Informatique en CPGE (2014-2015)
Résumé Python

1 Eléments de base

1.1 La fonction print

La fonction `print` permet d'afficher. Exemples : `print('bonjour')` ou `print(25+36)` ou `print('La somme est égale à',25+36)` ou encore `print('La somme est',x)` si x est une variable définie auparavant.

1.2 Types numériques

- le type `int` pour représenter les nombres entiers illimités ;
- le type `float` pour représenter les nombres réels et on écrit par exemple 5.2 ou -26.721. Les nombres du type `float` sont stockés dans la machine sous la forme de "nombres en virgule flottante" entre $-1,7 \times 10^{308}$ et $1,7 \times 10^{308}$.
- le type `bool` pour représenter les valeurs booléennes `True` et `False` ;
- le type `None` qui n'a qu'une seule valeur.

La fonction `type` permet de déterminer le type d'un objet ; les fonctions `int` et `float` permettent de convertir si c'est possible un objet d'un autre type respectivement en type `int` ou type `float`.

1.2.1 Opérations sur les types `int` et `float`

- l'addition + , la soustraction − , la multiplication * , l'exponentiation ** dont le résultat est un `float` si l'un des termes est un `float` et sinon un `int` ;
- la division / dont le résultat est toujours un `float` ;
- la division entière // , l'opération modulo % : utilisés avec des nombres entiers, on obtient le quotient et le reste, de type `int`, dans la division euclidiennne.
- `x+=y` est équivalent à `x=x+y` ; ce raccourci fonctionne de la même manière avec toutes les opérations ci-dessus.

1.2.2 Comparaison et opérateurs booléens

`x==y` renvoie `True` si x et y sont égaux.
`x!=y` renvoie `True` si x et y ne sont pas égaux.
Les opérateurs < , <= , > , >= ont le sens usuel.

Opérations logiques :
`a and b` donne `True` si a et b sont `True` et `False` sinon ;
`a or b` donne `False` si a et b sont `False` et `True` sinon ;
`not a` donne `True` si a est `False` et donne `False` si a est `True`.

1.3 Variables et affectation

Une **variable** permet d'associer un nom avec un objet. Si on écrit `pi = 3.14159`, on lie le nom `pi` à un objet de type `float` ; on a alors une variable dont le nom est "pi" et la valeur est 3,14159.

Une **affectation** associe le nom à gauche du signe = avec l'objet représenté par l'expression qui est à droite du signe = ; par exemple `aire=pi*5**2`.

Python autorise des affectations multiples : `x,y=3,7` associe x à 3 et y à 7.
Ceci permet de faire l'échange : `x,y=y,x` de manière très simple (et bien utile !).

1.4 Type str et fonction input

1.4.1 Type str

Le type `str` (string) est utilisé pour les chaînes de caractères. On utilise des guillemets simples ou doubles (`c='bonjour'`) ou on utilise la fonction `input` pour récupérer ce qui est entré au clavier.

1.4.2 Fonction input

La fonction `input` prend un argument (en général une chaîne de caractères) qui sera affiché à l'utilisateur) ; ce que l'utilisateur entrera au clavier pourra être stocker dans une variable toujours de type `str`.

```
>>>nom=input('Entrer votre nom :')
Entrer votre nom : toto
>>> print(nom)
toto
```

Pour récupérer un nombre, il sera nécessaire de procéder à une conversion à posteriori ou en même temps, par exemple : `a=int(input('Entrer un nombre entier : '))`.

1.4.3 Méthodes

`c.count(c1)` compte combien de fois la chaîne c1 apparait dans c ;
`c.find(c1)` renvoie l'index de la première occurence de la chaîne c1 dans c et renvoie -1 si c1 n'est pas dans c ;
`c.lower()` convertit toutes les lettres majuscules en minuscules ;
`c.upper()` convertit toutes les lettres minuscules en majuscules ;
`c.replace(ancien,nouveau)` remplace toutes les occurences de la chaîne `ancien` par la chaîne `nouveau` ;
`c.rstrip()` supprime l'espace à la fin d'une chaine ;
`c.split(sep)` coupe la chaîne c (`sep` est le délimiteur) et renvoie une liste de sous-chaînes de c. Si `c='bonjour, ça va?'`, alors `c.split(',')` renvoie `['bonjour', 'ça va?']`.

2 Instructions conditionnelles et boucles

L'**indentation** est capitale. Toutes les instructions au même niveau d'indentation appartiennent au même bloc.

2.1 Instructions conditionnelles

```
if (conditions1) :
    actions1
elif (conditions2) :
    actions2
...
else :
    actionsn
```

2.2 Boucles itératives conditionnelles

```
while (conditions) :
    actions
```

2.3 Boucles itératives non conditionnelles

```
for i in range(...) :
    actions
```

L'instruction `break` permet d'interrompre une boucle.
L'instruction `continue` permet d'éviter un passage dans la boucle.

3 Les types composés

3.1 les n-uplets

Ce sont les objets de type `tuple`. Les éléments sont indexés par un entier commençant à 0. On ne peut pas en modifier un élément.

`()` un tuple vide ;
`(5,)` un tuple à un élément ; (attention, la virgule est obligatoire)
`(5,'bonjour')` les éléments peuvent être de type diférents.

3.2 Les listes

Ce sont les objets de type `list`. Les éléments sont indexés par un entier commençant à 0. On peut en modifier les éléments.

`[]` une liste vide ;
`[5]` une liste à un élément ;
`[5,'bonjour']` les éléments peuvent être de type diférents.

Méthodes :
`L.append(x)` ajoute l'objet x à la fin de la liste L.
`L.count(x)` renvoie le nombre d'occurences de l'objet x dans la liste L.
`L.insert(i,x)` insère l'objet x dans la liste L à l'index i.
`L.remove(x)` supprime la première occurence de l'objet x dans la liste L.
`L.index(x)` renvoie l'index de la première occurence de l'objet x dans la liste L.
`L.pop(i)` supprime l'objet d'index i dans la liste L et le renvoie.
`L.sort()` trie les éléments de L.
`L.reverse()` inverse l'ordre des éléments de L.

3.3 Opérations sur les types itérables

Ces opérations sont applicables sur les objets de type `str`, `tuple` et `list`.

`for x in c` itère sur les éléments de c.
`x in c` teste si x est contenu dans c.
`c[i]` renvoie le (i+1)-ème élément de c. `c[0]` renvoie le premier élément.
On peut aussi indexer à partir du dernier élément qui est `c[-1]` ;
`c[-2]` renvoie l'avant-dernier élément de c, `c[-3]` ...
`c[début:fin]` renvoie une partie de c qui commence à l'index `début` et se termine à l'index `fin-1`.
`len(c)` renvoie la longueur de c.
`c1+c2` concatène c1 et c2.
`n*c` concatène n copies de c.

4 Les fonctions

4.1 Définition d'une fonction

```
def nomDeLaFonction(arguments):
    """ aide sur la fonction (facultatif) """
    corpsDeLaFonction
```

def est un mot clé du langage. Les arguments sont séparés par des virgules.
Le corps de la fonction est un bloc de code indenté par rapport à la ligne de définition.

Si le corps de la fonction contient l'instruction `return`, alors l'appel de la fonction est une expression qui a donc une valeur. S'il n'y a pas d'instruction `return` dans le corps de la fonction, alors l'appel de la fonction a la valeur "None". Ce type de fonction s'appelle une procédure.

4.2 Espace et portée des variables

4.2.1 Espace local

Cet espace contient les paramètres qui sont passés à la fonction et les variables définies dans son corps. Si la variable x n'existe pas dans l'espace local de la fonction, Python va regarder dans l'espace local dans lequel la fonction a été appelée, et là, s'il trouve bien la variable x il peut l'utiliser.

4.2.2 Portée d'une variable

Une fonction ne peut pas modifier, par affectation, la valeur d'une variable extérieure à son espace local. Dans le code suivant, la variable x utilisée dans la fonction est distincte de la variable x définie au début du programme (x=3) et n'existe plus après l'appel de la fonction. Après l'instruction `f(x)`, l'espace local de la fonction f est détruit.

```
x=3
def f(x):
    x+=2
    print(x)
f(x)  # affiche 5
print(x)  # affiche 3
```

De même la fonction d'échange suivante ne produit pas le résultat espéré :

```
a=2
b=5
def echange(x,y):
    x,y=y,x
echange(a,b)
print(a,b) #affiche 2 5
```

4.2.3 Variables globales

Il existe un moyen de modifier avec une fonction des variables extérieures à celle-ci. On utilise pour cela des **variables globales** avec le mot-clé **global**.

Le code suivant permet de modifier la variable x extérieure à la fonction.

```
x=3
def f():
    global x
    x+=2

print(x) #affiche 5
```

5 Les fichiers

5.1 Ouverture d'un fichier

On utilise la fonction **open** qui prend comme premier paramètre le nom du fichier et en second paramètre 'w' pour le mode "écriture", 'r' pour le mode "lecture" et 'a' pour le mode "ajout".

```
fic1=open('fichier1','w')
fic2=open('fichier2','r')
fic3=open('fichier3','a')
```

5.2 Fermeture d'un fichier

La fermeture d'un fichier est obligatoire, il y a une seule instruction :
`fic.close()`

5.3 Ecriture et lecture

5.3.1 Ecriture

On écrit des chaînes de caractères (type **str**) en utilisant la méthode **write**.
`fic.write('Bonjour, comment allez-vous?')`
Si on souhaite écrire sur plusieurs lignes, on utilise le caractère '\n' pour un retour à la ligne.

Si on relance le même programme, le fichier qui existe déjà est écrasé et réécrit. Pour ne pas l'écraser et écrire à la suite dans ce fichier, on l'ouvre avec l'instruction `fic=open('fichier1.txt','a')`.

Si les données à écrire sont de type **int** ou **float**, il suffit de les convertir préalablement en type **str**.

5.3.2 Lecture

Pour lire dans un fichier texte, on ouvre le fichier en lecture et on utilise la méthode **read**.
`fic.read(n)` lit n caractères, `fic.read()` lit tout le fichier.
L'objet "fic" est un ensemble de lignes, chacune étant une chaîne de caractères. On peut récupérer une ligne dans une variable de type **str** avec le code suivant :
`ligne=fic.readline()` lit la ligne courante et passe à la suivante.
On peut aussi récupérer toutes les lignes dans une liste. : `lignes=fic.readlines()`
On peut itérer sur chaque ligne : `for ligne in fic`

Si les données à récupérer sont des nombres, il suffit de convertir chaque mot en **int** ou en **float**. Si "fichier.dat' contient des lignes de deux flottants séparés par une tabulation ('\t'), le code est :

```
fic=open('fichier.dat','r')
for ligne in fic:
    coord=ligne.rstrip().split('\t') # coord est une liste de 2 mots
    x,y=float(coord[0]),float(coord[1])
```