Initiation à la programmation

Zhentao Li

18 février 2016

Les listes

Type list

- Similaire aux chaînes de caractères, mais à entrées divers.

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu']
>>> taille = [5.0, 1.0, 0.7, 2.0]
>>> mix = ['girafe', 5.0, 'dahu', 2]
>>> animaux
['girafe', 'hippopotame', 'singe', 'dahu']
>>> print animaux
['girafe', 'hippopotame', 'singe', 'dahu']
>>> taille
[5.0, 1.0, 0.7, 2.0]
>>> mix
['girafe', 5.0, 'dahu', 2]
```

Type list

- Similaire aux chaînes de caractères, mais à entrées divers.
- Création: `[element1, element2, element3]`

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu']
>>> taille = [5.0, 1.0, 0.7, 2.0]
>>> mix = ['girafe', 5.0, 'dahu', 2]
>>> animaux
['girafe', 'hippopotame', 'singe', 'dahu']
>>> print animaux
['girafe', 'hippopotame', 'singe', 'dahu']
>>> taille
[5.0, 1.0, 0.7, 2.0]
>>> mix
['girafe', 5.0, 'dahu', 2]
```

Indices

- On peut rappeler ses éléments par leur numéro de position **indice**

```
>>> animaux = ['girafe','hippopotame','singe','dahu']
>>> animaux[0]
'girafe'
>>> animaux[1]
'hippopotame'
>>> animaux[3]
'dahu'
```

Indices

- On peut rappeler ses éléments par leur numéro de position **indice**
- Les indice d'une liste de n éléments commence à 0 et se termine à $n-1$

```
>>> animaux = ['girafe','hippopotame','singe','dahu']
>>> animaux[0]
'girafe'
>>> animaux[1]
'hippopotame'
>>> animaux[3]
'dahu'
```

Opérations + et *

- + concaténation
- * duplication.

```
>>> animaux = ['aigle','ecureuil']
>>> animaux + animaux
['aigle', 'ecureuil', 'aigle', 'ecureuil']
>>> animaux * 3
['aigle', 'ecureuil', 'aigle', 'ecureuil', 'aigle',
'ecureuil']
```

Indiçage négatif

```
liste     : ['girafe', 'hippopotame', 'singe', 'dahu']
index pos :         0              1        2       3
index nég :        -4             -3       -2      -1
```

```
>>> animaux = ['girafe','hippopotame','singe','dahu']
>>> animaux[-4]
'girafe'
>>> animaux[-2]
'singe'
>>> animaux[-1]
'dahu'
```

Tranches

Sélectionner une **partie** de liste avec `une_liste[m:n+1]`

```
>>> animaux = ['girafe', 'hippo', 'singe', 'dahu', 'ornithoryn
>>> animaux[0:2]
['girafe', 'hippo']
>>> animaux[0:3]
['girafe', 'hippo', 'singe']
>>> animaux[0:]
['girafe', 'hippo', 'singe', 'dahu', 'ornithorynque']
>>> animaux[:]
['girafe', 'hippo', 'singe', 'dahu', 'ornithorynque']
>>> animaux[1:]
['hippo', 'singe', 'dahu', 'ornithorynque']
>>> animaux[1:-1]
['hippo', 'singe', 'dahu']
```

Les instructions `range()` et `len()`

L'instruction `range` vous permet de créer des listes d'entiers de manière simple et rapide.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,5)
[0, 1, 2, 3, 4]
>>> range(15,21)
[15, 16, 17, 18, 19, 20]
>>> range(0,1000,100)
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
>>> range(0,-10,-1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Les instructions `range()` et `len()`

L'instruction `range` vous permet de créer des listes d'entiers de manière simple et rapide.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,5)
[0, 1, 2, 3, 4]
>>> range(15,21)
[15, 16, 17, 18, 19, 20]
>>> range(0,1000,100)
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
>>> range(0,-10,-1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

L'instruction `len` vous permet de connaître la longueur d'une liste.

```
>>> animaux = ['girafe', 'hippo', 'singe', 'dahu' , 'ornithory
>>> len(animaux)
```

L'instruction `for`

Il arrive souvent que nous voulons une boucle pour traverser les éléments d'une liste ou chaîne. L'instruction `for` permet de simplifier

```
l = ['a', 'b', 'c', 'd']
indice = 0
while indice < len(l):
    element = l[indice]
    # Faire quelque chose avec l'element
    indice = indice + 1
```

à

```
l = ['a', 'b', 'c', 'd']
for element in l:
    # Faire quelque chose avec l'element
```

Autre exemple for

```
for lettre in "ciao":
    print lettre # c i a o

for x in [2, 'a', 3.14]:
    print x # 2 a 3.14

for i in range(5):
    print i # 0 1 2 3 4
```

Syntaxe

```
for variable in list:
    instruction1
    instruction2
```

L'instruction break

L'instruction `break` interrompt les boucles `for` comme pour les boucles `while`.

```
for x in range(1, 11):
    if x == 5:
        break
    print x,

print "\nBoucle interrompue pour x =", x
```

Résultat

```
1 2 3 4
Boucle interrompue pour x = 5
```

L'instruction continue

L'instruction `continue` fait passer à l'itération suivante les instructions `while` ou `for`.

```
for x in range(1, 11):
    if x == 5:
        continue
    print x,

print "\nLa boucle a sauté la valeur 5"
```

Résultat

```
1 2 3 4 6 7 8 9 10
La boucle a sauté la valeur 5
```

Mutation

Les `lists` sont *mutable*: la valeur d'une variable peut changer sans réassignation.

```
>>> liste1 = [1, 2, 3]
>>> liste2 = liste1
>>> liste2
[1, 2, 3]
>>> liste2[2] = 0
>>> liste1
[1, 2, 0]
```

Mutation

Les `lists` sont *mutable*: la valeur d'une variable peut changer sans réassignation.

```
>>> liste1 = [1, 2, 3]
>>> liste2 = liste1
>>> liste2
[1, 2, 3]
>>> liste2[2] = 0
>>> liste1
[1, 2, 0]
```

`liste2` a été réassignée mais pas `liste1` alors que sa valeur a changée! Essayez le même exercice avec d'autre types, `str` par exemple.

Mutation

Les `lists` sont *mutable*: la valeur d'une variable peut changer sans réassignation.

```
>>> liste1 = [1, 2, 3]
>>> liste2 = liste1
>>> liste2
[1, 2, 3]
>>> liste2[2] = 0
>>> liste1
[1, 2, 0]
```

`liste2` a été réassignée mais pas `liste1` alors que sa valeur a changée! Essayez le même exercice avec d'autre types, `str` par exemple.

Attention: certaines opérations renvoient des nouvelles listes alors que d'autre modifient la variable.

Type `tuple`

- Un `tuple` est une liste non-mutable.
- Création: Un `tuple` est créé par des parenthèses au lieu de crochets.

```
>>> t = (1, 'a', 2)
>>> t
(1, 'a', 2)
>>> t[1]
'a'
>>> t2 = (1,)
>>> t2
(1,)
```

Il faut au moins une virgule pour créer un `tuple`.

Unpacking

Pour récupérer tous les entrées d'une `list`, au lieu de

```
>>> animaux = ['girafe', 'hippopotame', 'singe']
>>> a = animaux[0]
>>> b = animaux[1]
>>> c = animaux[2]
```

nous pouvons utiliser la syntaxe suivante.

```
>>> animaux = ['girafe', 'hippopotame', 'singe']
>>> a, b, c = animaux
```

Opérations sur `list` et `tuple` (avancé)

D'autre opérations qui sont des mutations les `lists`: `append`, `extend`, `insert`, `pop`, `remove`, `reverse` et `sort`.

```
l = range(5)
l.append(10)
l2 = ('a', 1)
l.extend(l2)
l.remove(1)
element = l.pop()
premier_element = l.pop(0)
l.reverse()
l.sort()
```

Opérations sur `list` et `tuple` (avancé)

D'autre opérations qui sont des mutations les `lists`: `append`, `extend`, `insert`, `pop`, `remove`, `reverse` et `sort`.

```
l = range(5)
l.append(10)
l2 = ('a', 1)
l.extend(l2)
l.remove(1)
element = l.pop()
premier_element = l.pop(0)
l.reverse()
l.sort()
```

Attention: Une erreur récurrente est d'appeler `l2 = l.append(10)`. Cela met `l2` à `None`.
D'autre opérations sur les `lists` et `tuples`: `index` et `count`.

Compréhension de liste (très? avancé)

Un motif récurrent d'initialisation d'une liste est sa création par une boucle.

```
carres = []
for i in range(10):
    carres.append(i*i)
```

Dans l'intérêt de la lisibilité du programme, une nouvelle syntaxe a été ajoutée.

```
carres = [i*i for i in range(10)]
carres_impairs = [i*i for i in range(10) if i%2 == 1]
```

Cette notation ressemble à la notation d'ensemble en mathématiques.

Les fonctions

Définition d'une fonction

Rappel: Deux formes d'appel

- `nom_de_fonction(parametre1, parametre2, ...)`
- `variable.nom_de_fonction(parametre1, parametre2, ...)`

Nous allons maintenant voir comment définir de nouvelles fonctions. Les **fonctions** permettent de décomposer les programmes en sous-programmes et de réutiliser des morceaux de programmes.

Exemple de définition d'une fonction

```
def compter_lettre(lettre, texte):
    n=0
    for c in texte :
        if c == lettre :
            n += 1
    return "nombre d'occurences de la lettre %s : %s" \
               % (lettre, n)

print compter_lettre('e', 'je reviens')
```

Résultat

```
nombre d'occurrences de la lettre e : 3
```

Syntaxe

- Une fonction renvoie une valeur de sortie à partir de paramètres d'entrées.
- Syntaxe

```
def nom_de_la_fonction(parametre1, parametre2, ...):
    instruction1
    instruction2
    ...
```

- `return valeur` dans le bloc d'instructions: fin d'exécution et retour de la valeur.

Syntaxe

- Une fonction renvoie une valeur de sortie à partir de paramètres d'entrées.
- Syntaxe

```
def nom_de_la_fonction(parametre1, parametre2, ...):
    instruction1
    instruction2
    ...
```

- `return valeur` dans le bloc d'instructions: fin d'exécution et retour de la valeur.

Remarque(avancée): Une fois définie, une fonction peut être manipulée comme tout autre variable!

```
>>> print compter_lettre
<function compter_lettre at 0x7f485e9d45f0>
>>> type(compter_lettre)
<type 'function'>
```

Appel d'une fonction

- Une fois qu'une fonction `f` a été définie, elle peut être utilisée dans une expression particulière qu'on nomme un appel de fonction et qui a la forme `f(v1,v2,...,vn)`, où `v1`, `v2`, ..., `vn` sont des expressions dont la valeur est transmise au paramètres.
- Cela veut dire que d'abord `parametre1` est assigné la valeur `v1`, `parametre2` est assigné la valeur `v2`, et ainsi de suite. Ensuite le bloc d'instruction est exécuté ligne par ligne, jusqu'à la fin du bloc ou une instruction `return`.
- Si la fin du bloc est atteint, la valeur `None` est retournée.

Paramètres par défaut

Python offre un mécanisme d'instanciation des paramètres par défaut. On peut écrire la liste des paramètres en entête d'une définition de fonction. Par exemple

```
>>> def pluriel(mot, famille = 'standard'):
...     if famille == 'standard':
...         return mot + 's'
...     if famille == 's':
...         return mot
...     if famille == 'oux':
...         return mot + 'x'
...     if famille == 'al':
...         return mot[:-1] +'ux'

>>> print pluriel('maison')
'maisons'
>>> print pluriel('souris', 's')
'souris'
```

Paramètres par défaut

- Syntaxe

```
def nom_de_la_fonction(p_1, ..., p_k, d_1=expr_1,..., d_n=expr_n):
    instruction1
    instruction2
    ...
```

- Les k premiers paramètres doivent obligatoirement être précisés à l'appel de fonction mais pas les n derniers. L'appel de fonction se fait donc avec k arguments au minimum et $k+n$ arguments au maximum. Si un paramètre `d_i` n'est pas instancié explicitement, il prend la valeur par défaut de `expr_i`.
- Il est possible d'instancier certain paramètre et pas d'autre. Par exemple, avec un appel `pluriel(p_1, ..., p_k, d_4 = 10)`. `d_4` prends la valeur `10` au lieu de `expr_4`.

Variables locales et variables globales

- Les variables qui sont introduites dans la définition d'une fonction peuvent être utilisées dans la suite de la définition mais **pas à l'extérieur de la fonction**.
- Ces variables sont dites **locales** par opposition aux variables **globales** qui sont introduites à l'extérieur de la définition d'une fonction et qui peuvent être utilisées à l'intérieur comme à l'extérieur de cette définition.
- Lorsque le même nom est utilisé pour introduire une variable locale et une variable globale, Python distingue bien deux variables différentes mais à l'intérieur de la définition de la fonction, c'est à la variable locale auquel le nom réfère.

Variables locales et variables globales

```
>>> def f(x):
...     y=2
...     return x + y
>>> print f(3)
5
>>> print y
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <mo
    print y
NameError: name 'y' is not defined
>>> u = 7
>>> def g(v):
...     return u * v
>>> print g(2)
14
```

```
>>> def h(u):
...     return u
>>> print h(3)
3
>>> print u
7
>>> def k(w) :
...     u = 5
...     return w+u
>>> print k(3)
8
>>> print u
7
```

Les modules

Modules

- On peut ranger les définitions de fonctions se rapportant à une même application au sein d'un script commun baptisé red module.
- Un module est sauvegardé sous forme d'un fichier dont le nom a la forme `<nom du module>.py`.
- Pour utiliser un module, il faut se servir de l'instruction `import <nom du module>`.
- L'exécution de cette instruction consiste à exécuter le script définissant le module (ce script peut contenir des instructions autres que des définitions de fonctions).
- Pour importer un module, Python a besoin de connaître le chemin qui permet d'accéder au fichier correspondant. Ce chemin doit apparaître dans la liste des chemins possibles stockés dans la variable `path` du module `sys`.

Modules - Première méthode d'importation

```
>>> import random
>>> random.randint(0, 10)
9
```

Regardons de plus près cet exemple :

- L'instruction `import` vous permet d'importer toutes les fonctions du module `random`
- Ensuite, nous utilisons la fonction `randint(a,b)` du module `random`; attention cette fonction renvoie un nombre entier aléatoirement entre `a` inclus et `b` inclus.

Modules - Deuxième méthode d'importation

- Pour disposer d'une fonction du module:

```
>>> from random import randint
>>> random.randint(0, 10)
9
```

- Pour disposer de toutes les fonctions d'un module:

```
from math import *
racine = sqrt(49)
angle = pi/6
print sin(angle)
```

Normalement, `import *` est seulement utilisé dans l'interpréteur et déconseillé dans les scripts. C'est pour aider le lecteur à trouver la provenance des variables et fonctions du programme.

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard
- `os` : dialogue avec le système d'exploitation.

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard
- `os` : dialogue avec le système d'exploitation.
- `random` : génération de nombres aléatoires.

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard
- `os` : dialogue avec le système d'exploitation.
- `random` : génération de nombres aléatoires.
- `time` : permet d'accéder aux fonctions gérant le temps.

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard
- `os` : dialogue avec le système d'exploitation.
- `random` : génération de nombres aléatoires.
- `time` : permet d'accéder aux fonctions gérant le temps.
- `profile` : permet d'évaluer le temps d'exécution de chaque fonction dans un programme (profiling en anglais).

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard
- `os` : dialogue avec le système d'exploitation.
- `random` : génération de nombres aléatoires.
- `time` : permet d'accéder aux fonctions gérant le temps.
- `profile` : permet d'évaluer le temps d'exécution de chaque fonction dans un programme (profiling en anglais).
- `urllib` : permet de récupérer des données sur internet depuis python.

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard
- `os` : dialogue avec le système d'exploitation.
- `random` : génération de nombres aléatoires.
- `time` : permet d'accéder aux fonctions gérant le temps.
- `profile` : permet d'évaluer le temps d'exécution de chaque fonction dans un programme (profiling en anglais).
- `urllib` : permet de récupérer des données sur internet depuis python.
- `Tkinter` : interface python avec Tk (permet de créer des objets graphiques; nécessite d'installer Tk).

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard
- `os` : dialogue avec le système d'exploitation.
- `random` : génération de nombres aléatoires.
- `time` : permet d'accéder aux fonctions gérant le temps.
- `profile` : permet d'évaluer le temps d'exécution de chaque fonction dans un programme (profiling en anglais).
- `urllib` : permet de récupérer des données sur internet depuis python.
- `Tkinter` : interface python avec Tk (permet de créer des objets graphiques; nécessite d'installer Tk).
- `re` : gestion des expressions régulières.

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard
- `os` : dialogue avec le système d'exploitation.
- `random` : génération de nombres aléatoires.
- `time` : permet d'accéder aux fonctions gérant le temps.
- `profile` : permet d'évaluer le temps d'exécution de chaque fonction dans un programme (profiling en anglais).
- `urllib` : permet de récupérer des données sur internet depuis python.
- `Tkinter` : interface python avec Tk (permet de créer des objets graphiques; nécessite d'installer Tk).
- `re` : gestion des expressions régulières.
- `pickle` : écriture et lecture de structures python.

Quelques fonctions des modules courants

```
>>> math.pow(2,10)   #Exposant
1024.0
>>> code_retour = os.system("echo 123")  #Commande externe
123
>>> random.randint(0, 10)  #Choisis un entier au hasard
2
>>> random.choice(["a", 3, "d"])  #Choisis un element d'une liste
'd'
>>> time.time()  # Nombre de secondes depuis 1970-01-01
1393935277.116552
>>> time.sleep(3)  # Attendre 3 seconds
>>> profile.run("time.sleep(3)")  # Calcule le temps d'excution
         4 function calls in 0.000 seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 :0(sleep)
[...]
```

Quelques fonctions des modules courants

```
>>> handle = urllib.urlopen("http://www.google.com")
>>> html = handle.read()  # Lire le contenu d'une page comme un fichier
>>> print html
'<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage">
<head><meta itemprop="image" content="/images/google_favicon_128.png"><ti
tle>Google</title><script>(function(){\nwindow.google={kEI:"i8QVU8ejFqTt4
gTBvIDIAw",getEI:function(a){for(var b;a&&(!a.getAttribute||!(b=a.getAttr
[...]
>>> # Recherche des chaines de caractere dans un texte.
>>> # Dans cet exemple, on trouve les liens sur la page web'.
>>> re.findall('href="(.*?)"', html)
['/search?', 'http://www.google.fr/imghp?hl=fr&tab=wi', 'http://maps.
google.fr/maps?hl=fr&tab=wl', 'https://play.google.com/?hl=fr&tab=w8',
'http://www.youtube.com/?gl=FR&tab=w1', 'http://news.google.fr/nwshp?
[...]
>>> # Sauvegarde de variables dans des fichiers et lecture
>>> pickle.dump(liste, open("sauvegarde.pkl", "w"))
>>> listelue = pickle.load(open("sauvegarde.pkl"))
>>> listelue
['a', 'b', 'c']
```

Obtenir de l'aide

- Utiliser `help` pour obtenir de l'aide sur un module.

```
>>> import random
>>> help(random)
```

Obtenir de l'aide

- Utiliser `help` pour obtenir de l'aide sur un module.

```
>>> import random
>>> help(random)
```

- De même pour de l'aide sur une fonction.

```
>>> help(random.randint)
```

Obtenir de l'aide

- Utiliser `help` pour obtenir de l'aide sur un module.

```
>>> import random
>>> help(random)
```

- De même pour de l'aide sur une fonction.

```
>>> help(random.randint)
```

- Ça marche même sur des variables et des types.

```
>>> liste = [10, 20]
>>> help(liste)
>>> help(str)
```

Les fichiers

Fichiers: utilisation courante

```
$ cat exemple.txt
Ceci est la premiere ligne
Ceci est la deuxieme ligne
Ceci est la troisieme ligne
Ceci est la quatrieme et derniere ligne
```

```
>>> nom_fichier = "exemple.txt"
>>> contenu_fichier = open(nom_fichier).read()
>>> contenu_fichier
'Ceci est la premiere ligne\nCeci est la deuxieme ligne\nCe
ci est la
troisieme ligne\nCeci est la quatrieme et derniere ligne\n'
>>> contenu_a_ecrire = "hello world\n"
>>> open("exemple2.txt", "w").write(contenu_a_ecrire)
$ cat exemple2.txt
hello world
```

Autre exemple

Créez un fichier dans un éditeur de texte que vous sauverez dans votre répertoire avec le nom `exemple.txt`, par exemple :

```
Ceci est la premiere ligne
Ceci est la deuxieme ligne
Ceci est la troisieme ligne
Ceci est la quatrieme et derniere ligne
```

Exemple readlines

```
>>> file_in = open('exemple.txt','r')
>>> file_in
<open file 'exemple.txt', mode 'r' at 0x40155b20>
>>> lignes = file_in.readlines()
>>> lignes
['Ceci est la premiere ligne\n', 'Ceci est la deuxieme ligne\n',
'Ceci est la troisieme ligne\n', 'Ceci est la quatrieme et derniere ligne\n']
>>> for i in lignes:
...   print i
...
Ceci est la premiere ligne

Ceci est la deuxieme ligne

Ceci est la troisieme ligne

Ceci est la quatrieme et derniere ligne
>>> file_in.close()
>>> file_in
<closed file 'exemple.txt', mode 'r' at 0x40155b20>
>>> file_in.close()
```

Exemple read

```
Ceci est la premiere ligne
Ceci est la deuxieme ligne
Ceci est la troisieme ligne
Ceci est la quatrieme et derniere ligne
```

```
>>> file_in = open("exemple.txt","r")
>>> file_in.read()
'Ceci est la premiere ligne\nCeci est la deuxieme ligne\nCe
ci est la troisieme ligne\nCeci est la quatrieme et dernier
e ligne\n'
>>> file_in.close()
```

Exemple `tell`

```
Ceci est la premiere ligne
Ceci est la deuxieme ligne
Ceci est la troisieme ligne
Ceci est la quatrieme et derniere ligne
```

```
>>> file_in = open("exemple.txt","r")
>>> file_in.tell()
0L
>>> file_in.readline()
'Ceci est la premiere ligne\n'
>>> file_in.tell()
27L
>>> file_in.seek(0)
>>> file_in.tell()
0L
>>> file_in.readline()
'Ceci est la premiere ligne\n'
>>> file_in.close()
```

Exemple `write`

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu',
'ornithorynque']
>>> file_out = open('exemple2.txt','w')
>>> for i in animaux:
...     file_out.write(i)
...
>>> file_out.close()
>>>
$ cat exemple2.txt
girafehippopotamesingedahuornithorynque
$
```

Utilisation de fichiers

- Il est important de dissocier les données des programmes qui les utilisent en rangeant ces données dans des fichiers séparés.
- Pour utiliser un fichier dans Python, il faut commencer par l'ouvrir ce qui retourne un objet de type `file`.
- Le paramètre facultatif `<mode>` indique le mode d'ouverture du fichier :
  - `r` : mode lecture (le fichier doit exister préalablement)
  - `w` : mode écriture (si le fichier existe, les données sont écrasées, sinon le fichier est créé)
  - `a` : mode ajout (si le fichier existe, les données écrites vont l'être après celles existantes, sinon le fichier est créé)
- Si le mode est omis, le mode par défaut est `r`.