Introduction à la programmation en Python

I Modèle de calcul

❦ Définition. *Ordinateur*

Un *ordinateur* sera pour nous un *modèle de calcul* théorique possédant

1. une *unité de calcul* qui *exécute* les *instructions* de calcul.
2. une *mémoire* qui sauvegarde les *objets* du calcul.

La modélisation que l'on propose dans ce cours sera cohérente avec la machine virtuelle Python, elle-même construite sur une (en fait plusieurs) abstraction du fonctionnement du système électronique complexe qu'est l'ordinateur physique. On pourra donc concevoir les *programmes* en utilisant ce modèle.

❦ Définition. *Code*

Un *code* Python est un texte formé d'ensemble de mot-clés, de noms et de symboles opératoires *respectant les règles de syntaxe* Python.

❦ Définition. *Instructions*

Une *instruction* correspond à une étape de calcul élémentaire que l'on souhaite exécuter sur l'ordinateur et tient sur une ligne du code.

Une instruction a pour but de modifier le contenu de la mémoire de l'ordinateur, en ajoutant, enlevant ou modifiant des *objets*. Sinon, elle est *inutile*.

❦ Définition. *Expression*

Une *expression* est une portion de code qui calcule et *renvoie* un nouvel objet qui peut alors être utilisé pour une autre opération ou être sauvegardé dans la mémoire. Une expression *peut être aussi* une instruction.

Un code qui respecte la syntaxe pourra être exécuté par l'interpréteur Python. Dans le cas contraire, ce dernier enverra une *erreur de syntaxe* (`Syntax Error`).

II Exécution

❦ Définition. *Intérpréteur*

L'*interpréteur Python* est le logiciel qui *lit*, *traduit* et *exécute* un code Python.

Chaque fois que vous lancez l'interpréteur, l'ordinateur lui dédit une partie de sa mémoire pour l'exécution de vos code. Cette mémoire est nettoyée lorsque vous quittez l'interpréteur.

❦ Définition. *Console*

La *console* de votre logiciel de développement en Python, est l'interface qui vous permet d'envoyer directement une expression Python à l'interpréteur et d'en observer le résultat. Elle permet aussi d'envoyer une instruction pour modifier la mémoire.

❦ Définition. *Editeur*

L'*éditeur* quand à lui vous permet d'écrire un code, puis de choisir quand envoyer le code à l'interpréteur.

❦ Définition. *Erreur de Syntaxe*

Si l'interpréteur n'arrive pas à lire et traduire le code que vous lui envoyer, il s'arrêtera sur une *erreur de syntaxe*, `Syntax Error`.

L'interpréteur essayera de préciser à quel endroit du code il n'arrive plus à traduire votre code et, à son avis, quel est le problème. Parfois l'erreur n'est pas exactement celle que l'interpréteur prévoit, ni exactement à l'endroit où n'arrive pas à lire…

❦ Définition. *Erreur d'exécution*

Si l'interpréteur n'arrive pas à exécuter le code que vous lui envoyer, il s'arrêtera sur une *erreur d'exécution*.

Le nom de l'erreur dépend du problème et doit vous aider à comprendre d'où vient le problème et comment le résoudre.

III Objets

❦ Définition. *Type*

Chaque objet possède un *type* qui détermine les opérations qui sont possibles sur cet objet.

Entiers

❦ Définition. *Type Entier*

Le type *entier* (**`int`**) permet de représenter les nombres entiers relatifs. Il n'y a pas de limite sur les entiers représentable, si ce n'est la place occupée en mémoire par l'objet.

Les entiers peuvent être créés en écrivant leur représentation en base 10 en numérotation de position : `12`, `-144`.

Les opérations suivantes sont disponibles pour le type entier :

Addition	`+`	`12 + 42`
Soustraction	`-`	`12 - 42`
Inversion	`-`	`-12`
Multiplication	`*`	`12 * 42`
Quotient	`//`	`42 // 12`
Reste	`%`	`42 % 12`
Puissance entière	`**`	`42**12`

⚛ **Pour s'entraîner** : Exercice 1

Booléens

❦ Définition. *Type Booléen*

Le type *booléen* (**`bool`**) permet de représenter une valeur de vérité. Il y en a deux : `True` qui représente *vrai* et `False` qui représente *faux*.

Les opérations suivantes sont disponibles pour le type booléen :

Et logique	**`and`**	`True and False`
Ou logique	**`or`**	`True or False`
Non logique	**`not`**	`not True`

⚛ **Pour s'entraîner** : Exercice 2

❧ ❧ ❧

Flottants

❦ Définition. *Type Flottant*

Le type *flottant* (**`float`**) permet de représenter les nombres décimaux. La précision est cette fois limité. De manière simplifiée, la précision ne dépasse pas 10^{-16}.

Les flottants peuvent être créés en écrivant leur représentation décimale en numérotation de position : `12.12`, `-144.56`. S'il s'agit d'un nombre à partie décimale nulle, on peut l'écrire `12.` au lieu de `12.0`. On peut aussi utiliser la notation scientifique `1.2e12`.

Les opérations suivantes sont disponible pour le type flottant :

Addition	`+`	`12.2 + 42.5`
Soustraction	`-`	`12. - 42.67`
Inversion	`-`	`-12.9`
Multiplication	`*`	`12.345 * 42.1`
Division	`/`	`42.0 / 12.0`
Puissance	`**`	`42.4**0.12`

⚛ **Pour s'entraîner** : Exercice 3

❧ ❧ ❧

Le Rien

❦ Définition. *Type Flottant*

Le type *None* (`None`) permet de représenter le « rien ». Il n'y a qu'un objet de type `None`, c'est `None`.

⚛ **Pour s'entraîner** : Exercice 4

❧ ❧ ❧

Comparaisons numériques

Les entiers et les flottants peuvent être comparés entre eux. Le résultat de la comparaison est un booléen. Voici les opérateurs disponibles :

```
<, > # Inégalité strictes
3 < 4, 5 > 6
<=, >= # Inégalité large
3 <= 4, 5 >= 6
== # Egalité
2.2 == 2.0
!= # Différence
2.2 != 2.0
```

⚛ **Pour s'entraîner** : Exercice 5

❧ ❧ ❧

IV Variables

❦ Définition. *Variables*

Une *variable* est un *nom* associé à un *objet*.

Un objet peut avoir plusieurs noms, donc être associé à plusieurs variables. Mais un nom ne désigne qu'un seul objet.

Le nom ne devra utiliser que des caractères alphanumérique, ou _ pour remplacer les espaces.

❦ Définition. *Nommage et mémoire*

Un objet ne peut rester dans la mémoire que s'il possède un nom. Les objets renvoyés par les expressions ne vivent que le temps de l'exécution de l'instruction complète. Ceux qui ne sont pas nommés seront détruits.

❦ Définition. *Affectation*

L'*affectation* est l'opération consistant à donner un nom à un objet. C'est ce qui correspond à la création (ou la modification) d'une variable.

Elle s'écrit avec le symbole = : `variable = objet`.

La première fois qu'une variable est affectée à un objet, on dit qu'elle est *définie*.

❦ Définition. *Lecture*

L'utilisation d'une variable dans une expression, signifie implicitement que l'on souhaite faire une *lecture*, *i.e* récupérer l'objet associé à la variable dans la mémoire pour le modifier ou l'utiliser dans le calcul.

Lire une variable non définie provoque une erreur.

⚛ **Pour s'entraîner** : Exercice 6

❧ ❧ ❧

V Fonctions

❦ Définition. *Fonctions*

Une fonction est un objet qui représente un code Python paramétré.

Le code Python peut utiliser des variables qui n'y sont pas définies, mais qui sont annoncées comme paramètre de la fonction.

❦ Définition. *Appel de fonction*

Une fonction a pour but d'être *appelée*. L'appel de fonction consiste à définir pour chaque paramètre, l'objet qui lui sera associé, puis à exécuter le code avec les paramètres ainsi défini. Le code s'arrête en *renvoyant un objet*, résultat de l'appel de fonction. S'il n'y a pas d'objet à renvoyer, l'appel renvoie tout de même `None`.

Une fonction peut être définie à l'aide de la syntaxe suivante :

```
def nom_fonction(p1, p2, ..., pn) :
    # Code paramétré par p1, p2, ..., pn
    # ...
    return ... # La fonction renvoie un objet
```

Cette même fonction sera appelé comme suit :

```
resultat = nom_function(o1, o2, ..., on)
```

Avec `o1, o2, ..., on` des objets Python qui seront respectivement associés à `p1, p2, ..., pn` lors de l'exécution du code de `nom_function`.

⚛ **Pour s'entraîner** : Exercice 7, 8

❧ ❧ ❧

VI Environnements

❦ Définition. *Environnement*

Un *environnement* est un espace dans lequel sont sauvegardées les variables.

Dans un même environnement, deux variables ne peuvent donc pas être nommées de manière identique. Mais ça peut être le cas dans deux environnements différents.

❦ Définition. *Environnement Global*

L'*environnement global* est l'espace dans lequel vivent les variables définies en dehors de toute fonction. Ces variables sont dites *globales*.

❦ Définition. *Environnement Local*

Lors de l'appel d'une fonction, un *environnement local* se crée. Les paramètres ainsi que toutes les variables définies dans le code de la fonction existeront dans cet environnement-là. Ces variables seront dites *locales* à la fonction. L'environnement local sera détruit à la fin de l'appel.

Ainsi, les variables locales peuvent avoir des noms identiques dans différentes fonctions ou le même nom qu'une variable globale, tout en étant associées à des objets différents.

⚛ **Pour s'entraîner** : Exercice 9

❦ ❦ ❦

VII Flot d'exécution

❦ Définition. *Flot d'exécution*

Le *flot d'exécution* (ou flot de contrôle) d'un code, est la séquence ordonnée d'instruction qui seront exécutées par la machine de calcul.

Le comportement par défaut de la machine de calcul est de commencer par l'instruction à la première ligne de code, puis à chaque fois qu'une instruction a été exécuté, de passer à celle sur la ligne suivante. Et cela jusqu'à ce qu'il n'y ai plus de ligne de code à exécuter.

❦ Définition. *Appel*

Un appel de fonction modifie le flot d'exécution comme ceci :

1. La prochaine instruction exécutée sera la première ligne du code de la fonction appelée.
2. Après l'exécution de l'instruction **`return`** du code de la fonction, la prochaine instruction à exécuter sera celle sur la ligne suivant l'appel.

⚛ **Pour s'entraîner** : Exercice 10

Structure de contrôle conditionnelle.

❦ Définition. *Conditionnelle*

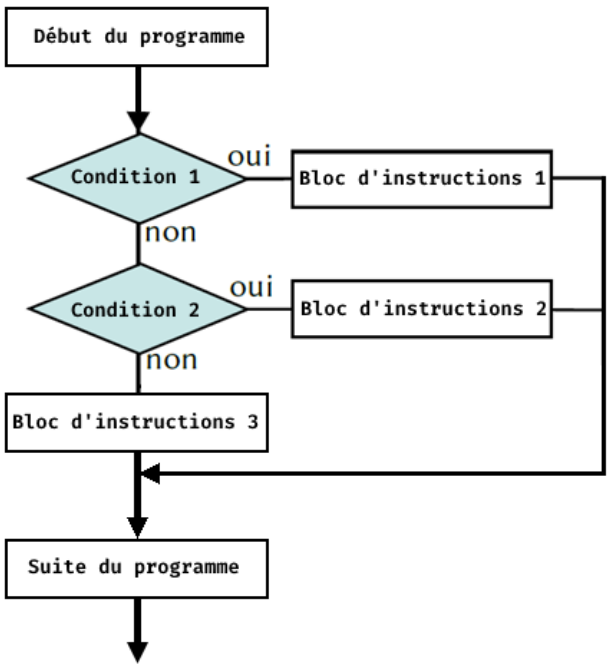
La structure de contrôle conditionnelle permet, selon le résultat d'une ou plusieurs expressions booléennes (des conditions), de choisir un bloc d'instruction à exécuter parmi plusieurs. Après la fin de l'exécution du bloc, l'instruction exécutée est celle qui suit directement la structure.

La syntaxe d'une structure conditionnelle est la suivante :

```
if c1 : # première condition
    ... # c1 est vrai
elif c2 : # deuxième condition
    ... # c1 est fausse et c2 est vrai
elif c3 : # troisième condition
    ... # c1 et c2 sont fausses et c3 est vrai
... # autant de elif qu'on veut
else :
    ...# toutes les conditions sont fausses.
```

Il peut n'y avoir aucun **elif**. Le **else** est facultatif. S'il n'est pas spécifié, la machine ne fera rien dans le cas à la ou les conditions sont fausses.

```
if condition1 :
    # Block d'instructions 1
elif condition2 :
    # Block d'instructions 2
else :
    # Block d'instructions 3
#Suite du programme
```



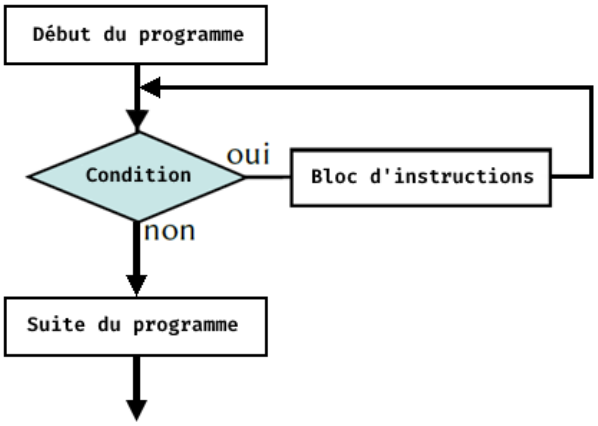
⚛ **Pour s'entraîner** : Exercice 11, 12

Structure de contrôle itérative.

❦ Définition. *Boucle*

La boucle permet, selon le résultat d'une expression booléenne (condition), de répéter un bloc d'instruction à exécuter. Lorsque la condition devient fausse, l'instruction exécutée est celle qui suit directement la boucle.

```
while condition :
    # Block d'instructions
    ...
#Suite du programme
```



⚛ **Pour s'entraîner** : Exercice 13

❦❦❦

VIII Algorithme pour le calcul des termes d'une suite récurrente

⚛ **Pour s'entraîner** : Section VI

Soit une suite récurrente :

$$\begin{cases} u_0 & = a \\ u_{n+1} & = f(u_n) \end{cases}$$

pour a et f fixés, où f est une fonction définissant le calcul de u_{n+1} en fonction de u_n.

L'algorithme qui prend en entrée n et qui calcul u_n procède comme suit :

1. On commence par u_0, terme courant de départ.
2. Si le terme courant est u_i, on calcule $u_{i+1} = f(u_i)$ qui devient le nouveau terme courant.
3. On recommence avec le nouveau terme courant, et ce tant que $i \neq n$

Plus formellement :

Algorithme Calcul de u_n

ENTRÉES : n un entier naturel
SORTIES/EFFETS : u_n
1. $i \leftarrow 0$ /* *rang initial* */
2. $u \leftarrow u_0$ /* *terme courant,* u_i */
3. **tant que** $i < n$ **faire**
4. $\quad u' \leftarrow f(u)$
5. $\quad u \leftarrow u'$
6. **fin tant que**
7. **renvoyer** u

Ou en `Python` :

```
def suite_u(n) :
    i = 0
    u = u0 # u0 doit être remplacer par le terme initial
    while i < n :
        u_suivant = f(u) # ici on remplace par le calcul du terme suivant
        # Le calcul peut prendre plusieurs lignes de code...
        u = u_suivant
    return u
```

L'algorithme s'adapte avec une suite récurrente d'ordre supérieure. Soit une suite récurrente :

$$\begin{cases} u_0 & = a_0 \\ u_1 & = a_1 \\ \cdots & \\ u_{p-1} & = a_{p-1} \\ u_{n+p} & = f(u_n, u_{n+1}, u_{n+p-1}) \end{cases}$$

pour $a_0, \cdots, a_{p-1}$ et f fixés, où f est une fonction définissant le calcul de u_{n+p} en fonction de $u_n, \cdots, u_{n+p-1}$.

L'algorithme qui prend en entrée n et qui calcul u_n procède comme suit :

1. On commence par $u_0, \cdots, u_{p-1}$, les terme p termes courants de départ.
2. Si les termes courant sont $u_i, \cdots, u_{i+p-1}$, on calcule $u_{i+p} = f(u_i, \cdots, u_{i+p-1})$ qui devient le nouveau $p^{\text{ème}}$ terme courant.
3. Les termes courants sont maintenant $u_{i+1}, \cdots, u_p$
4. On recommence avec les nouveaux termes courants, et ce tant que $i \neq n$

Plus formellement :

Algorithme Calcul de u_n

entrées : n un entier naturel
sorties/effets : u_n
1. $i \leftarrow 0$ /* *rang initial* */
2. $u \leftarrow u_0$ /* *terme courant,* u_i */
3. $u' \leftarrow u_1$ /* *terme courant,* u_{i+1} */
4. $\cdots$
5. $u^{(p-1)} \leftarrow u_{p-1}$ /* *terme courant,* u_{i+p-1} */
6. **tant que** $i < n$ **faire**
7. $\quad u^{(p)} \leftarrow f(u, u', \cdots, u^{(p-1)})$
8. $\quad u \leftarrow u'$
9. $\quad u' \leftarrow u''$
10. $\quad \cdots$
11. $\quad u^{(p-1)} \leftarrow u^{(p)}$
12. **fin tant que**
13. **renvoyer** u

Ou en Python :

```
def suite_u(n) :
    i = 0
    u = u0 # u0 doit être remplacer par le premier terme initial
    u_suivant = u1 # u1 doit être remplacer par le second terme initial
    ...
    u_suivantpm1 = upm1 # upm1 doit être remplacer par le p-1 ème terme initial
    while i < n :
        u_suivantp = f(u,u_suivant,...,u_suivantpm1)
        # calcul du terme suivant, sur eventuellement plusieurs lignes.
        u = u_suivant
        u_suivant = u_suivant2
        ...
        u_suivantpm1 = u_suivantp
    return u
```

Il faut parfois savoir faire apparaître une suite récurrente pour résoudre un problème qui ne semble initialement pas être le calcul d'une suite récurrente. Par exemple :

— calcul de $n!$: $(n+1)! = (n+1) \cdot n!$, $0! = 1$
— calcul de la somme des éléments d'une suite : $S_{n+1} = S_n + u_{n+1}$, $S_0 = u_0$
— calcul de la somme S des diviseurs de n : on pose

$$S_i = \sum_{0 \leq d \leq i,\ n \,\%\, d = 0} d$$

puis on a $S = S_n$, $S_0 = 0$ et

$$S_{i+1} = \begin{cases} S_i & \text{si } n \,\%\, (i+1) \neq 0 \\ S_i + (i+1) & \text{sinon} \end{cases}$$