3 - Les fonctions

Le style est une façon simple de dire des choses compliquées.

Jean Cocteau

Objectifs :

- les fonctions et méthodes prédéfinies
  - listes ;
  - chaînes ;
  - dictionnaires ;
- définition des fonctions.

Fichiers à produire : `TP3_1.py` `TP3_2.py` `TP3_3.py` `TP3_4.py` `TP3_5.py` `TP3_6.py` `TP3_7.py`

⚠ **Attention**
La notation ***[xxx]*** indique un **paramètre optionnel** que l'on peut donc omettre. Si on fournit le paramètre, on ne met bien sûr pas les crochets, qui sont uniquement là pour la documentation.

Cette notation est utilisée ici ainsi que dans la plupart des documentations informatiques, généralement associée à du texte en italique, comme dans le présent document.

3.1-Manip

Comme nous l'avons vu en cours, l'utilisation des fonction permet de structurer ses programmes, de réutiliser les parties communes et de mieux les maintenir.

Dans un premier temps nous allons explorer quelques *fonctions prédéfinies* avant d'apprendre à *écrire nos propres fonctions* (il existe bien d'autres fonctions et méthodes, se référer à la documentation Python pour la liste exhaustive).

3.1.1 - Des méthodes et fonctions prédéfinies

Sur les listes

Fonctions. Ces fonctions sont aussi applicables aux autres types de séquences de données (chaînes, tuples…) :

`len(L)` — Renvoie le nombre d'éléments dans `L`.

`sorted(L)` — Crée et renvoie une nouvelle liste, copie de `L`, triée.

`reversed(L)` — Génère les éléments de `L` en ordre inverse.

Méthodes non modificatrices. Ces méthodes renvoient un résultat sans modifier la liste :

`L.count(x)` — Renvoie le nombre d'occurrences de `x` dans `L`. `L.index(x)` — Renvoie l'indice de la première occurrence de `x` dans `L`.

Exercice : Essayez et commentez :

```
>>> L = [
"jean","marc","yacine","marc","herbert","marc","jean"]
>>> # Affichez la liste et sa longueur.
>>> L.count("marc")
>>> L.index("marc")
```

```
>>> for n in dir(L) :print(n)
```

Quels genres de renseignements nous donne la fonction `dir(L)` ?

Méthodes modificatrices. Ces méthodes, sauf `pop()` renvoient la valeur `None`.

`L.append(x)` — Ajoute l'élément x à la fin de `L`.

`L.extend(L2)` — Ajoute tous les éléments de la liste `L2` à la fin de `L`. `L.insert(i,x)` — Insère l'élément `x` à l'indice `i` dans `L`. `L.remove(x)` — Supprime la première occurrence de x dans L.

`L.pop([i])` — Renvoie la valeur de l'élément à l'indice `i` et l'ôte de `L`. Si `i` est omis alors supprime et renvoie le dernier élément.

`L.reverse()` — Renverse sur place les éléments de `L`.

`L.sort()` — Trie sur place les éléments de `L`. Par défaut tri croissant ; un argument optionnel `key` permet de trier suivant le résultat du calcul d'une fonction appliquée à chacune des valeurs ; un argument optionnel booléen `reverse` permet d'inverser le tri.

Rappel

Les chaînes de caractères ne sont pas modifiables et donc les méthodes qui *transforment* les chaînes en créent de *nouvelles* et les retournent.

S.count(sous_chaîne[, début [, fin]]) — Compte le nombre d'occurrences de **sous_chaîne** dans **S** entre **début** et **fin** (0 et len(S) par défaut).

S.find(sous_chaîne[, début [, fin]]) — Retourne la position de la première occurrence de **sous_chaîne** dans **S** entre **début** et **fin** (0 et len(S) par défaut). Retourne -1 si **sous_chaîne** n'est pas trouvée.

S.join(séquence) — Retourne une chaîne résultant de la concaténation des valeurs chaînes issues de **séquence** en insérant **S** comme séparateur entre les valeurs.

s.split([séparateur[,nb_max]]) — Retourne une liste de chaînes résultant du découpage de la chaîne **S**. Par défaut la séparation se fait sur les blancs (espaces, tabulations, retours à la ligne), mais on peut spécifier un autre **séparateur**. Le paramètre **nb_max** permet de limiter le nombre de découpes.

S.strip([caractères]) — Supprime les blancs (ou tout caractère dans **caractères**) au début et à la fin de **S**.

S.replace(ancienne, nouvelle[, nombre]) — Remplace chaque occurrence de la sous-chaîne **ancienne** par **nouvelle** dans **S**. Si **nombre** est donné, seules les **nombre** premières occurrences sont remplacées.

S.capitalize() — Transforme le premier caractère de **S** en majuscule et les suivants en minuscules.

S.lower() — Convertit toutes les lettres de **S** en minuscules.

S.upper() — Convertit toutes les lettres de **S** en majuscules.

S.center(largeur[,remplissage]) — Centre la chaîne dans une **largeur** donnée, en rajoutant si besoin est des espaces (ou le caractère **remplissage**) de part et d'autre.

S.zfill(largeur) — Ajoute si nécessaire des zéros à gauche de **S** pour obtenir la **largeur** demandée.

De plus, on dispose des méthodes booléennes suivantes, dites « de test de contenu» :

S.isalnum() — Alphanumérique uniquement (chiffres ou lettres).

S.isalpha() — Lettres uniquement. **S.isdigit()** — Chiffres uniquement.

S.islower() — Lettres minuscules uniquement. **S.isspace()** — Blancs uniquement.

S.istitle() — Forme de titre.

S.isupper() — Lettres majuscules uniquement.

Exercice : Essayez et commentez :

```
>>> s = "Joe Student"
```

```
>>> s.find("Stu")
>>> s.split()
>>> s.upper().center(80,"=")
```

Sur les conteneurs (séquences, ensembles, dictionnaires)

La duplication de ces conteneurs et du contenu nécessite des traitements spéciaux qui sont disponibles en utilisant le module ***copy***.

copy.copy(D) — Renvoie une copie en *surface* de **D**. Fournit une nouvelle valeur distincte de **D** et égale à **D**, dont le contenu reste partagé avec **D**.

copy.deepcopy(D) — Renvoie une copie en *profondeur* de **D**. Fournit une nouvelle valeur distincte de **D** et égale à **D**, dont le contenu est complètement dupliqué et est distinct de celui de **D**.

Sur les dictionnaires

Méthodes non modificatrices. Les méthodes **items()**, **keys()** et **values()** renvoient des vues sur le contenu du dictionnaire, utilisables par exemple dans les boucles **for** ou transformable en listes, avec des valeurs dans un ordre quelconque.

k in D — Est vrai (**True**) si **k** est une clé de **D**, faux (**False**) sinon.

D.items() — Renvoie une vue des *éléments* (paires clé/valeur) dans **D**. **D.keys()** — Renvoie une vue des *clés* dans **D**.

D.values() — Renvoie une vue des *valeurs* dans **D**.

D.get(k[, x]) — Renvoie **D[k]** si **k** est une clé de **D** ; sinon **x** (ou **None** si **x** n'est pas précisé).

Méthodes modificatrices :

`D.clear()` — Supprime tous les éléments de `D`.

`D.update(D1)` — Pour chaque clé `k` de `D1`, affecte `D1[k]` à `D[k]`.

`D.setdefault(k[, x])` — Renvoie `D[k]` si `k` est une clé de `D` ; sinon affecte `x` à `D[k]` et renvoie `x`.

`D.popitem()` — Supprime et renvoie un élément (paire clé/valeur) quelconque de `D`.

Exercice : Essayez (respectez bien les accents) et commentez chaque opération :

```
>>> d = { "à":'a',"é":'e',"ù":'u'}                # Affichez d !
>>> d["ê"] = 'e'                                  # Re-affichez d !
>>> d.update( { 'à':'A','è':'E'} )                # Re-re-affichez d !
>>> 'e' in d
>>> 'à' in d
```

Définition des fonctions

Une fonction est une instruction composée définie par :

- *une ligne d'en-tête formée :*
  - *du mot-clé* ***def***
  - *du nom de la fonction*
  - *de la liste des arguments entre parenthèses*
  - *du caractère deux-points*

- Le bloc d'instructions qui constitue le corps de la fonction est ensuite indenté par rapport à la ligne d'en-tête. Une fonction se termine grâce à une instruction `return`, ou à défaut à la fin de ses instructions ;
- L'instruction `return` permet de retourner une ou plusieurs valeurs à l'expression qui a appelé de la fonction ; un `return` sans valeur spécifiée ou une fonction sans `return` renvoie simplement la valeur prédéfinie `None` ; .
- entre l'en-tête et le corps on insère généralement une *chaîne de documentation*, ce qui vous est demandé pour les fonctions que vous définissez en TP.

Exemple élémentaire :

```
>>> def doubler(x):
...         "Retourne le double de <x>"
...         return x * 2
...
```

```
>>> doubler(7)                  # Pourquoi cet appel fonctionne-t-il avec un
entier...
>>> doubler(7.2)                # ...un flottant...
>>> doubler("bla") # ...une chaîne de caractères...
>>> doubler({1:'bleu',2:'rouge',3:'vert'}) # Et avec un dictionnaire ?
```

Programmation

Réorganisation de texte

Programme TP3_1.py

À partir d'une variable `chaine` contenant la chaîne suivante :

`"Un texte que je veux retravailler"`

Remplacer dans `chaine` le `"je veux"` par `"Joe veut"` (interdiction de coder en dur des index).

Découper ensuite `chaine` en une liste de mots, stockée dans une variable `mots`.

A partir de `mots`, créer une nouvelle chaîne qui contienne un séparateur `"+++"` entre chaque mot. Afficher cette nouvelle chaîne.

Semaine

Programme TP3_2.py

Affecter à une variable `semnum` un dictionnaire qui fasse correspondre les clés numéros des jours de la semaine (entiers 1 à 7) à leurs valeurs noms (chaînes lundi à dimanche).

Créer une seconde variable `semjours`, de type dictionnaire, vide.

En utilisant une boucle parcourant les *éléments* stockées dans `semnum`, remplire `semjours` en faisant correspondre les clés noms des jours à leurs valeurs numéros des jours dans la semaine.

Extraire à partir de `semjours` les noms des jours dans une liste `jours`, que l'on trie ensuite par ordre croissant.

Parcourir cette liste dans une boucle, et pour chaque jour afficher son nom ainsi que son numéro dans la semaine (trouvé via l'un des deux dictionnaires créés précédemment).

Calcul de π

Programme TP3_3.py

Écrire une fonction `monPi()` avec un argument `p_n`. Cette fonction retourne une valeur approchée de π, sachant que :

$$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$$

Attention, cette approximation est une quantité *réelle* : utilisez des *flottants*. Pour cela

vous avez besoin de la fonction « racine carrée » dont le nom en Python est **`sqrt()`** et qui est accessible depuis le module ***`math`***.

Par ailleurs, écrire un programme principal qui saisit un entier **`n`** positif non nul, qui appelle la fonction **`monPi()`** et qui affiche vos résultats ainsi que l'erreur relative (en pourcentage) entre votre valeur de π et celle fournie par le module ***`math`***.

Suite de Syracuse

On appelle suite de Syracuse toute suite d'entiers naturels vérifiant :

$$u_{n+1} = \begin{cases} \frac{u_n}{2}, & u_n \; est \; pair \\ 3u_n + 1, & u_n \; est \; impair \end{cases}$$

Par exemple, la suite de Syracuse partant de u_0=11 est :

11,34,17,52,26,13,40,20,10,5,16,8,4,2,1,4,2,1,4,2,1

En se bornant à 1, on appelle cette suite finie d'entiers le **vol** de 11. On appelle **étape** un nombre quelconque de cette suite finie. Par exemple 17 est une étape du vol de 11. On remarque que la suite atteint une étape maximale, appelée **altitude maximale** du vol. Par exemple 52 est l'altitude maximale du vol de 11.

Programme TP3_4.py

Écrire une fonction **`etapeSuivante()`** avec un argument **`p_u`**. Elle reçoit un entier strictement positif, calcule l'entier suivant dans la suite de Syracuse et le retourne. Par exemple **`etapeSuivante(5)`** doit produire **`16`** et **`etapeSuivante(16)`** doit faire **`8`**, etc.

Écrire une fonction **`vol()`** avec un argument **`p_uinit`**. Elle reçoit un entier initial strictement positif, crée une liste vide et, en utilisant la fonction **`etapeSuivante()`**, calcule et stocke dans cette liste les termes de la suite de Syracuse de l'entier initial jusqu'à 1 inclus. Cette fonction retourne la liste ainsi créée.

Écrire un programme principal qui demande un entier initial à l'utilisateur (pour les tests, utilisez des valeurs entre 2 et 100) puis qui, en utilisant les fonctions déjà définies, affiche proprement les termes de la suite de Syracuse correspondante.

Programme TP3_5.py

Écrire un programme principal qui demande un entier initial à l'utilisateur puis qui calcule et affiche l'altitude maximale de ce vol.

Pour cela, écrire une fonction **`altMaxi()`** avec un argument **`p_uinit`**. Elle reçoit un entier et produit l'altitude maximale de la suite de Syracuse commençant par **`u0 = p_uinit`**. Par exemple, en partant de **`u0 = 7`**, la suite de Syracuse est : 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 et par conséquent **`altMaxi(7)`** vaut **`52`**.

Notions de récursivité

La notion de récursivité en programmation peut être définie en une phrase

> *Une fonction récursive peut s'appeler elle-même.*

Par exemple, trier un tableau de N éléments par ordre croissant c'est extraire le plus petit élément puis trier le tableau restant à $N-1$ éléments.
Bien que cette méthode soit souvent plus difficile à comprendre et à coder que la méthode classique dite *itérative*, elle est dans certains cas l'application la plus directe et la plus simple de sa définition mathématique.

Voici l'exemple classique de définition par récurrence de la fonction factorielle :

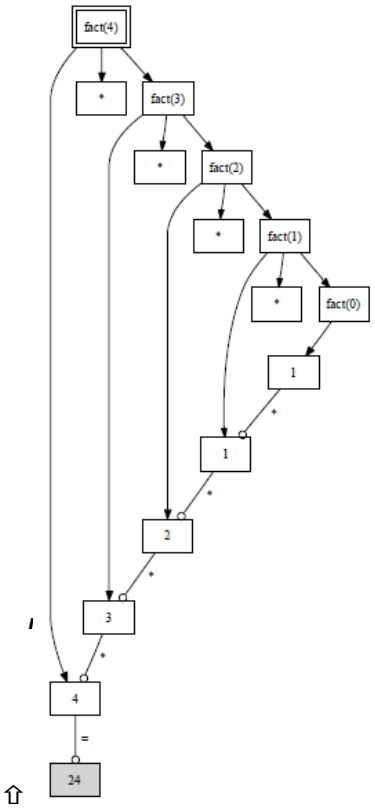
$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)!, & n \geq 1 \end{cases}$$

Remarque
Son code est très exactement calqué sur sa définition mathématique.

```
def
fact(p_n) :
if p_n == 0
:
return 1
else :
return p_n * fact(p_n-1)
```

Dans cette définition, la valeur de $n!$ n'est pas connue tant que l'on n'a pas atteint la condition terminale (ici `p_n == 0`). Le programme *empile* les appels récursifs jusqu'à atteindre la condition terminale puis *dépile* les valeurs.

Pour dérouler un programme récursif, il suffit de dérouler tous ses appels dans un tableau comme ci-dessous, jusqu'à atteindre la condition terminale sans appel (par exemple ici quand `n` vaut `0`). Ensuite on fait remonter, à partir de la ligne du bas (sans appel), les valeurs de retour trouvées (3e colonne) jusqu'en haut du tableau.



Suite de Fibonacci

FIBONACCI : une suite doublement récursive

On définit la suite de Fibonacci de la façon suivante :

Programme TP3_6.py

À faire :

1. dérouler le programme « à la main », avec `5` comme entrée, remplir et **rendre** le tableau ci-dessous ;
2. puis coder la suite de FIBONACCI dans le fichier `TP3_6.py` et vérifier les résultats du tableau avec une boucle d'affichage.

Calcul	*Appels*	*Valeur de retour*
fib(n)	*fib(n-1)+fib(n-2)*	*fib(n-1)+fib(n-2)=*
fib(5)	fib(4)+fib(3) ⇩	⇧
fib(4)	fib(3)+fib(2) ⇩	⇧
fib(3)	fib(2)+fib(1) ⇩	⇧
fib(2)	fib(1)+fib(0) ⇩	⇧
fib(1)	*sans appel*	⇧
fib(0)	*sans appel*	⇧