant à des contraintes susceptibles *d'évoluer*...  
donc de *maintenir* du code.