Numérique et sciences informatiques
Classe de première

A Langages et programmation
1 Premières notions de programmation en Python

I Les différents types d'objets et les opérations associées

• Ouvrons IDLE et la console comme une calculatrice.

I.1 Les opérations sur les nombres

a) Écriture des nombres et première opérations

• Écriture des nombres en décimal (avec les 10 chiffres habituels) :
* les entiers :

`– 267`

* les « nombres à virgules » :

`7.01` ou `+ 9.004`

! Utilisez le signe '.' pour la virgule !
Le fait de taper des opérations avec des nombres écrits avec une virgule ne génère pas de message d'erreur mais donne des résultats étranges. Ceci est dû au fait que les virgules sont dotées d'une autre signification que nous verrons plus tard.

* les « nombres à virgules » en écriture scientifique : ex.

`4.78e7` ou `4.78E7`

Remarque : Nous verrons plus tard qu'il est aussi possible d'écrire les nombres en binaire (base 2) ou en hexadécimal (base 16).

• On écrit les quatre opérations avec les caractères habituels : $+$, $-$, $*$ et $/$

• L'opération d'exponentiation (ou « puissance ») s'écrit avec l'opérateur `**` :

`78**3` signifie 78^3

• Le message d'erreur "`ZeroDivisionError`" apparaît lorsqu'on divise par 0 ou bien lorsqu'on élève 0 à une puissance négative.
Exemple :

```
>> 0**-5
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    0**-5
ZeroDivisionError: 0.0 cannot be raised to a negative
power
```

• Dans le cas d'une division, le résultat est parfois une valeur approchée.
Exemple :

```
>>> 2/3
0.6666666666666666
```

b) Deux types de nombre

• En Python, les objets que nous créons et manipulons ont tous un « type » : il appartiennent à une certaine catégorie qui définit leurs propriétés ainsi les opérations et les fonctions que

l'on peut leur appliquer. La fonction **`type(...)`** permet de connaître le type de chaque objet.

• Les nombres (dans l'utilisation la plus courante) sont répartis dans deux catégories :
* les objets de type **`int`** permettent de calculer sur les entiers (« int » est une abréviation de *integer* : entier en anglais) **;**
* les objets de type **`float`** permettent de calculer avec des « nombres à virgule » : (« float » est une abréviation de *floating point number* : nombre à virgule flottante).

L'interpréteur Python crée un nombre en lui attribuant un de ces deux types selon la façon dont il a été écrit.

Exemples :

```
>>> type(- 478)
<class 'int'>
>>> type(- 8.9)
<class 'float'>
>>> type(12.0)
<class 'float'>
>>> type(7e5)
<class 'float'>
```

Remarque : Un objet peut donc être de type **`float`** *même si sa valeur est un nombre entier* ! Son type dépend de la façon dont il a été écrit plutôt que de sa valeur.

• Pour les opérations d'addition, de soustraction, de multiplication :
* si les deux opérandes sont de type **`int`** alors le résultat est de type **`int`**.
* si au moins l'un des deux opérandes est de type **`float`** alors le résultat est de type **`float`**.

• Pour la division, le résultat est toujours de type **`float`**. Ceci est vrai même si les deux opérandes sont de type **`int`** et si le résultat est un entier.

Exemples :

```
>>> type(12/4)
<class 'float'>
>>> 12/4
3.0
```

• Pour l'exponentiation de la forme $a ** b$:
* si le nombre a est de type **`int`** et le nombre b est de type **`int`** *et positif* alors le résultat est de type **`int`** ;
* dans tous les autres cas, le résultat est de type **`float`**.

c) La division euclidienne

• Un rappel mathématique sur la division euclidienne :

Définition : Soient a et b deux nombres entiers où $b > 0$. Alors il existe un unique couple d'entiers $(q\ ;\ r)$ tel que

$$a = b \times q + r \text{ et } 0 \leq r < b$$

q est appelé le **quotient** et r le **reste** de la **division euclidienne** de a par b.

• En Python, si les nombres a et b sont de type **`int`** et $b > 0$ alors on obtient le résultat de la division euclidienne de a par b de la façon suivante :
* $a//b$ donne le quotient ;
* $a\%b$ donne le reste.

Exemples :

```
>> 57//4
14
>>> 57%4
```

```
1
>>> 4*14+1
57
```

• Si on effectue une division euclidienne par 0 on obtient un message d'erreur « `ZeroDivisionError` ».

Exemple :

```
>>> 5%0
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    5%0
ZeroDivisionError: integer division or modulo by zero
```

Remarque : Nous considérerons que ces opérateurs // et % ont vocation à n'être utilisé que lorsque l'on cherche le résultat d'une division euclidienne « ordinaire ». Si l'on applique les opérateurs // et % avec des opérandes qui ne sont pas de type **`int`** ou avec une deuxième opérande strictement négative, on n'obtient quand même un résultat. Mais, en pratique, nous éviterons de le faire car il est alors plus difficile à interpréter.

d) Les parenthèses et la priorité des opérations

• Les règles de priorité des opérations telles qu'elles sont utilisée dans les expressions mathématiques sont respectées en python comme dans tous les langages de programmation.

• Les parenthèses déterminent les opérations effectuées en premier.

• En l'absence de parenthèses :
  * la multiplication et la division sont prioritaires sur l'addition et la soustraction
  * l'opérateur d'exponentiation ** est prioritaire sur les autres opérations

Exemples :

```
>>> 8+3*5
23
>>> 5*2**3
40
```

I.2 Les chaînes de caractères

• Écrire une chaîne de caractère : on l'encadre par deux caractères ' ou bien deux caractères ".

• Le type d'une chaîne de caractère est **`str`**.

Exemple :

```
>>> type("bonjour tout le monde")
<class 'str'>
```

L'objet "89" est de type **`str`** et non de type **`int`** !

• Comment traiter les cas où la chaîne de caractère contient les caractères ' ou les caractères " ou bien les deux ?
  * Si la chaîne de caractères ne contient que des caractères ', on peut encadrer avec des ".
  * Si la chaîne de caractères ne contient que des caractères ", on peut encadrer avec des '.
  * Si la chaîne de caractères contient un caractère ' ou ", on peut les faire précéder du caractère \ (anti-slash) pour indiquer qu'il ne sert pas ici à fermer la chaîne de caractères.

```
>>> message = "Il m'a dit : \"C'est la première fois que je vois ça
!\" en agitant les mains..."
```

```
>>> print(message)
Il m'a dit : "C'est la première fois que je vois ça !" en agitant les
mains...
```

• La fonction **`len(...)`** permet de déterminer le nombre de caractères que comporte une chaîne de caractères. Attention, le caractère `espace` « » et tous les signes de ponctuations sont aussi comptés comme des caractères. La fonction **`len(...)`** renvoie un résultat de type **`int`**.

Exemple :

```
>>> len("Bonjour à tous, et bienvenu à ce cours de NSI !")
47
```

• Pour écrire le caractère « fin de ligne » dans une chaîne de caractère on utilise `\n`.

```
>>> maListe ="* Brésil\n* Afrique du sud\n* Inde\n* etc."
>>> print(maListe)
* Brésil
* Afrique du sud
* Inde
* etc.
```

• Deux opérations sur les chaînes de caractère :

* la concaténation s'effectue avec l'opérateur + si les opérandes sont tous deux de type **`str`**. Attention cette opération n' est pas commutative : l'ordre des opérandes change le résultat.

Exemples :

```
>>> "comment allez vous " + 'ce matin ?'
'comment allez vous ce matin ?'
>>> 'ce matin ?' + "comment allez vous "
'ce matin ?comment allez vous '
```

* la « démultiplication ».

Exemples :

```
>>> 3*"la"
'lalala'
>>> 3.0*"la"
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    3.0*"la"
TypeError: can't multiply sequence by non-int of type
'float'
```

Si on essaie d'additionner une chaîne de caractère et un nombre, on obtient un message d'erreur « TypeError ».

Exemple :

```
>>> 2 + "bonjour"
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    2 +"bonjour"
TypeError: unsupported operand type(s) for +: 'int' and
'str'
```

II Variables et affectations

II.1 La notion de variable et d'affectation

a) Qu'est-ce qu'une variable ?

• Un programme consiste à effectuer successivement une multitude d'opérations sur des objets (nombres, chaînes de caractères, booléens, etc.) en vue de résoudre un problème, effectuer une certaine tâche globale, etc.

• Pour cela, nous avons besoin de mémoriser le résultat de nos opérations pour pouvoir les réutiliser dans les opérations suivantes.

• En informatique, l'emplacement où l'on stocke le résultat d'une opération en vue de le réutiliser s'appelle une **variable**.

• Du point de vue du programmeur, une variable possède deux caractéristiques fondamentales :
* un **nom** : une suite de caractère ;
* une **valeur** : un objet qui lui est associé. Une variable « référence » un objet.

• Le nom d'une variable sera fixe, il définit son identité : *dans un contexte donné*, une seule variable peut porter ce nom et un autre nom correspondra toujours à une autre variable.

• Comme l'indique le terme 'variable', la valeur associée à une variable, c'est-à-dire l'objet qu'elle référence, est susceptible de changer au cours de l'exécution d'un programme.

• En python, la nouvelle valeur peut même être d'un autre type que celui de l'ancienne valeur. On dit donc que Python est un langage de programmation à **typage dynamique**.

b) Création d'une variable et première affectation

• Comment créer une variable en Python ? En tapant un nom de variable et en lui associant une valeur :

```
maVar = 53
```

• SI le programme rencontre ce nom de variable pour la première fois, il va faire deux choses :
* créer une variable en lui donnant le nom 'maVar' ;
* lui associer l'objet 53 de type **`int`**

• L'opération qui consiste à donner une valeur à une variable (que ce soit pour la première fois ou par la suite) s'appelle une **affectation**. Le signe qui demande à l'ordinateur d'effectuer cette affection est =.

Il ne faut surtout pas confondre le signe = qui permet d'effectuer une affectation avec le signe == qui permet de savoir si deux objets sont égaux.

• La syntaxe d'une affectation :
  * *à gauche* du signe =, on place le nom de la variable à modifier ;
  * *à droite* du signe =, on place une valeur ou une opération dont le résultat sera affecté à la variable.

Exemples :

```
>>> monTexte = "bonjour tout le monde !"
>>> monNombre = 7*9 + 18
```

On ne peut donc pas placer autre chose qu'un nom de variable à gauche de l'opérateur d'affectation =.

Exemple :

```
>>> 47 = uneVar
SyntaxError: can't assign to literal
```

II.2 Utilisation d'une variable déjà créée

a) Exploiter la valeur d'une variable ou modifier cette valeur

• Une fois qu'une variable a été créée, on peut ensuite faire à nouveau référence à cette variable, que ce soit :

* pour utiliser la valeur qu'elle contient dans un nouveau calcul ;
* ou pour lui affecter une nouvelle valeur ;

Exemples :

```
>>> rayon = 5.2
>>> aireCercle = 2*3.14*rayon**2
>>> rayon =7.4
>>> diametre = 2*rayon
```

• On peut utiliser une variable dans une instruction consistant à affecter une nouvelle valeur à une variable à l'aide d'un calcul faisant intervenir l'ancienne valeur.

Exemple :

```
>>> maVar = 12
>>> maVar = maVar**2
>>> maVar
144
```

• Dans la console IDLE, la variable continue à « exister » (c'est-à-dire à avoir un sens pour l'interpréteur) tant que la console n'a pas été fermée. Dans un programme, elle existe tant que le programme n'a pas entièrement été exécuté.

• Suivons pas à pas l'exécution d'une instruction telle que :

```
>>> aireCercle = 2*3.14*rayon**2
```

* d'abord, il calcule le résultat de l'opération `2*3.14*rayon**2` en récupérant la valeur associée à la variable `rayon` à ce moment précis de l'exécution du programme. Si la variable n'existe pas encore, il arrête l'exécution et envoie un message d'erreur.
* ensuite, si l'interpréteur rencontre le nom de variable `aireCercle` pour la première fois, il crée une variable de ce nom ;
* enfin, il affecte le résultat du calcul à cette variable.

• Ceci fonctionne de la même façon pour l'instruction

```
>>> maVar = maVar**2
```

Ici l'instruction ne pourra être exécutée que si la variable `maVar` existe déjà.

b) Afficher à l'écran la valeur d'une variable

• Comment connaître la valeur associée à une variable à un moment donné ? La réponse n'est pas la même selon que l'on se place dans la console IDLE ou dans le cadre de l'exécution d'un programme

* dans la console IDLE : il suffit de taper le nom de la variable puis sur la touche « entrer » ;
* dans le cadre de l'exécution d'un programme, on utilise une fonction **`print(...)`** pour afficher sa valeur à l'écran.

II.3 Règles à respecter pour créer un nom de variable en Python

• Un nom de variable doit respecter les règles suivantes :
  * être composé de lettres minuscules ou majuscules (de `a` à `z`), de chiffre (de `0` à `9`) ou du caractère `_` ;
  * ne pas commencer par un chiffre ;
  * ne pas être un mot-clé du langage Python (`if`, `else`, `elif`, `for`, `while`, …).

Exemple :

```
>>> 1re4 = 35
SyntaxError: invalid syntax
```

• En particulier, on ne peut pas placer de caractère − dans un nom de variable.
• Le langage Python permet de créer des variables avec des lettres accentuées mais *une bonne pratique consiste à ne pas faire usage de cette possibilité*. En effet, la plupart des langages de programmation refusent ce type de caractère.

• L'utilisation des noms de variables dans un programme explique pourquoi une chaîne de caractère doit toujours être écrite entre deux signes " ou deux signes '. En effet, toute autre succession de caractères alphanumériques *est interprétée comme un nom de variable ou un mot-clé du langage de programmation*.

La liste des mots-clés en Python 3

`and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield`.

II.4 Deux opérations d'affectation utiles en Python

a) Affectation multiple

• En Python, il est possible d'affecter *simultanément* des valeurs à deux variables différentes en utilisant les virgules. Cette opération s'effectue avec la syntaxe suivante :

```
var1, var2, …, varN = c1, c2, …, cN
```

A l'exécution de cette instruction, la variable *var1* recevra la valeur *c1*, la variable *var2* recevra la valeur *c2*, *… et* la variable *varN* recevra la valeur *cN*.

On peut utiliser autant de variables que l'on souhaite mais il est indispensable qu'il y ait exactement *autant* de variables à gauche du signe = que de valeurs à droite.

Exemple :

```
>>> message, nombre = 'coucou', 421
>>> message
'coucou'
>>> nombre
421
```

• L'affectation multiple permet notamment de faire un échange de valeurs entre deux variables *var1* et *var2* très facilement grâce à l'instruction suivante :

```
var1, var2 = var2, var1
```

Exemple :

```
>>> a, b =5, 12
>>> a, b = b, a
```

```
>>> a
12
>>> b
5
```

b) Opération sur une variable

• Si on veut augmenter une variable `var` d'une valeur `ajout` donnée on peut écrire l'instruction

```
var = var + ajout
```

mais, en Python, il existe une façon plus condensée d'écrire cette instruction :

```
var += ajout
```

• De la même façon, l'instruction

```
var = var * facteur
```

peut s'écrire :

```
var *= facteur
```

• On peut aussi utiliser de façon similaire les opérateurs `-=`, `/=`, `**=`, `//=` et `%=` mais il faut faire bien attention au caractère *non commutatif* de ces opérateurs.
Par exemple, `nb -= retrait` signifie `nb = nb - retrait` et non pas `nb = retrait - nb`.
Exemple :

```
>>> nb = 12
>>> nb /= 4
>>> nb
3.0
>>> nb -= 5
>>> nb
-2.0
```

• De même, on peut utiliser l'opérateur `+=` sur une chaine de caractère mais il faut prendre garde au fait que `message += suite` signifie `message = message + suite` et non pas `message = suite + message`
Exemple :

```
>>> texte = "bonjour"
>>> texte += " à tous"
>>> texte
'bonjour à tous'
```