Informatique 2 : Programmation de base en Python

I Algorithmique élémentaire

1 Définition

Les programmes que nous écrirons sont une succession d'opérations élémentaires permettant d'effectuer un calcul ou de résoudre un problème. Bien que nous écrirons tous ces programmes dans le langage Python, la stratégie de résolution est indépendante du langage choisi : cette stratégie est ce qu'on appelle un algorithme

Définition 1 (*Algorithme*)

Exemple 2

2 Structure d'un algorithme

Quand on rédige un algorithme, on distingue :

1. sa structure physique :

2. ses structures logiques :

Pour décrire un algorithme, on peut utiliser le **pseudo-langage** qui n'est pas directement du langage Python mais une écriture en « français » de la stratégie employée pour le problème demandée. On verra des exemples dans le 4.

3 Recherche des erreurs et expérimentations

La réalisation d'un programme, même simple, peut toujours être source d'erreurs de différents types :

1. des erreurs syntaxiques :

2. des erreurs sémantiques :

3. des erreurs à l'exécution :

Remarque 3

4 Des exemples d'algorithme écrits en pseudo-langage

Avant de voir les bases de la syntaxe en Python, on peut toujours s'entrainer à programmer dans un pseudo-langage. Il s'agit simplement de rédiger des instructions élémentaires de façon abstraite. Par exemple, pour additionner deux nombres entiers, on pourra écrire l'algorithme suivant :

Lire A, B
C $\leftarrow$ A + B
Écrire C

Exemple 4

Écrivons un algorithme en pseudo-langage qui, en fonction de $n \in \mathbb{N}^*$ retourne la somme

$$1 + 2 + \cdots + n.$$

Exemple 5

Écrivons un algorithme en pseudo-langage qui, en fonction $a, b, c \in \mathbb{R}$ renvoie les racines réelles éventuelles du polynôme $aX^2 + bX + c$.

II Types en Python et opérations de base

1 Gestion des variables et affectation

Dans de nombreux problèmes, on sera amené a définir des objets ou a mémoriser des valeurs au travers de **variables**. Pour définir une variable en Python, on utilisera le symbole d'affectation `=` de la manière suivante

```
nom_de_la_variable = valeur
```

Remarque 6

Pour plus de clarté, le nom des variables est souvent simple, écrit en minuscule et on respectera l'insertion d'espace autour du signe = pour l'affectation. De plus, on pourra prendre l'habitude de commenter les lignes de code grâce au symbole #.
On peut alors utiliser les variables définies pour d'autres instructions même si on fera attention à l'ordre dans lequel on a défini nos variables :

```
>>> a = 2; a*b ; b = 3
Traceback (most recent call last) :
(...)
NameError : name 'b' is not defined
```

```
>>> a = 2 ; b = 3 ; a*b
6
```

Python autorise la **réaffectation**, c'est-à-dire qu'on peut modifier la valeur donnée à l'**identificateur** :

```
>>> a = a+2; a
4
```

Par contre, pour libérer le nom d'une variable, on utilise la fonction `del` :

```
>>> del(a); a
Traceback (most recent call last) :
(...)
NameError : name 'a' is not defined
```

Remarque 7

2 Notion de type d'objet

Quand on définit une variable en Python, il n'est pas utile d'en préciser son **type**. En effet, pour chaque objet manipulé, Python lui associe automatiquement une **classe** qu'on obtient avec la fonction `type` :

```
>>> a = 2.0/4; a; type(a)
0.5
<type 'float'>
```

```
>>> b = 'informatique'; b; type(b)
'informatique'
<type 'str'>
```

On donne quelques exemples de classes :

- la classe des nombres entiers définie comme `class 'int'` pour *integer* ;
- la classe des nombres à virgule flottante définie comme `class 'float'` pour *floating number* ;
- la classe des chaînes de caractères définie comme `class 'str'` pour *string* ;
- la classe des listes (qui sera vu plus tard) définie comme `class 'list'` pour *list*.

Les classes précédentes sont les principales que nous verrons cette année même si nous en verrons quelques autres.

Exemple 8

3 Opérateurs « mathématiques »

Les quatre opérateurs mathématiques de base en Python sont `+,-,*,/,**,//,%`. Il faudra faire attention au fait que Python donnera des résultats différents suivant le type des objets sur lesquels agissent ces opérateurs. Précisons le rôle de chacun pour les quatre classes de l'exemple précédent :

action sur opérateur	entier `int`	flottants `float`	chaînes de caractères `str`	liste `str`
`+`				
`-`				
`*`				
`/`				
`**`				
`//`				
`%`				

Exemple 9

III Fonctions en Python

1 Notion de fonction

Une fonction en Python est une généralisation de la notion de fonction mathématique : celle-ci prend en argument un certain nombre d'objets (éventuellement aucun) pour en retourner d'autres après un calcul. Pour construire un tel programme dans le langage Python, il suffira de rédiger les séquences d'instruction dans l'éditeur de texte en respectant quelques règles :

- le nom de la fonction sera toujours écrit en minuscule et sera simple mais compréhensible ;
- rédiger des commentaires avec le symbole # tout au long du programme pour en faciliter sa compréhension, mais aussi sa construction,
- respecter la syntaxe du langage Python : saut de ligne, utilisation des symboles : et ; indentation car c'est cette structure physique qui sera interprétée par l'interface.

On compilera ensuite cette fonction avec la touche F5 pour pouvoir l'utiliser dans la console.

Remarque 10

2 Premier exemple de fonction : la fonction mathématique

On pourra utiliser l'éditeur de texte pour définir des fonctions mathématiques simples avant de les évaluer. Par exemple, on peut construire la fonction suivante :

```
def g(x):                    # on définit f dont l'argument formel est x
    return 1/(x**2+x)                    # et qui à x associe 1/(x^2+x)
```

Encore une fois, on fera attention à la syntaxe associée à cette commande. Sous Python, le **saut de ligne** après le symbole : et l'**indentation**, correspondant à 4 espaces ou à une tabulation, sont fondamentaux. L'évaluation de la fonction g s'effectue encore par un simple appel de celle-ci en donnant aux arguments des valeurs effectives :

```
>>> g(1), g(2)
(0.5, 0.16666666666666666)
```

Par contre, on devra veiller aux valeurs interdites ; à défaut, l'interface renverra une erreur à l'exécution :

```
>>> g(0)
Traceback (most recent call last) :
(...)
ZeroDivisionError : division by zero
```

Remarque 11

Remarque 12

On fera attention à ne pas appeler deux fonctions par le même nom.

Exemple 13

On considère le code suivant :

```
def g(x):
    return 1/(x**2+x)
def g(x):
    return x**2
```

puis on entre dans la console `>>> g(1); type(g)`. Expliquons ce qui est retourné.

Remarque 14

On peut bien sûr faire des fonctions dépendants de plusieurs arguments :

Exemple 15

Écrivons une fonction Python `module(a,b)` qui retourne le module d'un nombre complexe $z = a + ib$ où $a, b \in \mathbb{R}$.

Remarque 16

3 Variables globales et variables locales

En utilisant Python, lorsqu'on crée des variables et des fonctions dans la console ou l'éditeur de texte, en dehors de celles créées à l'intérieur d'une fonction, les noms de celles-ci sont enregistrées dans une table qu'on appelle table **globale**.

Pendant l'exécution d'une fonction `fct`, Python crée une table **locale** des symboles où seront inscrits :

1. les noms des arguments de la fonction `fct` ;
2. les noms des variables et fonctions créées pendant l'exécution de la fonction `fct`.

Au cours de l'exécution, si Python a besoin de la valeur d'une variable `x`, il va chercher le nom `x` dans la table **locale** des symboles. Si Python ne trouve pas cette variable, il essaie deux procédures successives :

1. il cherche `x` dans la table globale ;
2. si `x` ne se trouve pas dans la table globale, il renvoie un message d'erreur.

Exemple 17

On considère le code

```
a = 3
def g(x):
    a = 0
    return x * a
```

Expliquons ce que retourne la commande dans la console `>>> g(2) ; a`

Remarque 18

4 Différence entre `print` et `return`

Dans les fonctions précédentes, on a utilisé la commande `return` pour retourner une valeur avec les fonctions précédentes. Pourtant, on a vu en TP qu'on pouvait aussi utiliser la fonction `print`. Ces deux commandes ne donnent pourtant pas exactement la même chose.

La commande `print` ne fait que montrer à l'utilisateur une chaîne de caractères qui représente ce que le programme a effectué. L'ordinateur ne peut pas utiliser ce que renvoie `print`, cette commande n'est qu'une intercation avec l'utilisateur.

La commande `return` est un vrai retour de valeur par une fonction. Cette valeur, qui peut ne pas être vue par l'utilisateur peut, en revanche, être utilisée par l'ordinateur (comme variable ou dans d'autres fonctions par exemple).

Remarque 19

Exemple 20

Considérons le code suivant :

```
def fonction_avec_print():
    print 'utilisation de print'

def fonction_avec_return():
    return 'utilisation de return'

f1 = fonction_avec_print()
f2 = fonction_avec_return()
print f1
print f2
print type(f1)
print type(f2)
```

et expliquons ce qui est retourné.

Remarque 21

Exemple 22

Écrivons une fonction Python `somme_racine(x,y)` qui renvoie $\sqrt{x}+\sqrt{y}$ en affichant d'abord les racines de x et de y avant de les sommer.

IV Interaction avec l'utilisateur

1 Commande `input`

La plupart du temps, on cherchera à construire des **programmes fonctionnels**, c'est à dire des programmes qui, à partir d'**arguments formels**, renvoient un ou plusieurs **résultats**. Ainsi, les paramètres du programme seront évalués à l'appel de la fonction.

Cependant, d'autres commandes permettent d'interroger directement l'utilisateur dans la console : la commande `input` pour obtenir une valeur au cours du programme, et dont l'affectation par défaut est du type `str`, c'est-à-dire une chaîne de caractères. Par exemple, on peut réécrire le programme de l'exemple précédent, mais en demandant à l'utilisateur de renseigner a et b directement à l'exécution du programme :

```
def module():                  # ici la fonction n'attend aucun argument
    a,b = float(input('partie réelle? ')),float(input('partie
imaginaire? '))
    return sqrt(a**2+b**2)
```

On remarquera l'utilisation de la commande `float` qui convertit la chaîne de caractère entrée par l'utilisateur en un nombre à virgule flottante. Ainsi, si on lance le programme, on obtient :

```
>>> module()
partie réelle ? 3
partie imaginaire ? 4
5.0
```

Remarque 23

De la même façon, nous avons vu que la commande `return` interrompt l'exécution du programme pour renvoyer le résultat donné. Il est aussi possible d'afficher des valeurs au cours du programme, et ceci avec la fonction `print` :

```
def module():                  # ici la fonction n'attend aucun argument
    a,b = float(input('partie reelle? ')),float(input('partie
imaginaire? '))
    print('le module de ',a,'+i',b,'est ')
    return sqrt(a**2+b**2)
```

Exemple 24

2 Documentation d'une fonction

Si les commentaires sont très pratiques pour comprendre un module en langage Python, on peut aussi **renseigner une fonction** en ajoutant sa description dans l'entête du programme : on parle du **docstring**. Par exemple, si on souhaite calculer la distance entre deux points du plan, on peut renseigner l'entête entre guillemets et on prendra l'habitude de donner un exemple :

```
from math import *

def distance(A,B):
    """Renvoie la distance entre deux points.

    Exemple
    >>> distance([3,0],[0,4])
    5.0
    """

    return sqrt((B[0]-A[0])**2+(B[1]-A[1])**2)
```

On peut alors faire appel à la fonction `help` pour retrouver toutes les informations utiles avant de l'exploiter :

```
>>> help(distance)
Help on function distance in module _main_ :

distance(A, B)
    Renvoie la distance euclidienne entre deux points.

    Exemple
    >>> distance([3,0],[0,4])
    5.0
```

```
>>> distance([1,1],[2,2])
1.4142135623730951
```

Exemple 25

Écrire une fonction Python `poly(n,x)` qui renvoie, pour un flottant x et un entier n, le nombre $1+x^n$ On écrira une documentation pour la fonction.

V Codage

La mémoire est un dispositif électronique qui sert à stocker des informations. Physiquement, ces informations sont des impulsions électriques : un système binaire à deux états. On peut donc voir la mémoire comme un dispositif qui stocke un ensemble de 0 et de 1.

Les informations sont donc **codées** dans la mémoire sous forme binaire. Nous allons voir dans la suite comment on peut stocker des entiers puis des « réels » dans la mémoire de l'ordinateur.

1 Base de numération

Dans cette partie b est un entier supérieur ou égal à 2.

Théorème 26

Pour tout $N \in \mathbb{N}$, il existe un entier naturel $n \in \mathbb{N}$ et $x_0, x_1, \ldots, x_n$ inférieur ou égaux à $b-1$ uniques tel que

$$N = x_0 b^0 + x_1 b^1 + x_2 b^2 + \cdots + x_n b^n.$$

On ne montrera pas ce théorème de façon précise mais on va donner une méthode pour obtenir n, $x_0, x_1, \ldots, x_n$. Pour préciser dans quelle base on écrit un entier, on écrira :

$$N = \overline{x_0 x_1 x_2 \cdots}^b$$

Exemple 27

Ainsi, lorsqu'on veut coder un entier écrit dans la base $b = 10$ en binaire (base $b = 2$), il faut changer de base de numération. On ne donnera pas de théorème précis mais seulement trois exemples qui donneront une méthode générale.

Exemple 28 (*Passage de la base 2 à la base 10*)

Écrivons le nombre $\overline{1101}^2$ en base 10.

Exemple 29 (*Passage de la base 10 à la base 2*)

Écrivons le nombre $\overline{242}^{10}$ en base 2.

Remarque 30

Exemple 31 (*Passage de la base 2 à la base 6*)

Écrivons le nombre $\overline{11011}^2$ en base 6.

Remarque 32

2 Codage en mémoire : limitations

La mémoire est découpée en différentes parties pour stocker les entiers. Chaque partie a une taille limitée qu'on nomme **nombre de bits** où chaque bit peut prendre la valeur 0 ou 1.

Exemple 33

Néanmoins, ce type de situation ne permet a priori pas de coder d'autre nombres que des entiers alors qu'un ordinateur doit être capable de manipuler des entiers négatifs et des nombres décimaux ou plus précisément des nombres en **virgule flottante**. C'est la façon de mettre en mémoire ce type de nombre qu'on va voir dans la suite.

Théorème 34 (*Notation scientifique en base b*)

Soit $b \in \mathbb{N}$, $b \geq 2$. Soit $x \in \mathbb{R}$ un nombre décimal (ou flottant). Il existe un unique entier $k \in \mathbb{Z}$ et, un unique entier $s \in \{-1, 1\}$ et un unique entier y tel que

$$x = s \times y \times b^k.$$

Remarque 35

Exemple 36

On a désormais un moyen de coder en mémoire des nombres décimaux. On a le schéma suivant :

1. **Signe** :

2. **Exposant :**

3. **Mantisse :**

Remarque 37 (*Limitations du stockage en mémoire des nombres décimaux*)

Exemple 38

Expliquons le fonctionnement de la fonction suivante :

```
def mystere(n):
    q = n
    conv = ''
    while q > 0:
        r = q % 2
        q = q // 2
        conv = str(r) + conv
    return conv
```