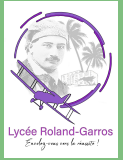


# Python en classe de seconde

Laboratoire de mathématiques du lycée Roland Garros



## Utilisation de la console, les variables

Il y a deux manières d'écrire du Python :

- **Mode interactif dans la console Python**, vous tapez une ligne dans la console puis vous appuyez sur entrer.
- **Dans un fichier texte** (fichier.py), vous écrivez une suite d'instructions (lignes) dans un fichier en vous servant d'un éditeur de texte et dites à Python d'exécuter les instructions du fichier dans la console (appelé aussi shell).

Durant cette séance nous allons seulement utiliser la console.

Évaluez les expressions encadrées ci-dessous, marquez les affichages obtenus et commentez les.

### I) Opérations et types de données

Les trois caractères >>> constituent le signal d'invite, lequel vous indique que Python est prêt à évaluer une expression.

```
>>> 5+3
>>> 2 - 9
>>> 7 + 3 * 4
>>> (7+3) * 4
>>> 20 / 3
>>> 20 // 3
>>> 20 % 3
>>> 17 % 4

>>> 2**3
>>> 2019**2019
>>> 2e3
>>> 8,7 / 5
>>> 5 + 3.0
>>> 0.1 + 0.7
>>> 3 < 5
>>> 5 < 3

>>> 0 < 3 < 8
>>> 1 + 1 = 2
>>> 1 + 1 == 2
>>> age
>>> "age"
>>> sqrt(16)

>>> from math import *
>>> sqrt(16)
```

Il existe en Python des **constantes**. Les valeurs fixes telles que les nombres, les chaînes etc sont appelées « constantes » parce que leur valeur ne change pas. Il y a :

- Les constantes numériques qui sont de type **entier** ou **flottant** (nombre à virgule), Python les reconnaît directement.
- Les **chaînes de caractères** à écrire entre deux symboles d'apostrophe simple ' ou double ".
- Les **booléens** qui sont soit vrai (True), soit faux (False). Ils servent à tester des conditions.

Vous avez fait des opérations sur les entiers (résultats exacts) et les flottants (résultats approchés), Python peut aussi faire des opérations sur des chaînes de caractères (il redéfinit les opérations) et des booléens.

```
>>> "age" + 1
>>> "age" + "1"

>>> "1" + "1"
>>> 3*"age"

>>> "a" in "age"
>>> "b" in "age"

>>> 3<5 and 4<2
>>> 3<5 or 4<2
```

En Python l'expression `type (objet)` permet de connaître le type d'un objet.

```
Entier
>>> type(3)

Flottant
>>> type(3.5)

Chaîne de caractères
>>> type("age")

Booléen
>>> type(3<5)
```

## II) Les variables

Une **variable** est une zone de la mémoire de l'ordinateur dans laquelle une **valeur** est stockée. Aux yeux du programmeur, cette variable est définie par un **nom**, alors que pour l'ordinateur, il s'agit en fait d'une adresse, c'est-à-dire d'une zone particulière de la mémoire. En Python, la **déclaration** d'une variable et son **initialisation** (c'est-à-dire la première valeur que l'on va stocker dedans) se font en même temps.

Quand on affecte un contenu à une variable, elle prend le **type** de l'objet qu'elle contient.

### 1) Noms de variables

Les noms des variables sont des noms que vous choisissez vous-même assez librement.

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (majuscules ou minuscules) et de chiffres, qui doit toujours commencer par une lettre;
- Seules les lettres ordinaires sont autorisées, pas d'espaces etc, seul le caractère spécial `_` est autorisé;
- La casse est significative, prenez l'habitude d'écrire l'essentiel des noms de variables en minuscules, n'utilisez les majuscules qu'à l'intérieur même du nom pour en augmenter la lisibilité, comme dans `tableDesMatières`;
- On ne peut pas utiliser non plus une liste de 33 mots qui sont utilisés par le langage.

### 2) Affectation (ou assignation)

Affecter une variable c'est lui attribuer une valeur. En Python, l'opération d'affectation est représentée par le signe `=`.

```
>>> n = 7
>>> msg = "Bonjour"
>>> pi_2 = 3.14
```

Après avoir effectué ces trois affectations, elles ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- Créer et mémoriser un **nom de variable**;
- Créer et mémoriser une **valeur** particulière;
- Attribuer à la variable le **type** de la valeur;
- Établir un **lien** entre le nom de la variable et sa valeur correspondante.

Il faut bien comprendre qu'il ne s'agit en aucune façon d'une égalité, Python aurait pu choisir un autre symbolisme, tel que  $n \leftarrow 7$ . Pour affecter 7 à n, on écrit `n = 7` et on peut lire « n reçoit 7 », on n'écrira pas `7 = n` !

La variable `n` peut ne pas toujours contenir la valeur 7, on peut la réaffecter en écrivant par exemple `n = 5`.

### 3) Afficher la valeur d'une variable

À la suite de ce que l'on vient de faire, on dispose donc de trois variables : `n`, `msg` et `pi_2`.

Pour afficher leur valeur à l'écran, il existe deux possibilités :

- Une variable est aussi une expression. Pour l'évaluer on entre son nom puis « entrer » et le résultat de l'évaluation est la valeur de la variable :

```
>>> n
>>> msg
>>> pi_2
```

- On utilise la fonction `print ()` (qui servira à l'intérieur d'un programme et aussi à afficher des expressions) :

```
>>> print (n)
>>> print (msg)
>>> print (pi_2)
```

#### 4) Affectations multiples

Sous Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple :

```
>>> x = y = 7
>>> x
>>> y
```

On peut aussi effectuer des affectations parallèles à l'aide d'un seul opérateur :

```
>>> a, b = 4, 8.33
>>> a
>>> b
```

#### 5) Opérateur et expression

On manipule les valeurs et les variables qui les référencent en les combinant avec des opérateurs pour former des expressions. Exemple :

```
>>> a, b = 7.3, 12
>>> y = 3*a + b/5
>>> y
```

Dans cet exemple, nous commençons par affecter aux variables  $a$  et  $b$  les valeurs  $7.3$  et  $12$ .

La seconde ligne de l'exemple consiste à affecter à une nouvelle variable  $y$  le résultat d'une expression.

Dans une affectation, ce que vous placez à la gauche du signe  $=$  doit toujours être un nom de variable. Ainsi par exemple, l'instruction  $m + 1 = b$  est tout à fait **illégale**.

Par contre, l'instruction  $a = a + 1$  est très fréquente en programmation, elle signifie en l'occurrence « augmenter la valeur de la variable  $a$  d'une unité » (ou encore : « incrémenter  $a$  »).

Que valent les variables à la fin des programmes suivants? Compléter les tableaux.

```
>>> a = 4
>>> a = a + 1
```

$a$

```
>>> a = 2
>>> m = 5
>>> a = m + a
>>> a = a + 1
>>> m = a*m
>>> m = 2*m
```

$a$	$m$

```
>>> a, b = 3, 7
>>> a = b
>>> b = a
```

$a$	$b$

```
>>> a, b = 3, 7
>>> b = a
>>> a = b
```

$a$	$b$

Proposer un programme qui puisse échanger le contenu de deux variables  $a$  et  $b$ .

#### 6) Composition

Jusqu'ici nous avons examiné les différents éléments d'un langage de programmation, à savoir : les variables, les expressions et les instructions, mais sans traiter la manière dont nous pouvons les combiner les unes aux autres.

Par exemple si vous savez comment additionner deux nombres et comment afficher une valeur, vous pouvez combiner ces deux instructions en une seule :

```
>>> print(17 + 3)
```

Cela n'a l'air de rien, mais cette fonctionnalité qui paraît si évidente va vous permettre de programmer des algorithmes complexes de façon claire et concise. Exemple :

```
>>> h, m, s = 15, 27, 34
>>> print("Le nombre de secondes écoulées depuis minuit est", h*3600 + m*60 + s)
```

## Utilisation d'un script, les fonctions informatiques

Lors de la première séance, on a travaillé directement dans la console, tout le travail effectué a été perdu (variables etc). Pour pouvoir sauvegarder des programmes, il va falloir rédiger les séquences d'instructions dans un **éditeur de texte**. Ainsi vous écrirez un **script** qui pourra être sauvegardé sous la forme fichier.py.

Il faudra ensuite appeler le fichier dans la console en faisant par exemple : `from fichier import *`

Une fonction informatique possède un **nom**, aucun, un ou plusieurs **arguments** et renvoie un ou plusieurs **résultats**.

On a déjà rencontré une fonction prédéfinie, la fonction `print()`.

On peut enregistrer les fonctions dans des scripts que l'on exécute dans la console.

On peut faire appel plusieurs fois à la même fonction en donnant des valeurs aux arguments (dans le bon ordre).

Il faut respecter la syntaxe comme sur l'exemple ci-dessous :

fichier.py

```
1 def nom(argument1, argument2, ...):
2     instructions
3     return resultat1, resultat2, ...
```

L'instruction `def` est une *instruction composée*, elle comporte une ligne d'en-tête terminée par un double point, suivie d'une ou plusieurs instructions indentées sous cette ligne d'en-tête (vous pouvez garder cette indentation ou la changer à condition de garder la même pour chaque ligne). Une fois l'instruction `return` exécutée, Python quitte la fonction.

Cas particuliers :

- Si l'on veut afficher plusieurs résultats sans sortir de la fonction, on peut utiliser `print` plusieurs fois.
- Si `return` est appelé sans argument, ou s'il n'est pas utilisé dans la fonction, la valeur retournée dans ce cas est l'objet `None`.

### Exercice 1 :

Écrire le script ci-dessous :

exo1.py

```
1 def f(x):
2     return 3*x-1.6
```

Exécuter le dans la console et compléter l'affichage :

```
>>> from exo1 import *
>>> f(1)

>>> 4*f(2)+5
```

Quelle est la nature de cette fonction?

Pour les exercices suivants, écrivez les scripts dans des fichiers, exécutez les dans la console comme pour l'exercice précédent et répondez aux questions.

### Exercice 2 :

On a programmé une fonction nommée `g`.

exo2.py

```
1 def g(a, u, x):
2     return a*x+u
```

① Quels sont les arguments de cette fonction?


② Quel va être l'affichage si on demande `g(1, 2, 3)` dans la console?

③ À quoi s'évalue l'expression `g(2, 1, 3) == 7` ?

### Exercice 3 :

En prévision des soldes, un commerçant s'apprête à modifier ses étiquettes.

- ① Un article coûte 40 € et subit une réduction de 30 %, quel est son nouveau prix?



- ② Voici le programme d'une fonction :

```
exo3.py
1 def solde(prix, baisse) :
2     return prix*(1-baisse/100)
```

- a) Quelle est la valeur affichée dans la console si on saisit `solde(40, 30)` ?



- b) Quel est le rôle de ce programme?



- c) Que doit saisir le commerçant pour solder à 60 % un article à 55 €?



### Exercice 4 :

```
exo4.py
1 def vecteur(A, B) :
2     (xA, yA) = A
3     (xB, yB) = B
4     return (xB-xA, yB-yA)
```


Quelle est la valeur affichée dans la console si on saisit `vecteur((2, 3), (5, 7))` ?

Quel est le rôle de cette fonction?



### Exercice 5 :

- ① Écrire une fonction `moyenne` (dans le fichier `exo5.py`) qui à deux nombres réels `a` et `b` associe leur moyenne arithmétique.



- ② Compléter la fonction `milieu` pour qu'elle renvoie les coordonnées du milieu du segment `[AB]`.

```
exo5.py
1 def milieu(A, B) :
2     (xA, yA) = A
3     (xB, yB) = B
4     return ( , )
```

- ③ Chercher une autre réponse à la question précédente en utilisant la fonction `moyenne`.

```
exo5.py
1 def milieu(A, B) :
2     (xA, yA) = A
3     (xB, yB) = B
4     return ( , )
```

# L'instruction if

Python exécute les instructions d'un programme les unes après les autres, dans l'ordre où elles ont été écrites à l'intérieur du script sauf lorsqu'il rencontre une instruction conditionnelle comme l'instruction `if` décrite ci-après. Une telle instruction va permettre au programme de suivre différents chemins suivant les circonstances. Pour la suite des séances tapez les scripts dans les cadres noirs, exécutez les et marquez les affichages en dessous.

## I) Exécution conditionnelle if

```
1 a=150
2 if a>100 :
3     print("a dépasse la centaine")
```

```
1 a=80
2 if a>100 :
3     print("a dépasse la centaine")
```

Lorsque vous finissez d'entrer la seconde ligne sans oublier de mettre le caractère « : », vous constatez que l'éditeur commence la troisième en mettant une indentation, c'est le même comportement que lorsque vous écrivez une fonction. L'expression que vous avez placée après `if` est ce que nous appellerons désormais une *condition*. L'instruction `if` permet de tester la validité de cette *condition* (qui est évaluée par un booléen). Si la *condition* est évaluée à vraie (True), alors l'instruction que nous avons indentée après le « : » est exécutée, si la condition est évaluée à fausse (False), cette ligne ne s'exécutera pas. Cas particuliers où la *condition* est le booléen True ou le booléen False :

```
1 if True :
2     print("Exécutée car cond vraie")
```

```
1 if False :
2     print("Non exécutée car fausse")
```

## II) Opérateurs de comparaisons et de logique

La *condition* évaluée après l'instruction `if` peut contenir les opérateurs de comparaison ou de logique suivants :

<code>x==2</code>	évaluée à vraie si x est égal à 2
<code>x!=4</code>	évaluée à vraie si x est différent de 4
<code>x&gt;-1</code>	évaluée à vraie si x est strictement supérieur à -1
<code>x&gt;=3</code>	évaluée à vraie si x est supérieur ou égal à 3
<code>x&lt;5</code>	évaluée à vraie si x est strictement inférieur à 5
<code>x&lt;=2</code>	évaluée à vraie si x est inférieur ou égal à 2
<code>not (x==3)</code>	évaluée à vraie si x est différent de 3
<code>1&lt;x and x&lt;3</code>	évaluée à vraie si x est dans l'intervalle ]1;3[
<code>x==2 or x&gt;3</code>	évaluée à vraie si x est égal à 2 ou x strictement plus grand que 3
<code>x in "age"</code>	évaluée à vraie si x vaut "a", "g" ou "e"

## III) Instructions composées, les blocs d'instructions

```
1 if 3<2 :
2     print("Exécutée si vraie")
3     print("mais celle-ci ?")
```

```
1 if 3<2 :
2     print("Exécutée si vraie")
3     print("mais celle-ci ?")
```

`if 3<2 :` Bloc 1

```
print("Exécutée si vraie") Bloc 2
print("mais celle-ci ?")
```

`if 3<2 :` Bloc 1

```
print("Exécutée si vraie") Bloc 2
print("mais celle-ci ?") Bloc 1S
```

La condition est fausse donc le bloc 2 n'est pas exécuté, on passe au bloc suivant. Un bloc est repéré par son indentation. Dans le premier cas il n'y a pas de bloc après le bloc 2 donc rien ne s'affiche, dans le deuxième cas on retourne dans le bloc 1 pour exécuter l'instruction.

## 1) Instructions imbriquées

### Prog1.py

```
1 if x>1 :
2     print("Plus que 1")
3     if x<100 :
4         print("Moins que 100")
```

### Prog2.py

```
1 if x>1 :
2     print("Plus que 1")
```

### Prog3.py

```
1 if x>1 :
2     print("Plus que 1")
3 if
4     print("Moins que 100")
```

① Tapez `x=110` dans la console puis exécutez Prog1.py. Affichage?

② Tapez `x=-10` dans la console puis exécutez Prog1.py. Affichage?

③ Est-ce que pour une certaine valeur de `x`, Prog1.py peut afficher seulement « Moins que 100 »?

④ Délimitez les blocs de Prog1.py

⑤ Complétez Prog2.py pour qu'il affiche « Moins que 100 » pour tous les nombres plus petits que 100.

⑥ Complétez Prog3.py pour qu'il soit équivalent à Prog1.py.

## 2) Instructions mutuellement exclusives

```
1 def f(x):
2     if x>2 :
3         return "Plus que 2"
4     else :
5         return "2 ou moins"
```

① Que renvoient `f(4)`, `f(-1)` et `f(2)` ?

② Que fait l'instruction `else` ?

③ Délimitez les blocs à partir de la deuxième ligne.

Créez une fonction `inverse` de paramètre `x` qui renvoie « 0 n'a pas d'inverse » si `x` vaut 0 et renvoie l'inverse de `x` sinon.

Créez une fonction `absolue` de paramètre `x` qui renvoie la valeur absolue de `x` c'est-à-dire `x` si `x` est positif et l'opposé de `x` si `x` est négatif.

On peut aussi créer un programme qui possède plus d'une instruction alternative en utilisant l'instruction `elif` (contraction de `else if`), les instructions du bloc qui le suivent ne seront exécutées que si la condition qui suit `elif` est évaluée à vraie et que les conditions précédentes sont évaluées à fausses :

```
1 def g(x):
2     if x<2 :
3         return "Petit"
4     elif x<10 :
5         return "Moyen"
6     else :
7         return "Grand"
```

① Que renvoient `g(4)`, `g(-1)` et `g(22)` ?

② Pour quelles valeurs de `x` la fonction `g` renvoie « Moyen »?

③ Délimitez les blocs à partir de la deuxième ligne.

On considère la fonction mathématique `h` définie pour tout nombre réel par :

$$h(x) = \begin{cases} -x-1 & \text{si } x \leq -2 \\ 0,5x+2 & \text{si } -2 < x \leq 4 \\ -0,5x+8 & \text{si } x > 4 \end{cases}$$

Programmez la fonction `h` en Python. Placez des points dans le graphique en vous aidant de cette fonction puis dessinez la courbe représentative de la fonction `h` sur `[-4; 8]`.

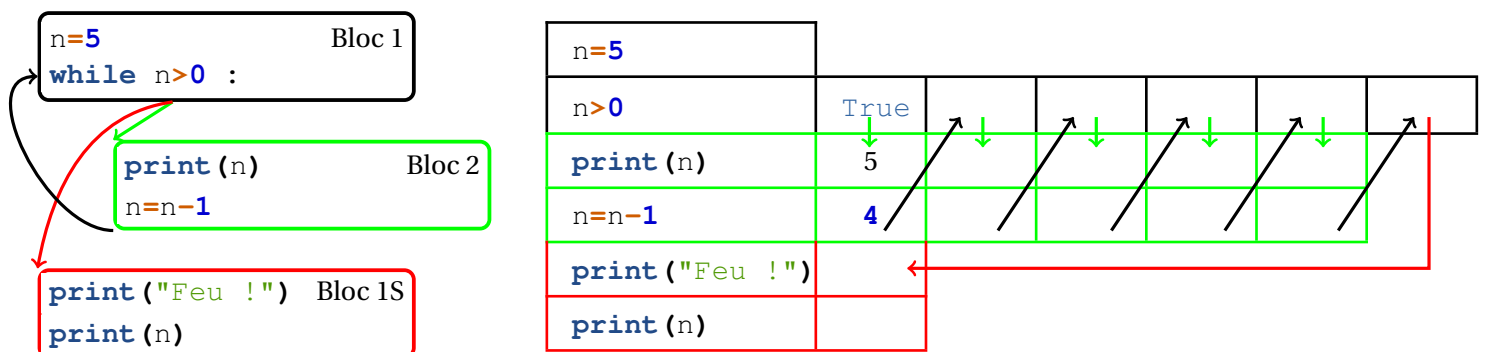
## Boucle non bornée, l'instruction while

```

1 n = 5
2 while n>0 :
3     print (n)
4     n=n-1
5 print ("Feu !")
6 print (n)

```

On remarque que la syntaxe de l'instruction `while` est la même que pour l'instruction `if` (condition à vérifier, les `:`, l'indentation du bloc suivant). Le mot `while` signifie « tant que » en anglais. Cette instruction utilisée à la seconde ligne indique à Python qu'il lui faut répéter continuellement (en boucle) le bloc d'instructions qui suit, tant que le contenu de la variable `n` reste strictement supérieur à `0`, une fois que la condition sera devenue fausse Python passera au bloc 1S.



On appelle *variable d'itération* une variable dont la valeur change à chaque parcours de la boucle et dont la valeur est testée pour pouvoir effectuer une nouvelle itération.

Une boucle `while` nécessite généralement trois éléments pour fonctionner correctement :

1. Initialisation de la *variable d'itération* avant la boucle (ligne 1).
2. Test de la *variable d'itération* associée à l'instruction `while` (ligne 2).
3. Mise à jour de la *variable d'itération* dans le corps de la boucle (ligne 4).

Il est donc important de bien initialiser la variable d'itération avant la boucle, car si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté. Si la condition est toujours vraie, alors le corps de la boucle est répété indéfiniment (d'où l'intérêt de la ligne 4).

**Bonjour**

Écrire un script qui affiche  $n$  fois « Bonjour » en colonne.

```


```

**1 à n**

Écrire un script qui affiche les  $n$  premiers nombres entiers en colonne.

```


```

**1 à n impairs**

Écrire un script qui affiche les nombres impairs inférieurs ou égaux à  $n$  en colonne.

```


```



### Triangle

Écrire un script qui dessine un triangle comme ci-dessous.

Rappel: `"a"+"b" → "ab"` et `"a"*3 → "aaa"`

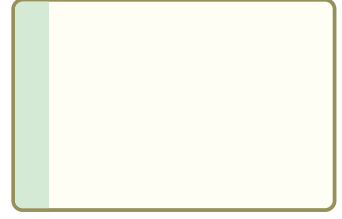
```
*  
**  
***  
****  
*****
```



### Triangle inversé et somme

Écrire un script qui dessine un triangle comme ci-dessous à  $n$  lignes et qui affiche en dessous le nombre total d'étoiles.

```
*****  
****  
***  
**  
*  
Il y a 15 étoiles
```



### Voiture

Au premier janvier 2020, le prix d'une voiture est de 18 960 €.

Chaque année, la valeur de cette voiture diminue de 20%.

On souhaite écrire un algorithme qui calcule le nombre d'années au bout duquel la valeur de cette voiture passe sous le seuil de 2 000 €.



### Baballe

Une balle chute de 400 pixels. À chaque rebond, la hauteur de la balle diminue de 10%. Réalisez un programme qui affiche la hauteur de chaque rebond tant qu'ils sont supérieurs à 5 pixels. Afficher le nombre de rebonds nécessaires pour avoir une hauteur inférieure à 5 pixels.



## Boucle bornée, l'instruction for

Une autre alternative à la boucle `while` couramment utilisée en informatique est la boucle `for` (qui signifie « pour » en anglais).

```
1 for lettre in "Bonjour" :
2     print (lettre)
```

`for ... in ...` : est un nouvel exemple d'*instruction composée*. La variable `lettre` est la *variable d'itération*, elle contiendra successivement tous les caractères qui composent la chaîne `"Bonjour"`, du premier jusqu'au dernier, à chaque itération de la boucle. Celle-ci est créée par Python la première fois que la ligne contenant le `for` est exécutée (si elle existait déjà son contenu serait écrasé). Une fois la boucle terminée, cette variable d'itération `lettre` ne sera pas détruite et contiendra ainsi la dernière valeur de la chaîne de caractères `"Bonjour"` (ici `"r"`).

```
1 for car in "Bonjour" :
2     print (car)
```

On peut voir sur l'exemple ci-contre que l'on peut choisir le nom que l'on veut pour la *variable d'itération*.

La boucle `for` itère toujours sur un objet dit *séquentiel* (c'est-à-dire un objet constitué d'autres objets).

### Les canetons

Dans un compte américain, huit petits canetons s'appellent respectivement : Jack, Kack, Lack, Mack, Nack, Oack, Pack, et Qack. Écrivez un petit script qui génère tous ces noms et les affiche à partir des deux chaînes suivantes : `prefixes="JKLMNOPQ"` et `suffixe="ack"`

Dans l'exemple ci-dessus, l'objet *séquentiel* était une chaîne de caractères (composée de caractères) et dans les exemples ci-dessous l'objet *séquentiel* sera une séquence d'entiers que l'on écrit à l'aide de `range()`.

Tapez les quatre programmes suivants en Python, marquez les affichages et faites un commentaire pour décrire ce qui se passe :

```
1 for i in range(4) :
2     print ("bla")
```

```
1 for i in range(4) :
2     print (i)
```

```
1 for i in range(12,16) :
2     print (i)
```

```
1 for i in range(5,15,3) :
2     print (i)
```

## Somme

Que fait le script ci-dessous? Programmez le.

```
1 s=0
2 for i in range(101):
3     s=s+i
4 print(s)
```

## Table de 5

Écrire un script qui affiche la table de multiplication de cinq (la table doit être affichée en colonne, en commençant par afficher :  $5 \times 0 = 0$  pour finir à  $5 \times 10 = 50$ ).

## Légende de Sissa

Connaissez-vous la légende de Sissa, l'inventeur du jeu d'échecs? Elle raconte que le roi indien Shirham lui aurait demandé quelle récompense il souhaitait pour avoir imaginé ce jeu qu'il appréciait. Sissa répondit ainsi : « Majesté je serais heureux si vous m'offriez un grain de riz que je placerai sur la première case de l'échiquier, deux grains pour la deuxième case, quatre grains pour la troisième, huit grains pour la quatrième et ainsi de suite pour les soixante-quatre cases. » Le roi se mit à rire tant sa demande lui semblait insignifiante.

Réalisez un programme permettant de calculer le nombre total de grains de riz.

## Scrabble

Le Scrabble<sup>®</sup>, jeu de société bien connu est un jeu de lettres où l'objectif est de cumuler des points, sur la base de tirage aléatoire de lettres, en créant des mots sur une grille carrée dont certaines cases sont primées. Dans la version française, il y a 102 jetons. Le joueur dispose de 7 lettres pour constituer un mot ou en compléter un. Voici la valeur de chaque lettre :

1 point	E, A, I, N, O, R, S, T, U, L
2 points	D, M, G
3 points	B, C, P
4 points	F, H, V
8 points	J, Q
10 points	K, W, X, Y, Z

Complétez la fonction ci-contre qui renvoie le nombre de points que rapporte au Scrabble un mot donné en argument (on suppose que le mot ne contient pas de lettres dans les cases primées). La reconnaissance des lettres est sensible à la casse, `mot.upper()` renvoie la chaîne `mot` en majuscules. Pour faire appel à cette fonction dans la console, il ne faudra pas oublier les guillemets autour du mot choisi car c'est une chaîne de caractères, exemple `scrabble("tortue")`.

```
1 def scrabble(mot):
2     mot=mot.upper()
3     points=0
4     for lettre in mot:
5         if lettre in "EAINORSTUL":
6             points=points+1
7
8
9
10
11
12
13
14
15
16
17     return points
```

# Aléatoire

Le module `random` permet de générer des nombres aléatoires, pour importer les fonctions qu'il contient, il suffit d'écrire au début du programme `from random import *`.

Le hasard joue un rôle important dans certains jeux pour qu'une partie ne ressemble pas à une autre.

Trois fonctions sont utiles :

Fonction	Effet
<code>randint(a, b)</code>	Renvoie un <b>entier</b> aléatoirement dans $[a; b]$ .
<code>random()</code>	Renvoie un <b>flottant</b> choisi aléatoirement dans l'intervalle $[0; 1[$ .
<code>uniform(a, b)</code>	Renvoie un <b>flottant</b> choisi aléatoirement dans l'intervalle $[a; b]$ .

## Jusqu'à 6

On lance un dé et on s'intéresse au nombre de lancers nécessaires pour obtenir un 6. Écrire un programme simulant cette expérience. Pour vous aider :

- Introduire la variable `dé` qui correspond au résultat du lancer du dé.
- Introduire la variable `compteur` qui compte le nombre de boucles utilisées.
- Connaissons nous à l'avance le nombre de lancers nécessaires? Quelle type de boucle utiliser?

## Pile ou face

On lance  $n$  fois une pièce de monnaie en l'air et on s'intéresse au nombre d'apparition de « face ». Compléter le programme suivant pour que la fonction `nombre_face(n)` retourne le nombre d'apparition de « face » parmi les  $n$  lancers.

```
1 def nombre_face(n):
2     face = 0
3     for i in range(n):
4         piece =
5         if :
6             face =
7     return(face)
```

## Le lièvre et la tortue

À chaque tour, on lance un dé à 6 faces. Si le 6 sort le lièvre gagne la partie, sinon la tortue avance d'une case. La tortue gagne quand elle a avancé 5 fois. Le jeu est-il à l'avantage du lièvre ou de la tortue?

Écrire un programme qui simule ce jeu, conjecturer lequel des deux a l'avantage. Démontrer cette conjecture à l'aide d'un arbre tronqué.