Chapitre 1 : bases de la programmation

C. Charignon

Table des matières

Première partie
Cours

1 Introduction

1.1 Fonctions calculables

Au début du XX° siècle, on s'est posé la question de savoir quelles étaient les fonctions « calculables ». Grosso modo, une fonction est dite calculable lorsqu'il est toujours possible de calculer son résultat, sans avoir besoin de deviner une astuce, autrement dit lorsqu'il existe une procédure claire permettant d'arriver à tous les coups au résultat.

Par exemple, le pgcd est une fonction calculable : vous connaissez l'algorithme d'Euclide, qui permet de le calculer sans se poser de question.

Remarque : Exponentielle n'est pas calculable au sens strict : on sait en trouver des valeurs approchées, mais pas de valeur exacte. Mais en réalité, dans la théorie des fonctions calculables, on ne peut pas vraiment prendre en compte les nombres réels : un cerveau humain ne peut pas retenir un nombre réel, car il lui faudrait connaître une infinité de décimales... Les réels sont des nombres qu'il est possible d'approcher autant que l'on veut mais qu'on ne peut pas connaître de manière exacte. C'est pourquoi la notion de fonction calculable ne parle en fait que de nombres entiers.

Notez que pour un ordinateur, une information n'est jamais rien d'autre qu'une suite finie de bits, qu'on peut considérer comme une suite de 0 et de 1, et donc comme un nombre entier écrit en base 2. On y reviendra dans un chapitre ultérieur, mais ceci explique qu'en informatique théorique on peut considérer qu'on n'utilise que des nombres entiers.

Donner une définition précise d'une fonction calculable faisait partie de la réflexion de cette époque. Les mathématiciens les plus célèbres à avoir travaillé sur ce sujet sont Alonzo Church et Alan Turing. Le point remarquable et qu'il a été prouvé que toutes les définitions qui ont été imaginées par divers mathématiciens se sont avérées équivalentes ! Une fonction calculable pour Church est aussi calculable pour Turing, et réciproquement.

Pour définir ce qu'est une fonction calculable, Alan Turing a défini une machine théorique, qu'on appelle depuis "machine de Turing", et les fonctions calculables sont les fonctions qu'il est possible de calculer à l'aide de cette machine.

Cette machine n'a jamais eu vocation à être construite concrètement, c'était juste un modèle théorique. C'est en 1945 que Jon Von Neumann [1] et ses collaborateurs proposent le premier modèle pratique d'une machine capable de calculer toutes les fonctions calculable : le premier ordinateur.

De nos jours, la quasi-totalité des ordinateurs utilisent encore l'architecture de Von Neumann. Ils sont donc capables de calculer toujours les mêmes fonctions, celles définies par Church et Turing.

1.2 Un exemple de fonction non calculable

Voici l'exemple le plus célèbre de fonction non calculable.

On a dit qu'on ne parlerait que de fonctions sur les nombres entiers. Mais n'importe quelle donnée enregistrée dans un ordinateur est une suite de 0 et de 1, qu'on peut finalement considérer comme un nombre entier écrit en base 2. Y compris un programme lui même ! L'exemple ci-dessous est un algorithme qui travaille sur les algorithmes.

Théorème 1.1. *Il n'existe pas d'algorithme permettant de vérifier si un autre algorithme s'arrête (c'est-à-dire ne "plante" pas).*

Autrement dit, si nous notons $\mathcal{A}$ l'ensemble de tous les algorithmes possibles, alors la fonction :

$$\begin{array}{rcl} \mathcal{A} & \to & \{\text{vrai}, \text{faux}\} \\ \text{un algorithme} & \mapsto & \left| \begin{array}{ll} vrai & si \text{ l'algorithme termine} \\ faux & sinon \end{array} \right. \end{array}$$

n'est pas calculable.

Démonstration :

Supposons par l'absurde qu'il existe un tel algorithme, appelons-le `termine`. Considérons alors le programme suivant :

1. Jon Von Neumann est connu pour ses travaux en informatique, mais aussi en mécanique quantique, et même en économie !

```
mystere
entrée: rien
sortie: le message "coucou"

début de l'algorithme:
1    si termine(mystere):
2        tant que 1=1:
3        fin du tant que
4    sinon:
5        renvoyer("coucou")
6    fin du si
fin de l'algorithme
```

Posons nous la question : est-ce que le programme `mystere` termine ?

- supposons que `mystere` termine. Suivons alors le fonctionnement de l'algorithme : comme il termine, `termine(mystere)` doit renvoyer `vrai`. Mais alors nous allons rentrer dans la boucle tant que de la ligne 2. Or cette boucle tant que continue de tourner tant que 1=1... Condition qui est toujours vraie ! Donc cette boucle tourne sans arrête et le programme plante ! Ceci contredit notre hypothèse comme quoi `mystere` est un programme qui termine.

- Supposons maintenant que le programme `mystere` plante. Suivons alors son exécution : `termine(mystere)` va renvoyer `faux`. On va donc passer à la ligne 5, renvoyer le message "coucou", puis atteindre sans encombre la fin de l'algorithme. Ainsi, `mystere` fonctionne parfaitement !
  Ceci contredit l'hypothèse comme quoi `mystere` est un programme qui plante !

Les deux possibilités sont toutes les deux absurdes. Ainsi notre hypothèse de début est impossible : le programme `termine` n'existe pas.

□

Remarque : Le principe de cette preuve est un classique de l'informatique, qui admet de nombreuses variantes. Une des plus connues : que pensez-vous de la phrase suivante : « Un barbier rase tous les hommes du village qui ne se rasent pas eux-même. » Qui rase le barbier ?

Ou encore : « Cette phrase est fausse ».

Remarque : Il existe un argument relativement simple (mais niveau deuxième année) pour prouver l'existence de fonctions non calculables : c'est que le nombre de fonctions est strictement plus grand que le nombre d'algorithme. En effet, un algorithme est une suite finie de symboles choisis dans un alphabet fini...

1.3 Algorithme

Rentrons dans le concret. Pour nous, une fonction sera dite calculable lorsqu'il existe une méthode pour calculer son résultat, quelle que soit la donnée d'entrée, décrite à l'aide des opérations suivantes :

- enregistrer et lire des données ;
- effectuer des opérations arithmétiques élémentaires ($+$ et $\times$) ;
- effectuer des tests ($<, =, >$), et choisir l'opération suivante à exécuter en fonction du résultat du test ;
- Répéter des opérations, tant qu'une certaine condition est réalisée.

En outre, cette méthode doit être décrite à l'aide d'un nombre *fini* des opérations ci-dessus, et doit donner le résultat en un temps *fini*.

Remarque : Concernant les opérations arithmétiques, en fait seul l'opération d'ajouter 1 (appelée *incrémenter*) est réellement nécessaire, car on peut programmer toutes les autres à partir de celle-ci. Mais de nos jours, les processeurs savent tous faire directement au moins les additions et les multiplications.

En négligeant les questions de mémoire disponible, n'importe quel appareil pouvant faire les opérations ci-dessus est capable de calculer *toutes* les fonctions calculables. Un tel appareil peut être appelé un ordinateur. Par exemple un téléphone ou une calculette programmable est tout à fait capable de calculer toutes les fonctions calculables.

Il est intéressant de noter que réciproquement, aucun ordinateur actuel n'est capable de faire mieux : aussi puissant soit-il aucun ordinateur actuel n'est capable de calculer une fonction qui ne soit pas calculable au sens de Turing, et

donc ne fait pas plus que votre téléphone, même s'il le fait plus vite, et vous affiche le résultat avec de plus jolies couleurs.

Une méthode de calcul telle que décrite ci-dessus s'appelle un *algorithme*.

Concrètement : Lorsque vous voudrez programmer un ordinateur pour qu'il calcule quelque chose, vous devrez détailler la procédure jusqu'à ce qu'il n'y ait *uniquement* des opérations élémentaires parmi celles listées ci-dessus.

Remarque : La phrase ci-dessus est un peu exagérée : en pratique, tous les langages de programmation modernes proposent un grand nombre de fonctions pré-programmées pour faire gagner du temps à l'utilisateur. Ainsi, si vous avez besoin par exemple d'une division euclidienne en Python, vous utiliserez `//`, et vous n'aurez pas à la programmer vous-même.

1.4 Langage de programmation

Nous ne communiquerons pas directement à l'ordinateur : nous traduirons nos algorithmes dans un langage appelé langage de programmation, et un programme appelé « compilateur » ou « interpréteur » se chargera de le traduire en instructions élémentaires compréhensibles par l'ordinateur. (Tâche délicate ne serait-ce que parce que les instructions élémentaires varient selon les processeurs !)

Comme dit ci-dessus, un langage de programmation, outre les opérations élémentaires vues au paragraphe précédent a déjà un certain nombre de fonctions pré-programmées pour faire gagner du temps à l'utilisateur. Il peut aussi permettre divers raccourcis syntaxiques. Ainsi, bien que tous les langages permettent de réaliser rigoureusement les mêmes programmes, par la théorie de Church-Turing, chacun va orienter l'utilisateur vers un style de programmation particulier.

En outre, certains langages sont plus pratique d'emploi que d'autres, mais au prix de performances amoindries. Ainsi le langage Python est-il considéré comme en moyenne 4 fois plus lent que C. Dans un gros projet, le coeur du programme sera souvent codé en C. En revanche, Python sera judicieux pour un projet pour lequel le temps de développement est plus important que le temps d'exécution.

Cependant rassurez-vous : lorsqu'on connaît un langage, il est beaucoup plus facile d'en étudier un nouveau.

Le langage de programmation utilisée en tronc commun sera Python, celui utilisé en option informatique Caml.

Lorsqu'on décrit un algorithme, on peut le donner dans tel ou tel langage, ou le décrire en français (on dit souvent « pseudo-code » ou « langage naturel »).

Dans la suite de ce chapitre, nous allons décrire plus précisément les fameuses « opérations élémentaires ».

1.5 Présentation de l'environnement de travail

Il est possible de taper ses programmes dans le premier éditeur de texte venu, par exemple blocnote sous windows, gedit sous linux [2]. Il est conseillé d'enregistrer le fichier avec l'extension .py pour signaler qu'il s'agit de code python.

Ensuite, il faut envoyer ce fichier à l'interpréteur [3] pour qu'il l'exécute. Pour ce, essentiellement taper `python monprogramme.py` dans un terminal.

Ceci étant, il est plus pratique d'utiliser un éditeur spécialement conçu pour taper du code Python. Il se chargera tout seul d'envoyer votre code au compilateur, et apportera de nombreuses fonctionnalités pratiques : afficher les commandes spéciales de différentes couleurs, chercher les fautes de frappe, retrouver les parenthèses correspondantes, suivre l'état des variables pendant l'exécution, etc...

De nombreux éditeurs sont disponibles, dont beaucoup sont libres. Sous linux vous pouvez utiliser par exemple geany ou emacs. Sous Windows : Spyder, IEP, pyscripter, idle, Thonny... Commun aux deux systèmes et assez complet, on trouve VsCodium.

La plupart des éditeurs que nous utiliserons ont deux fenêtres :

- Une petite, appelée *console*, utilisée pour rentrer des commandes l'une après l'autre, qui seront exécutées au fur et à mesure.
- Une grande, appelée *éditeur*, pour taper autant de lignes de code que nécessaire et n'envoyer le tout au compilateur que lorsqu'on est prêt. (Pensez à repérer le raccourci clavier car vous ferez ceci sans arrêt.)

2. En réalité, gedit est très largement supérieur à blocnote, et peut être tout à fait adapté à la programmation...
3. Python est un langage interprété et non compilé, nous en parlerons en fin d'année.

La console est utilisée pour tester une commande ou son programme. Mais c'est l'éditeur qu'on utilise pour écrire son programme.

2 Commentaires

Avant de commencer à décrire les opérations élémentaires, voici la syntaxe la plus utilisée en Python : toute suite de caractères tapés à la suite d'un « # » sera ignorée par l'ordinateur. On l'appelle un « commentaire ».

```
# Ceci est un commentaire
```

3 Fonctions

Avant de voir comment expliquer à l'ordinateur comment calculer une fonction, voyons comment définir la fonction elle-même.

De manière générale, une fonction est définie par :

- Un ensemble de départ ;
- Un ensemble d'arrivée ;
- Un moyen d'associer à chaque élément de l'ensemble de départ un élément de l'ensemble d'arrivée.

Les éléments de l'ensemble de départ seront appelés les arguments de la fonction, ceux de l'ensemble d'arrivée les valeurs renvoyées par la fonction.

3.1 En maths

En math, on rédige ainsi :

$$\textit{nom de la fonction} : \begin{array}{ccc} \textit{ensemble de départ} & \to & \textit{ensemble d'arrivée} \\ x & \mapsto & \textit{image de } x \end{array} .$$

Par exemple pour la fonction carré :

$$c : \begin{array}{ccc} \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & x^2 \end{array} .$$

Pour gagner du temps, on laisse souvent au lecteur le soin de deviner les ensembles de départ et d'arrivée. On écrit juste : $c : x \mapsto x^2$.

La lettre x utilisée ci-dessus est interchangeable avec n'importe quelle autre lettre (on dit parfois que c'est une « variable muette »). Notons également que cette variable x n'existe que dans le corps de la définition de la fonction. La rédaction suivante est fausse par exemple (bien que parfois employée au lycée...)

« *Soit* $c : x \mapsto x^2$*. Alors* $c'(x) = 2x$ *donc* c *est croissante sur* $\mathbb{R}^+$ *et décroissante sur* $\mathbb{R}^-$*.* »

Une rédaction correcte étant :

« *Soit* $c : x \mapsto x^2$*. Alors pour tout* $x \in \mathbb{R}$*,* $c'(x) = 2x$ *donc* c *est croissante sur* $\mathbb{R}^+$ *et décroissante sur* $\mathbb{R}^-$*.* »

3.2 En Python

```
def nom_de_la_fonction( arguments ):
    """ Description de la fonction (aide). """
    contenu de la fonction

    return résultat de la fonction
```

La description de la fonction est importante : sans elle l'utilisateur ne saura pas comment l'utiliser. Il faut donner les entrées et les sorties de la fonction, et s'il y a lieu, les conditions requises pour qu'elle fonctionne normalement. Par exemple pour la fonction inverse, préciser que l'argument doit être non nul.

Une fois définie, une fonction s'utilise en Python comme en maths, en mettant des parenthèses autour des arguments.

Exemple où on définit puis on utilise une fonction :

```python
def carre(x):
  """ Renvoie x**2"""
  return x*x

carre(3)
```

Le texte entre les triples guillemets est destiné à l'utilisateur. Il apparaîtra lorsqu'il appellera l'aide.

```python
def fonctionInutile():
    """ Cette fonction ne sert à rien"""
    return None

help(fonctionInutile)
```

Au passage dans cette exemple, on a une fonction qui n'a aucun argument. On met des parenthèses vide (et non pas aucune parenthèse !) Tapez ces deux lignes de code pour comparer :

```python
fonctionInutile
fonctionInutile()
```

En Python, on n'indique pas l'ensemble auquel appartiennent les arguments... Si on lui envoie un argument pour lequel la fonction n'est pas définie, il renverra un message d'erreur.

4 Variables, séquence, opérations arithmétiques

On commence par les opérations les plus facile.

Pour enregistrer une donnée, on utilise une « variable »[4]. En pratique n'importe quelle suite de lettres, de chiffres et de _ (souligné / underscore). Par exemple, en langage algorithmique :

$$toto \leftarrow 2$$

et en python :

`toto=2.`

Cette opération s'appelle *l'affectation*.

Ici, il faut se dire que l'interpréteur a choisi un emplacement libre dans la mémoire vive, a appelé cet emplacement « toto », et y a écrit le nombre 2.

À présent, à chaque fois que l'on tapera « toto », l'interpréteur remplacera immédiatement par « 2 ».

Vous pouvez ensuite changer la valeur enregistrée dans la variable en faisant une nouvelle affectation. Par exemple :

```
toto <- 3
tata <- 5
toto <- toto+tata
tata <- toto-tata
toto <- toto-tata
```

Que contiennent les variables « toto » et « tata » à l'issue de cette suite d'instruction ?

Au passage, vous l'aurez remarqué : pour enchaîner plusieurs instructions on va tout simplement à la ligne. Idem en python. Par contre, de nombreux langages de programmation (dont Caml utilisé en option) demandent de mettre un symbole, par exemple un point-virgule.

Enfin, pour les opérations arithmétiques :

- En algorithmique, on les écrit simplement comme en math
- En python, voici les principales :
  - ◇ addition : +
  - ◇ soustraction : -
  - ◇ multiplication : *
  - ◇ puissance : « `a**b` » signifie « a^b ».
  - ◇ Pour la racine carrée, le plus simple est d'utiliser la puissance 1/2 : `a**0.5`.

4. La notion de variable présente de nombreuses subtilité sur lesquelles il nous faudra revenir.

5 Tests

Voici comment on rédigera un test en pseudo-code :

```
si condition :
    à faire si la condition est remplie
sinon :
    à faire si la condition n'est pas remplie
fin

Suite du programme
```

et en Python :

```
if condition :
    #à faire si condition remplie
else:
    #à faire sinon

#à faire dans tous les cas (suite du programme)
```

Remarque : Ne pas oublier les deux points (:). Ils remplacent le « then » qu'on trouve dans d'autres langages. L'un des buts de Python est de fournir un code aussi concis que possible.

De plus, l'indentation est *indispensable*. Les instructions à faire si la condition est remplie sont celles qui sont indentées.

Comme de nombreux langages, Python dispose d'une syntaxe pour les situations où il y a plus que deux cas :

```
if condition 1:
    #à faire si condition 1 remplie
elif condition 2:
    #à faire si condition 1 non remplie et condition 2 remplie
else:
    #à faire si aucune des conditions précédentes n'est remplie

#à faire dans tous les cas (suite du programme)
```

On peut enchaîner autant de `elif` que nécessaire.
Exemple : Équations du second degré (voir le TD, **cf exercice :** 5)

La condition s'écrit généralement à l'aide de relations comme $\leq, \geq, =$ ... et des connecteurs logiques « et » et « ou ». Nous y reviendrons dans le chapitre suivant.

Élements de syntaxe Python :

- l'égalité s'écrit `==`. Et oui, puisque le `=` sert à l'affectation ! (Pas un choix très heureux, selon mon point de vue, mais bon...)
- $\leq$ s'écrit `<=`, similaire pour $\geq$
- « et » s'écrit **`and`**, « ou » s'écrit **`or`**. Ne pas oublier les parenthèses si nécessaire.

6 Boucles conditionnelles

L'intérêt principal d'un ordinateur est de pouvoir effectuer très vite un très grand nombre de tâches répétitives. Voici comment on gère la répétition.

En français :

```
tant que condition :
    à faire tant que la condition est remplie
fin
```

En Python :

```
while condition :
    à faire tant que la condition est remplie

à faire ensuite (suite du programme)
```

Ici aussi, l'indentation est indispensable : c'est elle qui permet à Python de reconnaître les instructions à répéter dans la boucle.

Soyez très soigneux pour déterminer la condition, car c'est elle qui permet à l'ordinateur de savoir quand s'arrêter ! Par exemple, que fait l'algorithme suivant ?

```
tant que 1=1 :
    afficher « bonjour »
fin
```

Et oui, il affiche des « bonjour » éternellement. Vous savez maintenant comment faire planter un ordinateur. À propos, si ceci vous arrive pendant un TP, cliquez sur le bouton « interrompre » (souvent une icône avec un carré rouge) pour arrêter l'ordinateur.

Exemple : On veut tirer au hasard des triplets pythagoriciens, c'est-à-dire trois nombres naturels a, b et c tels que $a^2 + b^2 = c^2$. Nous allons employer une méthode très naïve : tirer au hasard des nombres jusqu'à trouver un triplet qui fonctionne.

```
from numpy.random import randint
def tripletPyth(n):
    a=randint(n)
    b=randint(n)
    c=randint(n)
    while a*a + b*b != c*c:
        a=randint(n)
        b=randint(n)
        c=randint(n)
    # 'Lorsquon arrive à ce point du programme, 'cest que la condition de la boucle 'nest
        ↪ plus vérifiée, donc que a**2 + b**2 == c**2
    return a,b,c
```

Exemple : M. X va au casino avec la somme initiale de n euros. À chaque partie, son gain est un entier aléatoire entre -2 et 2 (inclus), tous les entiers étant équiprobables. Il s'arrête de jouer lorsqu'il est ruiné, ou lorsque son capital dépasse $2n$ euros.

Écrivons un programme pour simuler ceci. Le programme prendra en entrée l'entier n, et renverra une chaîne de caractère « gagné » ou « ruiné » selon l'issue du jeu.

```
def casino(n):
    """
    Entrée : n le capital initial de M. X.
    Sortie : le capital final de M. X."""
    capital = n
    while capital >0 and capital < 2*n:
        gain=rnd.randint(-2,3)
        capital += gain

   return capital
```

Pour poursuivre sur cet exemple, écrivons une fonction qui indique si M. X fini ruiné ou pas :

```
def ruiné(n):
    """ Entrée : le capital initial de M. X.
        Sortie : True si M. X est sorti ruiné du casion, et False sinon."""
    if casino(n) > 0:
        return False
```

```
    else:
        return True
```

Vous l'avez deviné : en Python, `True` signifie « Vrai », et `False` signifie « Faux ». Nous y reviendrons au chapitre suivant.

Amélioration : calculons également combien de parties M. X a joué. Pour ce faire, il suffit de rajouter une nouvelle variable `nbDeParties` qui se charge de compter le nombre de parties jouées. Une telle variable s'appelle un "compteur".

```
def casino(n):
    """ Renvoie le couple (M. X est ruiné, nombre de parties jouées)."""
    capital = n
    nbDeParties=0
    while capital >0 and capital < 2*n:
        gain=rnd.randint(-2,3)
        capital += gain
        nbDeParties+=1

   if capital==0:
       return (True, nbDeParties)
   else:
       return (False, nbDeParties)
```

N.B. Lorsqu'une fonction doit renvoyer plusieurs valeurs, on les sépare par des virgule. Sur cet exemple, on renvoie un couple formé d'un booléen et d'un entier.

Supposons maintenant que M. X se soit fixé de jouer au maximum 10 parties car sa femme l'attends pour dîner. Pour modéliser ceci, nous devons rajouter une condition dans le `while` : si le nombre de parties jouer atteint 10, M. X s'arrête.

```
def casino(n):
    """ Renvoie le couple (nombre de parties jouées, capital restant)."""
    capital = n
    nbDeParties=0
    while capital >0 and capital < 2*n and nbDePartie<10:
        gain=rnd.randint(-2,3)
        capital += gain
        nbDeParties+=1

   return ( n, capital)
```

cf exercice : 10, 11

7 Boucles inconditionnelles

7.1 Présentation de la boucle « pour »

Nous allons présenter maintenant une variante de la boucle « tant que » extrêmement utile : celle qui sert lorsqu'on sait à l'avance combien de fois on va répéter l'opération.

Commençons par un exemple : imaginons que nous voulions dire bonjour à chaque élève de classe. Mettons qu'il y ait 48 élèves dans la classe, l'opération « dire bonjour » sera donc répétée 48 fois : nous sommes dans le cas où on sait à l'avance combien de fois l'opération va être répétée.

Le plus pratique est alors d'utiliser une variable compteur, qui retiendra le numéro de l'élève où nous en sommes. A chaque fois que nous dirons bonjour à un élève, nous augmenterons le compteur de 1, et lorsqu'il atteindra 49, nous nous arrêterons. Enfin, nous commençons par l'élève numéro 1, donc au début notre compteur vaudra 1. Ceci donne :

Mais tous les langages de programmation proposent une syntaxe qui permet d'effectuer automatiquement la gestion du compteur. Autrement dit qui condense les lignes 1,2, et 4 de l'exemple ci-dessus. Il s'agit de la boucle « pour ». L'exemple ci-dessus peut être écrit à l'aide d'une boucle « pour » ainsi :

C'est beaucoup plus simple à taper et surtout : on est sûr que la boucle va s'arrêter ! En effet, avec une boucle conditionnelle une erreur ou un oubli est vite arrivé (oubli de la ligne qui augmente compteur de 1 très souvent) et le programme plante. Dans une boucle « pour », c'est l'ordinateur qui gère, il n'oubliera pas d'augmenter le compteur.

Citons à ce sujet ce théorème bien connu en informatique :

```
compteur← 1
tant que compteur < 48 :
    { Ici, compteur est le numéro du prochain élève à qui dire bonjour.
    dire bonjour à l'élève numéro 'compteur'
    augmenter compteur de 1
fin
```

```
pour compteur de 1 à 48 :
    dire bonjour à l'élève numéro 'compteur'
fin
```

Théorème 7.1. *(théorème des boucles « tant que »)*
La plupart du temps, une boucle « tant que » est fausse.

Remarque : Suite de ce théorème au chapitre « Preuves d'algorithme ».

En conclusion, oubliez l'exemple avec la boucle « tant que » et retenez le principe :

Lorsqu'on sait combien de fois il faudra répéter un calcul, on utilise une boucle « pour ».

Voici comment on rédige une boucle inconditionnelle simple en Python :

```
for compteur in range(d,a):
    à faire (a-d) fois
suite du programme
```

La fonction **range** renvoie intervalle semi-ouvert : si a et b sont deux entiers tel que $a \leq b$, alors **range(a,b)** représente pour Python l'intervalle *semi-ouvert* $[\![a, b[\![$. Ainsi, la ligne **for i in range(d, a):** se traduit en langage mathématique par « $\forall i \in [\![d, a[\![$ ».

L'intérêt de préférer les intervalles semi-ouverts est réel, et vous le comprendrez avec un peu de pratique.
Ainsi dans ce code, le corps de la boucle for est exécuté $a - d$ fois, et la variable `compteur` prendra toutes les valeurs de $[\![d, a[\![$.

Exemple : Reprenons le cas de M. X qui joue au casino. Supposons cette fois ci qu'il se fixe à l'avance le nombre de partie qu'il va jouer. On notera n ce nombre. On suppose également qu'il dispose d'assez d'argent pour éventuellement perdre les n parties.
Dans ce cas, le nombre de parties à jouer est connu à l'avance, nous pouvons donc utiliser une boucle « pour ».

```
def casino(n):
    """ Renvoie le gain total de M. X après n parties."""

  gainTotal=0
    for i in range(0,n):
        gain=rnd.randint(-2,3)
        gainTotal += gain


    return gainTotal
```

cf exercice : 10, 11

7.2 Sommes

Une utilisation fréquente et représentative d'une boucle « pour » est le calcul d'une somme.

Soit donc $(u_n)_{n\in\mathbb{N}} \in \mathbb{C}^{\mathbb{N}}$ une suite, écrivons l'algorithme qui étant donné $n \in \mathbb{N}$ calcule $\sum_{i=0}^{n} u_i$.

La méthode est d'utiliser une variable `somme` qui au départ contient 0 et à laquelle nous rajouterons au fur et à mesure tous les u_i. Nous aurons également besoin d'une variable `i` qui variera de 0 à n : cette variables sera gérée par une boucle « pour ».

Par exemple pour $\sum_{i=0}^{7} 2^i$:

```
somme ← 0
pour i de 0 à 7 :
    # somme contient Σ_{k=0}^{i-1} 2^k
    somme ← somme + 2^i
    # Maintenant, somme contient Σ_{k=0}^{i} 2^k
fin
renvoyer somme
```

En Python, écrivons une fonction prenant en entrée un entier n et renvoyant $\sum_{i=0}^{n-1} 2^i$.

Remarque : Je mets $n-1$ comme valeur d'arrivée pour rester dans l'esprit Python : la valeur d'arrivée est exclue !

```python
def sommeGeom(n):
    res=0
    for i in range(0,n):
        res = res + 2**i
    return res
```

Signalons enfin qu'il existe une syntaxe pratique pour rajouter quelque chose dans une variable : le `res= res+2**i` peut être remplacé par `res+=2**i`, que je trouve à la fois plus pratique, plus court, et plus lisible.

Bonus : version optimisée de la fonction précédente, en gardant en mémoire des puissances calculées au fur et à mesure.

Exercice : écrire une fonction `puissance` prenant en entrée un réel x et un entier positif n et renvoyant x^n.

cf exercice : 8

8 Récursivité

Il est possible d'exprimer la répétition sans utiliser de boucle, mais en relançant le programme. Reprenons l'exemple des triplets pythagoriciens. Voici pour mémoire le programme que nous avions écrit à l'aide d'une boucle conditionnelle.

```python
from numpy.random import randint
def tripletPyth(n):
    a=randint(n)
    b=randint(n)
    c=randint(n)
    while a*a + b*b != c*c:
        # On tire de nouveaux nombres a,b,c
        a=randint(n)
        b=randint(n)
        c=randint(n)
    # 'Lorsquon arrive à ce point du programme, 'cest que la condition de la boucle 'nest
        ↪ plus vérifiée, donc que a**2 + b**2 == c**2
    return a,b,c
```

Pour exprimer le fait de tirer de nouveau les nombres a, b, et c, nous allons cette fois relancer la fonction `tripletPyth` elle-même.

```python
from numpy.random import randint
def tripletPyth(n):
    a=randint(n)
    b=randint(n)
    c=randint(n)
    if a*a + b*b != c*c:
        # On tire de nouveaux nombres a,b,c
```

```
        return tripletPyth(n)
    else:
        retarn a,b,c
```

Autre exemple : calcul de puissance. Écrivons une fonction prenant un nombre x et un entier n et renvoyant x^n.

Commençons par une version avec une boucle. Comme on sait à l'avance combien d'opérations il faudra faire (en l'occurrence n), on peut utiliser une boucle for.

```
def puissance(x, n):
    """ Renvoie x**n """""
    res=1
    for i in range(n):
        res *= x
    return res
```

Passons à une version récursive. Soit $x \in \mathbb{N}$. L'idée est d'utiliser la définition par récurrence de la suite des puissances, à savoir :

$$\begin{cases} x^0 = 1 \\ \forall n \in \mathbb{N},\ x^n = x \times x^{n-1} \end{cases}.$$

Cette définition peut être traduite immédiatement en une fonction récursive :

```
def puissanceRéc(x, n):
    """ Renvoie x**n """
    if n==0:
        return 1
    else:
        return x*puissanceRéc(x, n-1)
```

De manière générale, toute relation de récurrence permettant de définir un objet mathématique donne lieu sans effort à une fonction récursive permettant de calculer cet objet.

Deuxième partie

Exercices

TP d'informatique, tronc commun
Programmation élémentaire

Dans les feuilles de TP, les étoiles (*) indiquent la difficulté. En outre, les points d'exclamation signalent les exercices qu'il est indispensable de savoir faire.

1 Affectations

Exercice 1. * Que contient la variable ?

On effectue les instructions suivantes :

```
x=2
y=3
x=x+y
y=x-y
y=x+2
```

Que contiennent les variables `x` et `y` ?

Exercice 2. * Échange de variables

1. On suppose définies deux variables `x` et `y`. Écrire une suite d'instructions permettant d'échanger le contenu de `x` et `y`.
2. Même question avec trois variables.

2 Fonctions

Exercice 3. *! Une fonction peut utiliser d'autres fonctions

Pour écrire chacune des fonctions demandées ci-dessous, hormis la première, on utilisera au moins une des fonctions précédentes.

1. Écrire la fonction `carre` qui à un flottant x associe x^2.
2. Écrire la fonction $x \mapsto x^4$. Il y a au moins deux possibilités... Quelle est la meilleure ?
3. Écrire la fonction $x \mapsto (x+1)^4$.
4. Écrire la fonction $x \mapsto 2 + 3x + 2x^2 + 4x^4$.

Exercice 4. ** Opérations sur les fonctions**

Cet exercice est prévu pour ceux qui savent déjà un peu programmer en Python. On demande d'écrire des fonctions qui manipulent des fonctions. Vous avez essentiellement deux possibilités :

- Il est possible de définir une fonction à l'intérieur d'une autre fonction, il y aura donc un `def` à l'intérieur d'un autre `def` (attention à l'indentation !). Par exemple :

```
def f(x):

  def g(x):
    return x+1

  return g(x)+g(x)**2
```

  Que calcule la fonction f ?

- Il existe aussi une manière raccourcie pour définir une fonction simple, qui utilise une syntaxe très proche du $x \mapsto f(x)$ employé en maths : c'est `lambda x: f(x)`. On peut réécrire le code précédent ainsi :

```
def f(x):
  g= lambda x: x+1
  return g(x)+g(x)**2
```

Enfin, signalons qu'une fonction est un objet comme un autre, qui peut être un argument ou une valeur renvoyée. Par exemple que fais la fonction suivante ?

```python
def mystère(f):
  g=lambda x: 2*f(x)
  return g
```

1. Écrire une fonction `sommeDeFonctions` prenant en entrée deux fonctions f et g et renvoyant la fonction $f+g$.
2. Écrire une fonction `compose` prenant en entrée deux fonctions f et g et renvoyant la composée $f \circ g$, définie par $f \circ g : x \mapsto f(g(x))$.
3. Définir la fonction `plusUn` : $x \mapsto x+1$ et la fonction carré. Puis en utilisant les fonctions précédentes, mais sans s'autoriser les mots clé `def` ni `lambda`, définir la fonction $x \mapsto (x+2)^2 + x + 1$.

3 Tests

Exercice 5. *! Trinôme du second degré

1. Écrire un algorithme prenant en entrée trois réels a, b, c et donnant en sortie les solutions de l'équation $ax^2+bx+c = 0$ d'inconnue $x \in \mathbb{R}$. Puis traduire cet algorithme en une fonction Python. On supposera dans cette question que $a \neq 0$.
2. (**) Maintenant, prendre en compte le cas où a peut être nul. Pour plus de clarté, il sera judicieux d'écrire une autre fonction `degre1` dont le but sera de résoudre les équations de degré 1. Ainsi, si $a = 0$, la fonction principale appellera `degre1` pour résoudre $bx + c = 0$, et sinon, elle appellera la fonction `degre2` de la question précédente.
3. *bonus :* Modifier le programme pour donner les éventuelles solutions complexes. Pour créer un nombre complexe sous Python : si x et y sont ses parties réelles et imaginaires, on peut taper `complex(x,y)`, ou bien `x + 1j*y` (l'expression `1j` représente le nombre imaginaire pur noté i en maths et j en physique).

4 Boucles

Exercice 6. *! Renvoyer deux entiers distincts

Écrire une fonction prenant en entrée un entier n et renvoyant deux entiers distincts de $[\![0, n[\![$ tirés au hasard uniforme.

Pour tirer un nombre au hasard dans $[\![0, n[\![$, on peut utiliser `rd.randint(0,n)` après avoir chargé la bibliothèque `numpy.random`, via `import numpy.random as rd`.

Exercice 7. *! Exemple très simple de boucle « pour »

Écrire une fonction qui prend en entrée un entier n et qui affiche n fois « bonjour ».

La commande pour afficher un message à l'écran est `print`. Donc ici, il faudra utiliser `print("bonjour")`.

Exercice 8. **! Sommes

1. Écrire une fonction prenant en entrée un entier n et calculant $\sum_{i=0}^{n-1} i$. Pour vérifier, comparer le résultat avec la formule vue en math.
2. De même, écrire une fonction calculant $\sum_{i=0}^{n-1} 2^i$.
3. De même écrire une fonction prenant en entrée un entier n et calculant la somme des entiers impairs de $[\![0, n[\![$. Pour tester si un entier i est impair, on pourra utiliser `if i % 2 == 1` (le symbole `%` indique la division euclidienne).
4. Écrire à présent une fonction prenant en entrée un entier n et renvoyant $n!$. Si vous ne l'avez pas encore vue en maths, la notation $n!$ se prononce « factorielle n » et signifie $1 \times 2 \times 3 \times .... \times n$, autrement dit $\prod_{i=1}^{n} i$.
5. De même écrire une fonction prenant en entrée un entier n et calculant $\sum_{i=0}^{n-1} \frac{1}{i!}$. Que se passe-t-il lorsque n tend vers $+\infty$?
6. (***) Combien la fonction précédente effectue-t-elle de multiplications ? Il est possible d'écrire une fonction qui calcule $\sum_{i=0}^{n-1} \frac{1}{i!}$ en faisant n multiplications… Si ce n'est pas le cas de la vôtre, améliorez-la !

Exercice 9. * Fonction générale pour calculer une somme**

Écrire une fonction prenant en entrée une fonction f, un entier $n \in \mathbb{N}$, et renvoyant $\sum_{i=0}^{n-1} f(i)$.

Exercice 10. **! Élevage de glomorphes

M. X élève des glomorphes à rayures en vue de réintroduire cette espèce dans le Béarn[5]. La première année, il en a 2. Chaque année, la population est multipliée par 6, et il en relâche 8 dans la nature.

1. Calculer le nombre de glomorphes après 6 ans. Vous pouvez utiliser Python comme une simple calculatrice ici.
2. Écrire un programme prenant un entier n en entrée et calculant le nombre de glomorphes après n année.
3. Écrire une fonction prenant en entrée un entier N et renvoyant le nombre de glomorphes de M. X l'année où la population a dépassé N.
4. Modifier la fonction précédente pour renvoyer le nombre d'années écoulées au moment où la population dépasse N glomorphes.
5. M. X se lance dans l'élevage de glomorphes à pois. Plus mignons mais moins prolifique, leur population n'est multipliée que par 4 chaque année. Il en relâche toujours 8 par an. Reprendre la question précédente.

Exercice 11. ** Marche aléatoire

Chloé la puce effectue un bond de 10cm chaque seconde. On suppose pour simplifier qu'elle se déplace uniquement sur une ligne. Ce bond peut être vers la droite ou vers la gauche de manière équiprobable.

1. Écrire une fonction prenant en entrée un entier i et renvoyant la position de la puce après i bonds.
2. Écrire une fonction qui indique au bout de combien de temps la puce a avancé de 1 mètre vers la droite.**N.B.** Cette fonction n'aura donc aucun argument.
3. Modifier cette fonction pour obtenir le nombre de bonds au bout duquel elle a avancé de 1 mètre depuis son point de départ (c'est-à-dire vers la droite ou la gauche).
4. Écrire à présent un deuxième programme qui prend en entrée un entier N, qui appelle la fonction précédente N fois et qui fais la moyenne des résultats obtenus.
5. (***) Enfin, écrire une fonction qui prend en entrée deux entiers i et N et qui calcule la distance maximale parcourue en i bonds, ce maximum étant calculé sur N tentatives.

Exercice 12. * Avec deux boucles : la suite de Syracuse**

Soit $p \in \mathbb{N}^*$. La suite de Syracuse de premier terme p est la suite u^p telle que :

$$u_0^p = p \text{ et } \forall n \in \mathbb{N}, u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}.$$

Une conjecture classique[6] affirme que quelque soit $p \in \mathbb{N}^*$, la suite u^p finit par retomber sur 1 (et à partir de là elle va boucler : 1,4,2,1,4,2,...)

On note $\text{syr}(p)$ le plus petit entier tel que $u^p_{\text{syr}(p)} = 1$, ce nombre est appelé le « temps de vol de u^p ».

Écrire une fonction prenant en entrée un entier P et calculant le maximum des temps de vols de u^p, pour $p \in [\![1, P[\![$.

Quelques indications

2 Utiliser une variable auxiliaire pour sauvegarder `x` pendant l'échange.

6 Tirer un premier nombre. Ensuite, tirer un second nombre autant de fois qu'il le faut pour qu'il soit différent du premier.

7 Vérifiez que le programme affiche « bonjour » n fois, et pas $n+1$ fois.

8
1. Comme en cours, créer une variable `res` (un « accumulateur ») appelée à contenir le résultat final.
2. Cette fois c'est un produit. Adapter la méthode vue pour les sommes.
3.
4. Pour gagner en rapidité, on peut utiliser une variable supplémentaire `Factorielle_i` qui contiendra à chaque instant $i!$.

10 1.

5. En effet, l'espèce s'y est éteinte au XVII° siècle.
6. Un moyen simple de devenir célèbre est donc de démontrer cette conjecture.

2. On ne sait pas combien de fois il faudra répéter les opérations. Utiliser une variable `nbDeGlomorphes` qui enregistre le nombre de glomorphes de M. X.
3. Maintenir une variable `anneesEcoulees` (un "compteur") qui compte le nombre d'années écoulées.
4.
5. Oui : il y un soucis...

11 1. Ici, on sait à l'avance combien de fois il faudra répéter les opérations.
2. Ici, on ne sait pas à l'avance combien de fois répéter les opérations. Utiliser une variable `nombreDeBonds` qui compte le nombre de bonds effectuées.
3.
4. Pour calculer la moyenne, il s'agit en premier lieu de calculer la somme des résultats obtenus.

12 Il y aura une boucle tant que et une boucle pour. On peut emboîter directement l'une dans l'autre dans une même fonction. Cependant, il sera plus lisible de créer une fonction auxiliaire `tempsDeVol`. Au passage, « plus lisible » signifie aussi « avec moins de risque d'erreur »...

Quelques solutions

1

3

4

1.
```python
def somme_fonctions(f, g):
    return lambda x: f(x) + g(x)

# Exemple d'utilisation :
def carré(x):
    return x*x
def cube(x):
    return x**3

h=somme_fonctions(carré, cube)
h(0)
```

2.
```python
def compose(f, g):
    return lambda x: f(g(x))
```

3.
```python
plusUn = lambda x:x+1

plusDeux = compose(plusUn, plusUn)
f = somme_fonctions( compose(carré, plusDeux), plusUn)
```

7

8 6) Pour tout $i \in \mathbb{N}$, le calcul de `factorielle(i)` nécessite i multiplications. Dès lors, pour tout $n \in \mathbb{N}$, le calcul de $\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(n-1)!}$ nécessite $0 + 1 + 2 + \dot{+}(n-1)$ c'est-à-dire $\frac{n(n-1)}{2}$ multiplications.

9
```python
def somm(f,n):
    res=0
    for i in range(0,n):
        res+=f(i)
    return res
```

Et pour tester cette fonction :

```python
def uExemple(n):
    return 2**n

somm(uExemple, 10)
```

Cet exemple renvoie $\sum_{i=0}^{n-1} 2^i$.

10

11

12