Exercices d'informatique - Série n° 2

Cours 3IN OC01 Série distribuée le 19.9.2017

1. Les fonctions *divmod* et *pow*.

Déterminer l'action des fonctions prédéfinies divmod et pow en exécutant le script suivant:

```
1  a=11
2  b=2
3  print divmod(a,b)
4  print pow(b,a)
5  print pow(b,a,3)
6  print (b**a)%3
```

2. Opérations sur les listes de nombres.

Déterminer l'action des fonctions min, max et sum sur les listes de nombres en exécutant le script suivant:

```
1  l=[-2,3.14,37,-786.2]
2  #
3  print l
4  print min(l)
5  print max(l)
6  print sum(l)
```

3. Opérations sur les listes.

Déterminer l'action de la fonction len et des opérations + et * sur les listes en exécutant le script suivant:

```
1  l1=[1,2,"bonjour"]
2  l2=[[3,list("texte")],[],7.56,"a"]
3  #
4  print l1
5  print l2
6  print len(l1), len(l2)
7  #
8  l=l1+l2
9  print l
10 #
11 l3=2*l1
12 print l3
```

4. Modification d'une liste.

Déterminer l'effet de chaque ligne du script suivant:

```
1  l=[1,2,3,4]
2  print l
3  #
4  l[1:3]=[22,33,44]
5  print l
6  #
7  l[1:4]=[8,9]
8  print l
9  #
10 l[0:2]=[]
11 print l
12 #
13 l=[123,156]+l
```

```
14 print l
15 #
16 del l[0:2]
17 print l
18 #
19 l[1:1]=list("texte")
20 print l
21 #
22 l[:]=[1,2]
23 print l
```

5. Modification d'une liste.

Déterminer l'effet de chaque ligne du script suivant:

```
1  l=list("Un texte")
2  print l
3  #
4  print l.count("e")
5  print l.index("t")
6  #
7  print l.append("vraiment petit")
8  print l
9  #
10 print l.extend(list("vraiment petit"))
11 print l
12 #
13 print l.insert(3,"Z")
14 print l
15 #
16 print l.remove("e")
17 print l
18 #
19 l=list("Un texte")
20 print l
21 #
22 print l.pop(5)
23 print l
24 #
25 l=list("Un texte")
26 print l
27 print l.reverse()
28 print l
```

6. La méthode *sort*.

Déterminer l'action de la méthode sort sur une liste en exécutant le script suivant:

```
1 l=list("Un petit texte")
2 print l
3 l.sort()
4 print l
5 #
6 l=["z",178,-2.34,1,123,"a",["a",23,-123]]
7 print  l
8 l.sort()
9 print l
```

7. Exceptions.

(1) Exécuter le script suivant:

```
1 n=raw_input('Tapez un nombre: ')
2 print int(n)
```

Que se passe-t-il si l'utilisateur ne tape pas un nombre entier mais des lettres ?

(2) Exécuter le script suivant:

```
1 n=raw_input('Tapez un nombre entier: ')
2 try:
3     print int(n)
4 except:
5     print "Ce n'est pas un nombre entier !"
```

Expliquer l'action des instructions `try` et `except`.

(3) Rédiger un programme qui demande à l'utilisateur de taper un nombre (entier ou pas). Si l'utilisateur tape un nombre, le programme affiche ce nombre et s'arrête. Sinon, il demande à nouveau à l'utilisateur de taper un nombre. Le programme ne s'arrête pas tant que l'utilisateur n'a tapé un nombre. Un exemple d'utilisation du programme se trouve sur la figure 1.

```
Tapez un nombre: 3675t
Ce n'est pas un nombre !
Tapez un nombre: 232zut32bg
Ce n'est pas un nombre !
Tapez un nombre: 2344.56
2344.56
```

FIGURE 1. Exercice 7

8. Monnaie optimale.

Soit C une liste de valeurs de pièces de monnaie. Par exemple, $C = \{1, 2, 5, 10, 20, 50, 100\}$. Étant donné un entier x, on aimerait trouver un ensemble de pièces choisies dans la liste, dont la somme des valeurs donne x. Par exemple, si $x = 325$, on a $x = 3 \cdot 100 + 1 \cdot 20 + 1 \cdot 5$. L'algorithme le plus simple permettant de résoudre ce problème s'appelle l'algorithme glouton. Son pseudo-code est donné par:

Données:	$C = \{1, 2, 5, 10, 20, 50, 100\}$ et x des entiers
$v \leftarrow 0$	
$t \leftarrow 0$	
Tant que	$v < x$ **effectuer**:
	Trouver le plus grand $c \in C$ tel que $c + v \leq x$
	$c_t \leftarrow c$
	$v \leftarrow v + c$
	$t \leftarrow t + 1$
Fin	
Retourner:	$c_0, c_1, \cdots c_{t-1}$

Transcrire cet algorithme dans le langage *Python*.

9. La factorielle.

Rédiger un script qui permette de calculer la factorielle d'un nombre en utilisant une boucle. **Rappel**: par exemple, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Voici un exemple de fonctionnement du programme:

```
n=3.14
Vous devez entrer un nombre entier !
```

```
n=a
Vous devez entrer un nombre entier !
n=5
n!= 120
```

10. L'instruction *def*.

Déterminer l'action de l'instruction **def** en exécutant les scripts suivants.

```
1 def bernard():
2     print "Bonjour"
```

```
1 def bernard():
2     print "Bonjour"
3 bernard()
```

11. L'instruction *def*.

Exécuter le script suivant:

```
1 def bernard(texte):
2     print texte
3 bernard("bonjour")
```

Que se passe-t-il si on remplace la ligne `bernard("Bonjour")` par `bernard()` ?

12. L'instruction *return*.

Exécuter le script suivant et déterminer l'action de chaque instruction.

```
1 def double(x):
2     y=2*x
3     return y
4 z=double(28)
5 print z
```

13. Les paramètres facultatifs d'une fonction.

Exécuter le script suivant et déterminer l'action de chaque instruction.

```
1 def essai(x,y=0):
2     return x+y
3 print essai(21)
4 print essai(21,2)
```

14. Attributs d'une fonction.

Exécuter le script suivant et déterminer l'action de chaque instruction.

```
1 def essai(x,y=0,z=1):
2     "Cette fonction calcule le double d'un nombre et ajoute 1"
3     return 2*x+y+z
4 print essai(21)
5 print essai.func_name
6 print essai.func_defaults
7 print essai.func_doc
```

15. Attributs d'une fonction.

Exécuter le script suivant et déterminer l'action de chaque instruction.

```
1 def essai(x):
2     essai.compteur=essai.compteur+1
3     return 2*x
4 essai.compteur=0
5 print essai(21)
6 print essai.compteur
7 print essai(33)
8 print essai.compteur
```

16. Variables locales.

Exécuter le script suivant et déterminer l'action de chaque instruction.

```
1 def essai(x):
2     y=2
3     print "Dans le corps de la fonction a=",a
4     return a*x+y
5 y=0
6 a=2
7 print essai(3)
8 print "y=",y
```

17. Variables locales.

Exécuter le script suivant:

```
1 def f():
2     y=2
3     print "y=",y
4 y=1
5 f()
6 print "y=",y
```

Expliquer pourquoi les deux valeurs affichées pour la variable `y` sont différentes.

18. Suites de Syracuse.

Soit f la fonction de $\mathbb{N}^*$ dans $\mathbb{N}^*$ définie par

$$f(n) = \begin{cases} \frac{n}{2} & \text{si } n \text{ est pair} \\ 3n+1 & \text{si } n \text{ est impair} \end{cases}$$

Pour $n_0 \in \mathbb{N}$, on définit une suite de nombres entiers comme suit:

$$n_1 = f(n_0),\ n_2 = f(n_1),\ n_3 = f(n_2),\ \cdots\ n_{k+1} = f(n_k)$$

Cette suite est appelée la suite de Syracuse de n_0.

(1) Calculer les 100 premiers éléments de la suite de Syracuse de 1.
(2) Rédiger un programme qui affiche les éléments u_k de la suite de Syracuse d'un nombre donné par l'utilisateur pour k entre 0 et k_0, où k_0 est le plus petit nombre pour lequel $u_{k_0} = 1$. Définir une fonction avec l'instruction **def** et utiliser une boucle **while**.
(3) La conjecture de Syracuse, appelée aussi problème $3x+1$, est l'hypothèse selon laquelle toutes les suites de Syracuse atteignent 1, en d'autres termes: pour tout nombre $n_0 > 0$, il existe k_0 tel que $u_{k_0} = 1$. Tester la conjecture de Syracuse pour quelques valeurs de n_0. La conjecture de Syracuse n'est toujours pas démontrée.

19. Suites de Syracuse.

Soit u_k la suite de Syracuse d'un nombre u_0. On définit

(1) **Le temps de vol** de la suite: c'est le plus petit nombre k tel que $u_k = 1$.
(2) **L'altitude maximale** de la suite: c'est le terme u_k le plus grand de la suite.
(3) **Le temps de vol en altitude** de la suite: c'est le plus petit nombre k tel que $u_{k+1} < u_0$.

Rédiger un programme qui affiche l'altitude maximale, le temps de vol et le temps de vol en altitude de la suite de Syracuse d'un nombre donné par l'utilisateur. Définir une fonction avec l'instruction **def** et utiliser une boucle **while** contenant seulement une ligne. Utiliser des attributs de la fonction.

20. L'instruction *global*.

Déterminer l'action de l'instruction **global** en exécutant les scripts suivants.

```
1 def gnu():
2     a=2
3     print "Dans le corps de la fonction, a=",a
4 a=0
5 gnu()
6 print "Dans le corps du programme, a=",a
```

```
1 def gnu():
2     global a
3     a=2
4     print "Dans le corps de la fonction, a=",a
5 a=0
6 gnu()
7 print "Dans le corps du programme, a=",a
```

21. L'instruction *import*.

Pour des programmes volumineux (surtout si ceux-ci sont rédigés par plusieurs informaticiens), il est important de pouvoir placer des fonctions dans des fichiers distincts du fichier contenant le programme principal. Sauvegarder le script suivant sous le nom "Pythagore.py".

```
1 def hypotenuse(x=0,y=0):
2     return (x**2+y**2)**0.5
```

Puis exécuter le script suivant:

```
1 from Pythagore import *
2 print hypotenuse(3,4)
```

Comparer avec l'exécution des scripts suivants:

```
1 def hypotenuse(x=0,y=0):
2     return (x**2+y**2)**0.5
3 print hypotenuse(3,4)
```

et

```
1 import Pythagore
2 print Pythagore.hypotenuse(3,4)
```

22. *Lambda*-**expressions.**
Exécuter les deux scripts suivants et comparer les résultats.

```
g=lambda x=3,y=4: (x**2+y**2)**(0.5)
print g()
print g(10,15)
```

```
def g(x=3,y=4):
    return (x**2+y**2)**(0.5)
print g()
print g(10,15)
```

23. La factorielle.
Rédiger un script qui permette de calculer la factorielle d'un nombre sans utiliser des boucles mais en définissant une fonction qui s'appelle elle-même. **Rappel**: On peut définir la factorielle par $n! = n \cdot (n-1)!$ et $1! = 1$. Par exemple, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

24. Fonction récursive.
On peut montrer que

$$2 = 1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^k} + \cdots$$

Vérifier ce résultat en rédigeant une fonction `f(n)` qui calcule

$$1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n}$$

La fonction ne doit pas utiliser de boucle mais doit faire appel à elle-même. Voici un exemple de fonctionnement du programme à réaliser:

```
Tapez un nombre entier: n=19
1+1/2+1/4+...+1/n^2= 1.99999809265
```

Indication: On peut définir la fonction `f(n)` par

(1) `f(0)`=1
(2) `f(n)`=`f(n-1)`+$\frac{1}{2^n}$

25. Fonction récursive.
On peut montrer que

$$\frac{\pi^2}{8} = 1 + \frac{1}{9} + \frac{1}{25} + \cdots + \frac{1}{(2k+1)^2} + \cdots$$

Vérifier ce résultat en rédigeant une fonction `f(n)` qui calcule

$$1 + \frac{1}{9} + \frac{1}{25} + \cdots + \frac{1}{(2n+1)^2}$$

La fonction ne doit pas utiliser de boucle mais doit faire appel à elle-même. Voici un exemple de fonctionnement du programme à réaliser:

```
Tapez un nombre entier: n=500
1+1/9+1/25+...+1/(2n+1)^2= 1.23320154831
pi^2/8= 1.23370055014
```

Indication: On peut définir la fonction `f(n)` par

(1) `f(0)`=1

(2) $f(n)=f(n-1)+\frac{1}{(2n+1)^2}$

26. Fonction récursive.

On peut montrer que

$$1 = \frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{k\cdot(k+1)} + \cdots$$

Vérifier ce résultat en rédigeant une fonction $f(n)$ qui calcule

$$\frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{n\cdot(n+1)}$$

La fonction ne doit pas utiliser de boucle mais doit faire appel à elle-même. Voici un exemple de fonctionnement du programme à réaliser:

```
Tapez un nombre entier: n=900
1/(1*2)+1/(2*3)+...+1/(n*(n+1))= 0.998890122087
```

Indication: On peut définir la fonction $f(n)$ par

(1) $f(1)=\frac{1}{2}$

(2) $f(n)=f(n-1)+\frac{1}{n\cdot(n+1)}$