Notes de cours
Semaine 2

Cours Turing

1 Les fonctions en Python

Les fonctions sont un outil essentiel en programmation. Cet outil permet de structurer les programmes et permet de réutiliser du code. Dans cette première partie, nous allons nous intéresser aux fonctions en Python : comment les appeler et comment les définir.

1.1 Appel de fonction

La semaine passée déjà, nous avions vu comment appeler des fonctions. Pour faire un appel de fonction, on faisait suivre le nom de la fonction à appeler par une paire de parenthèses. Entre les parenthèses sont spécifiés, dans l'ordre, les différents *arguments* de la fonction. Chaque argument est une expression.

```
print(3 * 4, 6 * 7)
```

Lors de l'évaluation d'un appel de fonction, la valeur de l'expression qui indique quelle fonction appeler est calculée en premier. La plupart du temps, cette expression est simplement le nom de la fonction (comme `print` dans le code ci-dessus), mais il se peut que la fonction à appeler soit le résultat d'une expression plus complexe, comme nous le montrerons par un exemple dans quelques paragraphes.

Une fois que l'expression qui indique quelle fonction appeler a été évaluée, les arguments sont calculés, un à un et de gauche à droite. Une fois toutes ces expressions évaluées, l'appel à la fonction est finalement effectué. Lors de cet appel, les instructions spécifiées par la fonction sont exécutées. Une fois ces instructions effectuées, l'appel de fonction est à son tour réduit à une valeur, qui dépend de la valeur éventuellement retournée lors de l'exécution des instructions de la fonction. Nous verrons sous peu comment spécifier quelles instructions doivent être effectuées lors d'un appel de fonction et comment donner une valeur de retour à la fonction.

Expression complexe pour spécifier la fonction à appeler Plus haut, nous avons mentionné le fait que l'expression qui détermine la fonction à appeler dans un appel de fonction pouvait être parfois plus complexe qu'un simple nom. Ci-dessous est un exemple d'un appel de fonction où la fonction à appeler est elle-même le résultat d'un calcul.

```
random.choice([print, input])("Entrez du texte (si possible) : ")
```

Ci-dessus est un appel de fonction. La fonction à appeler est elle-même le résultat d'un autre appel de fonction (en l'occurence un appel à la fonction `random.choice`). Dans ce cas précis, la fonction calculée par l'expression `random.choice([print, input])` sera soit `print` soit `input`, au hasard. C'est cette fonction, obtenue par un calcul, qui sera appelée ensuite avec `"Entrez du texte (si possible) : "` comme argument.

1.2 Définition de fonction

Dans la section précédente, nous avons vu comment appeler des fonctions et les règles qui régissent l'ordre d'évaluation lors d'un appel de fonction. Dans cette section, nous allons examiner comment définir des fonctions.

1.2.1 Syntaxe

Pour définir une fonction en Python, on utilise le mot-clé `def`. Sur la même ligne que ce mot-clé, on inscrit le nom de la fonction suivit de ses *paramètres* entre parenthèses. Finalement, on termine la ligne par un symbole : (deux points). En-dessous de cette ligne et indentées d'un cran sur la droite se trouvent les instructions de la fonction. On appelle ces instructions le *corps* de la fonction.

```
def dire_bonjour(nom, extra_poli):
    if extra_poli:
        print("Bonjour", nom, ".")
        print("J'espère que vous passez une belle journée.")
    else:
        print("Salut", nom, "!")
```

Remarque Formellement, une définition de fonction est une instruction. Une définition de fonction peut se situer à n'importe quel endroit où une instruction est attendue. Il est donc possible de définir des fonctions au sein de boucles ou d'autres fonctions, par exemple, bien que par défaut on préfèrera définir les fonctions au niveau principal du programme (le niveau sans indentation) si possible.

1.2.2 Exécution de la définition

Lorsqu'une instruction de définition de fonction est exécutée, la fonction est enregistrée en mémoire sous le nom spécifié. Il s'agit exactement du même procédé que lors d'une *assignation de variable*. Le nom de la fonction joue le rôle de nom de variable. La valeur stockée dans la variable est la *fonction elle-même*. En Python, les fonctions sont des valeurs.

Étant donnée la définition de la fonction `dire_bonjour` plus haut, l'instruction suivante permet d'afficher le type de la valeur stockée dans la variable `dire_bonjour`. En l'occurence, il s'agira du type *function*.

```
print(type(dire_bonjour))  # Affiche <class 'function'>
```

Comme les fonctions sont des valeurs en Python, est possible de les manipuler autrement qu'en les appelant directement. Il est par exemple possible de passer des fonctions comme arguments à d'autres fonctions, ou même de retourner des fonctions depuis d'autres fonctions.

1.3 Arguments, paramètres et exécution d'un appel de fonction

Lorsqu'un appel de fonction est évalué, le corps de la fonction est exécuté. Avant cela, une valeur est assignée à chaque paramètre de la fonction. La valeur assignée à un paramètre est la valeur de l'argument correspondant. Lors d'appels différents, les paramètres peuvent se voir assigner des valeurs différentes.

```
dire_bonjour("Albert", False)
dire_bonjour("Beatrice", True)
```

Dans l'exemple ci-dessus, deux appels sont faits à la fonction `dire_bonjour`. Dans le cas du premier appel, le paramètre `nom` se verra attribué la valeur `"Albert"` et `extra_poli` la valeur `False` avant l'exécution du corps de la fonction. Dans le cas du second appel, le paramètre `nom` aura comme valeur `"Beatrice"` et `extra_poli` la valeur `True` avant la nouvelle exécution du corps de la fonction.

1.4 Valeur de retour

Dans le corps d'une fonction, il est possible d'utiliser l'instruction `return` pour indiquer la valeur de retour d'une fonction. Après le mot clé `return` est indiquée une expression. Le résultat de l'évaluation de cette expression est utilisé comme valeur de retour de la fonction. La valeur de retour est la valeur qui est utilisée comme résultat de l'évaluation de l'appel de la fonction.

```
def additionner(a, b):
    return a + b
```

L'instruction `return`, lorsqu'elle est exécutée, met fin immédiatement à l'exécution du corps de la fonction. Toutes les instructions suivantes du corps de la fonction sont ignorées. Par exemple, dans le code ci-dessous, si une valeur autre que `"1234"` est donnée en argument à `verifier_mot_de_passe`, alors aucun affichage ne sera effectué car l'instruction `return` met fin à l'exécution du programme.

```
def verifier_mot_de_passe(mot):
    if mot != "1234":
        return False
    print("Mot de passe correct !")
    return True

# A pour valeur False, n'affiche rien.
verifier_mot_de_passe("suisse")

# A pour valeur True, affiche un message.
verifier_mot_de_passe("1234")
```

Notez au passage qu'il est possible d'utiliser plusieurs instructions `return` au sein d'une même fonction. La première à être exécutée donne la valeur de retour de la fonction et met fin immédiatement à l'exécution du corps de la fonction.

1.5 Portée des variables

Les paramètres d'une fonction, tout comme les variables assignées dans le corps de la fonction, sont visibles uniquement à l'intérieur de la fonction. On dit qu'elles ont une *portée locale*. Ces variables ne sont visible qu'au sein de la fonction et non à l'extérieur du corps de la fonction.

```
def verifier_mot_de_passe(mot):
    if mot != "1234":
        return False
    print("Mot de passe correct !")
    return True

verifier_mot_de_passe("hello")

# La variable mot n'est pas visible ici.
# L'instruction suivante donnera lieu à une erreur.
print(mot)
```

Toute variable du même nom que la variable locale se trouvant dans le contexte de la définition de la fonction sera cachée par cette variable locale. Les deux variables, bien qu'ayant le même nom, ne sont pas autrement liées. La modification d'une variable n'affectera pas l'autre variable. Voyez par exemple le code suivant qui met en avant ce phénomène.

```
x = 1
y = 1

def foo(x):
    # On ne peut pas lire le x et le y du contexte global ici.
    # Les deux variables sont cachées.
    y = 2

foo(2)

print(x)  # Affiche 1, le x du contexte global n'est pas modifié.
print(y)  # Affiche 1, le y du contexte global n'est pas modifié.
```

Si une variable est uniquement lue et jamais assignée dans le corps d'une fonction, alors la variable ne sera pas considérée comme locale mais sera à la place prise hors du contexte de la fonction, comme dans l'exemple suivant.

```
x = 1

def foo():
    print(x)

foo()  # Affiche 1

x = 2
foo()  # Affiche 2
```

Les mots-clés `global` et `nonlocal` permettent, au sein d'une fonction, de contourner les règles exprimées plus haut et de quand même modifier une variable définie hors du corps de la fonction. Le mot-clé `global` s'utilise pour accéder à une variable définie dans le contexte global, comme dans l'exemple ci-dessous.

```
x = 1

def foo():
    global x
    x += 1

print(x)  # Affiche 1
foo()
print(x)  # Affiche 2
foo()
print(x)  # Affiche 3
```

Le mot-clé `nonlocal` est plus technique. Il permet de modifier, au sein d'une fonction, une variable définie au sein d'une autre fonction dont le corps contiendrait la définition de la première fonction. Voici un exemple.

```
def foo():
    x = 1
    def bar():
        nonlocal x
        print(x)
        x += 1
    return bar

f1 = foo()
f1()  # Affiche 1
f1()  # Affiche 2

f2 = foo()
f2()  # Affiche 1
f2()  # Affiche 2
f2()  # Affiche 3

f1()  # Affiche 3
```

1.6 Pile d'appels

Il est possible de faire des appels de fonction dans le corps d'une fonction. Lors de l'évaluation d'un appel de fonction, il est donc possible qu'un autre appel de fonction soit évalué. Cet autre appel de fonction peut donner lieu lui aussi à d'autres appels, et ainsi de suite, comme dans l'exemple ci-dessous.

```
def foo(x):
    return bar(x) + 1

def bar(x)
    return baz(x) * 4

def baz(x):
    return 1 / x
```

Pour garder en mémoire les appels de fonctions débutés mais pas encore terminés, Python utilise ce qu'on appelle une *pile d'appels* (*call stack*, parfois simplement *stack*, en anglais) lors de l'exécution. Cette information est nécessaire afin que Python puisse reprendre là où il en était après l'évaluation d'un appel de fonction. Cette pile d'appels est affichée quand Python rencontre une erreur à l'exécution. On parle de *stack trace*. Cette information permet de reconstruire les différents appels de fonction en cours au moment de l'erreur. Ci-dessous est l'erreur obtenue lorsqu'on appelle la fonction `foo` définie plus haut avec la valeur `0` comme argument.

```
Traceback (most recent call last):
  File ".../exemple-stack-trace.py", line 10, in <module>
    foo(0)
  File ".../exemple-stack-trace.py", line 2, in foo
    return bar(x) + 1
           ^^^^^^
  File ".../exemple-stack-trace.py", line 5, in bar
    return baz(x) * 4
           ^^^^^^
  File ".../exemple-stack-trace.py", line 8, in baz
    return 1 / x
           ~~^~~
ZeroDivisionError: division by zero
```

Dans le message d'erreur ci-dessus, il est possible de retracer les différents appels effectués mais non résolus au moment de l'apparition de l'erreur (une division par 0).

1.7 Fonctions récursives

On appelle une fonction qui, du corps de la fonction, opère des appels à la fonction elle-même. Ces appels sont appelés des *appels récursifs*. La fonction `fact` ci-dessous est un exemple de fonction récursive.

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)
```

Dans le corps de la fonction se trouve un appel à la fonction elle-même. Ce genre d'appels est tout à fait autorisé en Python. Par exemple, l'appel suivant affichera bien la valeur de 100!.

```
print(fact(100))
```

Cependant, il y a quelques limitations à connaître. Chaque appel récursif se rajoute sur la pile, ce qui peut amener la pile à grandir rapidement et à prendre beaucoup de place en mémoire. Pour éviter cela, Python limite la profondeur maximale de récursion. Lorsque cette limite est dépassée, une erreur de type `RecursionError` est lancée par Python. Ainsi, avec la définition donnée plus haut de `fact`, un appel à `fact(10000)` donnera lieu au message d'erreur suivant :

```
Traceback (most recent call last):
  File ".../recursion-error.py", line 6, in <module>
    fact(10000)
  File ".../recursion-error.py", line 4, in fact
    return n * fact(n - 1)
               ~~~~~~~~~~
  File ".../recursion-error.py", line 4, in fact
    return n * fact(n - 1)
               ~~~~~~~~~~
  File ".../recursion-error.py", line 4, in fact
    return n * fact(n - 1)
               ~~~~~~~~~~
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Ainsi, forcément, à un moment donné, la fonction devra s'exécuter sans faire d'appels récursifs, sans quoi la fonction sera condamnée à ne pas s'exécuter correctement. Dans le cas de la fonction `fact` définie plus tôt, lorsque le paramètre `n` prend la valeur `0`, aucun appel récursif n'est effectué.

Notez que lorsqu'un nombre négatif est donné en argument à `fact`, une erreur de type `RecursionError` sera lancée par Python. En effet, le cas où `n` prend la valeur `0` n'est jamais atteint et les appels récursifs s'enchaînent jusqu'à atteindre la limite fixée par Python.

2 Récursivité

La récursivité est une technique de conception de programmes et de résolution de problèmes. L'idée est d'utiliser au sein d'une procédure de résolution d'un problème la procédure elle-même sur une ou plusieurs sous-parties du problème. Prenons l'exemple de la fonction récursive `fact` vue plus haut.

```
def fact(n):
    if n == 0:
        return 1  # Cas de base
    return n * fact(n - 1)  # Cas récursif
```

Pour calculer la valeur de $n!$, on distingue deux cas :

- Le cas où $n = 0$. Dans ce cas, on donne immédiatement la réponse et on ne fait pas d'appels récursifs. On parle du *cas de base*.
- Le cas où $n \neq 0$. Dans ce cas, on fait un appel récursif avec $n - 1$ comme argument, ce qui donne comme résultat la valeur pour $(n-1)!$. Ce résultat est ensuite utilisé pour construire le résultat de l'appel courant. Dans l'exemple présent, il suffit de le multiplier par n pour obtenir $n! = n \cdot (n-1)!$. On parle ici d'un *cas récursif*.

À noter que l'on part du principe dans ce raisonnement que l'appel récursif donne le bon résultat et termine sans erreur, ce qui ici est le cas uniquement si $n > 0$. Dans le cas où $n < 0$, alors l'appel récursif donnera lieu à une erreur, comme discuté plus haut.

Lorsque l'on utilise la technique de la récursivité pour concevoir des programmes, on se doit d'identifier des cas de bases, généralement des cas simples où la solution est immédiate, et des cas récursifs, où le problème peut se résoudre en combinants des solutions à des sous-parties du problème. La technique fonctionne si en réduisant de manière répétée le problème en ses sous-parties l'on tombe forcément sur des cas de base.

Considérons maintenant deux exemples plus en détail afin de mettre en lumière cette méthode de conception de programmes.

2.1 Étude de cas : Plus petit/plus grand

Prenons comme premier exemple approfondi la conception d'un programme qui doit deviner un nombre entre deux bornes a et b. À son tour, le programme doit faire une suggestion de nombre, à laquelle une réponse `"<"` ou `">="` sera donnée selon si le nombre à deviner est plus petit ou respectivement plus grand ou égal à la suggestion faite par le programme. Pour rappel, il s'agissait du dernier exercice de la série de la semaine passée.

Ce problème peut être résolu de manière récursive. Pour cela, on commence à identifier le cas de base, une *instance* du problème dont la solution est immédiate, triviale. Pour cet exemple, le cas où les deux bornes a et b sont égales sera notre cas de base. En effet, dans ce cas la solution est très simple : il y a une seule possibilité de réponse, la valeur de a (et aussi donc de b).

Dans les autres cas de figure, on essaie de résoudre le problème à partir d'une solution à une instance du problème plus simple. Cette solution s'obtient par un appel récursif. Dans ce cas précis, on commence par faire une suggestion m située entre les deux bornes a et b. Pour une plus grande efficacité, on utilisera $m = \lceil \frac{a+b}{2} \rceil$, où $\lceil x \rceil$ désigne la partie entiŕe supérieure

de x. Suivant la réponse donnée à cette suggestion, il faudra soit deviner un nombre entre a et $m-1$, soit deviner un nombre entre m et b. Dans les deux cas, il s'agit d'une instance du même problème, plus simple que l'instance de base. Pour résoudre ces sous-problèmes, on utilise simplement des appels récursifs. En terme de code, la fonction ainsi conçue prend la forme suivante :

```
def devine_nombre(a, b):
    if a == b:  # Cas de base.
        print("!", a)
    else:  # Cas récursif.
        m = (a + b + 1) // 2
        print(m)
        if input() == "<":
            devine_nombre(a, m - 1)  # Appel récursif.
        else:
            devine_nombre(m, b)  # Appel récursif.
```

La fonction ci-dessus est une fonction récursive qui résout le problème donné plus haut. Les suggestions sont faites par le biais de `print` et les réponses obtenues via `input`. À noter qu'il est aussi possible d'adapter cette implémentation afin de ne plus faire usage de fonction récursives, mais de boucles. Vous pouvez vous référer à la solution du dernier exercice de la semaine passée pour voir la version avec une boucle `while`.

2.2 Étude de cas : Jeu de Nim

Pour ce second exemple approfondi, considérons ensemble un jeu de la famille des *jeux de Nim*. Il s'agit d'un jeu où deux joueurs s'affrontent autour d'un tas d'allumettes. A tour de rôle, chaque joueur retire entre 1 à 3 allumettes. Le joueur qui retire la dernière allumette est déclaré gagnant.

Abordons ensemble le problème de jouer de manière optimale à ce jeu et concevons un programme qui, étant donné le nombre n d'allumettes restantes en jeu, devra nous indiquer s'il est possible de gagner à tous les coups, et dans ce cas, combien d'allumettes retirer. Pour cela, tentons d'utiliser la récursivité.

Commençons par identifier le cas de base. Lorsqu'il reste au plus trois allumettes, la solution est évidente. Il suffit de retirer toutes les allumettes en jeu pour gagner ! Le cas de base de notre approche récursive est donc très simple.

Lorsqu'il y a strictement plus de trois allumettes, la situation se complique. Il s'agira de notre cas récursif. Dans ce cas, il faudra choisir entre retirer 1, 2 ou 3 allumettes. Pour chacune de ces possibilités, il est intéressant de considérer la situation dans laquelle l'autre joueur se retrouverait. Si le joueur adverse se retrouve dans une situation où il n'a pas de coup gagnant à jouer suite au retrait de k allumettes, alors retirer k allumettes est un coup gagnant pour le joueur courant ! Si aucune possibilité de retrait d'allumettes ne place l'adversaire dans une situation où il ne peut pas gagner, alors la situation est considérée comme perdante. En effet, dans ce cas, quoi que l'on retire, l'adversaire pourra ensuite s'assurer la victoire en jouant de manière optimale.

Comment donc savoir si l'adversaire aura un coup gagnant ou non une fois un certain nombre d'allumettes retirées ? Via un appel récursif ! Considérez l'implémentation récursive suivante qui adopte cette approche.

```
def joue_nim(n):
    if n <= 3:  # Cas de base.
        return n
    else:  # Cas récursif.
        for i in range(1, 4):
            valeur_adverse = joue_nim(n - i)  # Appel récursif.
            if valeur_adverse is None:
                return i
        return None
```

Observons les résultats retournés par la fonction pour les nombres de 1 à 12.

```
1 -> 1
2 -> 2
3 -> 3
4 -> None
5 -> 1
6 -> 2
7 -> 3
8 -> None
9 -> 1
10 -> 2
11 -> 3
12 -> None
```

Pour 1 à 3 allumettes, le solution est évidente : Il faut retirer toutes les allumettes. Pour 4 allumettes, la situation est perdante : quoi que l'on retire, l'adversaire pourra ensuite s'assurer la victoire. Pour 5, 6 ou 7 allumettes, il suffit de retirer suffisamment d'allumettes afin que l'adversaire se retrouve ensuite avec 4 allumettes, une situation perdante. Pour 8 allumettes, quoi que l'on retire, l'adversaire sera dans une situation gagnante. Il n'y a donc pas de meilleur coup à jouer. Et ainsi de suite.

Question Au vu de ces résultats, pouvez-vous penser à une implémentation non récursive de cette fonction `joue_nim` ?

2.3 Récursivité et complexité temporelle

Comme pour tout programme, il est parfois intéressant de s'intéresser à la complexité temporelle d'un programme récursif. Déterminer la complexité temporelle d'un programme récursif peut s'avérer complexe. Dans cette section, nous allons simplement étudier la complexité temporelle d'un cas particulier de programme récursif : le programme pour deviner un nombre entre deux bornes abordé plus tôt dans ce cours. Le programme est reproduit ici :

```
def devine_nombre(a, b):
    if a == b:  # Cas de base.
        print("!", a)
    else:  # Cas récursif.
        m = (a + b + 1) // 2
        print(m)
        if input() == "<":
            devine_nombre(a, m - 1)  # Appel récursif.
        else:
            devine_nombre(m, b)  # Appel récursif.
```

Pour cet exemple, nous allons essayer de compter le nombre de propositions de nombres que fait le programme avant d'arriver au résultat en fonction du *nombre de candidats restants*, c'est-à-dire du nombre de solutions encore possibles. En effet, plus les bornes sont éloignées, plus il y aura de candidats possibles et plus il faudra d'étapes pour deviner le nombre [1].

Au vu du choix de la valeur `m` dans le cas récursif, le nombre de candidats diminue (à peu près) de moitié avec chaque appel récursif. On parle techniquement d'une recherche *dichotomique* (littéralement, « qui coupe en deux »).

Pour être précis, le nombre de candidats diminue juste « à peu près » de moitié car dans le cas d'un nombre impair de candidats ($n = 2k + 1$), l'appel récursif sera effectué soit sur k, soit sur $k + 1$ éléments, et $k + 1$ est un petit peu plus que la moitié de $2k + 1$. Quand le nombre de candidat est pair, ce problème ne se présente pas. Pour un nombre pair de candidats, le nombre de candidats diminue exactement de moitié lors de l'appel récursif, que la réponse soit `"<"` ou `">="`. Étudions donc dans un premier temps ce qui se passe lorsque le nombre de candidats donnés en entrée peut être divisé en deux parts égales à répétition, c'est-à-dire quand le nombre de candidats est *une puissance de 2*.

Ci-dessous sont présentés le nombre de propositions faites (à droite) en fonction du nombre de candidats (à gauche).

```
1 -> 0
2 -> 1
4 -> 2
8 -> 3
16 -> 4
32 -> 5
64 -> 6
```

Il est intéressant de constater que lorsque le nombre de candidats en entrée est de la forme 2^n, le nombre de propositions de nombres effectuées avant d'afficher la réponse est exactement n. En effet, avec une seule suggestion on diminue de moitié le nombre de candidats, et ce à chaque étape. Si l'on note le nombre de candidats $k = 2^n$, alors le nombre de propositions faites est de $\log_2(k) = n$. La fonction $\log_2$ représente le *logarithme* en base 2. Cette fonction permet de calculer l'inverse de 2 à la puissance un nombre. Cela représente en quelque sorte le nombre de fois qu'il est possible de diviser un nombre par 2 avant de retomber sur la valeur 1.

1. La preuve est laissée au lecteur.

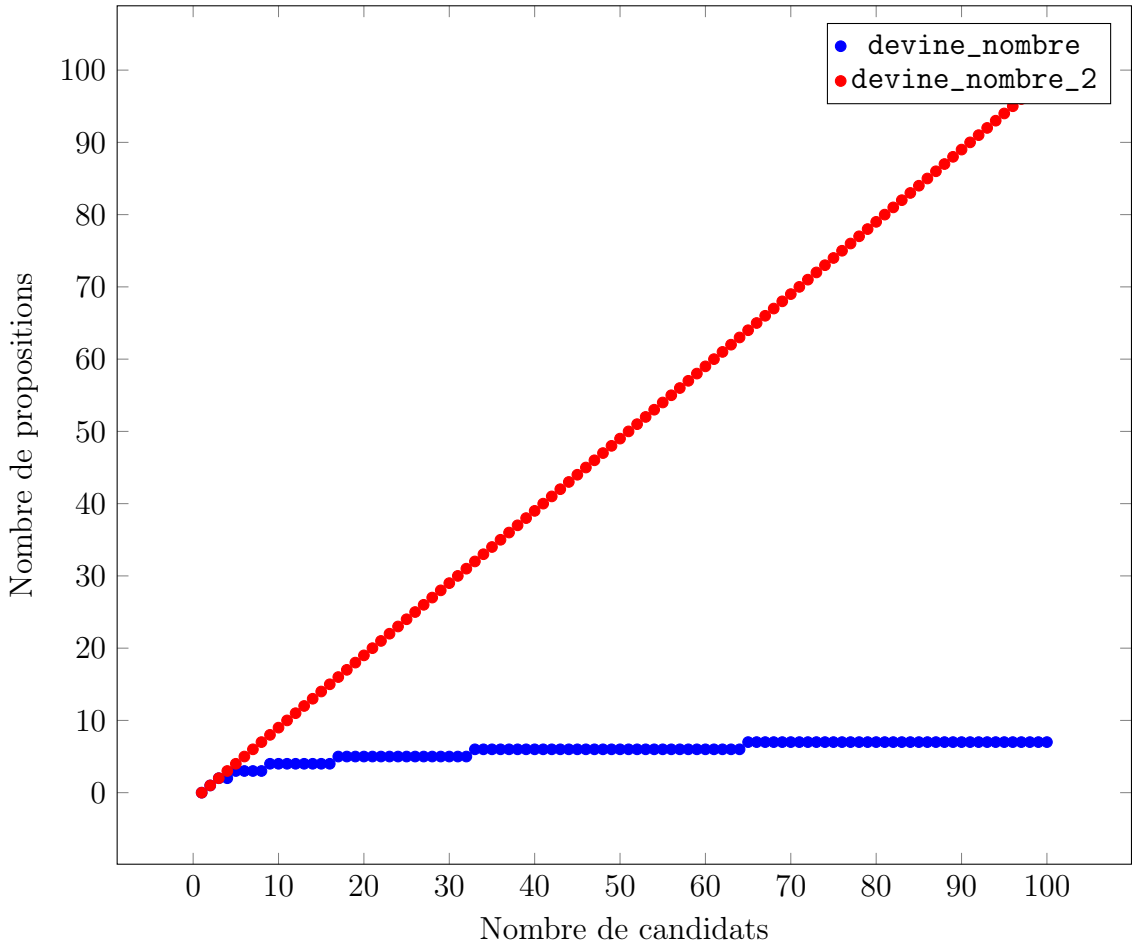
Pour les nombres n qui ne sont pas des puissances de deux, le nombre exact de propositions est plus compliqué à calculer et dépend aussi du nombre à trouver. Par contre, le nombre de propositions est facile à borner. Tout nombre entier positif qui n'est pas une puissance de 2 est compris entre deux puissances de 2 consécutives. Notons ces deux puissances de deux 2^d et 2^{d+1}, ainsi nous avons $2^d < n < 2^{d+1}$. Comme discuté plus tôt, on s'attend à ce que le nombre de propositions faites ne diminue pas lorsque le nombre de candidats augmente. C'est pourquoi on peut situer le nombre de propositions faites pour n entre le nombre de propositions faites pour 2^d et 2^{d+1}. Or, le nombre de propositions faites pour 2^d est de $d = \log_2(2^d)$ et pour 2^{d+1} de $d + 1 = \log_2(2^{d+1})$.

Par les propriétés du logarithme, on peut situer le nombre de propositions faites pour n candidats entre $\lfloor \log_2(n) \rfloor$ et $\lceil \log_2(n) \rceil$, et ce peu importe la valeur du nombre à deviner. On dit que l'approche a une complexité *logarithmique*.

Toutes les fonctions récursives ne présentent pas la même complexité temporelle ! Par exemple, considérons ensemble le programme suivant, qui permet aussi de deviner un nombre entre deux bornes de manière récursive mais qui adopte une stratégie très différente.

```
def devine_nombre_2(a, b):
    if a == b:  # Cas de base.
        print("!", a)
    else:
        m = a + 1
        print(m)
        if input() == "<":  # Cas de base.
            print("!", a)
        else:  # Cas récursif.
            devine_nombre_2(a + 1, b)  # Appel récursif.
```

L'implémentation ci-dessus est une autre solution au jeu *plus petit/plus grand*. Cependant, la complexité temporelle de cette fonction est différente de celle, logarithmique, vue plus tôt. Étant données les bornes a et b, la fonction propose un à un les nombres entre $a+1$ et b jusqu'à obtenir la réponse `"<"`, auquel cas le nombre a été deviné et est affiché. Au pire des cas, le nombre à deviner est égal à b. Dans ce cas, $b - a$ propositions de réponses seront faites. Si on note n le nombre de candidats possibles, $a - b$ correspond exactement à $n - 1$. On parle dans ce cas de complexité *linéaire*. Notez que le nombre d'appels au pire des cas augmente bien plus vite que dans le cas de l'autre implémentation, comme on peut le constater dans le graphique suivant.



Nous aurons l'occasion dans la suite du cours (et même dans le cadre de la série d'exercices du jour), de rencontrer des fonctions récursives avec des complexités temporelles très différentes.