Introduction à l'algorithmique et à Python

Objectifs

A la fin de la séquence d'enseignement les élèves doivent :

- être en mesure de manipuler un environnement de développement intégré (IDE)
- choisir un type de données en fonction d'un problème à résoudre
- concevoir l'en-tête (ou la spécification) d'une fonction, puis la fonction elle-même
- traduire un algorithme simple dans un langage de programmation

Table des matières

1 Langages de programmation **2**
- 1.1 Diversité de langage . . . 2
- 1.2 Niveau de langage . . . 2
- 1.3 Langage interprété et langage compilé . . . 2
- 1.4 Pourquoi Python ? . . . 2

2 Environnement de Développement Intégré (IDE) **3**
- 2.1 L'interpréteur . . . 3
- 2.2 L'éditeur . . . 3

3 Typage **3**
- 3.1 Différents types . . . 3
- 3.2 Les entiers . . . 4
- 3.3 Les flottants . . . 4
- 3.4 Les Booléens . . . 4

4 Affectation **5**
- 4.1 Mise en mémoire . . . 5
- 4.2 Pythoneries . . . 5

5 Types composés **5**
- 5.1 Les types immuables (non mutables) . . . 5
- 5.2 Les types mutables . . . 6

6 Fonctions **8**
- 6.1 Définition . . . 8
- 6.2 Portée des variables . . . 8
- 6.3 Fonction lambda . . . 9
- 6.4 Utilisation des bibliothèques . . . 9

7 Structures de contrôle **9**
- 7.1 Instructions conditionnelles . . . 9
- 7.2 Instructions itératives . . . 10

8 Utiliser des fichiers **10**
- 8.1 Ouvrir et fermer un fichier de type `.txt` . . . 10
- 8.2 Lire et écrire dans un fichier de type `.txt` . . . 11
- 8.3 Construire un tableur `.csv` . . . 11
- 8.4 Lire et écrire dans un fichier `.xlsx` . . . 11

1 Langages de programmation

1.1 Diversité de langage

Sur la page https://fr.wikipedia.org/wiki/Liste_des_langages_de_programmation sont regroupés par ordre alphabétique, les différents langages de programmations. On en compte plus de 600 !

Il faut faire la distinction entre un langage (comme **Python**) et son compilateur (comme Idle, Spider, Pyzo...).

1.2 Niveau de langage

On distingue des langages de différents niveaux :

- bas niveau (proche machine): permet de contrôler tout ce qui se passe dans la machine pendant l'exécution du code. C'est un langage machine lisible par un humain qui permet basiquement de manipuler explicitement des registres, des adresses mémoires, voire des instructions machines. Par exemple : assembleur, C...
- haut niveau (proche utilisateur) : permet à l'utilisateur d'utiliser des fonctions complexes sans se soucier de l'aspect machine (enfin un petit peu quand même). Pour son optimisation, il utilise parfois (souvent) des langages de bas niveau de façon transparente pour l'utilisateur. Par exemple : FORTRAN, Python, PHP, SQlite...

1.3 Langage interprété et langage compilé

On peut aussi trier les langages entre langages de programmation interprétés et langages de programmation compilés:

- **interprété :** son utilisation nécessite un interpréteur

  Code source $\xrightarrow{\text{interpréteur}}$ Résultat

  - **avantage :** Test immédiat
  - **inconvénient :** plus lent
- **compilé :** son implémentation exécutable requiert un compilateur.

  Code source $\xrightarrow{\text{Compilateur}}$ code machine $\xrightarrow{\text{exécuteur}}$ Résultat

  - **avantage :** Rapide
  - **inconvénient :** nécessite une première étape de compilation

La différence centrale entre compilé et interprété est la suivante : là où le compilateur traduit une bonne fois pour toute un code source en un fichier indépendant exécutable (donc utilisant du code machine ou du code d'assemblage), l'interprète est nécessaire à chaque lancement du programme interprété, pour traduire au fur et à mesure le code source en code machine.

Cette traduction à la volée est la cause première de la lenteur des langages dits interprétés.

1.4 Pourquoi Python ?

La réponse se trouve à l'adresse http://www.linux-center.org/articles/9812/python.html.

2 Environnement de Développement Intégré (IDE)

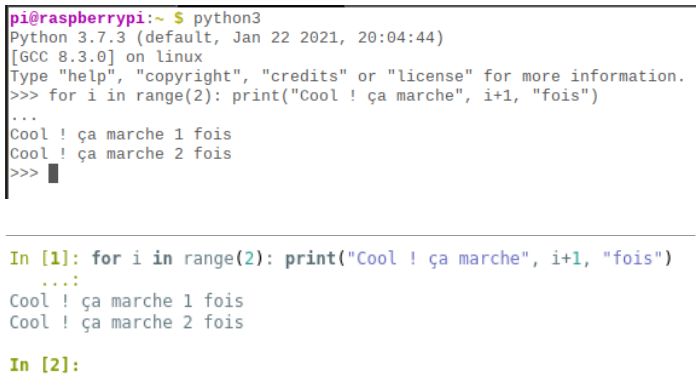
2.1 L'interpréteur

Nous utiliserons **Python** avec un interpréteur. Cela se présente sous la forme d'une fenêtre avec des >>> après lesquels on peut entrer des instructions :

```
>>> for i in range(2): print("Cool ! ça marche", i+1, "fois")

Cool ! ça marche 1 fois
Cool ! ça marche 2 fois
>>>
```

En pressant la touche `Entrée` du clavier, **Python** interprète les instructions et affiche le résultat.



2.2 L'éditeur

Pour lancer plusieurs fois des commandes dans l'interpréteur, il faut les retaper. Difficile donc de sauvegarder son travail et de le relancer. On utilise alors un éditeur dans lequel on va taper le texte.

Certes, on peut utiliser un simple bloc-note pour taper le code mais d'autres éditeurs proposent une coloration syntaxique. C'est beaucoup plus clair.

`Idle` présente une fenêtre pour l'éditeur et une fenêtre pour l'interpréteur. Pyzo regroupe en une fenêtre principale plusieurs fenêtres dont l'éditeur et l'interpréteur. Pour interpréter le programme (`.py`) de l'éditeur, il suffit de taper `F5`. On peut aussi interpréter dans une console le programme (.py) en tapant `python programme.py` ou en double-cliquant sur le fichier.

3 Typage

3.1 Différents types

On distingues différentes type de données :

`bool`	Booléen
`Int`	Nombre entier optimisé
`long`	Nombre entier de taille arbitraire
`float`	Nombre à virgule flottante
`complex`	Nombre complexe
`str`	Chaîne de caractère
`unicode`	Chaîne de caractère unicode

`tuple`	Liste de longueur fixe
`list`	Liste de longueur variable
`dict`	dictionnaire
`file`	Fichier
`NoneType`	Absence de type
`NotImplementedType`	Absence d'implementation
`function`	fonction
`module`	module

La fonction `type()` permet de connaître le type d'une variable.

3.2 Les entiers

Pas besoin de les entrer en binaire (ouf !), **Python** gère directement leur stockage en mémoire. Contrairement à votre calculatrice, il n'y a pas de taille limite pour le stockage de ces nombres (enfin, il faut quand même avoir suffisamment de place sur le disque dur...). Pour convertir un nombre en entier, on utilise la commande `int`.

Il est possible d'exercer les opérations suivantes sur les entiers :

- Opérations de base : `+ - *`
- Division entière (quotient) : `//`
- Reste (modulo) : `%`
- Exposant : `**`

Attention ! en **Python 2** la division entière se fait simplement avec `/` alors qu'en **Python 3** `/` transforme un rapport de nombres entiers en nombre flottant.

```
>>> type(4/2)
<class 'float'>
>>> type(4//2)
<class 'int'>
```

3.3 Les flottants

La norme IEEE 754 permet de définir le format de stockage des nombres réels (cf le cours sur le stockage des nombres).

Les opérateurs `+`, `-`, `*` ,`/` et `**` fonctionnent sur les flottants.

Remarque : si au moins un terme est flottant (donc s'il y a un `.` quelque part), le résultat sera flottant.

Attention ! Le stockage sous forme binaire conduit alors parfois à arrondir le nombre.

Pour réduire la taille d'affichage, on peut jouer sur la commande `format`.

Pour les nombres complexes, on utilise `j` comme en physique et en SI mais pas comme en math.

```
>>> 0.6 - 0.6
0.0
>>> 0.3 + 0.3 - 0.6
0.0
>>> 0.1 + 0.2 + 0.3 - 0.6
1.1102230246251565e-16
```

```
>>> format(16.7, '.2f')
'16.70'
>>> format(16.7, '.4f')
'16.7000'
>>> format(16.7, '.8f')
'16.70000000'
>>> format(16.7, '.16f')
'16.6999999999999993'
```

```
>>> (1j) ** 2
(-1 + 0j)
>>> type((-1 + 0j))
<class 'complex'>
```

3.4 Les Booléens

Pour les booléens, on distingue :

- deux éléments : `True` et `False`.
- trois opérations principales : `not`, `and`, `or`.
- six assertions : `== != < > <= >=`

Exemple :

```
>>> 1 == 2
False
>>> 1 + 2 == 3
True
>>> 0.1 + 0.2 == 0.3
False
>>> Ooouille...
```

Attention ! ne pas faire de test d'égalité sur deux flottants !!

4 Affectation

4.1 Mise en mémoire

L'affection se fait avec l'opérateur =. Il s'agit de définir un emplacement dans la mémoire, de lui donner une étiquette (le nom de la variable), et d'y stocker quelque chose.

Un nom de variable doit toujours commencer par une lettre et ne pas contenir de caractère spécial, d'accent,.... Seul le souligné _ est autorisé.

Remarque : **Python** respecte la casse, il fait la différence entre minuscule et majuscule.

La commande `id` permet de connaître l'adresse de stockage de la variable.

```
>>> a = 42
>>> b = a
>>> b
42
>>> id(a)
507100096
>>> id(b)
507100096 # meme adresse
>>> a = a + 3 # on change a
>>> b
42
>>> a
45
>>> id(a)
507100192 # a a change d'adresse
>>> id(b)
507100096 # n'a pas change d'adresse
```

4.2 Pythoneries

Il est courant d'avoir à ajouter ou retrancher des valeurs à une variable. **Python** propose des raccourcis syntaxiques : `+=`, `-=`, `*=`, `/=`, `//=`, `%=` et `**=`

`x` conserve la même adresse en mémoire.

```
>>> x = 3
>>> x += 4
>>> x
7
```

5 Types composés

5.1 Les types immuables (non mutables)

5.1.1 Les chaînes de caractères

Les chaîne de caractères (`str` de string en anglais) sont des suites finies de caractères. Elles sont délimitées par des apostrophes `'`, des guillemets `"` ou des triples guillemets `"""` si on veut écrire une même chaine sur plusieurs lignes.

Sur l'exemple précédent, les renvois à la ligne ont été remplacés par `\n` (`\t` affiche une tabulation). De même, `\'` et `\"` permettent de faire apparaître les caractères.

```
>>> 'Comment mettre l''apostrophe ?'

'Comment mettre lapostrophe ?'
>>> 'Comment mettre l'apostrophe ?'

SyntaxError: invalid syntax
>>> 'Comment mettre l\'apostrophe ?'

"Comment mettre l'apostrophe ?"
>>> """Puis écrire
sur plusieurs
lignes ? """

'Puis écrire\nsur plusieurs\nlignes ? '
```

On peut :

- accéder au $i^{\text{ème}}$ caractère d'une chaîne `s` en plaçant l'indice entre crochets `s[i]`. La numérotation commence à 0 et elle accepte des indices négatifs : -1 donne le dernier caractère, -2 l'avant dernier,...
- extraire des caractères d'une chaine pour en faire une sous-chaine `s[i:j]`
- concaténer des chaînes (les mettre bout-à-bout) grâce à l'opérateur `+`
- obtenir la longueur d'une chaîne grâce à l'opérateur `len()`
- convertir des objets en chaîne de caractère grâce à l'instruction `str()`
- évaluer des chaines de caractères grâces à l'instruction `eval()`
- faire des test d'inclusion avec l'opérateur `in`

```
>>> a = input()
1
>>> b = input()
2
>>> a
'1'
>>> b
'2'
>>> a + b
'12'
>>> eval(a) + eval(b)
3
```

Attention ! il n'est pas possible de la modifier. Les chaînes ne sont pas mutables !

On ne peut donc pas écrire `s[3]='x'` pour changer la $4^{\text{ème}}$ lettre de `s` en `x`.

Remarque : si `print` permet d'afficher des caractères ou des nombres, la commande `input` permet d'entrer uniquement du texte. Pour utiliser les valeurs numériques, il faut utiliser l'instruction `eval` (ou encore `int` pour convertir en entier ou `float` pour convertir en flottant).

5.1.2 Les tuples

Tuple est le mot anglais pour *n*-uplet : ce sont des éléments (pas forcément de même type) séparés par des virgules, entourés de parenthèses. **Exemple :** Tuple vide : `()`. Tuple à un élément : `(elem)`.

Les propriétés et opérations sont similaires à celles sur les chaînes.

Attention ! l'instruction `in` permet de savoir si le tuple contient un élément mais pas un *sous-tuple*.

5.2 Les types mutables

5.2.1 Les listes

Les listes se présentent comme les tuples, sauf qu'elles sont délimitées par des crochets. Mis à part les instructions `str` et `eval`, les opérations élémentaires sur les listes sont identiques à celles présentées pour les tuples.

Cependant, une différence fondamentale est qu'elles sont mutables : on peut changer leurs éléments.

Attention ! deux variables peuvent référencer une même liste. Il faut utiliser `L2 = L1.copy()` pour obtenir `L2` à partir de `L1` mais enregistré à une autre adresse.

L'instruction `L.append(x)` permet d'ajouter `x` à la liste `L` sans changer son adresse en mémoire. L'instruction `L.pop()` enlève le dernier élément d'une liste. Il est possible d'affecter cette élément à une variable pour en récupérer la valeur `der_val = L.pop()`.

Remarque : pas de panique ! C'est en pratiquant que les choses s'éclairent !

5.2.2 Les ensembles

On dispose en **Python** d'un type ensemble : set. Comme en maths, l'ordre des éléments n'importe pas, et ils n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide :** {} ou `set()`
- **Affectation :** `s = {1, 2, 3}` affecte l'ensemble `{1, 2, 3}` à `s`.
- **Cardinal :** `len(s)`
- **Appartenance :** `x in s`, `x not in s`.
- **Inclusion :** `s1.issubset(s2)` ou `s1 <= s2`
- **Union :** `s1.union(s2)` ou `s1|s2`
- **Intersection :** `s1.intersection(s2)` ou `s1 & s2`
- **Différence :** `s1.difference(s2)` ou `s1 - s2`
- **et aussi :** `s.add(x)`, `s.remove(x)`...

Pour la copie, les mêmes règles que pour les listes s'appliquent !

5.2.3 Les dictionnaires

C'est une structure de données de type dict qui permet d'associer une valeur à une étiquette (appelée clé) autre que l'indice d'une case comme pour les listes.

On crée un dictionnaire comme on crée un ensemble mais chaque élément d'un dictionnaire est composé de la clé suivie du symbole : puis de la valeur associée à la clé {`cle_1: val_1, cle_2: val_2`} :

```
>>> dic_sup = {"sup-1": "MPSI-1", "sup-2": "MPSI-2",
        "sup-3": "MP2I", "sup-4": "PCSI-1", "sup-5": "PCSI-2"}
>>> dic_sup["sup-5"]
'PCSI-2'
```

```
>>> dic_cub = {x:x**3 for x in range(5)}
>>> dic_cub
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}
>>> dic_cub[4]
64
```

```
>>> dic_cub[10] = 2
>>> dic_cub
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 10: 2}
```

Écrire `dic[cle]` pour un dictionnaire `dic` permet d'accéder à la valeur `val` associée à la clé `cle`. L'objet `dict` étant de type mutable, on peut alors changer la valeur associée à une clé. En revanche, les clés sont non mutables.

Les méthodes `keys` et `items` permettent respectivement d'obtenir un itérable des clés et un itérable des tuples (clé, valeur).

```
>>> a = dic_cub.items()
>>> a
dict_items([(0, 0), (1, 1), (2, 8), (3, 27), (4, 64), (10, 2)])
>>> type(a)
<class 'dict_items'>
```

L'instruction `list` permet de convertir l'itérable en liste :

```
>>> b = list(a)
>>> b
[(0, 0), (1, 1), (2, 8), (3, 27), (4, 64), (10, 2)]
>>> type(b)
<class 'list'>
```

Pour la copie, les mêmes règles que pour les listes s'appliquent.

6 Fonctions

6.1 Définition

Une fonction est un objet qui prend des valeurs en paramètres et, après calculs, manipulations et autres, renvoie un autre objet via l'instruction `return`. Un fonction correspond donc à un ensemble d'instructions que l'on pourra *appeler* plus loin dans le programme, ce qui évitera de réécrire plusieurs fois les mêmes instructions ou types d'instructions.

Une fonction s'introduit avec le mot-clé `def` et il ne faudra pas oublier le caractère `:` en fin de première ligne pour dire à Python qu'un nouveau bloc commence.

```
def nom_de_la_fonction(argument1, argument2, ...):
        """Description de la fonction"""
        <instruction1>
        <instruction2>
        ...
        return <instruction>
```

On peut remarquer que les lignes 2 à 4 des instructions ci-dessus sont décalées. On dit qu'elles sont indentées d'une tabulation (cf touche `tab` du clavier). L'indentation est capitale en **Python** . Un retour à la ligne signe la fin de la définition de la fonction.

L'instruction `return` peut être omise et tout ce qui se trouve après cette instruction n'est pas exécuté lors de l'appel à la fonction. On peut utiliser `return None` pour sortir de la fonction sans rien renvoyer ou ne pas mettre de `return` du tout. On parle alors de procédure et non de fonction.

Attention ! l'indentation (tabulations de 4 espaces couramment employée) doit être rigoureusement respectée pour être correctement interprétée.

On peut rajouter dans le corps de la fonction une ligne `assert <test>` renvoyant une erreur si le test n'est pas vérifié. Cela permet par exemple de s'assurer que les données entrées comme arguments de la fonction vérifient certaines conditions.

6.2 Portée des variables

6.2.1 Variables locales

Les variables utilisées à l'intérieur d'une fonction sont en général des variables locales : elle n'existent que dans le contexte de l'exécution de la fonction et disparaissent lorsque l'on sort de celle-ci.

Si une variable de même nom existait hors-contexte, celle-ci ne sera pas modifiée.

Attention ! ceci n'est valable que pour les variables de type **immuables**.

6.2.2 Variables globales

Il se peut que l'on ait besoin de modifier des variables définies hors-contexte à l'intérieure d'une fonction. Il est alors possible de la déclarer avec `global`.

6.3 Fonction lambda

Il peut parfois être pratique de définir une fonction en une ligne. On utilise alors la fonction `lambda` :

```
ma_fonc = lambda x, y : x ** y
```

Ainsi, si l'argument d'une fonction f_1 est une autre fonction f_2, si au moment de l'appelle de la fonction f_1, la fonction f_2 n'a pas été déclarée, il est possible de le faire à cet instant :

```
>>> def ma_fonc(f, x):
        return f(x) ** 2
>>> ma_fonc(lambda x : 3 * x, 3)
81 # (3 * 3) ** 2
>>> ma_fonc(lambda x : 2 * x, 3)
36 # (2 * 3) ** 2
```

6.4 Utilisation des bibliothèques

Après avoir défini des fonctions, il est possible d'utiliser des modules (appelés aussi bibliothèques) contenant des fonctions. Selon les modules, ces fonctions sont optimisées pour s'exécuter très rapidement.

- `from math import *` : on importe la totalité de la bibliothèque mathématique
- `from math import sqrt, sin, pi` : on importe uniquement les fonctions souhaitées
- `import math as m` : on écrira `2 * m.sqrt(3)` (resp. `m.sin(pi / 4)`) pour calculer $2.\sqrt{3}$ (resp. $\sin(\pi/4)$).

Nous utiliserons principalement les bibliothèques **math** (fonction mathématiques), **random** (tirage au sort), **numpy** (algèbre linéaire), **scipy** (calcul numérique) et **matplotlib** (pour tracer des courbes).

7 Structures de contrôle

7.1 Instructions conditionnelles

Lorsque dans un programme plusieurs cas se présentent et que des tests sont nécessaires, il est possible d'utiliser la structure suivante :

- if apparaît toujours sur la première ligne de la structure conditionnelle.
- elif apparaît si au moins un deuxième test doit avoir lieu.
- else apparaît pour utiliser tous les cas non traiter par `if` et `elif`.

```
if test_1 :
    instruction_1
elif test_2:
    instruction_2
    ....
elif test_n:
    instruction_n
else:
    instruction_n+1
```

7.2 Instructions itératives

7.2.1 Boucle conditionnelle `while`

On cherche parfois à faire des opérations jusqu'à ce qu'une **condition** soit atteinte. En programmation, on utilise alors la boucle conditionnelle while (tant que). Tant qu'une condition n'est pas vérifiée, on exécute plusieurs fois la même suite d'instructions.

```
def dectobin(n):
    r = ""
    while n! = 0:
        if n%2 == 0:
            r = "0" + r
        else:
            r = "1" + r
        n = n // 2
    return r
```

Attention ! il faut bien réfléchir à ce qu'on fait car si la condition de sortie de boucle n'arrive jamais, le programme tourne toujours...

Exemple : Conversion d'un nombre entier de la base décimale en base 2 :

Remarque : `break` permet de sortir directement de la boucle.

7.2.2 Boucle inconditionnelle `for`

Si on veut répéter une instruction sur un nombre fini de valeurs, on peut utiliser une boucle `for`. En **Python** , la syntaxe est :

```
for <variable> in <iterable>:
        <instruction>
```

où `<iterable>` peut être une liste, une chaîne de caractères, un tuple, un ensemble, un dictionnaire... ou :

- `range(n)` pour les entiers entre 0 et $n-1$ (il y a donc n termes)
- `range(k,n)` pour les entiers entre k et $n-1$
- `range(k,n,p)` pour les entiers entre k et $n-1$ avec un pas de p.

On peut s'en sortir avec une boucle `while`, mais c'est parfois plus rapide avec une boucle `for`:

```
for i in range (10):
    print(i)
```

```
i=0
while i < 10:
    print(i)
    i = i + 1
```

Il est possible de faire tenir les instructions précédentes en une seule ligne : `for i in range(10) : print(i)`

8 Utiliser des fichiers

8.1 Ouvrir et fermer un fichier de type `.txt`

Il est possible de lire et d'écrire dans un fichier texte. Pour cela, il faut créer un pointeur vers le document lors de l'appel du fichier en déclarant ce qu'on veut en faire :

- `'r'` ouvrir le document pour le lire
- `'w'` ouvrir le document pour y écrire en effaçant le fichier (si le document n'existe pas, il sera créé)
- `'r+'` lire dans le document et y écrire.

On ferme le fichier en ajoutant `.close()` au pointeur.

8.2 Lire et écrire dans un fichier de type `.txt`

On considère un fichier ouvert avec le pointeur `fic` (`fic = open(...)`).

Pour lire une ligne, on utilise `fic.readline()` et pour tout lire `fic.read()`. Pour lire 3 caractères `fic.read(3)`.

Il est possible de ranger toutes les lignes d'un fichier dans une liste en tapant `fic.readlines()`. Si on veut n'en lire que 3, on écrit plutôt `fic.readlines(3)`.

Pour écrire dans un fichier qu'on souhaite initialement vide, on utilise la méthode write() associée au pointeur :

```
fic =  open('fichier.txt', 'w')
fic.write('Voici comment mettre du texte dans le fichier.')
fic.close()
```

8.3 Construire un tableur `.csv`

Un fichier d'extension `.csv` est un tableur de complexité minimale. Les colonnes y sont séparées par des `;`. Un fichier `.csv` s'ouvre en tableur sous **Excel**, **OpenOffice**, **LibreOffice**, ...

Pour construire le fichier `.csv`, on peut utiliser le pointeur `fich` comme précédemment mais associé à `open('fichier.csv', 'w')`.

Reste alors à écrire les lignes en prenant soin d'insérer des `";"` entre chaque colonne du tableur espéré.

8.4 Lire et écrire dans un fichier `.xlsx`

Pour ma part, j'utilise le package `openpyxl`. Vous pouvez en retrouver les détails et des exemples ici.

Images

A suivre dans le cours dédié aux tableaux 2D...