notebook_1

September 13, 2018

1 Part 1. Calculer avec Python 3.x

Quelle est la différence entre Python et une calculette ? On commence cette leçon en montrant comment Python peut servir de **calculette**, puis nous verrons quelques éléments de base en programmation : types de données, variables, listes, et boucles. What is the difference between *Python* and a calculator? We begin this first lesson by showing how Python can be used **as** a calculator, and we move into some of the basic programming language constructs: data types, variables, lists, and loops.

1.1 Table des matières

1.2 Python comme calculette

Différentes sortes de données sont stockées avec des types différents en Python. Par exemple, si vous voulez travailler avec des entiers, vos données seront stockées avec type *int*. Un nombre réel doit être stocké en *float*. Il y a aussi les types booléens (True/False), les chaines de caractères (comme "Hello World!"), et beaucoup d'autres types.

On trouvera une référence complète pour les types numériques et les opérations arithmétiques dans la documentation Python officielle.

Les opérations arithmétiques en Python sont dénotés `+`, `-`, `*`, and `/`. Evaluez chacune des cellules suivantes pour voir comment Python traite ces opérations sur les *entiers*. A mesure que vous avancez dans cette leçon et les suivantes, essayez de *prévoir* ce que produira un calcul avant de l'effectuer.

```
In [ ]: 2 + 3
```

```
In [ ]: 2 * 3
```

```
In [ ]: 5 - 11
```

```
In [ ]: 5.0 - 11
```

```
In [ ]: 5 / 11
```

```
In [ ]: 6 / 3

In [ ]: 5 // 11

In [ ]: 6 // 3
```

Les résultats ne doivent pas être trop surprenants, mais les deux derniers demandent une petite explication. Python *interprète* le nombre 5 comme un *int* (entier) et 5.0 comme un *float* (décimal).

Python permet deux sortes de division. En Python 3.x une division avec un seul signe slash donne un résultat *float* ; avec deux signes slash on obtient le quotient *int* de la division euclidienne.

Ceci est utile mais on doit faire attention. **Ce tutoriel suppose qu'on utilise Python 3.x.** Si vous êtes en Python 2.x, attention : les deux symboles de la division se comportent autrement.

```
In [ ]: -12 // 5
```

Python vous permet de grouper des expressions entres parenthèses, et suit les priorités opératoires habituelles.

```
In [ ]: (3 + 4) * 5

In [ ]: 3 + (4 * 5)

In [ ]: 3 + 4 * 5   #  Prévoir le résultat avant d'exécuter
```

Ci-dessous, une cellule vide pour vous exercer. Rappel : pour créer une nouvelle cellule, faire **Echap a** ou **Echap b**; pour supprimer une cellule faire **Echap x**.

```
In [ ]: #  Cellule vide ; expérimenter.
```

En arithmétique, la division euclidienne est très importante. La division entière (deux signes slash) fournit le quotient, le symbole `%` fournit le reste.

```
In [ ]: 23 // 5  # quotient

In [ ]: 23 % 5  # reste
```

Remarquez ci-dessus le hashtag `#` pour commenter dans une ligne de code.

Python possède aussi une commande unique pour la division euclidienne. Le résultat est un *tuple*.

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

In [ ]: type(divmod(23,5))
```

La fonction *type* permet de connaitre le type des données.

```
In [ ]: type(3)

In [ ]: type(3.0)
```

```
In [ ]: type('Hello')

In [ ]: type([1,2,3])
```

Il est très important de connaitre letype des données que l'on manipule, et comment Python va opérer sur ces différents types.

```
In [ ]: 3 + 3

In [ ]: 3.0 + 3.0

In [ ]: 'Hello' + 'World!'

In [ ]: [1,2,3] + [4,5,6]

In [ ]: 3 + 3.0

In [ ]: 3 + 'Hello!'

In [ ]: #  Cellule vide, pour s'exercer.
```

Comme vous pouvez le constater, l'addition a un sens entre valeurs numériques ou entre chaines de caractères, mais pas entre un nombre et une chaine.

Par ailleurs, la multiplication a un sens entre un entier et une chaine ou une liste.

```
In [ ]: 3 * 'Hello!'

In [ ]: 0 * 'Hello!'

In [ ]: 2 * [1,2,3]
```

Pouvez vous créer une chaine de 100 A's (comme `AAA...`)? Essayez dans la cellule ci-dessous.

```
In [ ]: # créer ici une chaine de 100 A's
```

Les exposants en Python sont donnés par l'opérateur `**`.

```
In [ ]: 2**1000

In [ ]: 2.0**1000
```

Comme tout-à-l'heure, Python interprète l'operation (`**`) différéremment suivant les types. Avec des types entiers, le calcul est **exact**. Une bonne nouvelle, en Python, c'est que les entiers peuvent être de longueur arbitraire, comme ci-dessus, contrairement à des langages comme C/C++.

La notation e+301 signifie "multiplié par `10**301`"; cela s'appelle *notation scientifique*.

```
In [ ]: type(2**1000)

In [ ]: type(2.0**1000)
```

1.2.1 Exercises

1. Quels types avez vous vus, et sur quelles données sont ils utilisés ? Donner des exemples.
2. Comment se comporte la division `/` sur differents types ?
3. Même question avec la multiplication `*`.
4. Quelle est la différence entre 100 et 100.0, pour Python?

Double-cliquez cette cellule markdown pour l'éditer, et répondre aux questions de l'exercice.

1.3 Calculer avec des booléens

Un booléen (type *bool*) est le plus petit morceau de donnée possible ; il ne prend que les valeurs *True* or *False*. C'est en particulier utile pour tester des égalités ou inégalités entre nombres. Ci-dessous quelques exemples. Voir aussi la documentation Python officielle.

```
In [ ]: 3 > 2
```

```
In [ ]: type(3 > 2)
```

```
In [ ]: 10 < 3
```

```
In [ ]: 2.4 < 2.4000001
```

```
In [ ]: 32 >= 32
```

```
In [ ]: 32 >= 31
```

```
In [ ]: 2 + 2 == 4
```

Qui est le plus grand: 23^{32} or 32^{23}? Utilisez la cellule ci-dessous pour répondre !

```
In [ ]: #  Write your code here.
```

Les expressions <, >, <=, >= arsonte interprétées comme des **operations** avec un input numerique et un output booléen. Le symbole == teste si deux nombres sont égaux. Attention à ne pas confondre avec =qui a une signification toute différente (déclaration de variable, voir plus bas).

Using the remainder operator `%` and equality, we obtain a divisibility test.

```
In [ ]: 63 % 7 == 0  # 63 est-il divisible par 7 ?
```

```
In [ ]: 101 % 2 == 0  # 101 est-il pair ?
```

Utilisez la cellule dessous pour déterminer si 1234567890 est divisible par 3.

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

Entre booléens on peut utiliser les opérateurs and, or, not.

Ci-dessous les **tables de vérité** des opérateurs **and** et **or**.

| and | | True | False | |—–| |——|——-| | | | | | **True** | | True | False | | **False** | | False | False |

| or | | True | False | |—–| |——|——-| | | | | | **True** | | True | True | | **False** | | True | False |

```
In [ ]: True and False
```

```
In [ ]: True or False
```

```
In [ ]: True or True
```

```
In [ ]: not True
```

Utilisez les tables de vérité pour prévoir le résultat des expressions suivantes, avant d'évaluer le code.

```
In [ ]: (2 > 3) and (3 > 2)
```

```
In [ ]: (1 + 1 == 2) or (1 + 1 == 3)
```

```
In [ ]: not (-1 + 1 >= 0)
```

```
In [ ]: 2 + 2 == 4
```

```
In [ ]: 2 + 2 != 4  # Pour "not equal", Python utilise l'opérateur `!=`.
```

```
In [ ]: 2 + 2 != 5
```

```
In [ ]: not (2 + 2 == 5)
```

Python accepte quelques opérations entre booléens et nombres. Expérimentez ci-dessous et changez les valeurs.

```
In [ ]: False * 100
```

```
In [ ]: True + 13
```

L'aptitude de Python à interpréter les opérations suivant le contexte a ses avantages et inconvénients. D'un côté ça permet des codes plus courts, de l'autre ça peut diminuer la lisibilité du code. Les bons programmeurs visent à produire du code pas seulement court, mais aussi lisible.

Le Zen of Python est une série de 20 aphorismes pour programmeurs Python. Voici les sept premiers.

> 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.

1.3.1 Exercises

3. Comment savoir, en une ligne de code, si 2349872348723, est un multiple de 7 mais pas un multiple de 11 ?

4. L'opérateur `xor` veut dire "exclusive or". Sa table de vérité est : `True xor True = False` et `False xor False = False` et `True xor False = True` et `False xor True = True`. Implémenter `xor` à l'aide des opérateurs `and`, `or`, et `not`.

```
In [ ]: #  Espace pour répondre aux questions de l'exercice
```

1.4 Déclaration de variables

Une caractéristique essentielle en programmation est la déclaration de variables. Quand vous déclarez une variable, vous *stockez* une donnée en mémoire et vous donnez un *nom* à cette donnée. Tout ceci se fait à l'aide du symbole =.

```
In [ ]: e = 2.71828

In [ ]: e * e

In [ ]: type(e)
```

Un nom de variable doit commencer avec une lettre, ne doit pas contenir d'espaces, et ne doit pas être un mot Python existant (comme `True` par exemple). On peut utiliser des lettres (minuscules et majuscules), des chiffres et des underscores _.

Donc, comme nom de variable, `e` est valide mais `type` ne l'est pas. Les débutants sont tentés d'utiliser des abréviations courtes comme `dx` or `vbn`. Ce n'est pas recommandé ; utilisez plutôt des noms explicites comme `difference_x` ou `very_big_number`. Cela fera un code plus lisible pour vous et les autres !

Il y a différentes conventions de style pour les noms de variables. Nous utiliserons des noms en minuscule, et des underscores pour séparer les mots, suivant à peu près les Google's style conventions for Python code.

```
In [ ]: my_number = 17

In [ ]: my_number < 23
```

Après avoir déclaré une variable, sa valeur reste la même jusqu'à ce qu'elle soit modifiée. Vous pouvez modifier la valeur d'une variable avec une assignation simple. Après les lignes ci-dessus, la valeur de my_number est 17.

```
In [ ]: my_number = 3.14
```

Cette commande réaffecte la valeur de my_number à 3.14. Notez que cela change aussi le type! Elle remplace effectivement la valeur précédente et la remplace par la nouvelle valeur.

Il est souvent utile de changer la valeur d'une variable *de manière incrémentielle* ou *récursive*. Python, comme de nombreux langages de programmation, permet d'assigner des variables de manière auto-référentielle. Que pensez-vous de la valeur de S après les quatre lignes suivantes?

```
In [ ]: S = 0
        S = S + 1
        S = S + 2
        S = S + 3
        print(S)
```

Considérez maintenant les commandes suivantes

```
In [ ]: my_number = 17
        new_number = my_number + 1
        my_number = 3.14
```

Quelles sont les valeurs de my_number et new_number,après l'exécution de ces trois lignes ? Pour accéder à ces valeurs vous pouvez utiliser la fonction *print*.

```
In [ ]: print(my_number)
        print(new_number)
```

1.4.1 Exercises

1. What is the difference between = and == in the Python language ?
2. Imaginez deux variables `a` et `b`, et vous voulez permuter leurs valeurs. Comment faites-vous ?

```
In [ ]: #  Répondre ici, avec un exemple.
```

1.5 Listes et ranges

Python se distingue par le rôle central joué par les listes (type `list`). Les données d'une liste peuvent être de tout type. Plusieurs types sont possibles dans la même liste! La syntaxe de base d'une liste consiste à utiliser des crochets pour inclure les éléments de liste et les virgules pour séparer les éléments de la liste.

```
In [ ]: type([1,2,3])
```

```
In [ ]: type(['Hello',17])
```

Il y a un autre type appelé un tuple que nous utiliserons rarement (type `tuple`). Les tuples utilisent des parenthèses pour la clôture au lieu des parenthèses.

```
In [ ]: type((1,2,3))
```

Il existe un autre type de liste dans Python 3, appelé range. Les ranges sont un peu comme des listes, mais au lieu de mettre chaque élément dans un emplacement de mémoire, les ranges doivent simplement retenir trois nombres entiers: leur début, leur arrêt et leur pas.

La commande `range` crée un range avec un début, un arrêt et un pas donnés. Si vous ne saisissez qu'un seul entier en input, le range **commence à zéro** et utilise un **pas de un** et arrête **juste avant le numéro d'arrêt**.

On peut créer une liste à partir d'un range (en plaçant chaque terme du range dans un emplacement de mémoire), en utilisant la commande `list`. Voici quelques exemples.

```
In [ ]: type(range(10)) # Les ranges ont leur propre type en Python 3.x.  Pas en Py

In [ ]: list(range(10))
```

Une forme plus compliquée, à deux entrées, de la commande range produit une gamme d'entiers **commençant à** un nombre donné et **terminant avant** un autre nombre donné.

```
In [ ]: list(range(3,10))

In [ ]: list(range(-4,5))
```

C'est une source commune de difficulté pour les débutants en Python. Alors que le premier paramètre (-4) est le point de départ de la liste, la liste se termine juste avant le deuxième paramètre (5). Cela prend un peu de temps pour s'y habituer, mais les programmeurs Python expérimentés ont tendance à aimer cette convention.

La longueur d'une liste est accessible par la commande `len`.

```
In [ ]: len([2,4,6])

In [ ]: len(range(10))  # La commande len peut traiter des listes et des plages. Pa

In [ ]: len(range(10,100)) # Prévoir le résultat avant d'évaluer
```

La variante finale de la commande range (pour l'instant) est la commande à trois paramètres `range(a, b, s)`. Cela produit une liste comme le range (a, b), mais avec "un pas de s". En d'autres termes, il produit une liste d'entiers, commençant à a, augmentant de s d'une entrée à l'autre, allant jusqu'à (mais n'incluant pas) b. Il vaut mieux expérimenter un peu !

```
In [ ]: list(range(1,10,2))

In [ ]: list(range(11,30,2))

In [ ]: list(range(-4,5,3))

In [ ]: list(range(10,100,17))
```

Cela peut aussi être utilisé pour les ranges décroissants ; observer que le nombre final b dans la plage (a, b, s) n'est pas inclus.

```
In [ ]: list(range(10,0,-1))
```

1.5.1 Exercises

1. Si `a` et `b` sont des entiers, quelle est la longueur de `range(a,b)` ?
2. Combien de multiples de 7 sont compris entre 10 et 100? Vous pouvez répondre assez rapidement avec les commandes `range` et `len`.
3. Créez la liste `[1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5]` avec la commande range et une autre opération.

```
In [ ]: #  Espace pour répondre aux questions.
```

1.6 Itérer sur un range

Les ordinateurs sont excellents pour les tâches répétitives et simples. Nous examinons ici un moyen simple et commun de réaliser un calcul répétitif: la boucle `for`. La la boucle `for` itère à travers les éléments d'une liste ou d'un range, en effectuant une action pour chaque élément.

```
In [ ]: for n in [1,2,3,4,5]:
            print(n*n)

In [ ]: for s in ['I','Am','Python']:
            print(s + "!")
```

La première boucle, déroulée, exécute la séquence de commandes suivante.

```
In [ ]: n = 1
        print(n*n)
        n = 2
        print(n*n)
        n = 3
        print(n*n)
        n = 4
        print(n*n)
        n = 5
        print(n*n)
```

Essayez de dérouler la boucle ci-dessous et de prédire le résultat avant d'évaluer le code.

```
In [ ]: P = 1
        for n in range(1,6):
            P = P * n
        print(P)
```

Parfois, en particulier lors du débogage, il est utile d'inspecter chaque étape de la boucle pour voir ce que fait Python. Nous pouvons inspecter la boucle ci-dessus en insérant une commande print dans le champ (scope) de la boucle.

```
In [ ]: P = 1
        for n in range(1,6):
            P = P * n
            print("n is",n,"and P is",P)
        print(P)
```

Ici, nous avons utilisé la commande *print* avec des chaînes et des nombres. Dans Python 3.x, vous pouvez imprimer plusieurs éléments sur la même ligne en les séparant par des virgules. Les éléments à imprimer peuvent être des chaînes (entre guillemets simples ou doubles) et des nombres (int, float, etc.).

```
In [ ]: print("My favorite number is", 17)
```

Si nous déroulons la boucle ci-dessus, la séquence linéaire des commandes interprétées par Python est la suivante.

```
In [ ]: P = 1
        n = 1
        P = P * n
        print("n is",n,"and P is",P)
        n = 2
        P = P * n
        print("n is",n,"and P is",P)
        n = 3
        P = P * n
        print("n is",n,"and P is",P)
        n = 4
        P = P * n
        print("n is",n,"and P is",P)
        n = 5
        P = P * n
        print("n is",n,"and P is",P)
        print (P)
```

Analysons la boucle en détail.

```
P = 1
for n in range(1, 6):
    P = P * n  # commande dans le champ (ou scope) de la boucle.
    print("n is", n,"and P is", P)  # commande dans le champ de la boucle
print(P) # commande hors du champ de la boucle
```

La commande "for" se termine par un deux-points : et les deux **prochaines** lignes sont indentées. Les deux points et l'indentation sont des indicateurs de portée (ou **scope**). La portée de la boucle for commence après les deux points et inclut toutes les lignes indentées. La portée de la boucle for est ce qui se répète à chaque étape de la boucle (en plus de la réaffectation de `n`).

```
In [ ]: P = 1
        for n in range(1,6):
            P = P * n  # this command is in the scope of the loop.
            print("n is",n,"and P is",P)  # this command is in the scope of the loo
        print(P)
```

Si nous modifions l'indentation, cela change la portée de la boucle for. Prévoyez ce que fera la boucle suivante, en la déroulant, avant de l'évaluer.

```
In [ ]: P = 1
        for n in range(1,6):
            P = P * n
        print("n is",n,"and P is",P)
        print(P)
```

Les portées peuvent être imbriquées par une indentation. Essayez de créer une (double) boucle imbriquée qui imprime

```
1 a
```

```
2 a
3 a
1 b
2 b
3 b
```

```
In [ ]: # Ecrire votre boucle ici.
```

Parmi les langages de programmation les plus répandus, Python se distingue par l'utilisation de l'indentation. D'autres langages indiquent une portée avec des accolades ouvertes / fermées, par exemple, et l'indentation n'est qu'une question de style. En nécessitant une indentation pour indiquer la portée, Python supprime efficacement le besoin d'accolades et impose un style lisible.

Nous avons maintenant rencontré des types de données, des opérations, des variables et des boucles. Pris ensemble, ce sont des outils puissants pour le calcul! Faire les exercices suivants pour plus de pratique. Faire ensuite, au choix, les exercices en fin du chapitre 0 de An Illustrated Theory of Numbers – certains peuvent être résolus facilement en écrivant quelques lignes de code Python.

1.7 Exercises

1. Décrivez comment Python interprète la division avec reste lorsque le diviseur et/ou le dividende est négatif.
2. Quel est le reste lorsque 2^{90} est divisé par 91?
3. Combien de multiples de 13 y a-t-il entre 1 et 1000?
4. Combien y a-t-il de multiples impairs de 13 entre 1 et 1000?
5. Quelle est la somme des nombres de 1 à 1000 ?
6. Quelle est la somme des carrés, de 1^2 à 1000^2?