Cours d'informatique 1
Les bases de Python

I Introduction au langage Python

I.1 Bref aperçu

Le langage Python est un langage de programmation qui permet de donner des instructions à l'ordinateur, écrites de façon lisible par un humain. Il a les avantages suivants :

- Moderne et en pleine expansion, utilisé de plus en plus dans de nombreux milieux scientifiques. Pour une fois l'Éducation Nationale est vraiment à la pointe.
- Libre : n'importe qui peut l'installer sur son ordinateur, le tester, lire la documentation, le modifier et l'améliorer, contribuer au développement. La communauté Python est particulièrement grande.
- L'accent est mis sur la facilité à programmer et sur la lisibilité du programme. C'est idéal pour l'enseignement.
- Dynamique : en plus du mode script qui exécute tout un programme d'un seul coup, il y a un mode interactif qui permet de faire des tests très simplement, inspecter le contenu des variables, naviguer dans l'aide, exécuter son programme pas à pas pour trouver des erreurs.
- Haut niveau : le langage fournit de très nombreuses fonctions et constructions qui sont éloignées du fonctionnement interne de l'ordinateur mais beaucoup plus faciles à manipuler pour un humain (exemple : basiquement l'ordinateur ne sait faire que des calculs mathématiques simples et réserver des blocs de mémoire de taille fixée, pourtant les listes peuvent être concaténées, multipliées, on peut insérer, chercher…) En comparaison d'autres langages, un programme Python est souvent *beaucoup plus court*, *plus rapide à écrire*, mais fondamentalement *plus lent* : il fait beaucoup d'opérations par derrière en nous faisant croire que c'est facile !
- De très nombreuses librairies disponibles : des extensions et des fonctions qui permettent d'interagir avec les fichiers de l'ordinateur, le réseau, tracer des graphiques, traiter des statistiques… et tout est expliqué dans la documentation.

C'est donc aussi plus difficile pour un élève qu'un langage de calculatrices ou qu'un langage type *Scratch* (au collège avec le chat) car ceux-ci ont été conçus uniquement pour l'enseignement et n'ont aucune utilité professionnelle !

Les logiciels viennent avec leur propre documentation, écrite par ceux qui ont créé le logiciel, et qui décrit très précisément leur fonctionnement. *Tout* ne peut pas venir de l'enseignant : celui-ci connait avant tout les concepts généraux et a appris le langage, mais n'est pas responsable du logiciel lui-même… Il passe notamment du temps à lire les documentations, en anglais bien entendu !

Documentation Python officielle : https://docs.python.org/

Autre référence précise mais plus conviviale : https://www.w3schools.com/python/

I.2 Maniement du mode interactif

En mode interactif, les commandes rentrées après l'invite `>>>` sont exécutées unes par unes, le résultat est affiché à chaque fois, les variables sont enregistrées au fur et à mesure. Très pratique pour faire des tests rapides, explorer l'aide, comprendre d'exécution pas à pas de son programme.

Commandes intéressantes en mode interactif :

- `help()` : aide générale, ou aide sur une fonction, un module, un type…
- `type(x)` : connaitre le type de la variable `x`
- `who`, `whos` (certains logiciels comme Pyzo) : liste les variables actuellement enregistrées
- `del x` : supprime la variable `x`, qui disparait alors de la liste des variables enregistrées

I.3 Le mode script

En mode script, on peut écrire un programme d'un seul coup sur plusieurs lignes, puis l'exécuter. Les résultats ne sont pas affichés à chaque fois : il faut utiliser la commande `print()`. Les lignes démarrant par `##` servent à délimiter des *cellules*, pour garder tout dans un même fichier mais n'en exécuter qu'un morceau au choix.

Commandes intéressantes en mode script :

- `print()` : affiche le ou les arguments fournis, séparés par des virgules. Permet d'afficher directement de nombreux types, y compris les types composés `list`, `tuple`, …
- `input("`*texte*`")` : demande d'entrer quelque chose à l'utilisateur en affichant le texte, et retourne le résultat, **toujours de type `str`**
- `assert(`*condition*`)` : fait échouer le programme si la condition **n'est pas** vérifiée. Permet de protéger un programme contre des valeurs non désirées, ou de débugger le programme. Pour retourner un message d'erreur :
  `assert(`*condition*`), "`*message*`"`

II Variables et types

Une **variable** en Python a un **nom**, un **type** et un **contenu**.
Le nom est formé à partir notamment de toutes les lettres de `a` à `z`, minuscules et majuscules, des chiffres de `0` à `9` (mais pas en premier caractère !), et de l'underscore `_` pour séparer les mots. Accents possibles mais à éviter. La distinction entre minuscules et majuscules est prise en compte. Sont interdits les mots faisant déjà parti du langage, comme `def`, `return`, pour des raisons évidentes.
L'opération d'**affectation** sert à donner un contenu à la variable, on écrit par exemple `x = 3` et à partir de là la variable `x` est enregistrée et a la valeur `3`.

II.1 Les types simples

Le type entier `int` C'est le type des nombres entiers, avec un signe. Arbitrairement grands.

Le type flottant `float` C'est le type des nombres à virgules. La virgule s'obtient avec le point et la notation scientifique avec un `e` : `3.14`, `1.23e4`, `3e-10` ...
Les nombres sont stockés par l'ordinateur sous une certaine forme de notation scientifique de taille fixe. La précision est toujours limitée à une quinzaine de chiffres significatifs, mais la partie avec exposant varie de `e-308` à `e+308`.

Opérations sur les nombres Sur ces types sont disponibles :
- Les opérations `+`, `-`, `*`
- La division vers les flottants `/`
  (par exemple `6/2` vaut `3.0` qui est de type `float`, même si le résultat est un nombre entier)
- La division qui reste dans les `int` (euclidienne, avec reste) `//`, le reste `%`
  (par exemple `6//2` vaut bien `3`, et `7//2` aussi mais avec reste `7%2` qui vaut 1)
- La puissance `**`
- Les comparaisons `==`, `!=`, `<`, `>`, `<=`, `>=` qui retournent un booléen

Ne pas confondre la comparaison `==` et l'affectation `=`

Le type booléen `bool` Type ne contenant que deux éléments `True` (vrai) et `False` (faux).
Opérations possibles : `and`, `or`, `not` avec les règles de calcul habituelles sur les assertions.
C'est le type retourné par les comparaisons ainsi que par de nombreuses fonctions de « test ».

Le type vide L'objet `None` signifie qu'il n'y a rien... absence de valeur... pratique parfois comme valeur de retour d'une fonction ou bien pour initialiser une liste vide de n éléments.

Les nombres complexes `complex` Pas vu en cours, mais... c'est le type des nombres complexes. Le nombre i est noté `j` : `1j`, `-2 + 3j`, ... évidemment avec ses opérations (et la conjugaison !), toutes décrites dans `help(complex)`.

II.2 Les types composés

Le type des chaines de caractères `str` Type pouvant contenir du texte, écrit entre guillemets doubles `"texte"`, enregistré comme une liste de caractères individuels.
Opérations possibles sur une chaine `s` :
- Longueur : `len(s)`,
- `i`-ème élément : `s[i]` (à partir de 0)
- Concaténation (= mise bout à bout) : `+`
- Multiplication par un entier (= répéter `n` fois la chaine) : `s * n`
- La chaine vide est `""`

Ajoutons de plus : on peut insérer dans une chaine certains caractère spéciaux tels que :
- Comment insérer un guillemet double dans une chaine ? Le faire précéder de `\` : `"Une \"citation\" dans une chaine"`
- Insérer un symbole de retour à la ligne avec `\n` : `"ligne1\nligne2"`
- Insérer un caractère `\` : `"\\"`

Enfin on peut obtenir quelques informations sur une chaine `s` ou un caractère seul : les méthodes suivantes retournent un booléen
- `s.isalpha()` : est composée uniquement de lettres
- `s.isdigit()` : est composée uniquement de chiffres
- `s.isupper()`, `s.islower()` : est composée de majuscules, de minuscules

et d'autres dans `help(str)`.

Les listes `list` Écrites entre crochets, éléments séparés par des virgules :
`L = [x, y, ...]`
Opérations possibles :
- Comme pour `str` : longueur `len(L)`, `i`-ème élément `L[i]`, concaténation `+`, multiplication `L * n`
- Sélection d'éléments : `L[a:b]` (de `a` à `b`, exclus), `L[a:]`, `L[:b]` ; de plus les indices négatifs reviennent en arrière, `L[-1]` est le dernier élément
- Ajouter `x` à la fin de `L` : `L.append(x)`
- Supprimer le dernier élément, et le retourner : `x = L.pop()`
- Recopier la liste : `M = L.copy()`

... et bien d'autres dans `help(list)`.
Création de listes :

- Lister les éléments
- Conversion de types (ci-dessous)
- Boucle partant d'une liste vide `L = []` et itérant des `L.append()`
- Par multiplication : `L = [0] * n` ou `L = [None] * n`
- En compréhension : `[... for ... in ...]`
  ou même `[... for ... in ... if ...]`

Les tuples `tuple` Correspond au produit cartésien de deux ou plusieurs types. On peut l'*affecter* comme un produit cartésien : `t = (3, 4)`. Et *récupérer* les valeurs dans deux variables différentes : `(x, y) = t`.
Opérations possibles : en fait les mêmes que pour `list` et `str` : `len()`, `t[i]`, `+`, `t * n`. Quelle différence avec les listes alors ? Une fois créé, un tuple ne peut pas être modifié (pas de `t[0] = ...`). On l'utilise donc surtout pour :

- Une fonction qui retourne plusieurs arguments,
- Grouper deux ou trois valeurs ensemble, comme pour les coordonnées d'un point, d'un vecteur,
- Échanger des variables, sans passer par des variables intermédiaires :
  `(y, x) = (x, y)` : échange `x` et `y`
  au lieu de : `z = y` (sauvegarde de `y` dans `z`) puis `y = x` puis `x = z`
  `(x, y) = (y, x + y)` (plus compliqué : utile pour Fibonacci)

II.3 Conversions de type

À chacun des types ci-dessus correspond une fonction de conversion **vers** le type : `int()`, `float()`, `bool()`, `complex()`, `str()`, `list()`, `tuple()`.
Exemples intéressants :

- `input()` retourne toujours le type `str`. Pour demander un nombre entier à l'utilisateur :
  `n = int(input("Entrez un nombre : "))`
  (demander un nombre à virgule : idem avec `float`)
- Obtenir la représentation en texte d'un nombre `n` : `str(n)`
- Former une liste à partir d'un intervalle :
  `>>> list(range(10))`
  `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
  ou bien d'une chaine de caractères
  `>>> list("Bonjour")`
  `['B', 'o', 'n', 'j', 'o', 'u', 'r']`
- Idem pour former des tuples.

III Fonctions

Déclaration de fonction :

```
def nom(argument1, argument2, ...):
    """ documentation """
    instructions
```

où l'instruction `return` arrête la fonction et renvoie une valeur.
Conseils :

- Écrire des fonctions dès qu'un morceau de programme se répète avec des paramètres qu'on veut faire varier.
- Les fonctions prennent des arguments plutôt que d'utiliser `input`.
- Les fonctions calculent et retournent une valeur avec `return` plutôt que d'utiliser `print` à la fin. Cependant la commande `print` à l'intérieur est utile pour observer ce que fait la fonction au fur et à mesure : ne pas hésiter à en rajouter pour comprendre son programme, quitte à les enlever plus tard.
- Réfléchir à la logique du fait que, si `return` arrête la fonction, alors certains `else` deviennent inutiles ; cela permet aussi d'interrompre élégamment une boucle en cours de route.

IV Conditions et boucles

IV.1 Conditions

Syntaxe générale des conditions :

```
if condition1:
    instructions1
elif condition2:
    instructions2
elif ...
    ...
else:
    instructions sinon
```

Où les `elif` et le `else` ne sont pas obligatoires : si la condition n'est pas vérifiée le programme passe simplement à la suite.
La condition est en fait une valeur booléenne : une ligne telle que
`if f(x) == True:`
peut être simplement remplacée par
`if f(x):`

IV.2 Boucle `while`

Syntaxe générale de la boucle `while` :

```
while condition:
    instructions
```

Exemple de base : répéter `n` fois (`i` varie de 0 à $n-1$ inclus)

```
i = 0
while i < n:
    ...
    i = i + 1
```

Attention aux **boucles infinies** dans lesquelles la condition est toujours vraie... cela se produit par exemple si on oublie la ligne `i = i + 1` (**incrémentation** de la variable `i`).

Remarque : `i = i + 1` admet aussi le raccourci `i += 1`. D'autres opérations admettent aussi une telle syntaxe : il existe `+=` pour les listes ou chaines, `-=`, `*=` etc.

L'instruction `break` permet de sortir de la boucle en cours de route et passer à la suite.

L'instruction `continue` termine le passage actuel dans la boucle et revient au départ (le test de la condition, puis éventuellement le re-passage dans la boucle).

IV.3 Boucle `for`

Version de base : la boucle

```
for i in range(a, b):
    instructions
```

est **exactement équivalente** à

```
i = a
while i < b:
    instructions
    i = i + 1
```

et donc `b` est **exclus**. De plus `range(n)` est la même chose que `range(0, n)`.

Mais c'est bien plus ! Pour parcourir une liste `L` (ou une chaine, ou un tuple) on peut **itérer sur les indices**

```
for i in range(len(L)):
    ... (on a accès à i et à L[i])
```

mais aussi, plus direct et élégant, **itérer sur les éléments**

```
for x in L:
    ... (on a accès uniquement à l'élément x de L, un par un)
```

Remarques :

- La boucle `for` fonctionne pour tous les **objets itérables** (= qui peuvent fournir leurs éléments au compte-goutte, l'un après l'autre) : `list`, `str`, `tuple`, ... On peut d'ailleurs fabriquer ses propres objets itérables (Python niveau avancé !)
- `range(n)` est un objet de type `range` (intervalle), qui est itérable, et fournit les éléments uns par uns dans cet ordre `0`, `1`,..., `n-1`.
- `in` est le mot-clé qui teste (renvoie un booléen) l'appartenance d'un élément `x` à un objet itérable :

```
>>> 3 in range(10)
True
>>> 12 in range(10)
False
>>> "pain" in ["riz", "oeufs", "pain", "pâtes"]
True
```

V Quelques fonctions utiles

V.1 Bases

Dites *built-in* (encodées au cœur du langage)

- `abs()` : valeur absolue, ou module d'un nombre complexe
- `sum()` : somme des éléments (d'un objet itérable : liste, tuple, ...)
- `min()`, `max()` : comme leur nom l'indique, sur tout objet itérable

V.2 Mathématiques

Dans le module `math` : obtenues à partir de
`import math` (importe tout, les fonctions ont le préfixe `math.`) ou bien
`from math import ...` (importe les fonctions demandées, sans préfixe).

- `sqrt()` : racine carrée
- `pi` : constante mathématiques π
- `floor()` : partie entière ($\neq$ conversion de type `int()`), ainsi que `ceil()` : partie entière supérieure (plus grand entier supérieur ou égal)
- `sin()`, `cos()`, `tan()`, `exp()`, `log()`, `log10()` : comme leur nom l'indique
- `asin()`, `acos()`, `atan()` : les fonctions trigonométriques réciproques

V.3 Aléatoire

Dans le module `random` :
`import random` (préfixe `random.`) ou bien
`from random import ...` (importe les fonctions demandées, sans préfixe)

- `randint(a, b)` : nombre aléatoire, entier, entre `a` et `b` (tous les deux **inclus**, pas comme dans `range(a, b)` !)
- `randrange(a, b)` : idem mais `b` exclus, comme dans `range(a, b)`
- `random()` : nombre aléatoire flottant dans l'intervalle $[0, 1[$