notebook_2

October 4, 2018

1 Part 2: Fonctions en Python 3.x

Une propriété des langages de programmation est que le programmeur peut créer ses propres *fonctions*. Créer une *fonction*, c'est comme enseigner à l'ordinateur une nouvelle astuce. Généralement, une fonction recevra des données en entrée (*input*), exécutera un algorithme utilisant les données d'entrée et produira des données en sortie (*output*) lorsque l'algorithme se terminera.

Dans cette partie, nous explorons les *fonctions* Python. Nous explorons également les instructions de contrôle, qui permettent à un programme de se comporter de différentes manières pour différentes entrées. Nous introduisons également la boucle *while*, une boucle dont la répétition peut être plus soigneusement contrôlée qu'une boucle *for*. En tant qu'application de ces techniques, nous implémentons, de plusieurs manières, l'algorithme d'Euclide comme une fonction Python, pour trouver efficacement le PGCD d'entiers et pour résoudre des équations Diophantiennes linéaires. Cela complète le chapitre 1 du livre An Illustrated Theory of Numbers.

1.1 Table des matières

1.2 Démarrer avec les fonctions Python

Une *fonction* dans Python est une construction qui prend des données d'entrée, effectue certaines actions et produit des données de sortie. Il est préférable de commencer par quelques exemples et de décomposer le code. Voici une fonction `square`. Exécutez le code comme d'habitude en appuyant sur * shift-Enter * lorsque le bloc de code est sélectionné.

```
In [ ]: def square(x):
            answer = x * x
            return answer
```

Lorsque vous exécutez le bloc de code, vous ne voyez probablement rien se passer. Mais vous avez effectivement appris à votre ordinateur une nouvelle astuce, augmenter le vocabulaire des commandes qu'il comprend grâce à l'interpréteur Python. Vous pouvez maintenant utiliser la commande `square` comme vous le souhaitez.

```
In [ ]: square(12)
```

```
In [ ]: square(1.5)
```

Décomposons la syntaxe de cette function, ligne par ligne.

```
def square(x):
    answer = x * x
    return answer
```

La première ligne (déclaration de la fonction) commence par le mot réservé Python `def`. (Donc, n'utilisez pas `def` comme nom de variable!). Le mot `def` "définit" une fonction appelée `square`. Après le nom de la fonction `square` viennent les parenthèses contenant **l'argument** `x`. Les *paramètres* ou *arguments* d'une fonction font référence aux données d'entrée. Même si votre fonction n'a aucun argument, vous avez besoin de parenthèses, vides. L'argument `x` est utilisé pour nommer n'importe quel nombre entré dans la fonction `square`.

À la fin de la ligne de déclaration il y a un deux-points `:` et les deux lignes suivantes sont en retrait (indentées). Comme pour les boucles for, les deux points et l'indentation signalent la portée (*scope*). Tout ce qui se trouve sur les lignes en retrait est considéré comme inclus dans la portée de la fonction et est exécuté lorsque la fonction est utilisée ultérieurement.

La deuxième ligne `answer = x * x` est le début de la portée de la fonction. On déclare une variable `answer` et on lui affecte la valeur `x * x`. Donc, si l'argument `x` vaut `12`, alors `answer` sera évalué à `144`. La variable `answer`, déclarée dans la portée de la fonction, ne sera pas accessible en dehors de la portée de la fonction. C'est une variable **locale**.

La dernière ligne `return answer` contient le mot réservé Python `return`, qui termine la fonction et renvoie la valeur de la variable `answer`. Donc, quand vous utilisez la fonction avec la commande `square(1.5)`, l'argument `x` vaut `1.5`; la variable locale `answer` vaut `2.25` et ce nombre est renvoyé en sortie.

Une fonction ne doit pas toujours renvoyer une valeur de sortie. Certaines fonctions peuvent simplement afficher (`print`) des informations. Voici une fonction qui affiche le résultat de la division Euclidienne comme une phrase avec addition et multiplication.

```
In [ ]: def display_divmod(a, b):
            quotient = a // b # Integer division
            remainder = a % b #
            print("{} = {} * {} + {}".format(a, quotient, b, remainder))
```

```
In [ ]: display_divmod(23, 5)
```

Notez que cette fonction n'a pas de ligne `return`. La fonction se termine automatiquement à la fin de sa portée. La fonction utilise également le **formatage de chaîne** de Python. Cela a changé entre Python 2.x et 3.x, et ce notebook utilise la syntaxe Python 3.x. Le formatage de chaîne vous permet d'insérer des espaces réservés tels que {} dans une chaîne et de les remplir avec des valeurs.

```
In [ ]: print("My favorite number is {}".format(17))  # The .format "method" substi
```

```
In [ ]: print("{} + {} = {}".format(13, 12, 13+12))
```

La commande `format` est un exemple de **méthode de chaîne**. Cela a pour effet de remplacer tous les espaces réservés {} par ses entrées, en séquence. Il existe une syntaxe complexe pour ces espaces réservés, qui permet de faire correspondre des espaces réservés avec des valeurs dans des ordres différents et de formater différents types de valeurs. Voici la référence officielle pour le formatage de chaîne dans Python 3.x Python 3.x. Nous n'utiliserons que les fonctionnalités les plus élémentaires, exposées ci-dessous.

```
In [ ]: print ("The number {} comes before {}.".format(1, 2)) # This should be fami
        print ("The number {1} comes before {0}.".format(1, 2)) # What happens?
        print ("The number {1} comes before {1}.".format(1, 2)) # Got it now?
```

En plaçant un nombre dans l'espace réservé, comme `{1}`, vous pouvez remplir les espaces réservés avec les valeurs dans un ordre différent ou répéter la même valeur. La méthode de formatage prend plusieurs paramètres et ils sont numérotés: paramètre 0, paramètre 1, paramètre 2, etc. L'espace réservé `{1}` sera donc remplacé par le deuxième paramètre (paramètre 1). C'est déroutant au début, mais Python commence toujours à compter à zéro.

```
In [ ]: pi = 3.14159265
        print("pi is approximately {0}".format(pi))
        print("pi is approximately {0:f}".format(pi)) # The "f" in "0:f" formats th
        print("pi is approximately {0:0.3f}".format(pi)) # Choose 3 digits of preci
```

Si vous donnez des informations sur la manière dont l'espace réservé est utilisé, la méthode de formatage formatera mieux les choses pour l'impression. L'espace réservé `{0:f}` sera remplacé par le paramètre 0, et sera formaté d'une manière agréable pour les flottants (d'où le `f`). N'essayez pas de formater des choses en dehors de leur type !

```
In [ ]: print("{:d} is a pretty big integer.".format(2**100)) # d is the formatting
        print("{:f} is an integer, formatted like a float.".format(2**100))
        print("{:f} is a float, of course.".format(1/7))
        print("{:s} is a string.".format('Hi there!')) # s is the formatting code f
        print("{:d} will give us an error message.".format(1/7))

In [ ]: from math import sqrt  # Make sure the square root function is loaded.
        print("The square root of {0:d} is about {1:f}.".format(1000, sqrt(1000)))
```

1.2.1 Exercises

1. Comment signale-t'on la portée dans une fonction Python?
2. Écrivez une fonction appelée area_circle, qui prend en argument un rayon. La fonction doit retourner l'aire du cercle, sous forme de nombre à virgule flottante. Ajoutez ensuite une ligne à la fonction, en utilisant le formatage de chaîne, de sorte qu'elle imprime en outre une phrase de la forme "La surface d'un cercle de rayon 1.0 est 3.14159." (selon le rayon et l'aire calculée).
3. `format` est un exemple de méthode. Une autre méthode intéressante est `replace`. Essayez `"Python".replace("yth", "arag")` pour voir ce que ça fait.
4. Essayez les codes de mise en forme `%` et `E` (au lieu de `f`) pour un nombre à virgule flottante. Que réalisent ces codes ?
5. Voyez-vous une raison pour laquelle une fonction n'aurait aucun argument ?

```
In [ ]: # Utilisez cet espace pour travailler sur les exercices.
        # Rappelez-vous que vous pouvez ajouter une nouvelle cellule au-dessus / au
        # (la cellule aura une barre bleue à gauche), puis appuyez sur "a" ou "b" s
```

1.3 Instructions de contrôle

Il est important pour un programme informatique de se comporter différemment selon les circonstances. Les instructions de contrôle les plus simples, `if` et son compagnon `else`, peuvent être utilisées pour demander à Python d'effectuer différentes actions en fonction de la valeur d'une variable booléenne. La fonction suivante présente la syntaxe.

```
In [ ]: def is_even(n):
            if n%2 == 0:
                print("{} is even.".format(n))
                return True
            else:
                print("{} is odd.".format(n))
                return False

In [ ]: is_even(17)

In [ ]: is_even(1000)
```

La syntaxe large de la fonction devrait être familière. Nous avons créé une fonction appelée `is_even` avec un argument appelé `n`. Le corps de la fonction utilise l'**instruction de contrôle** `if n% 2 == 0:`. Rappelons que `n%2` donne le reste après la division de `n` par `2`. Ainsi, `n%2` est `0` ou `1`, selon que `n` est pair ou impair. Par conséquent, le **booléen** `n%2 == 0` est `True` si `n` est pair, et `False` si `n` est impair.

Les deux lignes suivantes (les premières instructions `print` et`return`) sont dans la **portée** de l'instruction `if <boolean>:`, comme indiqué par les deux points et l'indentation. L'instruction `if <boolean>:` demande à l'interpréteur Python d'exécuter les instructions dans la portée si le booléen est `True`, et d'ignorer les instructions dans le portée si le booléen est `False`.

Analysons le code de ce `if`.

```
if n%2 == 0:
    print("{} is even.".format(n))
    return True
```

Si `n` est pair, alors l'interpréteur Python affichera la phrase `n is even`. Ensuite, l'interpréteur retournera (output) la valeur `True` et la fonction se terminera. Si `n` est impair, l'interpréteur Python ignorera les deux lignes de portée.

Souvent, nous ne voulons pas que Python *ne fasse rien* quand une condition n'est pas satisfaite. Dans le cas ci-dessus, nous préférerions que Python nous dise que le nombre est impair. L'instruction `else:` indique à Python ce qu'il faut faire si l'instruction de contrôle `if <boolean>:` reçoit un booléen `False`. Analysons le code

```
else:
    print ("{} est impair".format(n))
    return false
```

Les commandes `print` et `return` sont dans la portée de l'instruction `else:`. Donc, lorsque l'instruction `if` reçoit un signal `False` (le nombre `n` est impair), le programme imprime une phrase de la forme `n est impair` et retourne la valeur `False` puis sort de la fonction.

La fonction `is_even` est une sorte de fonction "bavarde". Une telle fonction est parfois utile dans un environnement interactif, où le programmeur veut comprendre tout ce qui se passe. Mais si la fonction devait être appelée un million de fois, l'écran se remplirait de phrases imprimées ! En pratique, une fonction efficace et silencieuse `is_even` pourrait ressembler à ce qui suit.

```
In [ ]: def is_even(n):
            return (n%2 == 0)
```

```
In [ ]: is_even(17)
```

Une instruction `for` et une instruction `if`, utilisées ensemble, nous permettent d'effectuer une recherche par **force brute**. Nous pouvons rechercher, un par un, les facteurs d'un nombre afin de vérifier si celui-ci est premier. Ou nous pouvons chercher des solutions, une par une, à une équation jusqu'à ce que nous en trouvions une.

Une chose à noter: la fonction ci-dessous commence par un bloc de texte entre un guillemet triple (trois guillemets simples lors de la saisie). Ce texte est appelé un **docstring** et il est destiné à documenter ce que fait la fonction. L'écriture de docstrings claires devient importante lorsque vous écrivez des programmes plus longs, ou collaborez avec d'autres programmeurs ou encore lorsque vous souhaitez revenir des mois ou des années plus tard pour réutiliser un programme. Il existe différentes conventions de style pour docstrings; par exemple, voici les conventions docstring de Google. Nous adoptons une approche moins formelle.

```
In [ ]: def is_prime(n):
            '''
            Checks whether the argument n is a prime number.
            Uses a brute force search for factors between 1 and n.
            '''
            for j in range(2,n):  # the list of numbers 2,3,...,n-1.
                if n%j == 0:  # is n divisible by j?
                    print("{} is a factor of {}.".format(j,n))
                    return False
            return True
```

Une note importante: le mot-clé `return` **termine** la fonction. Donc, dès qu'un facteur est trouvé, la fonction se termine et affiche `False`. Si aucun facteur n'est trouvé, l'exécution de la fonction survit au-delà de la boucle et la ligne `return True` est exécutée et termine la fonction.

```
In [ ]: is_prime(91)
```

```
In [ ]: is_prime(101)
```

Essayez la fonction `is_prime` sur des nombres plus grands – essayez des nombres avec 4 chiffres, 5 chiffres, 6 chiffres. Où commence-t-elle à ralentir ? Avez-vous des erreurs lorsque les nombres sont trop grands ? Assurez-vous de sauvegarder d'abord votre travail, juste au cas où votre ordinateur serait bloqué !

```
In [ ]: # Experimentez ici avec la fonction is_prime.
```

Deux facteurs limitants sont étudiés plus en détail dans la prochaine leçon. Ce sont **le temps** et **l'espace** (espace mémoire de votre ordinateur). Lorsque la boucle de `is_prime` tourne encore et encore, cela peut prendre beaucoup de temps à votre ordinateur! Si chaque étape de la boucle ne prend qu'une nanoseconde (1 milliardième de seconde), la boucle prend environ une seconde lors de l'exécution de `is_prime (1000000001)`. Si vous essayiez `is_prime` sur un nombre beaucoup plus grand, comme`is_prime (2 ** 101 - 1)`, la boucle prendrait plus de temps que la durée de vie de la Terre.

L'autre problème qui peut survenir est un problème *d'espace*. Dans Python 3.x, astucieusement `range (2, n)` évite de stocker tous les nombres entre `2` et `n-1` en mémoire. Il se souvient simplement des bornes du range et de la manière de progresser d'un indice au suivant. Dans l'ancienne version, Python 2.x, la commande `range(2, n)` aurait tenté de stocker la liste complète des numéros `[2, 3, 4, .., n-1]` dans la mémoire de votre ordinateur. Votre ordinateur possède quelques (4, 8 ou 16, peut-être) gigaoctets de mémoire (RAM). Un gigaoctet est un milliard d'octets, et un octet est suffisant pour stocker un nombre compris entre 0 et 255. (Plus de détails à ce sujet plus tard!). Un gigaoctet ne suffira même pas à stocker un milliard de nombres. Donc, notre fonction `is_prime` aurait entraîné des problèmes de mémoire dans Python 2.x, mais dans Python 3.x, nous n'avons pas à nous soucier (pour l'instant) de l'espace.

1.3.1 Exercises

1. Créez une fonction `my_abs(x)` qui affiche la valeur absolue de l'argument `x`. (Notez que Python a déjà une fonction `abs(x)` intégrée).

2. Modifiez la fonction `is_prime` pour qu'elle affiche un message `Nombre trop grand` et renvoie `None` si l'argument d'entrée est supérieur à un million. (Notez que `None` est un mot réservé Python. Vous pouvez utiliser la déclaration à une ligne `return None`.)

3. Ecrivez une fonction Python `thrarity` qui prend un argument `n`, et affiche la chaîne `threeven` si `n` est un multiple de trois, `throdd` si `n` est un multiple de trois plus un, ou `thrugly` si n est un multiple de trois moins un. Exemple: `thrarity(31)` devrait renvoyer `throdd` et `thrarity(44)` devrait renvoyer `thrugly`. Astuce: étudiez la syntaxe `if/elif` du tutoriel Python officiel

4. Ecrivez une fonction Python `sum_of_squares(n)` qui trouve et imprime une paire de nombres entiers naturels x, y tels que $x^2 + y^2 = n$. La fonction utilisera une recherche par force brute et retournera `None` s'il n'existe aucune telle paire de nombres.

```
In [ ]: #  Utilisez cet espace pour les réponses aux questions.
```

1.4 Boucles while et implémentation de l'algorithme d'Euclide

Nous avons presque tous les outils nécessaires pour implémenter l'algorithme d'Euclide. Le dernier outil dont nous aurons besoin est la **boucle while**. Nous avons déjà vu la *boucle for*, ce qui est très utile pour itérer sur un range. L'algorithme d'Euclide implique la répétition, mais il n'y a aucun moyen de savoir à l'avance combien de pas il faudra faire. La boucle while nous permet de répéter un processus tant qu'une valeur booléenne (parfois appelée drapeau ou **flag**) est vraie. L'exemple de compte à rebours suivant illustre la structure d'une boucle while.

```
In [ ]: def countdown(n):
            current_value = n
```

```
    while current_value > 0:  # The condition (current_value > 0) is checke
        print(current_value)
        current_value = current_value - 1
```

```
In [ ]: countdown(10)
```

La syntaxe de la boucle while commence par `while <boolean>:` et les lignes indentées suivantes constituent la portée de la boucle. Si le booléen est `True`, la portée de la boucle est exécutée. Si le booléen est à nouveau True après, la portée de la boucle est à nouveau exécutée. Encore et encore et ainsi de suite.

Cela peut être un **processus dangereux** ! Par exemple, que se passerait-il si vous faisiez une petite faute de frappe et que la dernière ligne de la boucle while se lisait `current_value = current_value + 1` ? Les nombres augmenteraient et augmenteraient ... et la valeur booléenne `current_value > 0` serait **toujours** `True`. Par conséquent, la boucle ne se terminerait jamais. Des nombres de plus en plus grands défileraient sur votre écran d'ordinateur.

Vous pourriez paniquer dans de telles circonstances et éteindre votre ordinateur pour arrêter la boucle. Voici quelques conseils pour savoir si votre ordinateur est bloqué par une boucle qui ne s'arrête jamais:

1. Sauvegardez souvent votre travail. Lorsque vous programmez, assurez-vous que tout le reste est enregistré au cas où.
2. Sauvegardez votre travail de programmation (utilisez "Save and checkpoint" dans le menu "Fichier"), surtout avant de lancer une cellule avec une boucle pour la première fois.
3. Si vous restez bloqué dans une boucle sans fin, cliquez sur "Kernel ... Interrupt". Cela va souvent arrêter la boucle et vous permettre de reprendre là où vous étiez resté.
4. Vous pouvez essayer un "Force Quit" du processus Python, à l'aide du gestionnaire d'activité.
5. Enfin, le plus sûr, pensez à inclure systématiquement une condition de sécurité - par exemple, le nombre d'itérations doit être limité à 1000 - dans chaque boucle.

Maintenant, si vous vous sentez courageux, sauvegardez votre travail, changez la boucle while pour qu'elle ne se termine jamais et essayez de récupérer là où vous l'avez laissé. Mais sachez que cela pourrait provoquer un blocage ou un comportement erratique de votre ordinateur, un crash de votre navigateur, etc. Ne paniquez pas ; cela ne brisera pas votre ordinateur de manière permanente.

La boucle sans fin cause deux problèmes ici. L'un est avec votre processeur d'ordinateur, qui tournera essentiellement ses roues. Ceci est appelé busy waiting, et votre ordinateur sera essentiellement occupé à attendre pour toujours. L'autre problème est que votre boucle imprime de plus en plus de lignes de texte dans le bloc-notes. Cela peut facilement bloquer votre navigateur Web, qui essaie de stocker et d'afficher des milliards de lignes de chiffres. Alors soyez prêt pour les problèmes!

1.4.1 L'algorithme d'Euclide avec une boucle while

L'**algorithme d'Euclide** est un processus répété de division avec reste. En commençant par deux entiers `a` (dividende) et `b` (diviseur), on calcule le quotient `q` et le reste `r` pour exprimer `a = qb + r`. Ensuite `b` devient le dividende et `r` devient le diviseur, et on répète. Enfin, le **dernier reste non nul** est le plus grand diviseur commun (PGCD) de `a` et `b`.

Nous implémentons quelques variantes de l'algorithme d'Euclide. La première est une version "bavarde", pour montrer à l'utilisateur ce qui se passe à chaque étape. Nous utilisons une boucle while pour faire la répétition.

```
In [ ]: def Euclidean_algorithm(a,b):
            dividend = a
            divisor = b
            while divisor != 0:    # Recall that != means "is not equal to".
                quotient = dividend // divisor
                remainder = dividend % divisor
                print("{} = {} ({}) + {}".format(dividend, quotient, divisor, remai
                dividend = divisor
                divisor = remainder

In [ ]: Euclidean_algorithm(133, 58)

In [ ]: Euclidean_algorithm(1312331323, 58123123)
```

C'est très bien si l'on veut connaître toutes les étapes de l'algorithme d'Euclide. Si nous voulons simplement connaître le PGCD de deux nombres, nous pouvons être moins bavard. Nous retournons le dernier reste non nul après la fin de la boucle while. Ce dernier reste non nul devient le diviseur lorsque le reste devient zéro, puis il deviendra le dividende dans la ligne suivante (non imprimée). C'est pourquoi nous retournons la (valeur absolue) du dividende après la fin de la boucle. Vous pouvez insérer une ligne à la fin de la boucle, comme un dividende imprimé, un diviseur, le reste pour vous aider à suivre les variables.

```
In [ ]: def GCD(a,b):
            dividend = a # The first dividend is a.
            divisor = b # The first divisor is b.
            while divisor != 0:    # Recall that != means "not equal to".
                quotient = dividend // divisor
                remainder = dividend % divisor
                dividend = divisor
                divisor = remainder
            return abs(dividend)   #  abs() is used, since we like our GCDs to be po
```

Notez que la déclaration `return dividend` se produit *après* la portée de la boucle while. Donc, dès que *divisor* est égal à zéro, la fonction `GCD` retourne `dividend` et se termine.

```
In [ ]: GCD(111,27)

In [ ]: GCD(111,-27)
```

Nous pouvons affiner notre code de plusieurs manières. Tout d'abord, notez que la variable `quotient` n'est jamais utilisée! C'était bien dans la version bavarde de l'algorithme d'Euclide, mais ne joue aucun rôle dans la recherche du PGCD. Notre code amélioré se lit

```
def GCD(a,b):
    dividend = a
```

```
    divisor = b
    while divisor != 0:    # Recall that != means "not equal to".
        remainder = dividend % divisor
        dividend = divisor
        divisor = remainder
    return abs(dividend)
```

Maintenant, il existe deux astuces Python que nous pouvons utiliser pour raccourcir le code. La première est appelée **affectation multiple**. Il est possible de définir les valeurs de deux variables sur une seule ligne de code, avec une syntaxe comme ci-dessous.

```
In [ ]: x, y = 2, 3  # Sets x to 2 and y to 3.
```

Ceci est particulièrement utile pour les attributions autoréférentielles, car comme pour les affectations ordinaires, le côté droit est évalué en premier, puis lié aux variables du côté gauche. Par exemple, après la ligne ci-dessus, essayez la ligne ci-dessous. Utilisez des instructions print pour voir quelles sont les valeurs des variables après !

```
In [ ]: x, y = y, x #  Guess what this does!
```

```
In [ ]: print("x =", x) # One could use "x = {}".format(x) too.
        print("y =", y)
```

Maintenant, nous pouvons utiliser l'affectation multiple pour transformer trois lignes de code en une seule ligne de code. Car la variable `remainder` n'est utilisée que temporairement avant que sa valeur ne soit donnée à la variable `divisor`. En utilisant l'affectation multiple, les trois lignes

```
    remainder = dividend % divisor
    dividend = divisor
    divisor = remainder
```

peuvent être écrites en une seule ligne,

```
    dividend, divisor = divisor, dividend % divisor # Evaluations on the right occu
```

Voici notre nouvelle fonction GCD raccourcie

```
def GCD(a,b):
    dividend = a
    divisor = b
    while divisor != 0:    # Recall that != means "not equal to".
        dividend, divisor = divisor, dividend % divisor
    return abs(dividend)
```

L'astuce suivante implique la boucle while. La syntaxe habituelle a la forme `while <boolean>:` Mais si `while` est suivi d'un type numérique, par ex. while `while <int>:`, alors la boucle while s'exécutera tant que le nombre est différent de zéro! Par conséquent, la ligne

```
while divisor != 0:
```

peut être remplacée par

```
while divisor:
```

C'est seulement un truc. Cela ne va probablement rien accélérer, et cela ne rend pas votre programme plus facile à lire pour les débutants. Alors, utilisez-le si vous préférez communiquer avec des programmeurs Python expérimentés! Voici la fonction complète

```
def GCD(a,b):
    dividend = a
    divisor = b
    while divisor:    # Executes the scope if divisor is nonzero.
        dividend, divisor = divisor, dividend % divisor
    return abs(dividend)
```

L'amélioration suivante est un peu plus dangereuse pour les débutants, mais ça marche ici. En général, il peut être dangereux d'opérer directement sur les arguments d'une fonction. Mais dans ce contexte, c'est sûr et ça ne fait aucune différence pour l'interpréteur Python. Au lieu de créer de nouvelles variables appelées `dividend` and `divisor`, on peut manipuler `a` et `b` directement dans la fonction. Si vous faites cela, la fonction GCD peut être raccourcie comme suit.

```
In [ ]: def GCD(a,b):
            while b:    # Recall that != means "not equal to".
                a, b = b, a % b
            return abs(a)
```

```
In [ ]: # Essayez GCD sur de grands nombres et voyez comme ça tourne vite!
```

Ce code est essentiellement optimal, si l'on souhaite exécuter l'algorithme euclidien pour trouver le GCD de deux entiers. Il correspond presque au code GCD dans une bibliothèque Python standard. Celui-ci peut être légèrement plus rapide que notre code, mais il existe un compromis entre la vitesse d'exécution et la lisibilité du code. Dans cette leçon et les suivantes, nous optimisons suffisamment pour les besoins courants, mais pas trop afin de garder une bonne lisibilité.

1.4.2 Exercises

1. Modifiez la fonction is_prime en utilisant une boucle while au lieu de `for j in range(2,n):`. En quoi cela pourrait-il être une amélioration par rapport à la boucle for?
2. Modifiez la fonction `Euclidean_algorithm` pour créer une fonction qui renvoie le *nombre d'étapes* nécessaires à l'algorithme d'Euclide, c'est-à-dire le nombre de divisions avec reste.
3. Créez une fonction qui effectue la division avec un reste minimal. En d'autres termes, étant donné les entiers a, b, la fonction exprime a = qb + r, où r est un entier *positif ou négatif* inférieur à b/2. Utilisez une telle fonction pour créer un nouvel algorithme d'Euclide qui utilise le reste minimal.
4. Pour vous, quelle fonction `GCD(a, b)` offre le meilleur compris entre efficacité et lisibilité ?
5. Écrivez une fonction `LCM` (plus petit multiple commun) en utilisant la fonction `GCD`.

```
In [ ]: # Répondre ici aux exercices
```

1.5 L'algorithme d'Euclide étendu

L'algorithme d'Euclide étendu fournit, outre le PGCD d de deux nombres entiers a, b, les **coefficients de Bézout** x, y tels que $ax + by = d$.

1.5.1 Challenge

1. En vous aidant de l'article wikipedia ci-dessus (voir aussi la version en anglais), écrivez une fonction `EEA` (extended euclidean algorithm) qui prend en arguments deux entiers a, b et qui renvoie d, x, y, où d est le PGCD de a, b et x, y les coefficients de Bézout. Testez votre fonction sur différents exemples.

2. En quoi la fonction `EEA` peut-elle servir à résoudre l'équation Diophantienne linéaire ? Résoudre les équations Diophantiennes linéaires $102x + 45y = 3$ et $72x + 100y = 17$.

3. Ecrire une fonction `LDE` (linear Diophantine equation) qui prend en arguments des entiers a, b, c et renvoie un couple solution x, y de l'équation $ax + by = c$, s'il existe une solution, et qui renvoie `None` s'il n'existe pas de solution. En outre la fonction `LDE` devra afficher un message décrivant soigneusement toutes les solutions, lorsqu'elles existent, ou un message indiquant qu'il n'existe pas de solution, et pour quelle raison, lorsque c'est le cas.

```
In [ ]: # Répondre ici au challenge.
```