

Instructions générales

- En cas de problèmes, référez-vous au sujet, références bibliographiques, recherchez sur internet, copiez vos message d'erreurs dans google, etc... Si malgré tout vous restez sans réponses, posez votre question au chargé de TP.
- Utilisez l'éditeur de texte de votre choix pour les exercices. En cas d'hésitation, privilégiez gedit ou idle.
- Si ce n'est pas encore le cas, créez un dossier IN1260/.
- Dans ce dossier IN1260/, créez un nouveau sous-dossier pour la séance de TD, que vous nommerez TD01/.
- Au debut de chaque nouveau TD copiez le code de l'ancien TD avant d'effectuer des modifications. Par exemple: `cp -r TD1 TD2`
- Documentez et commentez précisément votre code : décrivez l'objectif général de l'algorithme, expliquez les points-clefs de son implémentation, détaillez les conditions normales d'exécution du script (c.-à-d. le fichier ".py") et exposez les cas extrêmes et les erreurs prévues.

Prérequis

- Algèbre linéaire
- Connaissances de bases en Python3
- Structures de données
- Récursivité

Objectif

Pendant cette première séance, nous allons nous familiariser avec le sujet ainsi que les outils de développement. Plus précisément nous allons:

- Introduire la notion de classe python et intrinsèquement celle d'objet et d'instance
- Utiliser des classes pour un regroupement fonctionnel de différentes entités
- Interpréter le code python

1 Introduction

La finalité de ce projet consiste à implémenter un lancé de rayons. Nous pouvons regarder ce travail comme une réponse à un regroupement d'exigences formalisées dans le cahier des charges qui fournit les spécifications fonctionnelles de ce qui doit être réalisé.

La validation du projet implique la réalisation de trois étapes:

- Pré-traitement
- Traitement
- Post-traitement

Théorie

2 Mise en place

Objectif: *Introduire des concepts avancés du langage.*

Python est un langage multi-paradigme. Malgré le fait que le langage sera regardé, au niveau de ce cours, comme un simple outil pour la réalisation du projet, la compréhension de quelques éléments plus avancés du langage faciliteront l'implémentation.

2.1 Une vue sur la réalisation des implémentations

Le génie logiciel (anglais software engineering) est une science de génie industriel qui étudie les méthodes de travail et les bonnes pratiques des ingénieurs qui développent des logiciels. Cette science fournit les critères d'évaluation qualitative ainsi que les moyens pour les atteindre. À partir de ces bonnes pratiques, nous allons retenir un sous-ensemble qui consiste en des éléments qui visent la lisibilité et l'évolutivité du logiciel final.

Un algorithme est une méthode générale pour résoudre un ensemble de problèmes. La réalisation de notre projet nécessitera l'utilisation d'un ensemble d'algorithmes connus et ad-hoc qui vont composer le lancé de rayons. Ces algorithmes se basent sur des structures de données. En informatique, une structure de données est une structure logique destinée à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement.

Le choix de structures de données est donc très important, qu'il soit des structures du langage Python ou des structures ad-hoc que vous allez introduire à l'aide de la notion de classe (voir section 2.1.1).

Une des bonnes pratiques consiste à séparer, autant que possible, la partie modèle (données) de la partie traitement. Nous allons également privilégier le regroupement fonctionnel et la factorisation du code.

Le regroupement fonctionnel offre une clarté du code à travers le rassemblement des variables, données, manipulations dans la même fonction, classe ou module. Le choix du nom de ce regroupement est également important. Il devrait être:

- succinct

- représentatif du traitement effectué
- suivre la même convention de nommage

La factorisation du code consiste à regrouper une partie de traitement qui s'applique plusieurs fois dans une fonction, par exemple, qui pourra s'appliquer à différentes valeurs des paramètres. C'est toujours une bonne idée de créer des fonctions une fois qu'on observe un traitement répétitif (une suite de plusieurs lignes qui se répète) pour des raisons de lisibilité (noms de variables explicites), ainsi que d'évolutivité (en cas de modification à apporter, elles seront effectuées à un seul endroit).

sources/module_ex1.py

```
#imports
import long_module_name as short_name
from module_name import *
from module_name import a_function

#def of classes and functions

def funDoesNothing(oneArgument):
    pass
```

Complements Python Création d'un programme hello world.

```
console: mkdir IS1260
console: cd IS1260
console: python3

>>> print("Hello World")
```

Voici une meilleure approche.

sources/hello_world.py

```
def printGenericHello(dest = None):
    """Prints "Hello " + the value of its argument and returns 0 if no error
    occurs or -1 otherwise.

    Args:
        dest: optional argument (of type string) – if a string is provided
        the function prints "Hello " followed by the string interpretation
        of the argument.

    Returns:
        0: if no error occurred
        -1: in case of non string argument
    """
    if (dest == None):
        print("Hello World")
    else:
        if (type(dest) == str):
            print("Hello " + dest)
        else:
            print("Hello ???")
            return -1
    return 0
```

```
def helloUnitTest():
    assert printGenericHello(0) == -1
    assert printGenericHello() is 0
    assert printGenericHello("") is not -1
    assert printGenericHello("World") is "Hello World", "return<print"

if __name__ == '__main__':
    printGenericHello()
    printGenericHello("World")

helloUnitTest()
```

Le résultat de l'interprétation `python3 hello_world.py`:

```
Hello World
Hello World
Hello ???
Hello World
Hello
Hello World
Traceback (most recent call last):
  File "sources/hello_world.py", line 34, in <module>
    helloUnitTest()
  File "sources/hello_world.py", line 28, in helloUnitTest
    assert printGenericHello("World") is "Hello World", "return<print"
AssertionError: return<print
```

Dans le code ci-dessous vous pouvez remarquer qu'une fonction a été créée à la place d'une seule ligne dans le but de rendre le code réutilisable pour d'autres types de *hello*.

La fonction porte un nom explicite qui indique sa fonctionnalité.

Une fonction spécifique teste le bon comportement de la fonction à travers les mots clés `assert`.

Un bloc `main` appelle les différentes fonctions. La syntaxe du début du bloc `main` est toujours la même:

```
if __name__ == '__main__':
```

Quand le fichier qui contient ce bloc est le fichier principal, c'est à dire celui sur lequel l'interprétation est lancée, le code de ce bloc est automatiquement interprété. C'est une manière plus propre que d'avoir des lignes de code sans indentation partout dans le code ou entre les fonctions. Ces lignes de code sans indentation doivent toujours appartenir à une fonction, ou se trouver dans le bloc `main`.

2.1.1 La notion d'objet

Définition de la notion de classe:

Une classe est un prototype d'un objet défini par l'utilisateur qui définit un ensemble d'attributs qui caractérisent tous les objets qui appartiennent à cette classe.

sources/Student1.py

```
class Student:
    pass
```

Le mot clé `pass` n'est pas nécessaire il sert juste à dire que le corps de la classe est vide (nous ne pouvons pas rien mettre, sinon une erreur de syntaxe sera détectée).

2.1.2 Instantiation d'une classe (I)

Pour instancier une classe il suffit d'écrire (exactement) le nom de la classe suivi par des parenthèses (et éventuels arguments si son constructeur en a). Nous allons stocker la classe (qu'on appelle une instance de la classe ou un objet) dans la variable `s`. Quel sera le type de `s`?

```
>>> s = Student()  
>>> type(s)
```

La classe définit un type ad-hoc, donc elle sera du type classe de Student: `<class 'Student'>`. De notre point de vue, une classe sera définie par des:

1. attributs
2. méthodes

Les attributs peuvent être vues comme de champs/propriétés de l'objet. Ils peuvent être de n'importe quel type (y compris d'autres classes). Les valeurs de champs définissent une certaine instance (ou objet) de la classe. (voir *name* dans l'exemple ci-dessous).

Les méthodes sont des fonctions spéciales qui sont définies dans le cadre d'une classe. Parmi les méthodes, nous avons distinguer un type spécial de méthode qui est appelée automatiquement de manière à engendrer une nouvelle instance (objet) de la classe. Cette méthode est appelée constructeur.

2.1.3 Création d'une classe (II)

De point de vue syntaxique, le constructeur est une méthode qui porte toujours le nom `__init__` (les doubles tirets `__` sont obligatoires). Nous allons créer une classe qui a un constructeur défini par l'utilisateur.

sources/Student2.py

```
class Student:  
    def __init__(self):  
        pass
```

Vous remarquerez le fait que la signature de cette méthode doit avoir au moins un argument, le mot clé `?self?`. C'est la manière de python d'indiquer que la méthode `__init__` n'est pas une méthode quelconque mais elle appartient à la classe Student. A l'intérieur d'une classe, le mot clé `self` fait référence à la classe même (Student dans notre cas). Le code ci-dessus est équivalent avec la première définition de Student car nous avons créé un constructeur qui ne fait rien.

Nous allons maintenant créer un attribut `name` pour la classe Student et définir un constructeur qui permet de créer des instances de Student en spécifiant son nom.

sources/Student3.py

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

Pour créer un objet de type `Student`, nous sommes maintenant obligés d'utiliser son constructeur qui prend un argument (et non pas deux, `self` sert uniquement pour indiquer que la fonction `__init__` est une méthode - et un constructeur aussi, vu son nom - de la classe `Student`)

```
>>> s = Student(?Mariana?)
```

Si nous essayons de créer une instance en utilisant le constructeur par défaut sans argument, nous obtenons une erreur.

```
>>> s = Student()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument: 'name'
```

Nous allons maintenant modifier la classe de manière à pouvoir déterminer si un étudiant a obtenu une note de passage ($note_{passage} \geq 10$). Nous allons donc ajouter un attribut `grade` et une méthode qui teste si la valeur de cet attribut est supérieure ou égale à 10.

sources/Student4.py

```
class Student:
    """A class that defines a student by its name and grade."""
    def __init__(self, name, grade):
        """Creates a named and graded instance of a student.
        Args:
            name: the name of the student
            grade: the integer that represents the grade of the student
        """
        self.name = name
        self.grade = grade

    def hasPassingGrade(self):
        """Determines if a student has a passing grade.
        Returns:
            True: if grade >= 10
            False: otherwise
        """
        return self.grade >= 10

if __name__ == '__main__':
    s = Student("Mariana", 13)
    print(s.hasPassingGrade())
    s = Student("Ted", 9)
    print(s.hasPassingGrade())
```

Pour accéder à une méthode d'un l'objet, nous devons d'abord le stocker dans une variable (s dans notre exemple) écrire le nom de la variable suivie d'un point `?.?` et du nom de la méthode avec ses arguments éventuels: `s.hasPassingGrade()`.

Pour accéder aux attributs d'une classe nous allons utiliser l'opérateur point également:

```
s = Student("Ted", 9)
print(s.name)
print(s.grade)
#reevaluate
s.grade = 10
print(s.grade)
```

qui affiche dans la console:

```
Ted
9
10
```

2.1.4 Méthodes spéciales

Une méthode particulière qui peut être ajoutée à chaque objet est la méthode `__str__` (**self**). L'ajout d'une telle méthode à un objet peut avoir plusieurs avantages:

- Conversion directe de l'objet à une représentation chaîne de caractères
- Appel compact et normalisé

Nous allons maintenant revisiter l'exemple précédant, en ajoutant une méthode d'affichage : `printStudent()`.

sources/Student5.py

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

    def hasPassingGrade(self):
        return self.grade >= 10

    def printStudent(self):
        print("Name = " + self.name + ", grade = " + str(self.grade))
```

Pour imprimer l'instance nous devons accéder à sa méthode d'impression. Le résultat attendu est :

```
>>> s = Student("Mariana", 13)
>>> s.printStudent()
Name = Mariana, grade = 13
```

Nous souhaitons maintenant avoir une représentation textuelle de l'objet, et pas une simple méthode d'impression.

sources/Student6.py

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

    def hasPassingGrade(self):
        return self.grade >= 10

    def __str__(self):
        return "Name = " + self.name + ", grade = " + str(self.grade)
```

Pour imprimer à l'écran les instances de `Student` il suffit maintenant d'appeler la fonction `print` sur une instance de `Student`:

```
>>> s = Student("Mariana", 20)
>>> print(s)
Name = Mariana, grade = 20
```

3 Pré-traitement

Objectif: *Réalisation d'un mini-lexer pour le traitement des fichiers d'entrée du projet*

Cette première étape consiste à réaliser un module qui permettra le chargement d'une scène, qui fera l'objet du rendu visuel basé sur le lancé de rayons, à partir d'un fichier localisé sur le disque dur.

Ce fichier d'entrée respecte une syntaxe stricte qui permettra une interprétation du contenu de la scène de manière unique, non-ambiguë.

Pour réaliser le chargement de la scène, le module de pré-traitement va lire, ligne par ligne, dans un fichier qui se trouve dans le répertoire courant. Ce fichier porte le titre `input.txt` par défaut, mais peut avoir un nom quelconque, spécifié par l'utilisateur.

Chaque ligne contient une information qui porte sur la nature des éléments qui constituent la scène ainsi que leur réglages et les paramètres de chaque instance.

3.1 Le format d'entrée

Le format d'entrée est structuré sous la forme de lignes qui contiennent des paramètres étiquetés avec les caractéristiques suivantes:

- Chaque ligne contient un seul type d'information
- Chaque ligne commence par une étiquette suivie du caractère `:` et la liste de ses paramètres
- Le caractère `:` est présent une seule fois dans la ligne
- Les étiquettes sont des littéraux dans l'ensemble : { `#`, `sph`, `tsph`, `tri`, `lgh`, `cam`, `out` } ;
- Les paramètres sont séparés par le caractère `:` ;
- Les paramètres peuvent être :
 - des chaînes de caractères ;
 - des valeurs numériques ;
 - des listes de valeurs numériques.
- les listes de valeurs numériques sont écrites entre des crochets, `[]` et leurs éléments sont séparés par des virgules.

Vous pouvez trouver un exemple de fichier d'entrée, intitulé `input.txt`, dans le listing ci-dessous.

sources/input.txt

```
#:      scene generator input file
sph:    [0, 0, 3] ; 1 ; [0, 0, 1.]; 0.5
tsph:   image.png ; [0, 0, 3] ; 1 ; [0, 0, 1.]; 0.5
tri:    [-1,-1,3] ; [0,2,3] ; [1,-1,4]; [0,1.,1.]; 0.5
lgh:    [4, 10, -10]; [1., 1., 1.]
cam:    200 ; 200
out:    image4.png
```

Les différents types d'éléments qui peuvent être lus à partir du fichier d'entrée et leur étiquettes sont les suivantes:

- #: des commentaires (ne changent pas l'interprétation) ;
- des objets à afficher :
 - sph: sphere
 - tsph: sphere texture
 - tri: triangle
- lgh: source de lumière (instances qui émettent les photons)
- cam: camera
- out: nom du fichier de sortie

La partie pré-processing consiste à implémenter une classe `RenderSetup` qui effectue la lecture et le chargement de la scène à partir d'un fichier texte.

```
from raytracer import *
from scene import *
from light import *
from camera import *
from scene import *

class RenderSetup:
    # FIXME
```

Depuis le fichier d'entrée vous pouvez extraire comme information à la fois le type de classes que vous allez devoir créer ainsi que le nombre et type d'arguments, dans la forme finale, de ces classes.

Dans un premier temps vous êtes invités à créer les classes sans fonctionnalités, mais uniquement déterminé par leur constructeur (et implicitement leur attributs) ainsi que par une méthode de conversion en chaîne de caractères. Chaque classe qui correspond à un objet lu à partir du fichier, devra être donc munie de la méthode `__str__(self)`.

Ajouter et implémenter maintenant les méthodes suivantes de la classe `RenderSetup`:

- `read(self, filename)` - effectue la lecture proprement dite à partir d'un fichier;
- `render(self)` - effectue le rendu textuel (pour l'instant) de la scène.

- un bloc main qui instancie la classe principale avec un nom de fichier `input.txt` qui se trouve dans le même dossier et qui appelle la méthode `render()`.

Le fonctionnement de la méthode `render()` se resume à afficher sur la sortie standard les différents objets qui constituent le lancé de rayons et leur paramètres.

Complements Python

Pour la lecture de fichiers vous pouvez utiliser la fonction `open()`. Pour obtenir de l'aide sur une fonction, consultez Google ou d'appeler la fonction `help` sur le nom de la fonction `help(open)`.

```
help(open)
open(file, mode='r', buffering=-1, encoding=None, errors=None,
      newline=None, closefd=True, opener=None)
Open file and return a stream. Raise IOError upon failure.

=====
Character Meaning
=====
'r'      open for reading (default)
'w'      open for writing, truncating the file first
'x'      create a new file and open it for writing
'a'      open for writing, appending to the end of the file if it exists
'b'      binary mode
't'      text mode (default)
'+'      open a disk file for updating (reading and writing)
'U'      universal newline mode (deprecated)
=====
```

D'autres fonctions/méthodes utiles:

- `split()` - méthode de la classe `str`, renvoie dans une liste les sous-chaînes séparées par un certain caractère ;
- `lstrip()` - méthode de la classe `str`, enlève les espaces qui se trouvent avant une chaîne de caractères;
- `rstrip()` - méthode de la classe `str`, enlève les espaces qui se trouvent après une chaîne de caractères.

Exemple:

```
s = "1, 3"
a, b = s.split(',', 2)
#a = 1, b = 3
```