IFT-1004 - Introduction à la programmation

Module 4.5 : Interlude

Honoré Hounwanou, ing.

Département d'informatique et de génie logiciel
Université Laval

Table des matières

Davantage de fonctionnalités

Davantage de fonctionnalités

Dans cette partie, nous donnons quelques trucs et astuces disponibles dans le langage Python dont nous avons glissé un mot informellement, ou non, sans les donner en notes de cours.

La programmation orientée objet

Pour développer des programmes plus simples de maintenance, d'amélioration et d'extension, une bonne pratique est de séparer le programme en fonctions et en modules.

Un paradigme de programmation offrant une solution à ce besoin est le *paradigme orienté objet*, dont nous allons présenter quelques composantes de base pour mieux poursuivre avec nos trucs et astuces.

Les objets

Un *objet* est une structure de données contenant des *attributs* (qui sont simplement des variables, aussi appelées *variables membres*), et qui répond à un ensemble de messages via des *méthodes* (qui sont simplement des fonctions qui agissent sur les variables membres).

Les objets

Imaginons par exemple un objet « employé », qui aurait comme attributs `numero`, `nom`, `prenom`, `salaire`, etc. Chacun de ces attributs est en fait une variable (donc avec une valeur et un type), associée à l'objet employé.

Un exemple de méthode serait `donner_promotion`, une fonction modifiant l'objet l'attribut `salaire` de l'employé.

Les classes

Une *classe* est la définition d'un objet. Elle contient la définition du contenu de l'objet (attributs et méthodes), ainsi que le code des méthodes. Une classe peut être vue comme un « moule » à objets : on l'utilise pour créer de nouveaux objets.

Une *instance* est une occurrence d'une classe. Il s'agit d'un objet ayant des valeurs d'attributs qui lui sont propres. On peut créer un nombre quelconque d'instances d'une même classe, et chaque instance contiendra un espace mémoire qui lui est propre (des valeurs de variables membres qui lui sont propres). Par exemple, une compagnie emploie plusieurs employés, avec des numéros, noms et salaires qui leur sont propres. Le type d'une instance est la classe qui a été utilisée pour le créer.

Les objets qu'on connaît déjà

Nous créerons plus tard dans la session nos propres objets. Par contre, nous connaissons et utilisons déjà plusieurs objets ! Les **chaînes de caractères**, les **listes**, les **dictionnaires** et les **fichiers** sont des objets, des instances de classes définies par Python !

```
# Instanciation d'une liste
a = [1, 2, 3, 4]

# Une autre instance de la classe liste.
b = [1, 2, 3, 4]

# Les deux objets a et b sont deux instances différentes.
# Modifier b n'aura pas d'impact sur a.
b[3] = 0
```

Appeler une méthode

Pour appeler une méthode à partir d'un objet, on utiliser le nom de l'objet, suivi d'un point, suivi du nom de la méthode.

```
nom_objet.nom_methode(arguments)
```

```
# Appel de la méthode append de la liste.
a.append(-5)

# Appel de la méthode count de la liste.
a.count(1)

# Appel de la méthode sort de la liste.
a.sort()
```

Trucs et astuces

Le formatage de chaînes

Python permet de formater des chaînes de caractères à l'aide de la méthode `format()`.

La chaîne de caractères doit contenir un certain nombre d'accolades `{}` qui seront remplacées par les arguments fournis à la méthode `format()`.

```
>>> ma_chaine = "Affichage de {}chaîne variable: {}."
>>> ma_chaine.format("", "ABCD")
'Affichage de chaîne variable: ABCD.'

>>> ma_chaine.format("ma ", "Bonjour")
'Affichage de ma chaîne variable: Bonjour.'
```

Le formatage de chaînes

On peut modifier l'ordre du formatage en indiquant un numéro entre les accolades.

```
>>> ma_chaine = "Affichage de {1}chaîne variable: {0}."
>>> ma_chaine.format("", "ABCD")
'Affichage de ABCDchaîne variable: .'

>>> ma_chaine.format("ma ", "Bonjour")
'Affichage de Bonjourchaîne variable: ma .'
```

Le formatage de chaînes

On peut également réserver un certain nombre d'espacements (minimum), de la manière suivante.

```
>>> ma_chaine = "{:10} | {:10}"
>>> for nom in ["Stark", "Snow", "Lannister"]:
...     for prenom in ["Ned", "John"]:
...         print(ma_chaine.format(prenom, nom))
...
Ned        | Stark
John       | Stark
Ned        | Snow
John       | Snow
Ned        | Lannister
John       | Lannister
```

Le formatage de chaînes

Il est également possible d'aligner les chaînes à gauche (<), à droite (>) ou au centre (^).

```
>>> ma_chaine = "{:^10} | {:^10}"
>>> for nom in ["Stark", "Snow", "Lannister"]:
...     for prenom in ["Ned", "John"]:
...         print(ma_chaine.format(prenom, nom))
...
   Ned     |   Stark
   John    |   Stark
   Ned     |    Snow
   John    |    Snow
   Ned     | Lannister
   John    | Lannister
```

Le formattage de chaînes

On peut finalement indiquer un type de données à formater, notamment `d` pour un entier ou `f` pour un réel. Dans le cas d'un réel, il est possible d'indiquer le nombre de chiffres après le point décimal en utilisant le format suivant :

```
>>> ma_chaine = "Entier: {:10d}, réel: {:10.4f}"
>>> ma_chaine.format(10, 10)
'Entier:         10, réel:    10.0000'

>>> ma_chaine.format(10, 10.02)
'Entier:         10, réel:    10.0200'
```

Autres méthodes pratiques

```
# Convertir une chaîne en lettres majuscules
nouvelle_chaine = ma_chaine.upper()

# Convertir une chaîne en lettres minuscules
nouvelle_chaine = ma_chaine.lower()

# Trouver l'index d'une sous-chaîne
mon_index = ma_chaine.index(ma_sous_chaine)

# Vérifier si un caractère est alphabétique ou numérique
ma_chaine.isalpha()
ma_chaine.isdigit()

# Séparer une chaîne en une liste de chaînes étant
# donné un séparateur
liste_chaines = ma_chaine.split(separateur)
```

Les méthodes sur les listes

Les listes ont également un ensemble de méthodes pratiques qui vous sont disponibles.

```
# Ajouter un élément à la fin d'une liste
ma_liste.append(element)

# Insérer un élément à un endroit quelconque
ma_liste.insert(index, element)

# Supprimer un élément
ma_liste.remove(element)

# Compter le nombre de fois qu'un élément est présent dans une liste
ma_liste.count(element)
```

Les méthodes sur les dictionnaires

Un dictionnaire est également un objet, nous connaissons déjà deux de ses méthodes : `keys()` et `values()`. Il existe également d'autres méthodes pratiques, telles que `clear()` qui vide le dictionnaire, et `copy()` qui en retourne une copie.

Les méthodes sur les fichiers

Nous connaissons déjà quelques méthodes pour manipuler les fichiers en Python, dont `read()`, `readline()`, `write()` et `close()`.

Il existe également d'autres méthodes pratiques dont `readlines()` qui retourne une **liste de chaînes de caractères** contenant les lignes du fichier, ainsi que `writelines()`, qui prend en argument une liste de chaînes de caractères à inscrire dans le fichier. **Attention** : malgré ce que son nom laisserait sous-entendre, les retours de charriots (caractères `\n`) ne sont pas ajoutés automatiquement.

Les boucles

Nous verrons ici deux instructions pratiques qui permettent de terminer une boucle ou le corps d'une boucle avant la fin. Ces instructions sont `break` et `continue`.

break

L'instruction `break` termine l'exécution d'une boucle. Voici plusieurs fonctions équivalentes utilisant des boucles différentes.

```
def element_appartient(liste, element):
    trouve = False
    i = 0
    while i < len(liste) and not trouve:
        if liste[i] == element:
            trouve = True
        i += 1
    return trouve
```

break

```
def element_appartient(liste, element):
    trouve = False
    i = 0
    while i < len(liste):
        if liste[i] == element:
            trouve = True
            break
        i += 1
    return trouve
```

break

```
def element_appartient(liste, element):
    i = 0
    while i < len(liste):
        if liste[i] == element:
            return True
        i += 1
    return False
```

break

```
def element_appartient(liste, element):
    for x in liste:
        if x == element:
            return True
    return False
```

continue

L'instruction `continue` termine l'exécution du bloc d'instructions courant dans une boucle, et continue l'exécution de la boucle avec la prochaine itération.

Voici plusieurs fonctions équivalentes utilisant des boucles différentes.

```
def afficher_pairs(n):
    i = 0
    while i < n:
        if i % 2 == 0:
            print(i)
        i += 1
```

continue

```
def afficher_pairs(n):
    for i in range(n):
        if i % 2 != 0:
            continue
        print(i)
```

assert

L'instruction `assert` permet de s'assurer qu'une certaine expression est vraie. Si celle-ci est fausse, une *exception est lancée* (nous y reviendrons).

```
def est_diviseur(a, b):
    assert b != 0, "L'argument b ne doit pas être 0"

    return a % b == 0
```

Cette instruction est très utile pour valider les arguments d'une fonction, ou pour faire des tests unitaires.