Introduction à la Programmation Python

Le langage [2/3]

- Opérateurs
- Structures de contrôle
- Fonctions
- Exceptions & gestionnaires de contexte
- Compréhensions de listes & expressions génératrices
- Modules
- Bonnes pratiques
- Python 3.x vs 2.x
- Le Zen de Python

Formation permanente du CNRS, Délégation Alsace
Février - Mars 2017

Auteurs :

- Vincent Legoll (vincent.legoll@iphc.cnrs.fr)
- Matthieu Boileau (matthieu.boileau@math.unistra.fr)

Opérateurs

Arithmétiques

`+, -, *, /, //, %, **`

Les classiques se comportent “normalement”, pas besoin d’entrer dans les détails.

Particularités de la division :

```
In [1]: # Avec des nombres entiers
        print(16 / 3)  # Quotient de la division euclidienne (produit un réel)
        print(16 // 3) # Quotient de la division euclidienne (produit un entier)
        print(16 % 3)  # Reste de la division euclidienne (produit un entier)

        # Avec des nombres flottants
        print(16. / 3)  # Division (produit un réel)
        print(16. // 3) # Quotient de la division (produit un réel)
        print(16. % 3)  # Reste de la division ou modulo (produit un réel)
```

```
5.333333333333333
5
1
5.333333333333333
5.0
1.0
```

Puissance :

```
In [2]: print(2 ** 10)
        # On peut aussi utiliser la fonction pow() du module math, mais celui-ci re
        import math
        print(math.pow(2, 10))
```

```
1024
1024.0
```

Logiques (retournent une valeur booléenne)

`and, or, not`

```
In [3]: print(True or False)
        print(True and False)
        print(not True)
        print(not False)
        print(not [])
        print(not (1, 2, 3))
```

```
True
False
False
True
True
False
```

Attention, ce sont des opérateurs “court-circuit” :

```
In [4]: a = True
        b = False and a  # b vaut False sans que a soit évalué
        c = True or a    # c vaut True, sans que a soit évalué
```

Pour s’en convaincre :

```
In [5]: True or print("nicht a kurz schluss")
        False and print("not a short circuit")
        print('on a prouvé que ce sont des opérateurs "court-circuit"...')
```

```
on a prouvé que ce sont des opérateurs "court-circuit"...
```

Exercice : Modifiez les valeurs True et False dans la cellule précédente, pour visualiser le fonctionnement de ces opérateurs.

Comparaison

`==, is, !=, is not, >, >=, <, <=`

L’évaluation de ces opérateurs retourne une valeur booléenne.

```
In [6]: print(2 == 2)
        print(2 != 2)
        print(2 == 2.0)
        print(type(2) is int)
```

```
True
False
True
True
```

On peut utiliser ces opérateurs avec des variables et des appels à des fonctions.

```
In [7]: x = 3
        print(1 > x)
        y = [0, 1, 42, 0]
        print(x <= max(y))
        print(x <= min(y))
```

```
False
True
False
```

On peut chaîner ces opérateurs, mais ils fonctionnent en mode “court-circuit” et l’opérande central n’est évaluée qu’une seule fois.

```
In [8]: x = 3
        print(2 < x <= 9) # équivalent à 2 < x and x <= 9

True
```

Attention : comparer des types non numériques peut avoir des résultats surprenants.

```
In [9]: # Chaînes de caractères
        print("aaa" < "abc")
        print("aaa" < "aaaa")
        print("22" > "3.0")

True
True
False
```

```
In [10]: # Listes
         print([1, 2, 3, 4] > [42, 42])
         print([666] > [42, 42])

False
True
```

Attention : comparer des types incompatibles peut avoir des résultats surprenants.

```
In [11]: # Cette cellule génère des erreurs
         print('chaîne:\t', "a" < 2)
         print('liste:\t', ["zogzog"] > 42)
         print('vide:\t', [] > 1)
         print('tuple:\t', [23, 24] >= (23, 24))
         print('dict:\t', [23, 24] >= {23: True, 24: "c'est pas faux"})


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-11-3b2bc39839bf> in <module>()
           1 # Cette cellule génère des erreurs
```

```
----> 2 print('chaîne:\t', "a" < 2)
      3 print('liste:\t', ["zogzog"] > 42)
      4 print('vide:\t', [] > 1)
      5 print('tuple:\t', [23, 24] >= (23, 24))


TypeError: unorderable types: str() < int()
```

Attention, l'égalité de valeur n'implique pas forcément que l'identité des objets comparés est la même.

```
In [12]: a = []
         b = []
         c = a

         print(a == b) # comparaison de valeur
         print(a is b) # test d'identité

True
False
```

Mais des variables différentes peuvent référencer le même objet.

```
In [13]: print(a == c) # comparaison de valeur
         print(a is c) # test d'identité

True
True
```

bits à bits *(bitwise)*

`|, ^, &, <<, >>, ~`

Ces opérateurs permettent de manipuler individuellement les bits d'un entier.

Ce sont des opérations bas-niveau, souvent utilisées pour piloter directement du matériel, pour implémenter des protocoles de communication binaires (par exemple réseau ou disque).

L'utilisation d'entiers comme ensemble de bits permet des encodages de données très compacts, un booléen (True, False) ne prendrait qu'un bit en mémoire, c'est a dire que l'on peut encoder 64 booléens dans un entier.

Description complète ici.

```
In [14]: val = 67 # == 64 + 2 + 1 == 2**6 + 2**1 + 2**0 == 0b1000011
         print(bin(val))

         mask = 1 << 0 # On veut récupérer le 1er bit
         print('le 1er  bit vaut', (val & mask) >> 0)
```

```
        mask = 1 << 1 # On veut récupérer le 2ème bit
        print('le 2ème bit vaut', (val & mask) >> 1)

        mask = 1 << 2 # On veut récupérer le 3ème bit
        print('le 3ème bit vaut', (val & mask) >> 2)

        mask = 1 << 6 # On veut récupérer le 7ème bit
        print('le 7ème bit vaut', (val & mask) >> 6)

        # Si on positionne le 4ème bit a 1 (on rajoute 2**3 = 8)
        newval = val | (1 << 3)
        print(newval)

        # Si on positionne le 6ème bit a 0 (on soustrait 2**7 = 64)
        print(newval & ~(1 << 6))
```

```
0b1000011
le 1er  bit vaut 1
le 2ème bit vaut 1
le 3ème bit vaut 0
le 7ème bit vaut 1
75
11
```

Exercice : Retournez une chaîne de caractères représentant le nombre contenu dans `var` écrit en notation binaire. Par exemple:

5 -> '101' 6 -> '110' 7 -> '111'

```
In [15]: var = 7
         # votre code ici
```

Solution : exos/num2bin.py

Assignation augmentée

```
+= -= /=
```

```
In [16]: a = 4
         a += 1  # <=> a = a + 1
         print(a)
         a //= 2
         print(a)
         a **= 3
         print(a)
         a %= 2
         print(a)
```

```
5
2
8
0
```

Compatibilité de type, coercition de type

Python effectue certaines conversions implicites, quand cela ne perd pas d'information (par ex. de entier court vers entier long).

```
In [17]: print(1 + 0b1)
         print(1 + 1.0)
         print(1.0 + 2 + 0b11 + 4j)

2
2.0
(6+4j)
```

Mais dans d'autres cas, la conversion doit être explicite.

```
In [18]: # Cette cellule génère une erreur
         print(1 + '1')


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-18-662f0778b45b> in <module>()
          1 # Cette cellule génère une erreur
    ----> 2 print(1 + '1')


        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Exercice :

Sans enlever le '+':

1. Corrigez le code de la cellule ci dessus, afin d'afficher la chaine 11
2. Corrigez le code de la cellule ci dessus, afin d'afficher le nombre 2
3. Corrigez le code de la cellule ci dessus, afin d'afficher le nombre 11

Solution : exos/type_conv.py

Priorité des opérateurs

Les opérateurs ont en python des priorités classiques.

Par exemple, dans l'ordre:

- puissance : $**$
- multiplication, division : $*$ et $/$
- addition, soustraction : $+$ et $-$

etc...

Utilisez des parenthèses quand cela aide a la lisibilité et à la clarté.

Une priorité explicitée avec des parenthèses est souvent plus facile à relire que s'il n'y en a pas et qu'il faut se remémorer les règles.

Pour plus d'informations, voir ici

Structures de contrôle

- La mise en page comme syntaxe
- `pass`
- tests conditionels : `if/elif/else`
- boucles
  - `for elt in liste`
  - `for idx in range(len(liste))`
  - `while`
  - `break`
  - `continue`

La mise en page comme syntaxe

- La mise en page est importante en python, c'est une différence majeure avec les autres langages (Java, C++, etc.)
- Python utilise l'indentation du code avec des caractères blancs (TAB ou ESPACE) plutôt que des mots clés (begin/end en pascal) ou des symboles (`{}` en java et C++). Cela permet de rendre le code plus compact.
- Elle va servir a délimiter des blocs de code sur lesquels les structures de contrôle comme les boucles ou les tests de conditions vont s'appliquer.
- De toute façon, dans les autres langages, on indente aussi le code pour l'aspect visuel et la lisibilité.
- L'indentation faisant partie de la syntaxe du langage, il faut y préter une grande attention, et être rigoureux quant au mélange de caractères blancs TAB et ESPACE. Car cela peut conduire à des erreurs à l'exécution voire des comportements erratiques.
- Dans un programme python, la durée de vie d'une variable est celle du bloc qui contient sa première assignation.

```
In [19]: if True:
             print("toutes")
             print("les")
             print("lignes")
             print("au même niveau d'indentation forment un bloc de code")
         print('et quand on remonte, on "termine" un bloc de code')
```

```
toutes
les
lignes
au même niveau d'indentation forment un bloc de code
et quand on remonte, on "termine" un bloc de code
```

Exercice : changez le True en False, et observez quelles lignes de code ne sont plus exécutées.

Pass

En cas de besoin d'un bloc de code qui ne fait rien, on utilise le mot clé `pass` (équivalent à NOP ou NO-OP)

Exemple : une boucle infinie

```
condition = True
while condition:
    pass
```

Tests conditionnels

Les instructions `if/elif/else` permettent d'exécuter des blocs d'instructions en fonction de conditions :

```
if <test1>:
    <bloc d'instructions 1>
[elif <test2>:
    <bloc d'instructions 2>]
[else:
    <bloc d'instructions 3>]
```

Pour les connaisseurs `elif` est similaire au `switch` en C et C++...

```
In [20]: if True:
             print("c'est vrai!")

c'est vrai!
```

```
In [21]: if False:
             print("je suis caché!")
         else:
             print("mais moi je suis en pleine lumière...")

mais moi je suis en pleine lumière...
```

```
In [22]: # Pour cet exemple, on itère sur les éléments d'un tuple (cf. boucle for p
         for position in 2, 9, 3, 1, 8:
             if position == 1:
                 print(position, "Or")
             elif position == 2:
                 print(position, "Argent")
             elif position == 3:
                 print(position, "Bronze")
             else:
                 print(position, "Vestiaires")
```

```
2 Argent
9 Vestiaires
3 Bronze
1 Or
8 Vestiaires
```

```
In [23]: taille = 1.90
         if taille >= 1.70: # La taille moyenne en France
             print('grand')
         else:
             print('petit')
```

```
grand
```

Exercices :

1. Editez la cellule pour y mettre votre taille et exécutez-la pour savoir si vous êtes grand ou petit.
2. Gérez le cas des gens de taille moyenne.
3. Utilisez la fonction input() pour demander sa taille à l'utilisateur.

Solution : exos/taille.py

Boucles

Les boucles sont les structures de contrôle permettant de répéter l'exécution d'un bloc de code plusieurs fois.

Boucle `while`

La plus simple est la boucle de type `while` :

```
while <condition>:
    <bloc d'instructions 1>
<bloc d'instructions 2>
```

Tant que `<condition>` est True, le `<bloc d'instructions 1>` est exécuté, quand la condition passe à False, l'exécution continue au `<bloc d'instructions 2>`.

```
In [24]: compteur = 3
         while compteur > 0:
             print('le compteur vaut :', compteur)
             compteur = compteur - 1
         print('le compteur a été décrémenté 3 fois et vaut maintenant', compteur)
```

```
le compteur vaut : 3
le compteur vaut : 2
le compteur vaut : 1
le compteur a été décrémenté 3 fois et vaut maintenant 0
```

Exercice :

1. Ecrivez une boucle `while` qui décompte les secondes pour la soirée du réveillon.
2. Allez voir par ici pour passer le temps...

```
In [25]: # Votre code ici
```

Solution : exos/decompte.py

Boucle `for`

Une boucle plus complexe : `for/in`

```
for <variable> in <iterable>:
    <bloc d'instructions 1>
<bloc d'instructions 2>
```

A chaque tour de boucle, la variable `<variable>` va référencer un des éléménts de l'`<iterable>`. La boucle s'arrête quand tous les éléménts de l'itérable ont été traités. Il est fortement déconseillé de modifier l'itérable en question dans le `<bloc d'instructions 1>`.

```
In [26]: invites = ('Aline', 'Bernard', 'Céline', 'Dédé')
         for personne in invites:
             print('Bonjour %s, bienvenue à la soirée de gala !' % personne)
         print('Maintenant tout le monde à été bien accueilli...')
```

```
Bonjour Aline, bienvenue à la soirée de gala !
Bonjour Bernard, bienvenue à la soirée de gala !
Bonjour Céline, bienvenue à la soirée de gala !
Bonjour Dédé, bienvenue à la soirée de gala !
Maintenant tout le monde à été bien accueilli...
```

Exercice : Rajoutez des invités à la fête. Vérifiez que tout le monde est accueilli correctement.

```
In [27]: # Maintenant, si nous avons reçu des réponses à notre invitation et stocke
         invites = {'Aline': True, 'Bernard': False, 'Céline': True, 'Dédé': True}
         for (personne, presence) in invites.items():
             if presence:
                 print('Bonjour %s, bienvenue à la soirée de gala !' % personne)
             else:
                 print('Malheureusement, %s ne sera pas avec nous ce soir.' % perso
         print('Maintenant tout le monde à été bien accueilli ou excusé...')
```

```
Bonjour Aline, bienvenue à la soirée de gala !
Bonjour Dédé, bienvenue à la soirée de gala !
Bonjour Céline, bienvenue à la soirée de gala !
Malheureusement, Bernard ne sera pas avec nous ce soir.
Maintenant tout le monde à été bien accueilli ou excusé...
```

Exercice : Rajoutez des invités à la fête. Certains ayant répondu qu'ils ne pourraient pas venir. Vérifiez que tout le monde est accueilli ou excusé correctement.

Si nous voulons itérer sur une liste mais avons besoin des indices liés a chaque élément, nous utilisons une combinaison de `range()` et `len()`.

```
In [28]: nombres = [2, 4, 8, 6, 8, 1, 0, 1j]
         for idx in range(len(nombres)):
             nombres[idx] **= 2
         # Les carrés
         print(nombres)
```

```
[4, 16, 64, 36, 64, 1, 0, (-1+0j)]
```

Il existe une forme raccourcie pour faire ce genre de choses, la fonction interne `enumerate()`

```
In [29]: nombres = [2, 4, 8, 6, 8, 1, 0]
         for (idx, item) in enumerate(nombres):
             nombres[idx] = bool(item % 2)
         # Les impairs
         print(nombres)

[False, False, False, False, False, True, False]
```

Note :

La fonction interne `range()` retourne un itérateur. C'est un objet qui se comporte comme une liste sans pour autant allouer la mémoire nécessaire au stockage de tous ses éléments. Le coût de création d'une vraie liste augmente avec sa taille (son empreinte mémoire aussi !).

Note de la note : En Python version 2.x, Il existait deux versions de cette fonctionnalité: `range()` et `xrange()`. La première retournait une vraie liste, allouée complètement, alors que `xrange()` retournait un itérateur.

```
In [30]: print(type(range(3))) # Like xrange() in python2
         print(repr(range(3))) # Like xrange() in python2
         print(type(list(range(3)))) # Like range in python2

<class 'range'>
range(0, 3)
<class 'list'>
```

Instruction `break`

Il est possible d'arrêter prématurément une boucle grâce a l'instruction `break`.

L'instruction `break` est utilisable indifférement dans les boucles `for` ou `while`.

```
In [31]: compteur = 3
         while True:  # Notre boucle infinie
             compteur -= 1
             print('Dans la boucle infinie! compteur =', compteur)
             if compteur <= 0:
                 break  # On sort de la boucle while immédiatement
             print('on contine, compteur =', compteur)
         print("c'était pas vraiment une boucle infinie...")

Dans la boucle infinie! compteur = 2
on contine, compteur = 2
Dans la boucle infinie! compteur = 1
on contine, compteur = 1
Dans la boucle infinie! compteur = 0
c'était pas vraiment une boucle infinie...
```

En cas d'imbrication de plusieurs boucles, l'instruction `break` sort de la plus imbriquée (la plus proche).

```
In [32]: for i in (1, 2, 3):
             for j in (1, 2, 3, 4):
                 if i == 2:
                     break
                 print("i, j = %d, %d" % (i, j))
```

```
i, j = 1, 1
i, j = 1, 2
i, j = 1, 3
i, j = 1, 4
i, j = 3, 1
i, j = 3, 2
i, j = 3, 3
i, j = 3, 4
```

Instruction `continue`

Si, dans une boucle, on veut passer immédiatement à l'itération suivante, on utilise l'instruction `continue`.

```
In [33]: compteur = 9
         while compteur > 0:
             compteur -= 1
             if compteur % 2:
                 compteur /= 2
                 print('impair, on divise :', compteur)
                 continue # retourne immédiatement au début de la boucle
             print("pair, RAS")
         print("c'est fini...")
```

```
pair, RAS
impair, on divise : 3.5
impair, on divise : 1.25
impair, on divise : 0.125
impair, on divise : -0.4375
c'est fini...
```

Fonctions

Les fonctions permettent de réutiliser des blocs de code plusieurs endroits différents sans avoir a copier ce bloc.

En python, il n'y a pas de notion de sous-routine. Les procédures sont gérées par les objets de type fonctions, avec ou sans valeur de retour.

```
def <nom fonction>(arg1, arg2, ...):
    <bloc d'instructions>
    return <valeur>  # Instruction optionnelle
```

On distingue :

- les fonctions avec `return` des fonctions sans `return`
- les fonctions sans arguments (pour lesquelles `()` est vide) des fonctions avec arguments `(arg1, arg2, ...)`

Fonctions sans arguments

Fonction sans `return`

Pour définir une fonction :

```
In [34]: def func():  # Definition de la fonction
             print('You know what?')
             print("I'm happy!")
```

Pour utiliser une fonction que l'on a défini :

```
In [35]: func()  # 1er Appel de la fonction
         func()  # 2eme appel
         func()  # 3eme appel, etc...

You know what?
I'm happy!
You know what?
I'm happy!
You know what?
I'm happy!
```

Exercice : Ecrivez une fonction nommée "rien" qui ne fait rien et appelez là deux fois.

```
In [36]: # Votre code ici
```

Solution : exos/pass.py

Fonction avec `return`

```
In [37]: def func():  # Definition de la fonction
             return "I'm happy"  # La fonction retourne une chaine de caractère

         print("1er appel:")
         func()  # 1er Appel de la fonction : la valeur retournée n'est pas utilise
         print("2eme appel:")
         ret_val = func()  # Le retour du 2eme appel est stocké
         print("La fonction func() nous a renvoyé la valeur:", ret_val)
```

```
1er appel:
2eme appel:
La fonction func() nous a renvoyé la valeur: I'm happy
```

Exercice : Ecrivez une fonction nommée "donne_rien" qui retourne la chaine de caractères 'rien'. Appelez-là et affichez sa valeur de retour.

```
In [38]: # Votre code ici
```

Solution : exos/donne_rien.py

Fonctions avec arguments

Pour définir une fonction qui prend des arguments, on leur donne juste des noms entre les parenthèses de la ligne `def`. Ces paramètres seront définis comme des variables à l'intérieur de la fonction et recevrons les valeurs passées lors des appels de celle-ci.

```
In [39]: def somme(x, y):
             return x + y
```

Pour utiliser cette fonction avec diverses valeurs, il suffit de l'appeler plusieurs fois :

```
In [40]: print(somme(1, 2))
         print(somme(4, 7))
         print(somme(2 + 2, 7))
         print(somme(somme(2, 2), 7))
```

```
3
11
11
11
```

Exercice : Définissez une fonction nommée "chevalier", qui prend un paramètre, et qui répète (avec `print`) la chaîne de caractères 'Ni!' ce nombre de fois, et appelez cette fonction pour vérifier que ce `chevalier(3)` dit bien Ni trois fois comme il convient !

Voici quelques exemples montrant comment cette fonction doit se comporter:

```
chevalier(1)
Ni!
chevalier(3)
Ni!Ni!Ni!
chevalier(6)
Ni!Ni!Ni!Ni!Ni!Ni!
```

```
In [41]: # Votre code ici
```

```
In [42]: # Vérifions que tout fonctionne bien:
         #chevalier(1)
         #chevalier(3)
         #chevalier(6)
```

Solution : exos/chevalier.py

Exercice : Ecrivez une autre fonction, nommée "chevalier_ret", qui prend deux paramètres : un nombre et un booléen, et qui retourne une chaine de caractères constituée du nombre de répétitions de la chaine 'ni!' ou 'NI!' en fonction du paramètre booléen. Appelez cette fonction et affichez sa valeur de retour.

Voici quelques exemples montrant comment cette fonction doit se comporter:

```
a = chevalier(1, True)
print(a)
NI!
a = chevalier(3, False)
print(a)
ni!ni!ni!
a = chevalier(6, True)
print(a)
NI!NI!NI!NI!NI!NI!
```

```
In [43]: # Votre code ici
```

```
In [44]: # Vérifions que tout fonctionne bien:
         #a = chevalier_ret(1, True)
         #print(a)
         #a = chevalier_ret(3, False)
         #print(a)
         #a = chevalier_ret(6, True)
         #print(a)
```

Solution : exos/chevalier_ret.py

Utilisation de valeurs par défaut

```
In [45]: def somme(x, y=1):
             return x + y

         print(somme(1, 2))
         print(somme(4))  # Si la valeur de y n'est pas spécifiée, le paramètre 'y
```

```
3
5
```

Note : Les arguments ayant une valeur par défaut doivent être placés en dernier.

Utilisation des arguments par leur nom

```
In [46]: print(somme(y=7, x=4)) # L'ordre peut être changé lors de l'appel si les a

11
```

Capture d'arguments non définis

```
In [47]: def func(*args):
             print(args) # args est un tuple dont les éléments sont les arguments p

         func("n'importe", "quel nombre et type de", "paramètres", 5, [1, 'toto'],

("n'importe", 'quel nombre et type de', 'paramètres', 5, [1, 'toto'], None)


In [48]: def func(**kwargs):
             print(kwargs)  # kwargs est un dictionnaire dont les éléments sont les

         func(x=1, y=2, couleur='rouge', epaisseur=2)

{'couleur': 'rouge', 'x': 1, 'epaisseur': 2, 'y': 2}
```

On peut combiner ce type d'arguments pour une même fonction :

```
In [49]: def func(n, *args, **kwargs):  # cet ordre est important
             print("n =", n)
             print("args =", args)
             print("kwargs =", kwargs)

         func(2, 'spam', 'egg', x=1, y=2, couleur='rouge', epaisseur=2)

n = 2
args = ('spam', 'egg')
kwargs = {'couleur': 'rouge', 'x': 1, 'epaisseur': 2, 'y': 2}
```

Espace de nommage et portée des variables

1er exemple

On veut illustrer le mécanisme de l'espace de nommage des variables :

```
In [50]: def func1():
             a = 1
             print("Dans func1(), a =", a)

         def func2():
             print("Dans func2(), a =", a)

         a = 2
         func1()
         func2()
         print("Dans l'espace englobant, a =", a)
```

```
Dans func1(), a = 1
Dans func2(), a = 2
Dans l'espace englobant, a = 2
```

Cet exemple montre que :

1. Une variable définie localement à l'intérieur d'une fonction cache une variable du même nom définie dans l'espace englobant.
2. Quand une variable n'est pas définie localement à l'intérieur d'une fonction, Python va chercher sa valeur dans l'espace englobant (cas de `func2()`).

2ème exemple

On veut illustrer le mécanisme de portée des variables au sein des fonctions :

```
In [51]: def func():
             a = 1
             bbb = 2
             print('Dans func(): a =', a)

         a = 2
         func()
         print("Après func(): a =", a)
```

```
Dans func(): a = 1
Après func(): a = 2
```

Les variables définies localement à l'intérieur d'une fonction sont détruites à la sortie de cette fonction. Ici, la variable `b` n'existe pas hors de la fonction `func()`, donc Python renvoie une erreur si on essaye d'utiliser `b` depuis l'espace englobant :

```
In [52]: # Cette cellule génère une erreur
         print(bbb)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-52-316ca8dc9d86> in <module>()
      1 # Cette cellule génère une erreur
----> 2 print(bbb)


NameError: name 'bbb' is not defined
```

Fonctions *built-in*

Ces fonctions sont disponibles dans tous les contextes. La liste complète est détaillée ici. En voici une sélection :

- `dir(obj)` : retourne une liste des toutes les méthodes et attributs de l'objet `obj`
- `dir()` : retourne une liste de tous les objets du contexte courant
- `eval(expr)` : analyse et exécute la chaîne de caractère `expr`

```
In [53]: a = 1
         b = eval('a + 1')
         print("b est de type", type(b), "et vaut", b)

b est de type <class 'int'> et vaut 2
```

- `globals()` : retourne un dictionnaire des variables présentes dans le contexte global
- `locals()` : idem `globals()` mais avec le contexte local
- `help(obj)` : affiche l'aide au sujet d'un objet
- `help()` : affiche l'aide générale (s'appelle depuis l'interpréteur interactif)

- `input(prompt)` : retourne une chaîne de caractère lue dans la console après le message `prompt`

```
In [54]: reponse = input('Ca va ? ')  # Seule la variante input() fonctionne dans u
         if reponse.lower() in ('o', 'oui', 'yes', 'y', 'ok', 'da', 'jawohl', 'ja'):
             print('Supercalifragilisticexpialidocious')
         else:
             print('Faut prendre des vacances...')


---------------------------------------------------------------------------

StdinNotImplementedError                  Traceback (most recent call last)

<ipython-input-54-317b328c8e11> in <module>()
```

```
----> 1 reponse = input('Ca va ? ')  # Seule la variante input() fonctionne dan
      2 if reponse.lower() in ('o', 'oui', 'yes', 'y', 'ok', 'da', 'jawohl', 'j
      3     print('Supercalifragilisticexpialidocious')
      4 else:
      5     print('Faut prendre des vacances...')


/opt/conda/lib/python3.5/site-packages/ipykernel/kernelbase.py in raw_input
687         if not self._allow_stdin:
688             raise StdinNotImplementedError(
--> 689                 "raw_input was called, but this frontend does not suppo
690             )
691         return self._input_request(str(prompt),


StdinNotImplementedError: raw_input was called, but this frontend does not
```

- `len(seq)` : retourne la longueur de la séquence `seq`
- `max(seq)` : retourne le maximum de la séquence `seq`
- `min(seq)` : retourne le minimum de la séquence `seq`

- `range([start=0], stop[, step=1])` : retourne une liste d'entiers allant de `start` à `stop - 1`, par pas de `step`.

```
In [55]: print(list(range(10)))
         print(list(range(5, 10, 2)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 7, 9]
```

- `repr(obj)`: affiche la représentation de l'objet `obj`.
- `reversed(seq)` : retourne l'inverse de la séquence `seq`
- `sorted(seq)` : retourne une séquence triée à partir de la séquence `seq`
- `sum(seq)` : retourne la somme des éléments de la séquence `seq`

Exercices sur les fonctions

Exercice 1

Ecrire une fonction `stat()` qui prend en argument une séquence d'entiers et retourne un tuple contenant :

- la somme
- le minimum
- le maximum

des éléments de la liste

```
In [56]: def stat(a_list):
             # votre fonction
             pass

         stat([1, 4, 6, 9])
```

Solution : exos/mystat.py

Exercice 2 :

Ecrire une fonction qui prend en paramètre une liste de prix hors taxes (HT) et qui retourne la somme toutes taxes comprises (TTC).

```
In [57]: def ttc(liste_prix_ht):
             # Votre fonction
             pass

         print(ttc([12, 56, 99, 1, 128]))
```

```
None
```

Solution : exos/ttc.py

Exercice 3 : Ecriture d'un *wrapper* de fonction

Noël pointe son nez, amusez-vous avec les boules de décoration !

Soit une fonction `boule()` capable d'accrocher une boule de couleur à la position `(x, y)` d'un sapin.

Exercice inspiré du Mooc de l'INRIA Python : des fondamentaux à l'utilisation du langage

```
In [58]: def boule(x, y, couleur='bleue'):
             print("J'accroche une boule en ({}, {}) de couleur {}".format(x, y, co

         # On place la première boule sur le sapin
         boule(1, 2)
         # Puis une autre, etc.
         boule(3, 4)
```

```
J'accroche une boule en (1, 2) de couleur bleue
J'accroche une boule en (3, 4) de couleur bleue
```

Ecrire une fonction *wrapper* `boule_or()` qui crée des boules dorées en appelant la fonction `boule()`. Dans le futur, on souhaite modifier la fonction `boule()` pour lui faire accepter un nouvel argument `rendu (brillant, mat, etc.)`. La fonction `boule_or()` devra continuer à fonctionner après cette modification de la fonction `boule()` et intégrer la nouvelle fonctionnalité `rendu` sans qu'il soit nécessaire de la modifier.

```
In [59]: def boule_or(x, y):
             # Votre code ici
             pass

         # On place une boule en or sur le sapin
         boule_or(2, 2)
```

Maintenant, on met à jour la fonction `boule()` :

```
In [60]: def boule(x, y, couleur='bleue', rendu='mat' ):
             print("J'accroche une boule en ({}, {}) de couleur {} et de rendu {}.'
         boule(1, 3, couleur='jaune', rendu='brillant')
```

```
J'accroche une boule en (1, 3) de couleur jaune et de rendu brillant.
```

Vérifier que votre fonction `boule_or()` marche encore et gère la nouvelle fonctionnalité :

```
In [61]: boule_or(3, 1, rendu='brillant') # doit retourner :
         # J'accroche une boule en (3, 1) de couleur or et de rendu brillant.
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-61-7b0890870e4d> in <module>()
----> 1 boule_or(3, 1, rendu='brillant') # doit retourner :
      2 # J'accroche une boule en (3, 1) de couleur or et de rendu brillant.


TypeError: boule_or() got an unexpected keyword argument 'rendu'
```

Solution : exos/boule.py

Exceptions

Pour signaler des conditions particulières (erreurs, évenements exceptionnels), Python utilise un mécanisme de levée d'exceptions.

```
In [62]: # Cette cellule génère une erreur
         raise Exception
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-62-4f63ccde8a3b> in <module>()
      1 # Cette cellule génère une erreur
----> 2 raise Exception

Exception:
```

Ces exceptions peuvent embarquer des données permettant d'identifier l'évenement producteur.

```
In [63]: # Cette cellule génère une erreur
         raise Exception('Y a une erreur')
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-63-d514012debf9> in <module>()
      1 # Cette cellule génère une erreur
----> 2 raise Exception('Y a une erreur')

Exception: Y a une erreur
```

La levée d'une exception interrompt le cours normal de l'exécution du code et "remonte" jusqu'à l'endroit le plus proche gérant cette exception.

Pour intercepter les exceptions, on écrit :

```
try:
    <bloc de code 1>
except Exception:
    <bloc de code 2>
```

```
In [64]: try:
             print('ici ca fonctionne')
             # ici on détecte une condition exceptionnelle, on signale une exceptio
             raise Exception('y a un bug')
             print('on arrive jamais ici')
         except Exception as e:
             # L'excécution continue ici
             print("ici on peut essayer de corriger le problème lié à l'exception :
         print("et après, cela continue ici")

ici ca fonctionne
ici on peut essayer de corriger le problème lié à l'exception : Exception('y a un b
et après, cela continue ici
```

Exemple illustrant le mécanisme de remontée des exceptions d'un bloc à l'autre :

```
In [65]: def a():
             raise Exception('coucou de A')

         def b():
             print('début B')
             a()
             print('fini B')

         try:
             b()
         except Exception as e:
             print("l'exception vous envoie le message :", e)

début B
l'exception vous envoie le message : coucou de A
```

> **Exercice** : Ecrivez une fonction qui demande à l'utilisateur un fichier a ouvrir, et qui gère correctement les fichiers inexistants. Ensuite cette fonction affichera la première ligne du fichier. Finalement la fonction retournera une valeur booléenne indiquant que le fichier a été ouvert ou non.

Attention: sous windows, par defaut les extensions de fichier sont cachées...

```
In [66]: # Votre code ici
```

exos/filefail.py
Pour plus d'informations sur les exceptions, se référer ici

Les gestionnaires de contexte

Pour faciliter la gestion des obligations liées à la libération de ressources, la fermeture de fichiers, etc... Python propose des gestionnaires de contexte introduits par le mot clé `with`.

```
In [67]: with open('interessant.txt', 'r') as fichier_ouvert:
             # Dans ce bloc de code le fichier est ouvert en lecture, on peut l'uti
             print(fichier_ouvert.read())
         # Ici, on est sorti du bloc et du contexte, le fichier à été fermé automat

Si vous lisez ce texte alors
...
vous savez lire un fichier avec Python !
```

```
In [68]: # Cette cellule génère une erreur
         print(fichier_ouvert.read())


        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-68-7b3d6f0bd331> in <module>()
          1 # Cette cellule génère une erreur
    ----> 2 print(fichier_ouvert.read())


        ValueError: I/O operation on closed file.
```

Exercice : Reprenez le code de l'exercice précédent, et utilisez `with` pour ne pas avoir à utiliser la méthode `close()`.

```
In [69]: # Votre code ici
```

Solution : exos/filewithfail.py

Il est possible de créer de nouveaux gestionnaires de contexte, pour que vos objets puissent être utilisés avec `with` et que les ressources associées soient correctement libérées.

Pour plus d'informations sur la création de gestionnaires de contexte, voir ici.

Les compréhensions de listes

Python a introduit une facilité d'écriture pour les listes qui permet de rendre le code plus lisible car plus concis.

```
In [70]: # Ce code construit une liste ne contenant que les éléments pairs de la li
         Liste1 = list(range(10))
         print(Liste1)
         ListePaire = []
         for i in Liste1:
             if (i % 2) == 0:
                 ListePaire.append(i)
         print(ListePaire)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
```

```
In [71]: # Ici, on fait la même chose, en liste...
         ListePaire = [i for i in Liste1 if (i % 2) == 0]
         print(ListePaire)
```

```
[0, 2, 4, 6, 8]
```

Cette concision peut être utile, mais n'en abusez pas, si vous commencez a avoir une compréhension de liste trop complexe a écrire en une simple ligne, faites le "normalement", avec les boucles et conditions explicites.
Plus d'informations ici.

Les expressions génératrices

C'est une forme d'écriture, très proche des compréhensions de listes, mais qui ne crée pas de nouvel objet liste immédiatement. Les items sont produits à la demande.

```
In [72]: tuplePairs = (i for i in Liste1 if (i % 2) == 0)
         print(tuplePairs)
         print(list(tuplePairs))

<generator object <genexpr> at 0x7f912035aba0>
[0, 2, 4, 6, 8]
```

Plus d'informations ici.

Modules

- Python fournit un système de modularisation du code qui permet d'organiser un projet contenant de grandes quantités de code et de réutiliser et de partager ce code entre plusieurs applications.
- L'instruction `import` permet d'accéder à du code situé dans d'autres fichiers. Cela inclut les nombreux modules de la librairie standard, tout comme vos propres fichiers contenant du code.
- Les fonctions et variables du module sont accessibles de la manière suivante :

  . .([][,]...)

```
In [73]: # Pour utiliser les fonctions mathématiques du module 'math'
         import math

         pi = math.pi
         print('%.2f' % pi)
         print('%.2f' % math.sin(pi))
```

```
3.14
0.00
```

Pour créer vos propres modules, il suffit de placer votre code dans un fichier avec l'extension '.py', et ensuite vous pourrez l'importer comme module dans le reste de votre code.

Il y a un fichier mon_module.py a coté du notebook, il contient du code définissant `ma_variable` et `ma_fonction()`.

```
In [74]: import mon_module
         print(mon_module.ma_variable)
         mon_module.ma_fonction()  # On accede ainsi à l'attribut ma_fonction() du
```

```
27
un appel à ma_fonction()
```

On peut importer un module sous un autre nom (pour le raccourcir, en général) :

```
In [75]: import mon_module as mm
         mm.ma_fonction()
```

```
un appel à ma_fonction()
```

Note : un module n'est importé qu'une seule fois au sein d'une même instance Python.

> **Exercice** : Modifiez le code contenu dans le fichier mon_module.py, et reexécutez la cellule ci-dessus.

Pour plus d'informations sur les modules, allez voir ici.

Quelques modules de la stdlib

La librairie standard de Python est incluse dans toute distribution de Python. Elle contient en particulier une panoplie de modules à la disposition du développeur.

`string`

- `find()`
- `count()`
- `split()`
- `join()`
- `strip()`
- `upper()`
- `replace()`

`math`

- `log()`
- `sqrt()`
- `cos()`
- `pi`
- `e`

`os`

- `listdir()`
- `getcwd()`
- `getenv()`
- `chdir()`
- `environ()`
- `os.path : exists(), getsize(), isdir(), join()`

`sys`

- `argv`
- `exit()`
- `path`

Mais bien plus sur la doc officielle de la stdlib !

Bonnes pratiques

Commentez votre code

- pour le rendre plus lisible
- pour préciser l'utilité des fonctions, méthodes, classes, modules, etc...
- pour expliquer les parties complexes

Habituez-vous assez tôt aux conventions préconisées dans la communauté des utilisateurs de python. Cela vous aidera a relire plus facilement le code écrit par d'autres, et aidera les autres (et vous-même !) à relire votre propore code. Collaborez !

Ces conventions sont décrites dans le document PEP n°8 (Python Enhancement Proposal). L'outil pep8 permet d'automatiser la vérification du respect de ces règles.

Exercice :

1. Lisez le PEP8, et corrigez toutes les fautes commises dans ce notebook
2. Envoyez le résultat à votre formateur

Organisez vos modules en packages

Un package est un répertoire qui contient des modules (fichiers `.py`) et un fichier `__init__.py`. Une arborescence de packages permet de les organiser de manière hiérarchique.

On peut accéder aux sous-packages avec la notation :

```
import <package1>.<subpackage1>[.<subsubpackage>]...
```

Pour plus d'informations sur la construction de packages, voir ici

Environnements de développement intégrés :

- pydev
- eclipse
- spyder

Gestionnaires de versions

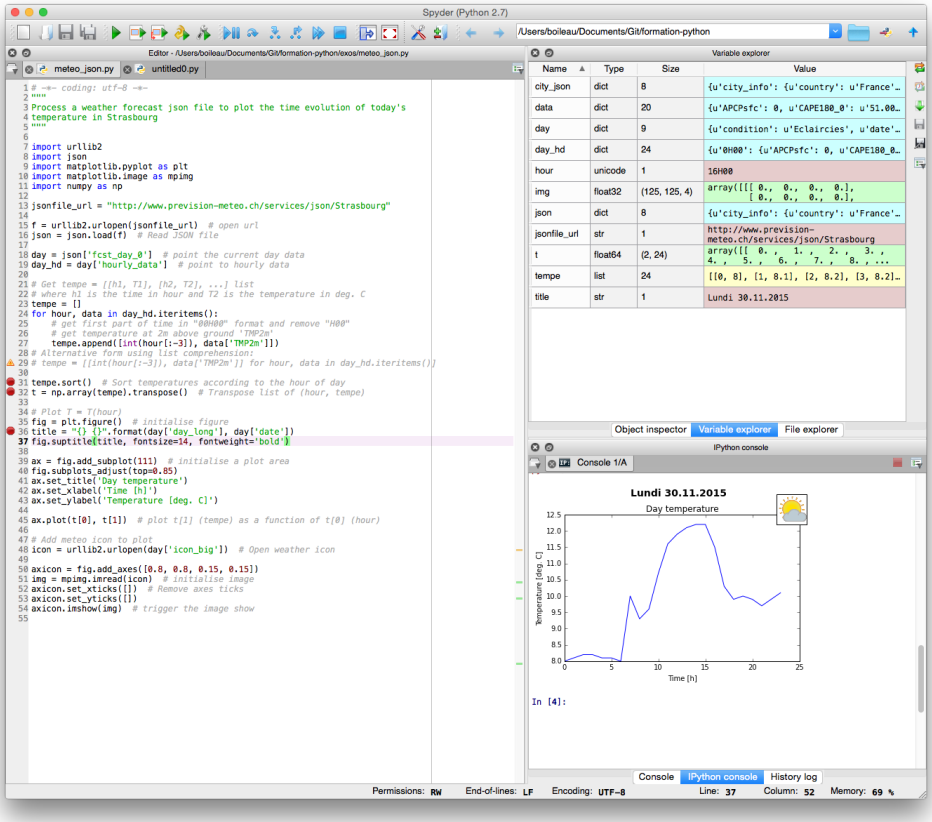
- git
- subversion

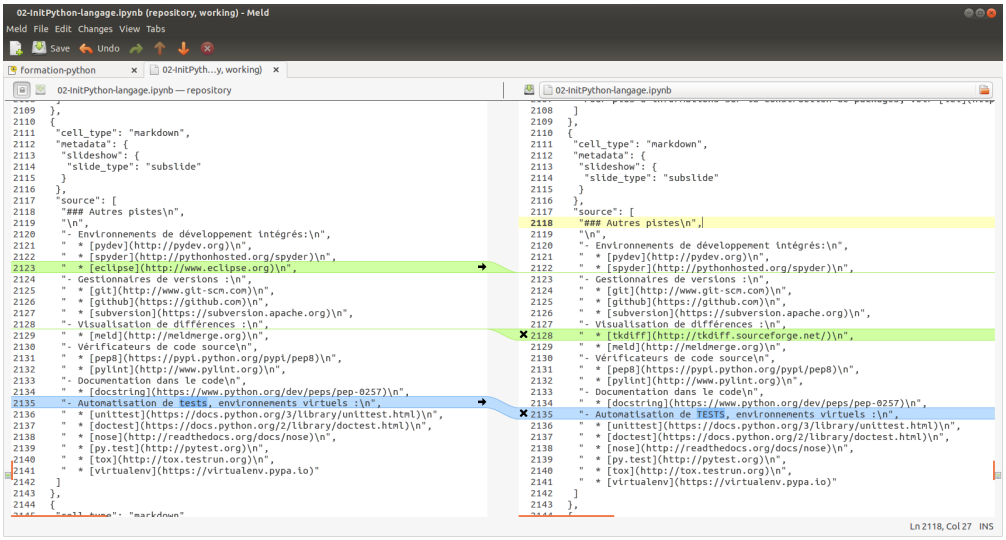
Héberger vos dépôts de sources

- github
- gitlab

Visualisation de différences :

- tkdiff
- kdiff3
- meld - écrit en python !


Spyder (Python 2.7)
Editor - /Users/boileau/Documents/Git/formation-python/exos/meteo_json.py
meteo_json.py
untitled0.py
Variable explorer
Name
Type
Size
Value
city_json
dict
8
{u'city_info': {u'country': u'France'…
data
dict
20
{u'APCPsfc': 0, u'CAPE180_0': u'51.00…
day
dict
9
{u'condition': u'Eclaircies', u'date'…
day_hd
dict
24
{u'0H00': {u'APCPsfc': 0, u'CAPE180_0…
hour
unicode
1
16H00
img
float32
(125, 125, 4)
array([[[0., 0., 0., 0.],
[0., 0., 0., 0.],
json
dict
8
{u'city_info': {u'country': u'France'…
jsonfile_url
str
1
http://www.prevision-meteo.ch/services/json/Strasbourg
t
float64
(2, 24)
array([[0. , 1. , 2. , 3. ,
4. , 5. , 6. , 7. , 8. , ...
tempe
list
24
[[0, 8], [1, 8.1], [2, 8.2], [3, 8.2]…
title
str
1
Lundi 30.11.2015
Object inspector
Variable explorer
File explorer
IPython console
Console 1/A
Lundi 30.11.2015
Day temperature
Temperature [deg. C]
Time [h]
In [4]:
Console
IPython console
History log
Permissions: RW
End-of-lines: LF
Encoding: UTF-8
Line: 37
Column: 52
Memory: 69 %


02-initPython-langage.ipynb (repository, working) - Meld
Meld File Edit Changes View Tabs
Save
Undo
formation-python
02-initPyth...y, working)
02-initPython-langage.ipynb — repository
02-initPython-langage.ipynb
Ln 2118, Col 27
INS

Vérificateurs de code source

- pep8
- pylint

Documentation dans le code

- docstring

Automatisation de tests, environnements virtuels :

- unittest
- doctest
- nose
- py.test
- tox
- virtualenv

Python 3.x vs 2.x

C'est le futur, et incidemment aussi le présent voire même le passé...

- Quoi : une version qui casse la compatibilité ascendante
- Pourquoi : nettoyage de parties bancales du langage accumulées au fil du temps
- Python 3.0 est sorti en 2008
- Python 2.7 est sorti en 2010 : EOL, fin de vie, (mal-)heureusement longue à venir
- Un certain nombre de choses n'a pas encore été converti pour fonctionner avec
- Les distributions linux majeures proposent encore la 2.X par défaut, mais la 3 est disponible en parallèle
- Une partie, la moins disruptive, à quand même été portée vers 2.6 et 2.7 pour aider à la transition
- Les tutoriels, et autres documentations disponibles sur internet ne sont pas forcément migrées
- Pour un nouveau projet, recherchez un peu avant de vous lancer, pour vérifier les besoins en librairies externes
- Les implémentations tierces d'interpréteurs python peuvent avoir des degrés variables de compatibilité avec les versions 3.x
- Les modules comportant des extentions en C sont plus compliqués à porter.

Différences notables entre Python 2 et Python 3

- Division entière
- `print()`
- variable à durée de vie plus stricte (boucles, etc...)
- toutes les classes sont du nouveau type
- Les chaînes de caractères sont en UTF-8 par défaut & encoding(s) & byte() interface
- stdlib changée
- `range()` vs `xrange()`
- outils `2to3.py, 3to2, python-modernize, futurize`
- `pylint --py3k`
- module de compatibilité: six

Plus d'informations sur le wiki officiel.

Philosophie du langage : le zen de Python

PEP 20

```
In [76]: import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Exercices complémentaires

Listes

Ecrivez la fonction `odds()` qui prend une liste de nombres, et qui retourne une liste ne contenant que les nombres impairs.

- `odds([1,2,3,4,5]) => [1,3,5]`

```
In [77]: def odds(numbers):
             # Votre code ici
             pass
```

Solution : exos/odds.py

Chaines de caractères

Ecrivez les fonctions :

- `majuscules('azERtyUI') -> 'AZERTYUI'`
- `minuscules('azERtyUI') -> 'azertyui'`
- `inverse_casse('azERtyUI') -> 'AZerTYui'`
- `nom_propre('azERtyUI') -> 'Azertyui'`

```
In [78]: # Votre code ici
```

Solution : exos/chaines.py

Récursion

Les fonctions dites "récursives" sont des fonctions qui font appel à elles-même, en résolvant une partie plus petite du problème à chaque appel, jusqu'à avoir un cas trivial à résoudre.

Par exemple pour calculer : "la somme de tout les nombres de 0 jusqu'à x", on peut utiliser une fonction récursive:

La somme de tous les nombres de 0 à 10 est égale à 10 plus la somme de tous les nombres de 0 à 9, etc...

```
In [79]: def sum_to(x):
             if x == 0:
                 return 0
             return x + sum_to(x - 1)

In [80]: print(sum_to(9))

45
```

La fonction mathématique factorielle est similaire, mais calcule : "le produit de tout les nombres de 1 jusqu'à x".

```
In [81]: def fact(x):
             if x == 1:
                 return 1
             return x * fact(x - 1)

In [82]: print(fact(5), fact(9))

120 362880
```

La fonction mathématique qui calcule la suite des nombres de Fibonacci, peut être décrite comme suit:

- `fibo(0) = 0`
- `fibo(1) = 1`

Et pour toutes les autres valeurs:

- `fibo(x) = fibo(x - 1) + fibo(x - 2)`

Exercice : écrivez une fonction récursive `fibo(x)` qui renvoie le x-ième nombre de la suite de Fibonnaci.

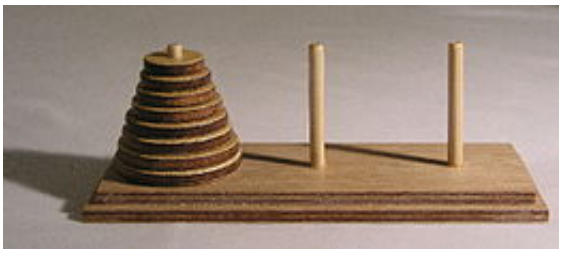
```
In [83]: def fibo(x):
             # Votre code ici
             pass

In [84]: print(fibo(9))

None
```

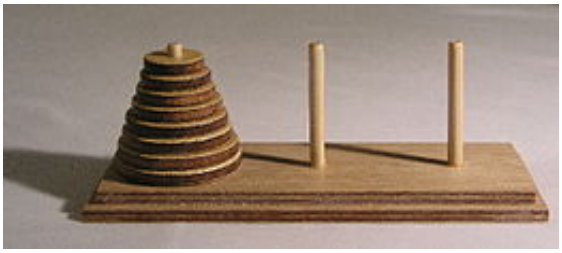
Solution : exos/fibo.py

Les tours de hanoï



Le but de ce jeu est de déplacer la pile de disques de la première colonne à la dernière, en s'aidant de la colonne intermédiaire, en suivant ces règles simples:

- On ne peut déplacer qu'un seul disque à la fois
- On ne peut placer un disque que sur un autre qui est plus grand



Cliquer ici pour la version animée

Exemple de solution pour une pile de quatre disques :
Pour cet exercice, on va représenter les colonnes avec des listes :

```
In [85]: source = [4,3,2,1]
         interm = []
         destin = []
```

Le but est de tout déplacer vers la destination :

```
In [86]: # Cette cellule génère une erreur
         assert (source, interm, destin) == ([], [], [4,3,2,1]), "Le résultat n'est

        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-86-617d1ecb5bd9> in <module>()
          1 # Cette cellule génère une erreur
    ----> 2 assert (source, interm, destin) == ([], [], [4,3,2,1]), "Le résultat n'


        AssertionError: Le résultat n'est pas correct
```

Nous allons utiliser pour déplacer un disque d'une colonne à une autre, la fonction `move(a, b)` :

```
In [87]: # Ne pas modifier cette fonction, utilisez-la dans votre fonction hanoi()
         def move(a, b):
             if not a:
                 raise Exception('Interdit: la colonne source est vide !')
             item = a.pop()
             if b and item > b[-1]:
                 raise Exception('Interdit: le disque {} est trop grand pour la col
             b.append(item)
             print('Déplaçons {}: source={}, interm={}, destin={}'.format(item, sou
```

Exercice : écrivez une fonction récursive `hanoi(taille, col_src, col_tmp, col_dst)` qui va déplacer les disques de la colonne `col_src` à la colonne `col_dst` en utilisant la fonction `move()`.

```
In [88]: def hanoi(size, source, interm, destin):
             # Votre code ici
             pass
```

Pour vérifier son fonctionnement :

```
In [89]: hanoi(len(source), source, interm, destin)

         assert (source, interm, destin) == ([], [], [4,3,2,1]), "Le résultat n'est

        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-89-2f4d5add0baa> in <module>()
          1 hanoi(len(source), source, interm, destin)
          2
    ----> 3 assert (source, interm, destin) == ([], [], [4,3,2,1]), "Le résultat n'


        AssertionError: Le résultat n'est pas correct
```

Solution : exos/hanoi.py