Langages Dynamiques

Cours 4

kn@lri.fr
http://www.lri.fr/~kn

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

université PARIS-SACLAY

Plan

Description

- Créé par Guido van Rossum en 1991
- 1991 : Python 1.0
- 2000 : Python 2.0
- 2008 : Python 3.0
- 2020 : Python 2.0 n'est plus supporté, version actuelle 3.8

Dans ce cours, on utilisera exclusivement la version 3 du langage

C'est aussi cette version qui est maintenant au programme de 1ère et Tale

Caractéristiques du langage

- Langage généraliste : traitement de données, interfaces graphiques, réseau, jeux, calcul scientifique, intelligence artificielle, ...
- Langage interprété : la commande `python3` permet d'exécuter des scripts Python
- (et bien d'autres qu'on va découvrir au fur et à mesure)

Un premier programme

On considère le fichier `salut.py`

```
LIMIT=40                                    #On définit une variable globale
entree = input("Quel est votre age ?") #On lit une entrée de l'utilisateur
age = int(entree)                           #On la convertit en nombre

if age >= LIMIT:                            #On teste la valeur et on affiche
    print ('Salut, vieux !')
else:
    print ('Salut, toi !')
```

On peut exécuter ce programme dans un terminal :

```
$ python3 salut.py
Quel est votre age ? 38
Salut, toi !
$
```

Qu'y a t'il dans ce programme ?

- Définition de variables
- Entrées (de l'utilisateur) et affichages (dans la console)
- Manipulation de constantes (nombres, textes, ...)
- Tests de valeurs

On remarque qu'il n'y a pas de notion de fonction principale (genre main).

La boucle d'interaction

Le programme `python3` possède aussi un **mode interactif** qui permet d'évaluer des instructions, comme un shell.

Il suffit de lancer la commande `python3` sans argument.

```
$ python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 1
2
>>> 3 * 10
30
>>> x = 42
>>> x + 10
52
>>>
```

On peut quiter avec `CTRL-d`

Mode programme et mode intéractif

Le mode intéractif attend des instructions Python et les exécute au fur et à mesure.
Il peut être utilisé pour tester des petits morceaux de programmes.

Le mode programme : `python3` interprète les lignes du fichier passé en argument.

Plan

Les entiers (type `int`)

En Python, les entiers peuvent être de taille arbitraire. Ils sont signés (on peut représenter les nombres positifs et négatifs) :

```
>>> 1
1
>>> -149
-149
>>> 1111111333333333332999999999999013000000000000033333333333333333
1111111333333333332999999999999013000000000000033333333333333333
```

Opération sur les entiers

Symbole	Description
+	addition
–	soustraction
*	multiplication
/	division exacte (résultat à virgule)
/ /	division entière (résultat entier)
%	modulo
**	puissance

```
>>> 1 + 1
2
>>> 27 - 100
-73
>>> 3 * 4
12
>>> 4 / 3
1.3333333333333
>>> 4 // 3
1
```

Opération sur les entiers (suite)

```
>>> 10 % 4
2
>>> 2 ** 100
1267650600228229401496703205376
>>>  3 + 4 * 7
31
>>>  (3 + 4) * 7
49
>>> 1 / 0

Traceback (most recent call last):
  File "", line 1, in
ZeroDivisionError: division by zero
```

Conclusion sur les entiers

- Python supporte des entiers signés de taille arbitraire
- Les opérations arithmétiques `+`, `-`, `*`, `%` et `**` sont naturelles
- Il existe deux version de la division : exacte (`/`) et entière (`//`)
- Les priorités sont aussi celles utilisées en mathématiques (`*`, `/`, `//`, `%` sont plus prioritaires que `+`, `-`)
- On peut utiliser des parenthèses pour grouper les opérations
- Certaines opérations peuvent provoquer des erreurs (ex: division par 0)

Les nombres à virgule (type `float`)

En Python, les « nombres à virgule » ont une précision limitée. On les représente en utilisant la notation scientifique :

```
>>> 1.5
1.5
>>> -12.3423e13
-123423000000000.0
>>> 1.5555555555555555555555555
1.5555555555555556
```

Remarque : `-12.3423e13` = `-12.3423` × 10^{13} = `-123423000000000.0`

Attention : En Python, comme dans de nombreux langages, calculer avec des nombres à virgule (nombres *flottants*) peut provoquer des erreurs d'arrondi.

Opérations sur les nombres à virgule

On utilise les mêmes opérations que sur les entiers. Il n'y a qu'une seule division (`/`)

```
>>> 1.5 + 1.5
3.0
>>> 3.141592653589793 * 2
6.283185307179586
>>> 10.5 / 3
3.5
>>> 1.2 + 1.2 + 1.2
3.5999999999999996
>>> 4.5 ** 100
2.0953249170398634e+65
>>>  1.0 / 0
Traceback (most recent call last):
File "", line 1, in
ZeroDivisionError: float division by zero
```

Les booléens (type `bool`)

Les booléens sont représentés par : `True` et `False`.
Les opérations sur ces valeurs sont la négation (`not`), le « ou logique » (`or`) et le « et logique » (`and`).

On peut manipuler ces objets en Python, comme on le fait avec des entiers, des nombres à virgule ou des chaînes de caractères.

```
>>> True
True
>>> False
False
>>> not (True)
False
>>> True or False
True
>>> True and False
False
```

Tableau (type `list`)

Un tableau permet de stocker une **collection ordonnée et finie** de valeurs et d'accéder **efficacement à un élément arbitraire** de la collection.

- `[e1, …, en]` : définition d'un tableau
- `t[i]` : accède au $i^{ème}$ élément du tableau `t`. Comme dans presque tous les langages, les indices commencent à 0.
- `t[i] = e` : mise à jour du $i^{ème}$ élément du tableau `t`.
- `len(t)` : longueur du tableau `t`

```
>>> tab = [1, 3, 5, 4, 19, 2]
>>> tab
[1, 3, 5, 4, 19, 2]
>>> tab[4]
19
>>> tab[4] = 42
>>> tab
[1, 3, 5, 4, 42, 2]
```

Opérations avancées

- `t1 + t2` : concaténation de deux tableaux (renvoie un nouveau tableau avec les éléments de `t1` et `t2` bout à bout).
- `t * n` : concatène `n` fois le tableau `t` avec lui même.
- `t[i:j]` renvoie une copie du tableau prise entre les indices `i` et `j`

```
>>> t1 = [1, 2, 3]
>>> t2 = [4, 5, 6]
>>> t1 + t2
[1, 2, 3, 4, 5, 6]
>>> t3 = t1 + t2
>>> t3[0] = 10
>>> t3
[10, 2, 3, 4, 5, 6]
>>> [0] * 10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> t3[2:4]
[3, 4, 5]
```

Langage objet

Python est un langage Objet et tout type de données en python est une classe. On définira plus tard la notion d'objet, d'héritage et de méthodes dans le cadre de Python.

On se contente dans la suite de donner des méthodes utilitaires.

Interface impérative des tableaux

Les tableaux Python sont des tableaux redimentionnables. Possédants une interface impérative. Parmis les méthodes on trouve :

- `t.append(e)` ajoute l'élément `e` en fin de tableau
- `t.pop()` retire et renvoie le dernier élément du tableau
- `t.insert(i, e)` ajoute l'élément `e` à l'indice `i` dans tableau
- `t.clear()` vide le tableau
- `t.remove(e)` retire la première occurrence de `e` dans `t`

Chaînes de caractères (type `str`)

Le type `str` représente les chaînes de caractères.

```
>>> 'Bonjour, ça va bien ?'
'Bonjour, ça va bien ?'
```

On ne montre que quelques opérations sur les chaînes de caractères :

Symbole	Description
`+`	concaténation
`*`	répétition
`len(`*s*`)`	nombre de caractères

Opérations sur les chaînes

```
>>> 'Bonjour' + ', ça va bien ?'
'Bonjour, ça va bien ?'
>>> 'Bonjour' * 3
'BonjourBonjourBonjour'
>>>len('Bonjour')
7
```

Attention : il existe des opérations plus complexes sur les chaînes, et plusieurs façons d'écrires celles ci. On verra cela au fur et à mesure.

Séquences d'échappement

Que se passe-t-il si on veut insérer un caractère « ' » dans une chaîne ?

```
>>> 'C'est moi!'
  File "", line 1
    'C'est moi!'
       ^
 SyntaxError: invalid syntax
```

On doit indiquer que le « ' » ne marque pas la fin de la chaîne. On utilise une **séquence d'échappement** :

```
>>> 'C\'est moi!'
'C\'est moi!
>>> print('C\'est moi!')
 C'est moi!
>>>
```

Attention, le programmeur saisit **deux**caractères (`\'`) mais Python n'en retient qu'un seul (`'`).

D'autres séquences d'échappement

Comment saisir un caractère `\` dans une chaîne ?
Imaginons qu'on veuille le mettre en dernière caractère d'une chaine :

```
>>> 'Caractère antislash: \'
 File "", line 1
   'Caractère antislash: \'
                         ^
  SyntaxError: EOL while scanning string literal
```

Il faut aussi échapper le caractère `\`

```
>>> 'Caractère antislash: \\'
'Caractère antislash: \\'
```

Il existe d'autres séquences d'échappement :`\n`(retour à la ligne), `\uxxxx` (code Unicode en base 16), ...

```
>>> '\u0041\n\u0042'
'A\nB'
```

D'autres chaînes

En Python, on peut aussi délimiter une chaîne de caractères par `"` ou encore par `"""`

- Les chaînes `"` sont comme les chaînes `'`. Elles permettent juste d'éviter les `\` si on a beaucoup de `'` : `"Plein d'apostrophes c'est super !"`
- Les chaînes `"""` permettent aussi les retour à la ligne dans la chaîne

```
>>> """Je peux
... écrire une chaîne
... sur
... plusieurs
... lignes!"""
'Je peux\nécrire une chaîne\nsur\nplusieurs\nlignes!'
```

Manipulation des chaînes

On peut accéder aux n^{ème} caractère d'une chaîne comme si c'était un tableau

```
>>> t = 'Bonjour'
>>> t[3]
'j'
```

L'opération `[...]` renvoie une chaîne de taille 1 contenant uniquement le caractère ce trouvant à cet endroit.

Par contre, il n'est pas possible de mettre à jour une chaîne :

```
>>> t[3] = 'X'
Traceback (most recent call last):
  File "", line 1, in
TypeError: 'str' object does not support item assignment
>>>
```

Autres opérations

La fonction `chr(n)` renvoie le caractère dont le code Unicode est `n`.
La fonction `ord(s)` affiche le code Unicode du premier caractère de la chaîne `s`

```
>>> chr(65)
'A'
>>> chr(945)
'α'
>>> chr(128169)
'💩'
>>> ord('B')
66
>>> ord('🐎')
128526
```

⚠: pour les « emoji », il faut que les polices soient installées correctement sinon on risque de voir un caractère de susbtitution « 🞜 »

Classe `str`

Certaines opérations sur les chaînes sont en fait des méthodes de la class `str`.

```
>>> t='Bonjour, ça va ?'
>>> t.upper()
'BONJOUR, ÇA VA ?'
>>> t.lower()
'bonjour, ça va ?'
>>> t.split(' ')
['Bonjour,', 'ça', 'va', '?']
>>> u=['Oui', 'ça', 'va' ]
>>> "_".join(u)
'Oui_ça_va'
```

Attention, il est encore un peu tôt pour justifier de cette notation. Donc on ne l'explique pas pour l'instant.

Fonctions sur les chaînes

Si on a une chaîne `t` :

- `t.upper()` : renvoie une copie de la chaîne en majuscules
- `t.lower()` : renvoie une copie de la chaîne en minuscules
- `t.split(c)` : découpe la chaîne en utilisant le premier caractère de `c` comme s'éparateur et renvoie les éléments dans un tableau
- `t.join(tab)` : prends les chaînes du tableau `tab` et les concatène en les séparant par `t`.

C'est équivalent à `tab[0] + t + tab[1] + t + … + t + tab[len(tab)-1]`

Conversions de type

En python, tous les constructeurs des classes de base permettent de convertir des valeurs en ce type, si la conversion a un sens :

```
>>> int("123")
123
>>> str(25.4)
'25.4'
>>> bool(45)
True
>>> float(18)
18.0
>>> float(9999999999999999)
1e+16
>>> int('Hello')
Traceback (most recent call last):
  File "", line 1, in
ValueError: invalid literal for int() with base 10: 'Hello'
```

Plan

Définition de variables

Une **variable** est un moyen de donner un **nom** au résultat d'un calcul.
En Python, une variable est une suite de caractères qui commence par une lettre ou un « _ » et contient des lettres, des chiffres ou des « _ ».
On définit une variable avec le symbole « = ».

```
>>> x = 314
>>> y = 2500 * 2
>>> x
314
>>> y
5000
>>> x + y
5314
>>> toto_A1 = x + y
>>> toto_A1 + 10
5324
```

Affichage des erreurs

Lorsqu'une erreur se produit en mode programme, le programe Python s'interrompt.
Le message d'erreur indique la ligne ou se situe l'erreur et la raison de cette dernière.

En mode interactif, la ligne n'est pas affichée (c'est forcément la dernière ligne saisie), et l'invite attend de nouvelles instructions Python.

Les comparaisons

Les booléens servent à exprimer le résultat d'un **test**. Un cas particulier de test sont les comparaisons. Les opérateurs de comparaisons en Python sont :

Symbole	Description
==	égal
!=	différent
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

Attention : dans les premiers cours on ne comparera que des nombres. Les comparaisons d'autres types (chaînes de caractères par exemple) seront expliquée plus tard.

Les comparaisons (exemples)

Le résultat d'une comparaison est toujours un booléens (`True` ou `False`) :

```
>>> 2 == 1 + 1
True
>>> 3 <= 10
True
>>> x = 4
>>>  x > 3 and x < 8
True
>>>  not (x == 4)
False
```

Attention : ne pas confondre « = » (affectation d'une variable) et « == » (égalité).

Instruction `if/else`

La syntaxe de l'instruction `if` est :

```
if e:
    i1
    i2
    ...
else:
    j1
    j2
    ...
suite
```

- `e` est une expression booléene (dont le résultat est `True` ou `False`)
- Les instructions i_n sont exécutée si `e` vaut `True`
- Les instructions j_n sont exécutée si `e` vaut `False`
- La partie `else:` est optionnelle
- Suite est exécuté après la dernière instruction i_n ou j_n
- Les instructions i_n et j_n **doivent être décalées de `4` espaces**.

Blocs d'instructions

Python est un langage dans lequel l'indentation est **significative**. C'est à dire qu'on ne peut pas mettre des retours à la ligne ou des espaces n'importe où.
L'indentation indique des **blocs** d'instructions qui appartiennent au même contexte. Exemple :

```
x = 45
if x < 10:
    print ("on a testé x")
    print ("x est plus petit que 10")
```

Le code ci-dessus n'affiche rien.

```
x = 45
if x < 10:
    print ("on a testé x")
print ("x est plus petit que 10")
```

Le code ci-dessus affiche « `x est plus petit que 10` ». L'absence d'indentation mets le deuxième `print` en dehors du `if`

Comparaison avec C++

En Python, l'indentation joue le même rôle que les accolades en C++

```
int x = 45
if (x < 10) {
    print ("on a testé x");
    print ("x est plus petit que 10");
}
```

```
int x = 45
if (x < 10) {
    print ("on a testé x");
}
print ("x est plus petit que 10");
```

(attention, il n'y a pas de fonction `print` prédéfinie en C++, c'est juste pour l'exemple)

Commentaires

On peut ajouter des commentaires dans un fichier Python.

Un commentaire est toute séquence de caractères qui commence par « # » jusqu'à la fin de la ligne.

```
if x > 10 and x < 100:
    # x est dans les bonnes bornes de température
    y = x * 10
else:
    # trop froid ou trop chaud, on réinitialise y
    y = 0
```

L'instruction `while`

En Python, l'instruction `while` permet de répeter un **bloc d'instructions** tant qu'une condition est vraie :

```
while e:
    i1
    …
    in
isuite
```

Le bloc d'instructions i_1, ..., i_n est réptété tant que `e` s'évalue en `True`.

Le bloc d'instructions i_1, ..., i_n est appelé le **corps** de la boucle.

Boucle `for`

Le langage python ne possède qu'une boucle de type « for each » qui itère sur les éléments d'une collection :

```
for var in col:
    i1
    …
    in
isuite
```

La variable `var` prend tour à tour les valeurs de la collection `col`. Cette dernière peut être un tableau, une chaîne, ...

La fonction `range`

On peut faire une boucle `for` sur les entiers en utilisant la fonction `range(i, e, k)` qui renvoie une collection des entiers compris entre `i` (inclus) et `e`(exlus) par pas de `k`.
Évidemment, cette collection n'est pas construite explicitement, et il est idiomatique d'écrire :

```
for i in range(0, 10):
    …
```

i_{suite}

Rattrapage d'exceptions

On veut parfois vouloir gérer une erreur au moment où elle se produit. On peut pour cela utiliser la construction `try/except`.

```
try:
    i1
    …
    in
except E:
    j1
    …
    jm
```

Le bloc i_1, ..., i_n est exécuté. Si une instruction lève l'exception *E*, alors il s'interrompt et le bloc j_1, ..., j_m est exécuté.

On utilisera cette construction à des endroits bien choisis, sans en abuser (généralement indiqués par l'énoncé de l'exercice). En général pour rattraper une `ValueError` levée par la fonction `int()`.

Plan

Appels de fonction

En Python, un appel de fonction ce note naturellement `f(`e_1`, …, `e_n`)`.
Certaines fonctions sont définies avec des paramètres nommés. On peut dans ce cas les appeler avec : `f(`x_1 `=` e_1`, …, `x_k`=`e_k`)`.
Une fonction peut mélanger parmètres nommés et paramètres anonymes. Dans ce cas, les paramètres anonymes sont toujours donnés en premier.

Les fonctions en Python (exemple)

En Python, le mot clé `def` premet de définir une fonction

```
def f(x,y, z):
    i1
    …
    in
```

L'instruction `return` ou `return e` permet de quitter une fonction, en renvoyant une valeur.

Passage par valeur

Python, fait du **passage par valeur** des arguments aux fonctions. Cela signifie que les arguments sont copiés (sur la pile). Si une fonction modifie ses arguments, les modifications sont locales à la fonction.

```
def f(a):
    a = 42
    print(a)

a = 18
f(a)     #affiche 42
print(a) #affiche 18
```

Dans une fonction, les paramètres se comportent **comme des variables locales**

Attention avec les tableaux

Python fait du passage par valeur, mais la valeur d'un tableau est **son adresse en mémoire**.

```
def f(tab):
    tab[0] = 42

tab = [1,2,3]
f(tab)
print(tab) #affiche [42, 2, 3]
```

On peut donc modifier les cases du tableau, mais pas la variable contenant le tableau:

```
def f(tab):
    tab = "toto"

tab = [1,2,3]
f(tab)
print(tab) #affiche [1, 2, 3]
```

Variables locales

Revenons aux fonctions en Python

```
def sumproduct(a, b, c):
    tmp = a + b
    tmp2 = tmp * c
    return tmp2
```

Les variables `tmp` et `tmp2` sont des **variables locales à la fonction**

- On ne peut pas y accéder depuis l'extérieur
- Elle « commencent à exister » quand on rentre dans la fonction
- Elle « cesse d'exister » quand on sort de la fonction

Variables globales

Une variable définie en dehors d'une fonction est une **variable globale**

```
NOMBRE_UN = 1

def suivant(x):
    return x + NOMBRE_UN

print(suivant(1))       # affiche 2
NOMBRE_UN = 17
print(suivant(1))       # affiche 18
```

Attention :

```
NOMBRE = 42
def change():
    NOMBRE = 666
    print ("Dans la fonction", NOMBRE)
    return

change()                                #affiche Dans la fonction 666
print ("Hors de la fonction", NOMBRE) #affiche Hors de la fonction 42
```

Variables globales depuis une fonction

Lorsque l'on écrit `x = e` dans une fonction (i.e. si `x=e` apparaît n'importe où dans la fonction)

- Si l'instruction `global x` a été donnée au début de la fonction alors la variable globale `x` sera crée ou modifiée
- Sinon la variable locale `x` sera crée ou modifiée

Lorsque l'on utilise une variable `x` dans une fonction

- Si une variable locale `x` existe, sa valeur est utilisée
- Sinon si une variable globale `x` existe (et que `x=e` n'apparaît pas dans la fonction), sa valeur est utilisée
- Sinon erreur : variable non définie

Exemples

```
X = 1
Y = 2

def f1(a, b):
    X = a  #X est locale
    Y = b  #Y est locale

def f2(a, b):
    global X, Y
    X = a #X global modifié
    Y = b #Y global modifié


def f3():
    return X+Y #pas de variable
               #locale, va chercher
               #les globales
```

```
def f4(a):
    X = X + a
#erreur ! X = ... indique que le
#X est local pour toute la fonction.
#mais en faisant X + a
#X n'est pas défini !
    return X
```

Plan

Les tuples (1)

Python propose le type de donnée de tuple (ou n-uplet). On écrit simplement les expressions en utilisant des parenthèses et des virgules.

```
>>> point = (1.5, -3.19)
>>> point
(1.5, -3.19)
>>> point[0]
1.5
>>> point[1]
-3.19
>>> x, y = point
>>> x + y
 -1.69
>>> point[0] = 2.2
 Traceback (most recent call last):
 File "", line 1, in <module>
 TypeError: 'tuple' object does not support item assignment
```

Les tuples (2)

- La fonction `len(t)` renvoie le nombre de composantes
- `()` est le tuple de taille 0, `(v, )` un tuple de taille 1 contenant `v`

(La virgule seule est obligatoire, sinon `(v)` est compris comme `v` entouré de parenthèses « mathématiques ».

- On peut utiliser le `+` et le `*` comme pour des tableaux

```
>>> t = (1, 2) + (3, 4, 5)
>>> len(t)
5
>>> t
(1, 2, 3, 4, 5)
>>> (42, ) * 5
 (42, 42, 42, 42, 42)
>>> (42) * 5
 210
```

Les tuples (exemple)


```
#On représente des points par un couple (x, y)
from math import sqrt

def add_point(p1, p2):
    return (p1[0] + p2[0], p1[1] + p2[1])

def mult_point(p, k):
   return (p1[0] * k, p1[1] * k)

def norm_point(p):
    x, y = p
    return sqrt(x ** 2 + y ** 2)
…
```

Le type None

En Python, la constante `None` est une valeur spéciale indiquant une « absence de valeur ». Peut être utilisée dans plusieurs cas :

- Une fonction qui ne fait pas de `return` renvoie la valeur `None`
- Dans une fonction, on peut vouloir renvoyer `None` plutôt que lever une erreur.

Exemple :

```
def moyenne(tab):
    if len(tab) == 0:
        return None
    total = 0
    for i in range(len(tab)):
        total += tab[i]
    return total / len(tab)


moyenne([1, 2, 3, 4])  #renvoie 2.5
moyenne([])            #renvoie None
```

Le type None (2)

On peut tester qu'une valeur est `None` avec l'opérateur d'égalité (==). Si une fonction peut renvoyer `None`, alors il faut **toujours** tester son résultat avant de l'utiliser, sinon on risque des erreurs:

```
moy = moyenne(tab)
if moy == None:
    print("Erreur, tableau vide !")
else:
    print("Moyenne au carré:", moy * moy)
```

Sans le test, on aurait calculé `None * None` dans le cas d'un tableau vide, ce qui aurait provoqué une erreur.

Les dictionnaires

Python propose le type de donnée de dictionnaire.
Il est similaire aux tableaux, mais les indices peuvent être (presque) n'importe quel type de données Python.
On définit un dictionnaire vide par des `{ }`
On peut pré-remplir le dictionnaire avec la notation `{ k1:v1, …, kv:vn }`.

```
>>> jours = { 'lundi':1, 'mardi':2, 'mercredi':3 }
>>> jours['mardi']
2
>>> jours
{'lundi':1, 'mardi':2, 'mercredi':3}
>>> jours['jeudi'] = 4
>>> jours
 {'lundi':1, 'mardi':2, 'mercredi':3, 'jeudi' : 4}
>>> jours['jeudi'] = 42
>>> jours
 {'lundi':1, 'mardi':2, 'mercredi':3, 'jeudi' : 42}
```

Les dictionnaires (2)

Accéder à une clé inexistante est similaire à faire un accès invalide dans un tableau. L'opérateur `in` permet de tester si une clé est dans le dictionnaire :

```
>>> jours['toto']
Traceback (most recent call last):
  File "", line 1, in <module>
KeyError: 'toto'
>>> 'mardi' in jours
True
>>> 'toto' in jours
>>> jours
 False
```

On peut utiliser d'autres types de valeur pour les clés (entiers, booléens). L'utilisation la plus fréquente reste les chaînes de caractères.

Attention, comme les tableaux, les dictionnaires sont mutables!

Modules en Python

En Python, chaque fichier `.py` définit un **module**, c'est à dire un ensemble de fonction et de variables.

Par défaut, on ne peut référencer que des fonctions et variables du fichier (≡ du module) dans lequel on se trouve.

La directive `import` permet d'importer tout ou partie des fonctions et variables d'un module.

import

La première chose que l'on peut faire est d'importer tout un module.

```
import math

y = math.sin(math.pi / 2)
z = math.sqrt(499)
t = math.log(29)
```

Lorsque l'on écrit `import toto`, l'interprète Python cherche dans le répertoire courant, puis dans les répertoire systèmes (dans cet ordre par défaut) un fichier `toto.py`. S'il le trouve, il l'évalue :

- Toutes les instructions se trouvant directement dans le fichier sont exécutées
- Toutes les fonctions et les variables définies dans ce fichier sont accessibles en les préfixant avec `toto.`

Attention, pour cette raison, il ne faut jamais appeler un de ses fichiers comme un fichier de la bibliothèque standard !

`import` partiel

Parfois, on ne souhaite importer qu'un petit nombre de fonctions.
On peut utiliser la directive `from … import` :

```
from math import sin, sqrt, pi

y = sin(pi / 2)
z = sqrt(499)
```

Dans le code ci-dessus, seul `sin`, `sqrt` et `pi` sont visibles.

La forme : `from foo import *` importe tous les symboles, sans préfixe. Elle est à proscrire dorénavant. Pourquoi ?

Pourquoi utiliser des modules ?

Il y a deux aspects contradictoires :

- Toutes les fonctions doivent avoir un nom distinct
- On doit utiliser des noms les plus courts mais les plus descriptifs possibles.

« Your variable names should be short, and sweet and to the point » (L. Torvalds)
Exemple :

- Le module `math` de Python définit une fonction logarithme. Elle s'appelle `log`
- Le module `logging` de Python définit une fonction permettant d'afficher des messages d'erreurs dans la console et dans des fichiers. Elle s'appelle `log`
(c'est le terme en anglais)

Sans système de module, on aurait du utiliser une convention arbitraire par exemple `math_log` et `console_log`. C'est moche.

Pourquoi import * c'est mal ?


```
from logging import log, WARNING, ERROR
from math import *


…
log (WARNING, "attention !") #Erreur  utilise math.log()
…
log (ERROR, "erreur fatale !")
```

Dans le code ci-dessus, on a masqué involontairement la fonction `log` du module `logging`, par une fonction qui fait complètement autre chose.

Plan

Affichages avec `print`

La fonction `print(…)` affiche tous ses arguments anonymes les uns à la suite des autres, séparés par des espaces et avec un retour à la ligne final. Certains arguments nommés permettent de modifier le comportement de la fonction

- `sep=` permet de choisir la chaîne de séparation (par défaut `' '`)
- `end=` permet de choisir la chaîne de fin de ligne (par défaut `'\n'`)
- `file=` permet de choisir le fichier de sortie (par défaut `sys.stdout`. On peut le remplacer par `sys.stderr` ou un fichier ouvert par `open(…)`)
- `flush=` permet de forcer les écritures dans le fichier (par défaut vaut `False`, les écritures sont faites selon des conditions sytème)

Chaînes formatées

À partir de Python 3.6, une syntaxe existe pour les chaînes de format : `f"…"` (le `f` fait partie de la syntaxe).
Une telle chaîne peut contenir du code entre accolades :

- `{e:f}` où `e` est une expression arbitraire et `:f`, optionnel est une indication de format comme pour `printf` en C
- `{{` ou `}}` pour insérer une accolade ouvrante ou fermante

```
>>> x, y = 1, 41
>>> f"{x} + {y} = {x+y}"
'1 + 41 = 42'
>>> f"{x} + {y} = 0x{x+y:x} en hexa"
'1 + 41 = 0x2a en hexa'
```

Pour les versions antérieures, la méthode `str.format(…)` fourni des fonctionalités similaires, mais plus limitées.

Lecture et écriture de fichiers

En python, la fonction `open(chemin, mode)` permet d'ouvrir un fichier pour le lire.

- `chemin` est une chaîne de caractères contenant le chemin vers le fichier. Le chemin peut être absolu (commener par `/`) ou relatif. Il ne peut **pas** contenir de caractères spéciaux du shell tel que ~ ou des motifs glob (`[a-z]*.txt`)
- `mode` est une chaîne de caractères indiquant le mode d'ouverture :

`r`	Le fichier est ouvert en lecture seule.
`w`	Le fichier est ouvert en écriture seule. Le fichier est créé s'il n'existe pas et vidé de son contenu s'il existe.
`w+`	Comme `w`, mais aussi accès en lecture.
`a`	Le fichier est ouvert en écriture et lecture. Le fichier est créé s'il n'existe pas. Le contenu est conservé si le fichier existe.
`a+`	Comme `a`, mais aussi accès en lecture.

Attention aux erreurs

Les opérations sur les fichiers peuvent lever des exceptions (des erreurs)

- Si le chemin n'existe pas
- Si le fichier est un répertoire (et pas un fichier)
- Si on a pas les droits nécessaires (par exemple pas les droits en écriture et qu'on ouvre avec le mode `w`)
- Si on essaye d'écrire dans le fichier et qu'il n'y a plus de places sur le disque

On pourra utiliser `try`/`except` : pour rattraper certaines de ces erreurs.

Opérations sur les fichiers

Le résultat de `open(...)` est une valeur spéciale, appelée **descripteur de fichier**. C'est un objet opaque qui possède de nombreuses opérations. On se limite aux plus simples.
On suppose que dans le répertoire courant, on a un fichier `test.txt`.

```
>>> f = open("test.txt", "r")
>>> f
<_io.TextIOWrapper name='test.txt' mode='r' encoding='UTF-8'>
>>> lignes = f.readlines()
>>> lignes
['Heureux qui, comme Ulysse, a fait un beau voyage,\n',
 'Ou comme cestuy-là qui conquit la toison,\n',
 "Et puis est retourné, plein d'usage et raison,\n", 'Vivre entre ses parents le
>>>
```

L'opération `f.readlines()` renvoie le tableau de toutes les lignes du fichier `f`. Les retours à la ligne sont conservés.

Opérations sur les fichiers (2)

```
>>> for i in range(len(lignes)):
...     ligne[i] = ligne[i].upper()
>>> lignes
['HEUREUX QUI, COMME ULYSSE, A FAIT UN BEAU VOYAGE,\n',
 'OU COMME CESTUY-LÀ QUI CONQUIT LA TOISON,\n',
 "ET PUIS EST RETOURNÉ, PLEIN D'USAGE ET RAISON,\n", … ]
>>> f2 = open ("test2.txt", "w")
>>> f2.writelines(lignes)
>>> f2.close()
>>> # on quitte Python et on est dans le terminal

$ ls
test.txt test2.txt
$ cat test2.txt
  …
PLUS MON LOIR GAULOIS, QUE LE TIBRE LATIN,
PLUS MON PETIT LIRÉ, QUE LE MONT PALATIN,
ET PLUS QUE L'AIR MARIN LA DOULCEUR ANGEVINE.
```

Opérations sur les fichiers (3)

L'opération `f.writelines(tab)` écrit le tableau de chaînes de caractères dans le fichier `f`. Lorsque l'on a fini, il faut refermer le fichier en appeleant `f.close()` sinon il se peut que certaines lignes ne soient pas écrites dans le fichier.

Construction `with as`

Lors de l'utilisation de certaines ressources (fichiers, connexions réseau, ...) il est souvent utile de pouvoir dire « lorsque la ressource n'est plus utilisée, libérer la ressource ». Une construction spéciale existe en Python :

```
with open("mon fichier", "rw") as f:
    lines = f.readlines()
    for l in range(len(lines)):
        lines[i] = lines[i].upper()

    f.writelines(lines)
```

Dans le code ci-dessus, le fichier `f` est refermé (avec `f.close()`) lorsque l'on quite le bloc `with`, de quelque façon que ce soit (fin du bloc, exception, return, ...).