Cours d'algorithmique et programmation 2

Cours 1

Anne Parrain

UFR des Sciences
Licence Sciences et Technologie
mentions Mathématiques et Informatique
Semestre 2

Année universitaire 2017–2018

Section 1

Présentation de l'unité

Anne Parrain - anne.parrain@univ-artois.fr
Bureau C 304

- Responsable de l'unité Algo 2
- Responsable du semestre 2 des licences mathématiques et informatique

Objectifs de l'unité

Dans cette unité, vous allez :

- revenir sur la notion de référence contenue dans une variable
- revenir sur les fonctions (passage de paramètres)
- étudier des structures de données classiques en informatique (piles, files, . . .) ainsi que la notion de classe
- étudier des algorithmes classiques de tris (non récursifs)
- étudier la conception et la programmation d'applications avec une interface graphique
- étudier la librairie graphique `tkinter`

Objectifs de l'unité (2)

À la fin de cette unité, vous devriez être capable :

- de spécifier et d'écrire des fonctions
- de proposer une structure de données adaptée à la représentation d'un problème
- de proposer une décomposition fonctionnelle d'un problème
- de faire de la programmation événementielle
- d'écrire une application graphique en respectant les principes de base de décomposition Modèle-Vue-Contrôleur

Crédits et charge horaire

ECTS : 6 - 5 heures / semaine

CM : 1h30/sem $\times$ 12 sem $=$ 18 h (Semaines 1–12)

TD : 1h45/sem $\times$ 12 sem $=$ 21 h (Semaines 1–12)

TP : 1h45/sem $\times$ 12 sem $=$ 21 h (Semaines 2–13)

Équie pédagogique

- enseignants en TD :
  - Saïd Jabbour : groupe 1
  - Anne Parrain : groupe 2 / groupe 3
- enseignants en TP :
  - Johan Koitka / Thomas Caridroit : groupe 1
  - Johan Koitka / Anne Parrain : groupe 2
  - Johan Koitka / Yazid Boumarafi : groupe 3

Modalités de contrôle de connaissance

- une note de contrôle continu **CC** obtenue à partir de
  - d'interrogations en TD régulières. 5 interrogations, les 4 meilleures notes sont conservées ;
  - d'une note de projet (et éventuellement de contrôles en TP)
  - les tps rendus seront évalués et notés comme un bonus sur la note de contrôle continu
- une note d'examen **EX**, en session 1 comme en session 2
- la note finale de l'unité est **max(EX ;(CC+EX)/2)**

L'unité en ligne sur la plateforme Moodle de l'université

Moodle : **`http://foad.univ-artois.fr/`**

Vous trouverez :

- support des cours ;
- exercices ;
- contrôles continus ;
- notes.

clé d'inscription : **ALGO2-1718**

Lectures supplémentaires

Site web : Télécharger Python :
`http://python.org`

Site web : La documentation de référence de Python :
`https://docs.python.org/3/`

Site web : **How to think like a computer scientist - Learning with Python**
Jeffrey Elkner, Allen B. Downey, and Chris Meyers
Pour Python version 2
Disponible gracieusement : **`http://www.openbookproject.net/thinkcs/python/english2e/`**

Livre en français : **Apprendre à programmer avec Python 3**
Gérard Swinnen. Eyrolles, 3e édition. Disponible gracieusement : `http://inforef.be/swi/python.htm`

Films : **Tous les films des Monty Python !**
à défaut de pouvoir vous servir dans le cadre d'Algo 2, ça ne peut qu'être bénéfique pour votre anglais

Environnement de travail en TP

- sous Linux
- avec un éditeur de texte (`gedit`, `geany`, ...)
- test et exécution des scripts python depuis la console
- utilisation de `python3`
- et de librairies multi-plateformes

Cours 1

C’est parti. . .

Au programme aujourd'hui : un retour sur quelques éléments de base

- retour sur la notion d'espace de noms et de modules
- retour sur la portée des variables
- retour sur les paramètres des fonctions
- types mutables et non mutables

Section 2

Espaces de noms et Modules

Les espaces de noms et les modules (1)

- Lorsqu'on lance l'interprète, l'espace de noms est `__main__`.
- Toutes les variables ou fonctions définies dans l'interprète peuvent être appelées directement.
- la fonction `dir()` retourne la liste des noms définis dans l'espace de noms courant.

```
>>> a = 3
>>> a
3
>>> def f(n) :
...     return 2*n
...
>>> f(3)
6
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'a', 'f']
>>> __name__
'__main__'
```

Les espaces de noms et les modules (2) - À vous !

- Lorsqu'on définit des variables et des fonctions dans un fichier `mon_module.py` ...

```
1 a = 3
2 print("Espace de nom courant : ",__name__)
3
4 def f(n) :
5   return a*n
```

On appelle `mon_module` dans l'interprète pour pouvoir l'utiliser

```
>>> import mon_module
```

Comment faire appel à la fonction `f` ou à la variable `a` ?

Les espaces de noms (3) et les modules - Réponse

- a et f ont été définis dans l'espace de noms mon_module

```
>>> import mon_module
Espace de nom courant : mon_module
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> mon_module.a
3
>>> mon_module.f(2)
6
>>> dir(mon_module)
['__builtins__', '__doc__', '__file__', '__name__', ...
'__spec__', 'a', 'f']
>>> dir()
['__builtins__', '__doc__', '__name__', ...
'__package__', '__spec__', 'mon_module']
>>> __name__
'__main__'
```

Les espaces de noms (4) et les modules

Pour appeler une variable ou une fonction définies dans un module

- à l'intérieur du module : forme courte (sans préfixe)
- à l'extérieur du module (s'il a été importé) : forme préfixée
- intérêt : il ne peut pas y avoir de conflit de noms !
  Dans le projet de semestre 1, c'est ce qui nous a permis de définir dans chacun des modules `scenario`, `avatar`, *et* `caisse` *des fonctions* `init()`, `dessine()` *et* `update()`

Les espaces de noms (5) et les modules

Exemple :

```
>>> import caisse
>>> caisse1 = caisse.init(5,12)
# une caisse en ligne 5 et colonne 12
>>> caisse1
{'vit_c': 0, 'pos_c': 12, 'vit_l': 0, 'pos_l': 5}
>>>
>>> import avatar
>>> joueur = avatar.init(6,12,0,{})
# le joueur est en ligne 6, colonne 12,
# en direction du nord, avec un scenario vide
>>> joueur
{'vit_c': 0, 'pos_c': 12, 'direc': 0, 'pos_l': 6,
 'scen': {}, 'vit_l': 0}
```

Les espaces de noms (5) et les modules

- un module permet de regrouper des définitions de fonctions, de variables. . .
- ces fonctions portent sur un même thème : c'est ce qu'on appelle aussi une librairie (`pygame`, `math`, `random`, . . .)
- la variable `__name__` contient le nom de l'espace de nom courant

```
>>> __name__
'__main__'
```

Les espaces de noms (6) et les modules - À vous !

Exemple : contenu d'un fichier `mon_module2.py`

```
1 a = 3
2 def mon_nom() :
3     print(__name__)
4     print(a)
5
6
7 print(__name__)
```

Que se passe-t-il ? Complétez les `...`

```
>>> import mon_module2
...
>>> monNom()
...
>>> mon_module2.monNom()
...
```

Changer d'espace de noms

Pour incorporer dans l'espace de noms courant

```
>>> from mon_module2 import *
>>> a
3
```

Cela peut-être pratique. Exemple, contenu d'un fichier `mon_module3.py` :

```
1 from random import randint
2
3 def bonjour_alea() :
4     noms = ['maurice', 'albert', 'anatole', \
5           'ginette', 'colette']
6     print("Bonjour", noms[randint(0,len(noms)-1)])
```

Changer d'espace de noms (2)

On peut ensuite l'utiliser par :

```
>>> import mon_module3
>>> mon_module3.bonjour_alea()
Bonjour colette
```

ou bien par :

```
>>> from mon_module3 import *
>>> bonjour_alea()
Bonjour maurice
```

Attention, car du coup :

```
>>> randint(7,10)
8
```

`import` ou `from ... import` ?

- toujours privilégier la forme `import nom_module`, et utilisation des noms de fonctions et variables longs
  - évite les conflits de noms
  - clarifie où sont les différentes fonctions
  - *c'est ce que vous avez fait dans le projet !*
- lorsqu'il y a utilisation de nombreuses fonctions d'un module, pour éviter d'alourdir le code, utiliser la forme `from ... import ...`
  - peut faciliter la lecture du code
  - préciser dans le `import` les seules fonctions utilisées lorsque cela est possible

Les applications Python - À Vous !

Dans le fichier `mon_module4.py` ci-dessous, quel est le sens des deux dernières lignes ?

```
1 def main() :
2   ...
3
4
5 if __name__ == '__main__' :
6   main()
```

Quelles peuvent être les différentes utilisations de `mon_module4.py` ?

Les librairies et les applications Python

- Premier cas de figure : vous voulez lancer l'application définie par la fonction `main()` de `mon_module4` :

```
moi@maMachine$ python3 mon_module4
```

- Deuxième cas de figure : vous voulez utiliser les fonctions de `mon_module4` dans un autre module :

```
1 import mon_module4
```

- Troisième cas de figure : vous voulez utiliser directement dans l'interprète les fonctions de `mon_module4`

```
>>> from mon_module4 import *
```

Section 3

Fonctions

Portée des variables au sein d'un module

- Rappel : **une variable est un nom qui va contenir une référence vers un objet** (un entier, une liste, une chaîne de caractères. . .).

- Rappel : **une variable est définie lors de sa première affectation. Elle n'existe pas auparavant**.

Portée des variables au sein d'un module ou d'une fonction

- une variable définie au niveau d'une fonction n'est accessible que dans cette fonction. C'est une variable locale.
- une variable définie au niveau d'un module est accessible partout dans le module (et à l'extérieur si elle est importée). C'est une variable globale : elle est accessible en lecture seulement par simple appel.
  - Au niveau d'un module, les seules variables globales définies doivent être des constantes.

Portée des variables - À vous !

```
1  CST1 = 1.8
2  CST2 = 32
3
4  def celsius_vers_fahrenheit(temp_cel) :
5      cpt = CST2
6      temp = CST1 * temp_cel + cpt
7      return temp
8
9  def dummy() :
10     print('dummy :',cpt)
11
12 if __name__ == '__main__' :
13     for temp in [5,15,25] :
14         tf = celsius_vers_fahrenheit(temp)
15         print('Celsius :',temp,'- Fahrenheit :',tf)
16     print(temp)
17     dummy()
```

Les paramètres des fonctions

- Les paramètres d'une fonction sont des **noms** qui ont une portée **locale** à la fonction
- On peut les voir comme des variables locales qui seraient définies au moment de l'appel de la fonction
- Si un paramètre d'une fonction porte le même nom qu'une variable globale à un module, alors seul le paramètre est **accessible** dans la fonction
- Les paramètres peuvent recevoir une valeur par défaut dans la déclaration de la fonction. *Uniquement pour certains types de données.*

Que se passe-t-il ? - À vous !

```
1  def somme_valeurs(uneListe,ajout=0) :
2      som = ajout
3      for elt in uneListe :
4          som += elt
5      return som
6
7  if __name__ == '__main__' :
8      maListe = [12, 15, 17]
9      res = somme_valeurs(maListe)
10     print(res)
11     autreListe = [1, -6, -10]
12     res = somme_valeurs(autreListe,10)
13     print(res)
```

L'instruction `return`

- Les fonctions retournent une valeur par l'instruction `return`
- Lorsque l'instruction `return` est exécutée, les instructions suivantes dans la fonction **ne sont pas exécutées**
- Si on veut retourner plusieurs valeurs :
  - on n'écrit pas plusieurs `return`
  - mais on rassemble les valeurs dans une liste, un tuple, un dictionnaire... avant de les retourner

L'expression `return` (2)

```
1  def contient_des_entiers_pairs(lentiers) :
2      '''teste si une liste ne contient
3      que des entiers pairs.
4      Argument :
5      - lentiers : list(int)
6      Retour :
7      boolean
8      '''
9      for elt in lentiers :
10         if elt%2 != 0 :
11             return False
12     return True
```

Que se passe-t-il pour :

```
1  contient_des_entiers_pairs([8,12,4,6])
2  contient_des_entiers_pairs([8,3,12,4,6])
```

L'expression return (3)

```
1  def dates_naissance(lpers,nom) :
2      '''retourne les dates de naissance des
3      personnes qui s'appellent comme nom.
4      Argument :
5      - lpers : list(dictionnaires de personnes)
6      - nom : str
7      Retour :
8      list(str)
9      '''
10     res = []
11     for pers in lpers :
12         if pers['nom'] == nom :
13             date_naiss = str(pers['jour'])+"/"+
14                          str(pers['mois'])+"/"+
15                          str(pers['annee'])
16             res.append(date_naiss)
17     return res
```

L'expression return (4)

```
1  def coords(grille,valeur) :
2      '''retourne le premier numero de ligne et de colonne
3      ou on trouve valeur. Retourne (-1,-1) sinon.
4      Argument :
5      - grille : list(list)
6      - valeur : objet
7      Retour :
8      int,int
9      '''
10     for i in range(len(grille))  :
11         for j in range(len(grille[i])) :
12             if grille[i][j] == valeur :
13                 return i,j
14     return -1,-1
```

Section 4

Types mutables et types non mutables

Les listes

Une liste est une structure de données qui permet de mémoriser une collection d'objets, quelque soit leur type.

```
>>> langages = ["python","scheme","c","java","php"]
>>> langages[0]
'python'
>>> langages[3]
'java'
>>> len(langages)
5
>>> langages[0] = "Python" # On peut modifier une liste
>>> langages
['Python', 'scheme', 'c', 'java', 'php']
>>> type(langages)
<type 'list'>
```

Pour ajouter un élément

On peut utiliser la **méthode** append qui permet d'ajouter un élément à une liste.

```
>>> langages.append("assembleur")  # ajoute l'element
            # 'assembleur' au bout de la liste langages
>>> langages
['Python','scheme','c','java','php','assembleur']
>>> len(langages)
6
```