Chapitre 7. Boucle conditionnelle, représentation des nombres

7.1 Boucle conditionnelle

La boucle `while` (tant que) permet de répéter une partie de programme tant qu'une condition est vraie. Attention, cela signifie que si vous vous trompez, éventuellement votre programme va *planter*, c'est-à-dire répéter indéfiniment la boucle `while` sans jamais s'arrêter, si la condition ne devient jamais fausse. Vous pouvez alors appuyer sur Control-c pour interrompre python.

Par exemple,

```
s = 0
for x in range(10):
  s = s + x
```

peut s'écrire

```
s = 0
x = 0
while x < 10:
  s = s + x
  x = x + 1
```

Certains problèmes ne peuvent cependant pas se résoudre à l'aide d'une boucle `for` mais uniquement à l'aide d'une boucle `while`, le nombre d'itérations n'étant pas connu a priori. Considérons par exemple le problème suivant.

Exercice 7.1.1 On place 100 euros sur un compte bancaire avec 1% d'intérêts par an. Écrire une suite d'instructions calculant le nombre d'années nécessaires pour que ce placement soit au minimum doublé.

Essayez de résoudre ce problème en utilisant une boucle `for`. On constate que l'on ne sait pas à l'avance combien de tours il faut effectuer (c'est justement l'objectif du problème). Utilisez donc à la place une boucle `while`.

Exercice 7.1.2 On considère la fonction suivante :

```
def mystere(n:int):
    s = 0
    while n > 0 :
        s = s + (n % 10)
        n = n // 10
    return s
```

1. Quelle est la valeur de `mystere(2501)`. De façon générale, que calcule la fonction `mystere` ?

2. Pourquoi est-il plus pratique d'utiliser un `while` qu'un `for` pour coder la boucle de la fonction `mystere` ?

3. Écrire une fonction `nombreDeChiffres(n: int) -> int` qui retourne le nombre de chiffres contenus dans l'écriture décimale du nombre entier `n` (supposé non nul).

   Par exemple, la valeur de `nombreDeChiffres(2501)` est 4.

4. Écrire une fonction `plusGrandChiffre(n: int) -> int` qui retourne le plus grand chiffre contenu dans l'écriture décimale du nombre entier `n`.

   Par exemple, la valeur de `plusGrandChiffre(2501)` est 5.

Exercice 7.1.3 (extrait épreuve de mathématiques Bac S, 2019) Soit A un nombre réel strictement positif. On considère l'algorithme suivant :

```
i = 0
while 2**i <= A:
  i = i + 1
```

où "`**`" représente l'élévation à la puissance. On observe que la variable `i` contient la valeur 15 en fin d'exécution de cet algorithme. L'affirmation suivante est elle vraie ou fausse (justifier) :

$$2^{15} \leq A < 2^{16}$$

Exercice 7.1.4 [extrait DST 2015-16]

Dans la fonction `mystere` suivante, `a` et `n` sont deux entiers positifs supérieurs ou égaux à 0.

```
def mystere(a:int, n:int):
    i = 1
    x = a
    y = n
    while y > 0:
        if y % 2 == 1:
            i = i * x
        x = x * x
        y = y // 2
    return i
```

1. Simuler l'exécution de `mystere(a,n)` lorsque `a` est égale à `2` et `n` est égale à `8` en donnant les valeurs successives des variables `i`, `x` et `y` dans le tableau suivant :

   `mystere(2,8)`

i	
x	
y	

   Quelle est la valeur retournée par la fonction dans ce cas ?

2. Que retourne la fonction lorsque `a` et `n` prennent les valeurs suivantes ?

a	2	3
n	6	4
valeur retournée		

3. Comment interpréter la valeur retournée par la fonction `mystere` en général, pour deux valeurs `a` et `n` quelconques ?

7.2 Recherche à l'aide d'une boucle conditionnelle

Exercice 7.2.1 Pour simuler le lancer d'un dé à 6 faces marquées avec les valeurs de 1 à 6, on peut utiliser l'appel `randrange(1,7)`, après avoir ajouté à son code la ligne :

```
from random import randrange
```

En utilisant cette fonction, écrire le code des fonctions suivantes :

1. `obtenirUn6() -> int`, qui ne prend aucun paramètre, et retourne le nombre de lancés effectués avant d'obtenir le nombre 6.

2. `obtenirUnDouble() -> int`, qui retourne le nombre de lancés effectués pour obtenir deux fois consécutivement le même nombre.

7.3 Représentation des nombres

7.3.1 Entiers

Alors que les humains représentent les nombres en *décimal*, c'est-à-dire en base 10 (notamment parce que l'on a 10 doigts), les ordinateurs représentent les nombres en binaire, c'est-à-dire en base 2. Ils écrivent ainsi les nombres avec seulement les chiffres 0 ou 1, chaque chiffre est appelé bit.

Avec 1 chiffre décimal on peut écrire les nombres de 0 à 9, avec 2 chiffres décimaux on peut écrire les nombres de 0 à 99, plus généralement avec n chiffres décimaux, on peut écrire les nombres de 0 à $10^n - 1$.

De même, avec 1 bit on peut écrire les nombres de 0 à 1, avec 8 bits (un octet) on peut écrire les nombres de 0 à $2^8 - 1 = 255$, avec 32 bits on peut écrire les nombres de 0 à $2^{32} - 1 = 4\,294\,967\,295$.

python sait en fait représenter des nombres entiers de taille arbitraire, il utilise autant de bits que nécessaires pour cela [1].

Par ailleurs, pour les chaînes de caractères, python utilise un octet par lettre [2]. Pour les images, il utilise 3 octets par pixel (un pour R, un pour G, un pour B). Une image de 1000 pixels sur 1000 occupe donc a priori 3 méga-octets [3].

7.3.2 Non entiers

Lorsqu'un nombre n'est pas entier, au lycée vous avez vu la notation scientifique :

$$0,25 = 2,5 \times 10^{-1}$$
$$0,033 = 3,3 \times 10^{-2}$$
$$etc.$$

Ici, on a gardé 2 chiffres significatifs. Les ordinateurs utilisent également ce genre de notation, appelée à virgule flottante (car la position de la virgule est variable, en anglais `float`). Python utilise la précision fournie par le processeur de l'ordinateur (64 bits) ce qui permet d'obtenir environ 16 chiffres décimaux significatifs.

C'est *environ* car le processeur utilise des puissances de 2 plutôt que des puissances de 10 (car avec ses transistors il compte en base 2), et est alors obligé d'arrondir à sa propre manière.

Par exemple, voici ce qui se passe avec python lorsque l'on essaie de mettre de plus en plus de chiffres dans un nombre non entier :

1. arrondi à un multiple de 32
2. sans compter les questions de lettres accentuées, les idéogrammes chinois, etc.
3. On utilise ensuite typiquement une compression JPEG ou PNG pour réduire la taille, mais c'est toute une autre histoire !

```
>>> 0.123456789
0.123456789
>>> 0.1234567890123456
0.1234567890123456
>>> 0.12345678901234567
0.12345678901234566
>>> 0.123456789012345678
0.12345678901234568
>>> 0.1234567890123456789
0.12345678901234568
```

Le processeur est coincé, il n'a pas la place pour stocker plus d'une quinzaine de chiffres, il est obligé d'arrondir.

De toutes façons le calcul en virgule flottante ne peut pas être de précision infinie, il suffit d'essayer de calculer `1/3` :

```
>>> 1/3
0.3333333333333333
```

De nouveau le processeur arrondit.

Si l'on essaie de calculer une racine carrée :

```
>>> from math import sqrt
>>> x = sqrt(2)
>>> x
1.4142135623730951
```

$\sqrt{2}$ n'est même pas rationnel, le processeur est obligé d'arrondir. Si l'on essaie de retrouver 2 :

```
>>> x*x
2.0000000000000004
```

On ne retombe pas juste...

Un autre point surprenant est que puisque le processeur calcule en binaire, ce qui ne tombe pas sur une puissance de deux ne tombe pas "juste". Par exemple :

```
>>> 0.1+0.2
0.30000000000000004
```

Ni `0.1` ni `0.2` ne sont des puissances de deux ou des sommes de puissances de deux, et le résultat `0.3` non plus. Le processeur a dû arrondir, mais il ne pouvait pas savoir quel sens était le plus approprié, il en a choisi un dont le résultat est certes surprenant pour nous :)

Pire, si l'on compare les résultats :

```
>>> 0.1 + 0.2 == 0.3
False
```

En effet, à cause des arrondis choisis par le processeur, la partie gauche de la comparaison vaut `0.30000000000000004`, ce qui n'est effectivement pas égal à `0.3`...

Pour comparer des nombres à virgule flottante, il faut donc plutôt vérifier si la différence est considérée comme suffisamment petite :

```
>>> 0.1 + 0.2 - 0.3 < 0.00000000001
True
```

Exercice 7.3.1 TP Note : étudier la section 7.3 depuis son début jusqu'ici !

On a vu que $\frac{1}{3}$ ne peut pas être représenté exactement. Mais apparement on parvient à retomber juste quand on remultiplie par 3 [4] :

```
>>> (1/3)*3
1.0
```

Mais c'est juste de la chance ! Avec 998 cela ne passe pas :

```
>>> (1/998)*998
0.9999999999999999
```

Écrire une fonction `fraction() -> int` qui calcule le plus petit i pour lequel le calcul $(1/i) * i$ ne retombe pas juste à `1.0`.

Exercice 7.3.2 TP Écrire une fonction `chiffres() -> int` qui estime le nombre de chiffres décimaux que l'ordinateur peut effectivement stocker. Il suffit pour cela de calculer $1+(1/(10^i))$ pour i de plus en plus grand, et s'arrêter lorsque python considère en fait que c'est égal à `1.0` (à cause de l'arrondi).

Utilisez maintenant plutôt $1+(1/(2^i))$, vous obtenez 53, il se trouve que c'est effectivement le nombre de chiffres significatifs utilisés par le processeur en binaire !

7.4 Exercices de révisions et compléments

Exercice 7.4.1 (extrait DST 2014-2015) Pour tout entier $n \geq 0$, le *nombre de Cullen d'indice* n est le nombre $n \times 2^n + 1$.

Les nombres de Cullen grandissent de manière très rapide, on souhaite étudier pour quel n on atteint une certaine borne x. On va faire cela de manière assez aggressive avec une boucle `while`.

1. Écrire une fonction `cullen(n:int) -> int` prenant en paramètre un entier `n` et retournant le nombre de Cullen d'indice `n`. On rappelle que, en python, le calcul de la valeur puissance a^k s'écrit `a**k`.

2. Écrire une fonction `indiceCullenSup(x:int) -> int` prenant en paramètre un entier `x`, et retournant le plus petit entier n tel que le nombre de Cullen d'indice n soit strictement supérieur à `x`.

3. Écrire une fonction `indiceCullenInf(x:int) -> int` prenant en paramètre un entier `x`, et retournant le plus grand entier n tel que `x` soit strictement supérieur au nombre de Cullen d'indice n.

Exercice 7.4.2 Écrire une fonction `logarithme(a:int, n:int) -> int` qui retourne le plus petit entier k tel que $a^k > n$ (on supposera $a > 1$). *Note :* python sait calculer a^k (notation `a**k`), mais il existe une solution simple et (légèrement) plus efficace qui utilise seulement la multiplication.

Comment modifier cette fonction pour qu'elle calcule le plus grand entier k tel que $a^k \leq n$ (c'est la définition habituelle du logarithme *entier*) ?

4. Ce qui parait surprenant quand nous effectuons le calcul en décimal, cf https://www.pinterest.fr/pin/619456123731421333/

Primalité

Exercice 7.4.3 L'objectif est d'écrire un *test de primalité*, c'est à dire une fonction `premier(n:int) -> bool` qui retourne `True` si l'entier naturel $n > 1$ est premier et `False` sinon. Pour cela on parcourt les nombres entre 2 et $n-1$ pour chercher un diviseur d de n : dès que l'on en trouve un on arrête les calculs, n n'est pas premier.

1. Écrire la fonction `premier(n:int) -> bool` à l'aide d'une boucle `for` implémentant cet algorithme.

2. Tester cette fonction en affichant tous les nombres premiers inférieurs à 100.

3. Modifier la fonction `premier` pour transformer la boucle `for` en boucle `while`.

4. Si on ne trouve pas de diviseur $d \leq \sqrt{n}$, on est sûr que n est premier : pourquoi ? Comment effectuer le test $d \leq \sqrt{n}$ sans utiliser de fonction « racine carrée » ? (qui est très coûteuse à appeler) Modifier la fonction `premier` en conséquence.

5. Améliorer [5] `premier` en traitant à part le cas où n est pair ; dans le cas où n est impair, il suffit ensuite de chercher un diviseur d impair.

Exercice 7.4.4 On suppose dans cet exercice que l'on dispose de la fonction `premier` décrite dans l'exercice 7.4.3, que son temps de calcul est proportionnel à $\sqrt{n}$ lorsque n est premier, et qu'il est négligeable lorsque n n'est pas premier, ce qui est très souvent le cas (la plupart des entiers possèdent un petit facteur, découvert très vite lors de l'exécution de `premier(n)`). [6]

1. Écrire une fonction `premierSuivant(n:int)->int` qui calcule le plus petit nombre premier $p > n$.

2. Sachant que le calcul de `premierSuivant(10**12)` prend environ une seconde, quel est l'ordre de grandeur maximal de n pour que le calcul de `premierSuivant(n)` dure moins d'une minute ? moins d'une heure ? moins d'une journée ?

   Quelle est la durée approximative du calcul de `premierSuivant(2**40)` ? Même question pour 2^{50}.

Exercice 7.4.5 (extrait DST 2015-16)

1. Écrire une fonction `facteurImpair(n:int) -> int`, qui, étant donné un entier naturel `n` non-nul, renvoie le plus grand diviseur impair de `n`. Le résultat sera obtenu par une succession de divisions de `n` par 2 tant que `n` est pair. Par exemple, `facteurImpair(504)` retournera 63, puisque 504 est pair, 252 (504 divisé par 2) et 126 (252//2) sont pairs, et 63 (126//2) est impair.

2. Écrire une fonction `puissanceDiviseur(p:int,n:int) -> int`, qui, étant donné un entier naturel `n` non-nul, renvoie la plus grande puissance de `p` qui divise `n`.

   Par exemple, `puissanceDiviseur(2,504)` retournera `8`, puisque 504 est divisible par $8 = 2^3$, mais pas par $16 = 2^4$.

5. il existe des tests de primalité sophistiqués *radicalement* plus efficaces que le test naïf décrit dans l'exercice 7.4.3 ; ils permettent de tester en quelques secondes la primalité d'un nombre dont l'écriture décimale comporte plusieurs centaines de chiffres.

6. Par ailleurs, l'écart moyen entre deux premiers consécutifs est environ $\ln(n)$, il y aura donc un nombre de tels appels qui est négligeable devant $\sqrt{n}$.

3. On suppose avoir la fonction `premierSuivant(n:int) -> int` qui renvoie le premier nombre premier strictement supérieur à `n`. Par exemple, `premierSuivant(2)` vaut 3, `premierSuivant(3)` et `premierSuivant(4)` valent 5.

   En utilisant la fonction `premierSuivant(n)`, écrire une fonction `decompositionFacteursPremiers(n:int) -> list` qui renvoie la liste des nombres constituant la décomposition de `n` en un produit de facteurs premiers.

   Cette décomposition est obtenue en divisant `n` par 2 autant de fois que possible, en continuant de la même façon avec 3, puis avec 5, avec 7, et ainsi de suite jusqu'à ce que `n` ait la valeur 1.

   Par exemple, `decompositionFacteursPremier(504)` retournera la liste `[2,2,2,3,3,7]`, puisque $504 = 2 * 252$, $252 = 2 * 126$, $126 = 2 * 63$, et 63 est impair. Ensuite, $63 = 3 * 21$, $21 = 3 * 7$ et 7 n'est plus divisible par 3. Comme 7 n'est pas divisible par 5, 5 n'apparaît pas dans la liste. Enfin, $7 = 7*1$, il n'y a plus rien à décomposer, donc la liste est complète.

   Pour ajouter des valuers à la liste, vous pouvez utiliser `L = L + [x]`

   Attention, inutile de chercher à utiliser les fonctions des questions précédentes.

Suites

Exercice 7.4.6 Une suite de Syracuse est définie par récurrence de la façon suivante :

$$\begin{cases} u_0 & = a \\ u_{n+1} & = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{sinon.} \end{cases} \end{cases}$$

où a est un entier naturel strictement positif.

1. Calculer à la main les premiers termes de la suite pour $a = 5$ puis pour $a = 7$.

2. Écrire une fonction `syracuse(a:int, n:int) -> int` qui calcule le terme de rang `n` de cette suite lorsque le premier terme u_0 est égal à `a`. Tester cette fonction en donnant pour `a` la valeur 5, puis la valeur 7 et plusieurs valeurs pour `n`.

3. Que se passe-t-il lorsque pour une valeur de n, u_n est égal à 1 ?

4. On conjecture (consulter par exemple Wikipedia) qu'une suite de Syracuse finit toujours par atteindre la valeur 1. Écrire une fonction `longueur(a:int) -> int` qui calcule et retourne la première valeur de n telle que $u_n = 1$ lorsque le premier terme u_0 vaut a.

5. Vérifier la conjecture pour tous les entiers $a < 100$ (utilisez une boucle bien sûr, en utilisant `print` pour montrer les résultats !) Parmi ces valeurs de a, quelle est celle qui fournit une suite de longueur maximale ?

6. Utiliser la fonction *syracuse* de la question 2 pour écrire la fonction *longueur* de la question 4 est une idée naturelle. Etudier dans ce cas combien de fois on exécute le calcul :

   $$u_{n+1} = u_n/2 \quad \text{si } u_n \text{ est pair,} \quad u_{n+1} = 3u_n + 1 \quad \text{sinon,}$$

   pour calculer longueur(27). Améliorer le code de la fonction `longueur(a)` pour éviter les calculs inutiles.

7. Écrire la fonction `listeSyracuse(a)` qui calcule et retourne la *liste*

$$[u_0 = a, u_1, u_2, ..., u_n = 1]$$

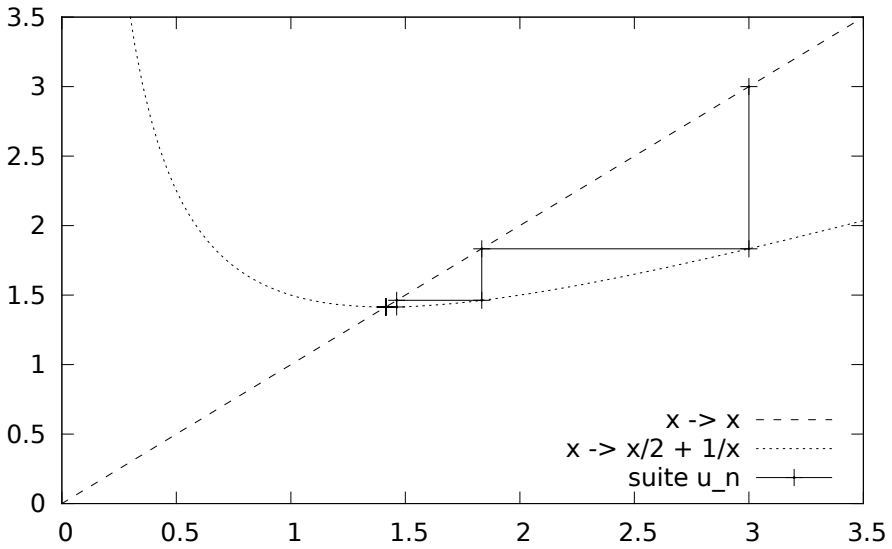
où u_n désigne le premier terme égal à 1. Par exemple, avec $a = 7$ on obtient la liste [7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1].

8. Écrire une fonction `hauteur(a:int)` qui calcule et retourne la valeur maximale de u_n lorsque le premier terme u_0 vaut a ; par exemple `hauteur(7)` vaut 52.

Exercice 7.4.7 Comment calculer $\sqrt{2}$ avec k chiffres après la virgule ? Une méthode mathématique simple est d'utiliser la suite u_n suivante :

$$\begin{cases} u_0 & = 3 \\ u_{n+1} & = \frac{u_n}{2} + \frac{1}{u_n} \end{cases}$$

On peut notamment afficher le graphe suivant pour observer la convergence.



1. Écrire une fonction `suite(n:int) -> float` qui calcule et retourne le terme de rang n de cette suite. Notes : ne vous embêtez pas à stocker toutes les valeurs ! une seule variable suffit pour garder le résultat intermédiaire.

2. Utilisez une boucle pour afficher les 10 premiers termes de la suite. Une valeur approchée de $\sqrt{2}$ est 1.414213562373095048[...]. On constate effectivement que la suite converge vers cette valeur, mais sans pouvoir toutefois l'atteindre : le nombre de chiffres des valeurs de l'ordinateur est limité !

3. Plus généralement, pour calculer $\sqrt{x}$, on peut utiliser la méthode de Newton : la suite

$$\begin{cases} u_0 & = x \\ u_{n+1} & = \frac{1}{2}(u_n + \frac{x}{u_n}) \end{cases}$$

où u_0 est une première estimation grossière de $\sqrt{x}$, on a utilisé x lui-même pour simplifier. Écrire une fonction `sqrt(x:float) -> float` qui retourne une approximation de $\sqrt{x}$ en calculant u_{10} et imprimant les valeurs intermédiaires u_n au passage. Tester avec 2, 3, 10.

4. Tester avec 1000. On constate que la convergence est lente, on n'est pas sûr que 10 itérations suffisent. Remplacer la boucle `for` par une boucle `while` pour continuer le calcul tant que la différence entre deux termes consécutifs est plus grande que 0.0000001. Tester avec 1000000.

Dessin d'une fractale

Exercice 7.4.8

$$\begin{cases} z_0 = c \\ z_{n+1} = z_n \times z_n + c \end{cases}$$

Le langage python permet de manipuler facilement les nombres complexes :

- Pour initialiser une variable complexe il suffit d'écrire : `c = complex(re,im)` où `re` désigne la partie réeelle de `c` et et `im` la partie imaginaire de `c`.
- Les opérateurs arithmétiques classiques (+, -, *, /) fonctionnent naturellement sur les complexes.
- La fonction `abs(c:complex)` permet de récupérer le module de `c`.

1. Définir une fonction `suiteComplexe(x:float, y:float, size:int, n:int) -> complex` qui retourne la valeur de n-ième terme de la suite (z_n) avec $c = (x+iy)*3/size-(2+1.5i)$.
2. Cette suite est souvent divergente, mais d'une manière irrégulière. On s'intéresse au cas où le module de z_n dépasse 2, et surtout au nombre d'itérations n nécessaire pour cela. On se limitera cependant à 256 itérations.

   En s'inspirant de la fonction `suiteComplexe`, définir avec une boucle `while` une fonction `suiteComplexeDiverge(x:float, y:float, size:int) -> int` qui retourne le plus petit entier n tel que l'on n'a plus la condition « $n < 256$ et le module de z_n est inférieur à 2 ».
3. Définir la fonction `mandelbrot(size:int)` qui retourne une image de taille $(size, size)$ telle que le niveau de gris du pixel de coordonnées (x, y) est déterminé par la valeur calculée par `suiteComplexeDiverge(x,y, size)`.

   Dessiner l'image retournée par `mandelbrot(256)`.
4. Définir la fonction `mandelbrotCouleur(size:int)` sur le même schéma que la fonction `mandelbrot(size)` mais cette fois-ci la couleur de chaque pixel est :

   $$((n*8)\,\%\,256,\ (n*32)\,\%\,256,\ (n*64)\,\%\,256)$$

   où n est la valeur retournée par la fonction `suiteComplexeDiverge(x, y, size)`.

7.5 L'essentiel du chapitre

7.5.1 Boucle conditionnelle

```
while (condition):
    instructions exécutées tant que
    condition est vraie
```

Exemple :

```
i = 1
while i < n:
    i = i * 2
```

7.5.2 Précision de calcul

En python, les nombres entiers sont représentés de manière exacte quelle que soit leur taille.

Les nombres non entiers utilisent par contre une virgule flottante, ce qui limite leur précision, en pratique à une bonne quinzaine de chiffres.