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()`

Retour sur les fonctions

Qualités d'une fonction

1. **Cohérence** : Chaque fonction devrait implanter une seule tâche.
2. **Indépendance** : Le résultat d'une fonction ne doit dépendre que de ses arguments.
3. **Concision** : garder les fonctions courtes et **simples** pour en faciliter la lisibilité.

Note Si on ajoute une étoile à un argument (e.g. `fonc(a, b, *c)` le dernier argument va récupérer tout les arguments hors de position ou supplémentaire. On ne peut pas avoir plus qu'un argument étoilé.

Note On peut forcer que des arguments soient spécifiés par nom (e.g. `fonc(a, b, *, c = 0)` en incluant une étoile comme frontière. Ici, `c` doit être spécifié par nom.

Note Si on insère 2 étoiles devant un argument, il va créer un dictionnaire des arguments nommés non définis par la fonction (e.g. `fonc(**kargs)`).

Note Si on ne spécifie pas de `return` dans une fonction, elle retourne `None`.

On peut utiliser `lambda` pour créer de fonctions anonymes (e.g. `lambda x: x**2`).

On peut utiliser la fonction `yield`, au lieu de `return`, pour retourner des valeurs intermédiaires.

Complexité algorithmique

Croissance :

- linéaire $O(n)$
- quadratique $O(n^2)$
- cubique $O(n^3)$

Classes et objets

Contexte

Les classes sont le fondement de la ***programmation orientée objet***. Nous étudions 5 notions de classes.

1 Encapsulation

Une classe *réunit* des ***données*** et des ***fonctions*** applicables sur ces données. On dit qu'elle ***encapsule*** les données avec une interface publique permettant de facilement les manipuler.

Un classe permet, entre autre, de définir ses propres ***types*** de données et les opérateurs applicables sur ses données.

Attributs d'une classe

Un ***attribut*** d'une classe est un membre de la classe. Pour accéder à l'attribut `x` d'un objet `y`, on écrit `y.x`. Une classe constitue un *gabarit* qui permet d'engendrer des objets particuliers qu'on nomme des ***instances*** de lla classe.

2 Héritage

On peut construire une classe à partir d'une autre classe. Dans une tel cas, la classe ***hérite*** tous les attributs de son ancêtre—c'est la classe ***dérivée***.

Par exemple, soit la *classe de base* une forme géométrique. Alors, un triangle, rectangle et ellipse sont tous des classes dérivées de la forme géométrique.

Contexte

La programmation orientée objet vise à favoriser la réutilisation du code. C'est une entité ***cohérente*** et***indépendante***.

Classe en Python

Un énoncé `class` est exécutable et produit un objet de type `class` qu'il affecte à la variable `nom` :

```
class nom(classe_de_base):
    var = valeur   # variable de classe (partagée par toutes les instances)
    ...
    def fonc1(self, ...):   # fonction membre
        self.membre = valeur # variable d'instance (spé
```

Il peut contenir des variables ***locales*** à la classe qui seront partagées par toutes ses objets (instances). Également, il contient des fonctions membres, dont le premier argument `self` réfère toujours à l'instance pour laquelle la fonction est appelée.

Note Une fonction membre définie dans une classe se nomme une ***méthode***.

Opérateurs

Une classe peut définir ses propres ***opérateurs*** en définissant une fonction avec un nom spécial de la forme `__NOM__`.

Le ***constructeur*** est une fonction spéciale qui permet d'initialiser les objets de la classe et porte toujours le nom `__init__`. Il y existe plusieurs méthodes spéciales :

- Opérations mathématiques (`add`, `sub`, `mul`) et la division (`__truediv__`).
- Égalités lt, gt, le, ge, eq (==), ne (!=).
- Opérateur [] en lecture (`getitem`) ou en écriture (`setitem`) qui permettent d'utiliser implicitement les crochets.
  - Par exemple, `def __setitem__(self, index, newArg): self.data[index] = newArg` et `def __getitem__(self, index): self.data[index] = newArg`.
- Longueur (`len`), conversion en texte (`str`) et la conversion en booléen (`bool`).

__add__

__sub__

__mul__

Modules

`import module` ou `from module import fun1, fun2`. Dans le deuxième cas, on importe directement les fonctions dans notre namespace et il n'est pas nécessaire de les précéder de `module.`.

Un « *package* » est un ***groupe de modules***. Pour qu'un répertoire soit reconnu comme un package par Python, il doit contenir un fichier nommé `__ init__ .py`.

Exceptions

Traitement

Contexte

Lorsqu'une exception est ***soulevée***, mais non traitée par notre programme, l'interpréteur affiche la ***trace*** (« *traceback* ») du programme. Si on désire traiter nous même les exceptions risquant d'être soulevées par les énoncés à exécuter, on doit les placer dans un bloc défini par l'énoncé `try`. Puis, on peut dire à Python quoi faire en insérant un bloc `except` après le `try`. On peut également soulever nous-mêmes une erreur à l'aide de l'énoncé `raise`.

Utilisation :

```
try :
    # exécution normale
    # insérer du code à rouler comme si il n'y avait pas d'exc
    ...
except nom1 [as valeur]:
    # exécuté pour une exception de type nom1
    ...
except (nom2, nom3) [as valeur]:
    # exécuté pour une exception nom2 OU nom3
    ...
else :
    # exécuté si aucune exception
    ...
finally :
    # exécuté dans tous les cas
    ...
```

Types d'erreurs et leur héritage :

BaseException le type de base pour toutes les exceptions

Exception le type de base pour les exceptions non liées au système

ArithmeticError **OverflowError** lorsque le nombre est trop grand pour être représenté

ZeroDivisionError lors d'une division par zéro

LookupError **IndexError** lorsqu'un indice est invalide (e.g. liste)

KeyError lorsqu'une clé est invalide (e.g. dictionnaire)

AssertionError lors de l'échec d'un énoncé assert

EOFError lorsque la fonction input atteint la condition de fin de fichier

ImportError lorsque l'importation échoue (module introuvable)

StopIteration lorsqu'il n'y a plus d'autres valeurs pour un itérateur (ou une fonction génératrice)

Raise

Le syntaxe de l'énoncé Raise est : `raise < objet >`. Cette fonction interrompt l'exécution de la fonction en cours et soulève une exception.

Note L'énoncé raise peut seulement soulever des erreurs de type `BaseException`.

Assert

Le syntaxe de l'énoncé Assert est : `assert < expressino >[, < objet >]`. Cette fonction permet de tester une expression et de soulever une exception si elle est fausse. Par exemple, on peut l'utiliser pour tester le domaine de valeurs entrées dans une fonction de densité.