Programmation en Python

Anne Garcia-Sanchez

M2i - CFA CCI Avignon - cyber2dev

14 mai 2024

programmation procédurale

exemple:

pour enregistrer des données sur un étudiant, choix de différents types de données possibles: tuples, listes, dictionnaires

```
def get_student():
    name = input("Nom: ")
    section = input("Section: ")
    return {"name": name, "section": section}


student = get_student()
if student["name"] == "paul":
    student["section"] = 'L3'
print(f"{student['name']}, section {student['section']}")
```

classes

- classes en POO: permettent de créer nos propres types de données et de leur donner un nom
- classe: peut être vue comme un moule pour un type de données
- création d'objets: instances de la classe

premier exemple utilisant une classe

première lettre de nom de classe en capitale

définition d'attributs

```
class Student:
    ...

def get_student():
    student = Student()
    student.name = input("Nom: ")
    student.section = input("Section: ")
    return student

student = get_student()
print(f"{student.name}, section {student.section}")
```

méthodes

création de fonctions appelées *méthodes* dans la classe `Student` qui déterminent le comportement d'un objet de la classe `Student`

exemple: initialisation

`self` permet d'accéder à l'objet qu'on est en train de construire

```
class Student:
    def __init__(self, name, section):
        self.name = name
        self.section = section

def get_student():
    name = input("Nom: ")
    section = input("Section: ")
    return Student(name, section)

student = get_student()
print(f"{student.name}, section {student.section}")
```

méthodes

POO: encapsulation de toutes les fonctionnalités liées à la classe créée

exemple: validation des entrées

```
class Student:
    def __init__(self, name, section):
        if not name:
            raise ValueError("nom manquant")
        if section not in ["L3", "M1", "M2"]:
            raise ValueError("section invalide")
        self.name = name
        self.section = section
```

méthode *string*

`__str__`: méthode built-in liée à `print` à adapter à la classe

```
class Student:
    def __init__(self, name, section):
        if not name:
            raise ValueError("nom manquant")
        if section not in ["L3", "M1", "M2"]:
            raise ValueError("section invalide")
        self.name = name
        self.section = section

    def __str__(self):
        return f"{self.name} en {self.section}"


def get_student():
    name = input("Nom: ")
    section = input("Section: ")
    return Student(name, section)

student = get_student()
print(student)
```

méthodes

```python
class Student:
    def __init__(self, name, section):
        self.name = name
        self.section = section

    def __str__(self):
        return f"{self.name} en {self.section}"

    def back_to_school(self):
        match self.section:
            case "L3":
                return "04/09/2023"
            case "M1":
                return "18/09/2023"
            case "M2":
                return "04/09/2023"
            case _:
                return "inconnu"

def get_student():
    name = input("Nom: ")
    section = input("Section: ")
    return Student(name, section)

student = get_student()
print(student)
print("rentrée 2023: ")
print(student.back_to_school())
```

propriétés, décorateurs, *Getter*, *Setter*

renforcement du code avec des *propriétés*

propriétés définies à l'aide de *décorateurs* de fonctions

décorateurs = fonctions qui modifient le comportement de fonctions

mise en place de *Getter* et *Setter* pour gérer l'affectation avec des valeurs valides

propriétés, décorateurs, *Getter*, *Setter*

```python
class Student:
    def __init__(self, name, section):
        if not name:
            raise ValueError("nom manquant")
        self.name = name
        self.section = section

    def __str__(self):
        return f"{self.name} en {self.section}"

    # Getter
    @property
    def section(self):
        return self._section

    # Setter
    @section.setter
    def section(self, section):
        if section not in ["L3", "M1", "M2"]:
            raise ValueError("section invalide")
        self._section = section

def get_student():
    name = input("Nom: ")
    section = input("Section: ")
    return Student(name, section)

student = get_student()
student.section = "n'importe quoi" # devient impossible
print(student)
```

__name__

student.py - /home/anne/Documents/PYTHON/DEMO/OOP/student.py (3.10.12)

File Edit Format Run Options Window Help

```
class Student:
    def __init__(self, name, section):
        self.name = name
        self.section = section

    def __str__(self):
        return f"{self.name} en {self.section}"

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    section = input("Section: ")
    return Student(name, section)

if __name__ == "__main__":
    main()
```

mystudent.py - /home/anne/Documents/PYTHON/DEMO/OOP/mystudent.py (3.10.12)

File Edit Format Run Options Window Help

```
import student

stud = student.Student('paul', 'L3')
print(stud)
```