Modularité

En python, le concept de « modularité » recouvre trois choses :

1) Le concept de programmation « modulaire », qui consiste à découper des tâches complexes en différentes fonctions et fichiers cohérents, de manière à pouvoir les réutiliser plus facilement.
2) Les « modules » python : un fichier python est un module.
3) Les « packages » python : un ensemble de fichiers/modules/packages regroupés dans un dossier et contenant un fichier $\_\_init\_\_.py$.

I) Modularité, interfaces

➢ Modularité

Le principe de modularité est particulièrement important dans tout développement logiciel. Il simplifie les tests, la maintenance du code, et permet de réutiliser facilement des éléments de logique à différents endroits d'un même projet. Ce principe peut opérer à différents niveaux :

- Grouper le code dans des fonctions pour éviter la duplication de code
- Grouper parfois des fonctions dans des classes (v. chapitre sur le POO, à venir)
- Grouper des fonctions et/ou des classes par thèmes, dans des fichiers (= modules !)
- Grouper des fichiers en bibliothèques (= packages)

➢ Interfaces

Tout module expose une « interface » qui décrit l'ensemble des valeurs et/ou opérations (≈ fonctions) utilisables avec ce module. C'est l'ensemble des éléments que l'utilisateur du module peut réutiliser.

Nota : le concept d'interface ne se limite pas aux modules !

II) Les modules en python

II.1) Modules disponibles par défaut

De nombreux modules sont préinstallés ou peuvent être installés via une commande du type « *pip install ...* ». Ces modules sont accessibles dans n'importe quel fichier python.

Exemples :

```
from math import inf, cos, atan2
import turtle
```

II.2) Modules créés par un utilisateur

Si on ne considère que les cas les plus simples d'utilisation de « modules » en python (on laissera de côté les « packages »), un module n'est qu'un simple fichier python !

« linked_list.py » ⇔ *module « linked_list »*

Noms de modules : conventions

- Un nom de module ne doit pas contenir d'espaces ou de caractères accentués.
- Il est recommandé d'utiliser des noms en minuscules uniquement, et de préférence sans underscores (sauf si cela facilite la compréhension du nom de module).

Documentation de modules :

Un fichier de module est sensé commencer par un docstring donnant le nom du module, puis décrivant son utilisation.

L'interface du module est accessible via l'instruction $help(module\_name)$.

II.3) Importations/utilisations

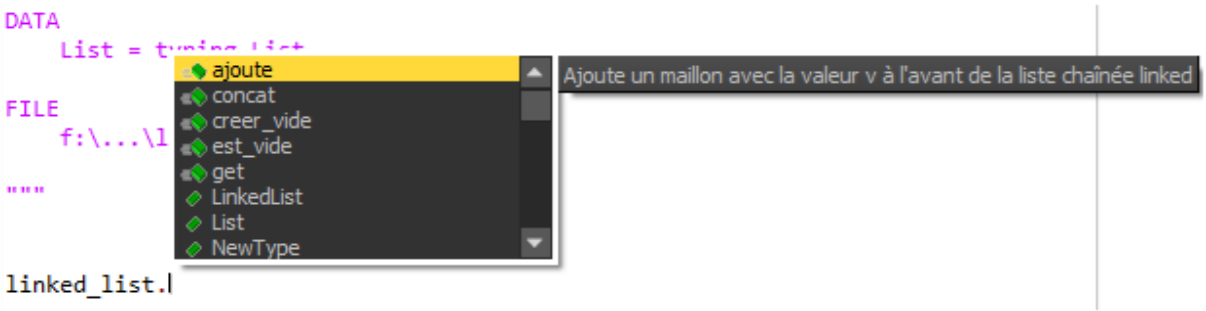
Une fois le fichier du module enregistré au format « .py », il peut être importé dans un autre fichier python situé dans le même dossier, en utilisant une des syntaxes suivantes :

➢ ***Module importé en tant qu'espace de noms :***

```
import linked_list

vide = linked_list.creer_vide()
a = linked_list.ajouter_tete(3, vide)
```

Avantage : Explicite / pas de risque d'écraser des fonctions/variables ayant le même nom dans différents modules / accès facile aux « suggestions » quand on tape le code :



Inconvénient : Rend souvent le code plus long à taper

- ***Module importé avec aliasing :***

```
import linked_list as Linked

vide = Linked.creer_vide()
a = Linked.ajouter_tete(3, vide)
```

Avantage : Explicite / pas de risque d'écraser des fonctions ou variables ayant le même nom dans différents modules / en général utilisé pour raccourcir le nom du module

Inconvénient : Peut rendre le code moins lisible si l'alias est trop court (`import linked_list as L` et plus loin dans le code : `L.ajouter_tete(3,a)` ⇒ on se demande bien ce qu'est `L`, ici…)

- ***Importation partielle (seulement certains éléments du module) :***

```
from linked_list import creer_vide, ajouter

vide = creer_vide()
a = ajouter_tete(3, vide)
```

Avantage : Code encore plus court…

Inconvénient : On perd le côté explicite ! (on ne sait pas d'où provient la fonction si on n'a pas l'import sous les yeux)

Nota : On peut également utiliser des alias pour chaque chose importée :

```
from math import cos as cosinus, degrees, sin as sinus
```

- ***Importation du fainéant… :***

```
from linked_list import *

vide = creer_vide()
a = ajouter_tete(3, vide)
```

Avantage : il n'y a pas plus court en termes de nombre de caractères…

Inconvénient : pas explicite, pas traçable, pas d'aliasing, gros risques de collisions de noms

⇒ en résumé : ***PAS BIEN !!***

III) Annexe : conventions de styles pour les identifiants

(=« casse » = « case », en anglais)

Afin de faciliter la compréhension d'un code pour une personne devant relire/modifier le programme qu'on écrit, chaque langage propose une série de recommandations de manière à identifier plus facilement ce qui peut se cacher derrière un identifiant.

En python, les recommandations sont les suivantes :

Type de casse	*snake_case*	*snake_case*	*UPPER_CASE*	*PascalCase*	/
Pour…	Variables	Fonctions	Constantes (pour des choses jamais réassignées par la suite)	Classes	Données non utilisée dans le code
Exemples	a $n0$ var_name	$func$ cut_this	PI $VITESSE\_LUM$	$LinkedList$ $GrayMater$	_ *(vous le croiserez de temps en temps)*

Remarques :

- Les identifiants commençant par un simple ou un double underscore signifient généralement que le « machin » qui lui correspond est sensé être « privé » (⇒ à ne pas utiliser ailleurs que dans le fichier/la fonction/la classe/l'élément où il est défini).

- Un autre type de casse est très utilisé dans d'autres langages : le $camelCase$. Jamais utilisé en python, mais c'est la norme utilisée pour les noms de variables ou fonctions à la place du $snake\_case$, en Java, Javascript, …