CONTRIBUTEURS

Introduction

Étapes de développement

1. L'analyse
   - Quel est le *problème* ?
   - Que veut l'*utilisateur* ?
   - Quel est son *budget* ?
   - Quelles sont les **conséquences** d'une *erreur* ?
2. La conception
   - Comment résoudre le problème ?
   - Quelles sont les *structures de données* appropriées ?
   - Quels sont les *algorithmes* nécessaires ?
   - Quelles sont les *interfaces* requises ?
3. La programmation
   - Implantation de la solution développée aux étapes précédente, en utilisant un ou plusieurs langages de programmation.
4. Les tests d'intégration
   - L'intégration des différents modules en un tout cohérent ;
   - Les procédures de tests qui permettent d'établir la validité et la fiabilité du logiciel.

L'approche hiérarchique pour traiter des données

- les **programmes** sont composés de **modules** ;
- les **modules** contiennent des **énoncés** ;
- les **énoncés** contiennent des **expressions** ;
- les **expressions** créent et manipulent les **données**.

Syntaxe de base

Affectation énoncé (e.g. `pi = 3.1415`) ayant habituellement 3 éléments :

1. un nom de variable (e.g. `pi`) appelé **identifieur** ;
2. l'opérateur (e.g. =) ;
3. une **valeur** affectée à la variable (e.g. 3.1415) appelée **expression**.

Commentaires Des commentaires sont des lignes du code, qui commencent par #, qui ne sont pas exécutées.

Opérateurs arithmétiques

- `+` addition ;
- `-` soustraction ;
- `*` multiplication ;
- `/` division régulière ;
- `//` division entière ;
- `%` reste de la division entière ;
- `**` exponentiation.

Fonction `int` retourne la partie entière d'un nombre. 3 types de nombres : entiers, flottants et complexes. Lorsque l'on effectue une opération arithmétique entre certains nombres, Python conserve le type le plus général. de même pour `float`
Module de maths a plus d'opérateur (e.g. sin, cos, sqrt) et est importé avec `import math`. Puis, on utilise ses fonctions avec, p. ex., `math.sqrt()`.

Fonctions de base

`print`

Permet d'afficher à la console la valeur d'une ou de plusieurs expressions.

- permet aussi avec toute sorte d'options de spécifier la façon dont cette ou ces valeurs seront affichées.

`input`

Permet de lire ce que vous entrez au clavier et retourne le résultat sous la forme d'une chaîne de caractères.

Fonctions

Introduction

Qualités d'une fonction

1 Cohérence

Une fonction est ***cohérente*** si elle accomplit une seule tâche. On doit pouvoir résumer en peu de mots ce qu'accomplit la fonction.

2 Indépendance

Une fonction est ***indépendante*** si sa sortie dépend uniquement de ses entrées (arguments) et d'aucune autre variable. Il ne faut pas définir des fonctions qui dépendent de *variables globales*.

3 Concision

La *concision* consiste à limiter la longueur des fonctions. Plus la fonction est courte, plus elle sera facile à comprendre pour un humain.

Définition

De façon générale, on définit une fonction en Python avec `def` :

```
def name(arg1, arg2, ..., argn):
        #       bloc d'énoncés indentés
        return expression       #       optionnel
```

Booléens

Les opérateurs suivants permettent de comparer les valeurs respectives de deux objets :

$<$ inférieur ;

$>$ supérieur ;

$<=$ inférieur ou égal ;

$>=$ supérieur ou égal ;

$==$ égal ;

$!=$ pas égal.

Les opérateurs suivants permettent de combiner une ou plusieurs expressions booléennes :

and conjonction ($\cap$) ;

or disjonction ($\cup$) ;

not négation ($A^{\complement}$).

Énoncés conditionnels

Forme générale du `if`

```
if expression_1 :
        #       bloc d'énoncés 1
elif expression_1 :
        #       bloc d'énoncés 2
#       ...
elif expression_n :
        #       bloc d'énoncés n
else :
        #       bloc d'énoncés n + 1
```

On peut aussi utiliser l'opérateur `if else` pour définir une variable. Par exemple, pour l'expression `a = y if x else z` si `x` est vraie, alors `a = y` sinon `a = z`.

Éléments de procédure

Types d'énoncés

Séquentiels Conditionnels

Énoncé répétitif

L'***énoncé répétitif*** permet au programmeur de faire des boucles, c'est-à-dire de **répéter** les énoncés d'un bloc d'énoncés tant qu'une certain **condition** demeure vraie.

Boucle `for`

```
for cible in itérable :
    # bloc principal d'énoncés indentés
    if condition :
        break     # facultatif
    if condition :
        continue  # facultatif
else :  # facultatif
    # bloc supplémentaire d'énoncés indentés
```

Notes :

- La **clause** `else` est exécutée lorsque la **boucle** `for` se termine normalement.
- L'**énoncé** `break` placé à l'*intérieur* d'une boucle met fin à l'exécution de la boucle **sans avoir traité toutes les valeurs** de l'`itérable`.
- L'**énoncé** `continue` placé à l'*intérieur* d'une boucle met fin à l'**itération *courante*** de la boucle.

En bref :

`break` sortir d'une boucle **prématurément** avant sa terminaison normale. Par exemple, si on détermine que les itérations restantes de la boucle ne sont pas utiles ou nécessaires.

`continue`

Boucle `while`

```
while expression :
    <bloc principal d'énoncés indentés>
    if condition :
        break     # facultatif
    if condition :
        continue  # facultatif
else :  # optionnel
    <bloc supplémentaire d'énoncés indentés>
```

Notes :

- Tant que l'**expression** est vraie, le bloc exécute.

Note On devrait toujours utiliser la boucle for et non la boucle while.

Fonctions utiles

`range(start = 0, stop, step = 1)` : générer une séquence de chiffres (équivalent de `seq` en `R` sauf que la fonction retourne un objet de classe "range".

Types de données

Dictionnaire

Le dictionnaire Python permet de stocker des **associations** entre une **clé d'accès** et une **valeur**. La clé d'accès doit être ***immuable*** et est souvent une chaine de caractères.

On définit un dictionnaire par une paire d'accolades `{'clé': valeur}`

Fonctions du dictionnaire

```
a = {'spam': 25, 'eggs': 37, 'coffee': 254}
print(a)
print(a.get('spam'))
print(a.copy())
print(a.keys())
print(a.values())
print(a.items())
```

Ensemble (« *set* »)

L'ensemble de Python correspond à la définition mathématique d'un ensemble. Il contient une collection de **clés unique** ; c'est donc un dictionnaire sans valeurs !

Opérations sur les ensembles

```
x = set('spam')
y = {'h', 'a', 'm'}
x | y   # réunion
x & y   # intersection
x - y   # différence, diffère selon
        #  l'ordre (x - y vs y - x) !
x ^ y   # différence symétrique, pareille
        #  peu importe l'ordre
```

Fichiers

On utilise la fonction `open(<fichier>, <mode>)` où on peut avoir 4 modes :

`'r'` mode lecture (défaut), le fichier doit exister. (« *read* »)

`'w'` mode écriture, le fichier est créé au besoin et remplacé s'il existe déjà. (« *write* »)

`'x'` mode écriture, comme 'w', mais retourne une erreur si le fichier existe déjà.

`'a'` mode écriture, on ajoute au fichier préexistant ou créé le fichier s'il n'existe pas. (« *append* »)

On écrit dans le fichier avec `<fich>.write('text')`. Puis, après nos ajouts on ferme le fichier avec `<fich>.close()`, puisque les écritures se font de façon asynchrones. Si on ne `close` pas le fichier on ne peut pas être certain qu'il sera modifié. Pour se simplifier la vie, on peut utiliser `with open('fichier.txt', <mode>) as fich: fich.write(<texte>)`.

On lit un fichier avec `<fich>.read()`