Python 102

Plus loin avec Python

Squelette d'un module

- Même un fichier destiné à être utilisé comme un script doit être importable
- Un fichier important un module ne devrait subir d'effet de bord en exécutant les fonctionalités du module importé
- En Python des outils comme ***pydoc*** ou ceux de tests unitaires ont besoin d'importer vos modules
- Votre code doit toujours vérifier ***if __name__ == '__main__'*** avant d'exécuter votre programme principal
- Ainsi, ce programme principal ne sera pas exécuté lorsque le module est importé
- Un squelette de module

Complément sur les listes

- Le type liste dispose de méthodes supplémentaires
- Voici la liste complète des méthodes des objets de type liste
  - ***list.append(x)***
    - Ajoute un élément à la fin de la liste. Equivalent à ***a[len(a):] = [x]***
  - ***list.extend(iterable)***
    - Étend la liste en y ajoutant tous les éléments de l'iterable. Équivalent à ***a[len(a):] = iterable***
  - ***list.insert(i, x)***
    - Insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc ***a.insert(0, x)*** insère l'élément en tête de la liste, et ***a.insert(len(a), x)*** est équivalent à ***a.append(x)***
  - ***list.remove(x)***
    - Supprime de la liste le premier élément dont la valeur est x. Une exception est levée s'il existe aucun élément avec cette valeur

Complément sur les listes

- ***list.pop([i])***
  - Enlève de la liste l'élément situé à la position indiquée, et le renvoie en valeur de retour. Si aucune position n'est indiqué, ***a.pop()*** enlève et renvoie le dernier élément de la liste (les crochets autour du i dans la signature de la méthode indiquent bien que ce paramètre est facultatif, et non que vous devez placer des crochets dans votre code
- ***list.clear()***
  - Supprime tous les éléments de la liste, équivalent à ***del a[:]***
- ***list.index(x[, start[, end]])***
  - Renvoie la position du premier élément de la liste ayant la valeur x (en commençant par zéro). Une exception ***ValueError*** est levée si aucun élément n'est trouvé.
  - Les arguments optionnels start et end sont interprétés de la même manière que dans la notation des tranches, et sont utilisés pour limiter la recherche à une sous-séquence particulière. L'index renvoyé est calculé relativement au début de la séquence complète, et non relativement à start
- ***list.count(x)***
  - Renvoie le nombre d'éléments ayant la valeur x dans la liste

Complément sur les listes

- ***list.sort(key=None, reverse=False)***
  - Trie les éléments sur place, (les arguments peuvent personnaliser le tri, voir ***sorted()*** pour leur explication)
- ***list.reverse()***
  - Inverse l'ordre des éléments de la liste, sur place
- ***list.copy()***
  - Renvoie une copie superficielle de la liste. Équivalent à ***a[:]***

- Un exemple utilisant la plupart de ces méthodes

Utiliser les listes comme des piles

- Les méthodes des listes rendent très facile leur utilisation comme des piles, où le dernier élément ajouté est le premier récupéré (« dernier entré, premier sorti », ou LIFO pour « last-in, first-out »)
- Pour ajouter un élément sur la pile, utilisez la méthode append()
- Pour récupérer l'objet au sommet de la pile, utilisez la méthode pop(), sans indicateur de position

lists-as-piles.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4
5  def main():
6      """Lists as piles
7      """
8
9      # Create a stack
10     my_stack = [1, 2, 3, 4]
11     print("my_stack", my_stack)
12
13     # Push values on the stack
14     my_stack.append(5)
15     my_stack.append(6)
16     my_stack.append(7)
17     print("my_stack", my_stack)
18
19     # Pop values from the stack
20     print("Poped value", my_stack.pop())
21     print("my_stack", my_stack)
22     print("Poped value", my_stack.pop())
23     print("my_stack", my_stack)
24     print("Poped value", my_stack.pop())
25     print("my_stack", my_stack)
26     print("Poped value", my_stack.pop())
27     print("my_stack", my_stack)
28
29
30 if __name__ == '__main__':
31     main()
32
```

Utiliser les listes comme des files

- Il est également possible d'utiliser une liste comme une file, où le premier élément ajouté est le premier récupéré (« premier entré, premier sorti », ou FIFO pour « first-in, first-out »)
- Toutefois, les listes ne sont pas très efficaces pour ce type de traitement
- Alors que les ajouts et suppressions en fin de liste sont rapides, les opérations d'insertions ou de retraits en début de liste sont lentes (car tous les autres éléments doivent être décalés d'une position).
- Pour implémenter une file, utilisez donc la classe ***collections.deque*** qui a été conçue pour fournir des opérations d'ajouts et de retraits rapides aux deux extrémité

queues.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from collections import deque
5
6
7  def main():
8      """Queues
9      """
10
11     # Create a queue
12     my_queue = deque([1, 2, 3])
13     print(my_queue)
14
15     # Append element to the right
16     my_queue.append(10)
17     print(my_queue)
18
19     # Append element to the left
20     my_queue.appendleft(100)
21     print(my_queue)
22
23     # Pop from the right
24     print("Deque", my_queue.pop())
25     print(my_queue)
26
27     # Pop from the left
28     print("Deque", my_queue.popleft())
29     print(my_queue)
30
31
32 if __name__ == '__main__':
33     #
34     # Call the main() function
35     #
36     main()
37
```

Compréhensions de listes

- Les compréhensions de listes fournissent un moyen de construire des listes de manière très concise
- Une application classique est la construction de nouvelles listes où chaque élément est le résultat d'une opération appliquée à chaque élément d'une autre séquence, ou de créer une sous-séquence des éléments satisfaisant une condition spécifique
- Par exemple, supposons que l'on veuille créer une liste de carrés
- Notez que cela créé (ou écrase) une variable nommée x qui existe toujours après l'exécution de la boucle
- On peut calculer une liste de carrés sans effet de bord, avec :

```
>>> carres = []
>>> for x in range(10):
...     carres.append(x**2)
...
>>> carres
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

```
>>> carres = [x**2 for x in range(10)]
>>> carres
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

Compréhensions de listes

- Une compréhension de liste consiste en crochets contenant une expression suivie par une clause for, puis par zéro ou plus clauses ***for*** ou ***if***
- Le résultat sera une nouvelle liste résultat de l'évaluation de l'expression dans le contexte des clauses ***for*** et ***if*** qui la suivent
- Par exemple, cette compréhension de liste combine les éléments de deux listes s'ils ne sont pas égaux (Equivalent, en plus concis que la première solution)

```
>>> combines = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...             if x != y:
...                     combines.append((x,y))
...
>>> combines
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
>>>
```

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Compréhensions de listes

- Les compréhensions de listes peuvent contenir des expressions complexes et des fonctions imbriquées

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
>>>
```

Compréhensions de listes imbriquées

- La première expression dans une compréhension de liste peut être n'importe quelle expression, y compris une autre compréhension de liste

```
>>> matrice = [
...     [1,2,3,4],
...     [5,6,7,8],
...     [9,10,11,12]
... ]
>>> [[row[i] for row in matrice] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
>>>
```

Compréhensions de listes imbriquées

- Les exemples suivants sont équivalents

```
>>> transpose = []
>>> for i in range(4):
...     transpose.append([row[i] for row in matrice])
...
>>> transpose
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
>>>
```

```
>>> transpose = []
>>> for i in range(4):
...     transposed_row = []
...     for row in matrice:
...             transposed_row.append(row[i])
...     transpose.append(transposed_row)
...
>>> transpose
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
>>>
```

Compréhensions de listes imbriquées

- Dans des cas concrets, il est toujours préférable d'utiliser des fonctions natives plutôt que des instructions de contrôle de flux complexes
- La fonction ***zip()*** ferait dans ce cas un excellent travail

```
>>> list(zip(*matrice))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
>>>
```

L'instruction *del*

- Il existe un moyen de retirer un élément d'une liste à partir de sa position au lieu de sa valeur : l'instruction ***del***
- Elle diffère de la méthode pop() qui, elle, renvoie une valeur
- L'instruction ***del*** peut également être utilisée pour supprimer des tranches d'une liste ou la vider complètement (ce que nous avions fait auparavant en affectant une liste vide à la tranche)

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
>>>
```

Tuples & séquences

- Nous avons vu que les listes et les chaînes de caractères ont beaucoup de propriétés en commun, comme l'indexation et les opérations sur des tranches
- Ce sont deux exemple de séquences (voir Types séquentiels — list, tuple, range)
- Comme Python est un langage en constante évolution, d'autres types de séquences y seront peut-être ajoutés
- Il existe également un autre type standard de séquence : le tuple
- Un tuple consiste en différentes valeurs séparées par des virgules

```
>>>
>>> t = 12345, 54321, 'hello!'
>>> t
(12345, 54321, 'hello!')
>>> # Les tuples peuvent etre imbriques
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Les tuples sont imuables
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # mais ils peuvent contenir des objets muables
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
>>>
```

Tuples & séquences

- Comme vous pouvez le voir, à l'affichage les tuples sont toujours encadrés par des parenthèses, de façon à ce que des tuples imbriqués soient interprétés correctement
- Ils peuvent être entrés avec ou sans parenthèses, même si celles-ci sont souvent nécessaires (notamment lorsqu'un tuple fait partie d'une expression plus longue)
- Il n'est pas possible d'affecter de valeur à un élément d'un tuple ; par contre, il est possible de créer des tuples contenant des objets mutables, comme des listes
- Si les tuples peuvent sembler similaires aux listes, ils sont souvent utilisés dans des cas différents et pour des raisons différentes
- Les tuples sont immutables et contiennent souvent des séquences hétérogènes d'éléments qui sont accédés par « déballage » (voir plus loin) ou indexation (ou même par attributs dans le cas des ***namedtuples***)
- Les listes sont souvent mutable, et contiennent des éléments homogènes qui sont accédés par itération sur la liste

Tuples & séquences

- Un problème spécifique est la construction de tuples ne contenant aucun ou un seul élément : la syntaxe a quelques tournures spécifiques pour s'en accommoder
- Les tuples vides sont construits par une paire de parenthèses vides
- Un tuple avec un seul élément est construit en faisant suivre la valeur par une virgule (il n'est pas suffisant de placer cette valeur entre parenthèses)
- Pas très joli, mais efficace
- L'instruction ***t = 1, 2, 3*** est un exemple d'un emballage de tuple : les valeurs 1, 2 et 3 sont emballées ensemble dans un tuple
- L'opération inverse est aussi possible

```
>>> empty = ()
>>> singleton = 'Bonjour',
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('Bonjour',)
>>>
```

```
>>> t = 1,2,3
>>> t
(1, 2, 3)
>>> x, y, z = t
>>> x
1
>>> y
2
>>> z
3
>>>
```

Tuples & séquences

- Ceci est appelé, de façon plus ou moins appropriée, un déballage de séquence et fonctionne pour toute séquence placée à droite de l'expression
- Ce déballage requiert autant de variables dans la partie gauche qu'il y a d'éléments dans la séquence
- Notez également que cette affectation multiple est juste une combinaison entre un emballage de tuple et un déballage de séquence

Les ensembles

- Python fournit également un type de donnée pour les ensembles
- Un ensemble est une collection non ordonnée sans élément dupliqué
- Des utilisations basiques concernent par exemple des tests d'appartenance ou des suppressions de doublons
- Les ensembles supportent également les opérations mathématiques comme les unions, intersections, différences et différences symétriques
- Des accolades, ou la fonction set() peuvent être utilisés pour créer des ensembles
- Notez que pour créer un ensemble vide, {} ne fonctionne pas, cela créé un dictionnaire vide
- Utilisez plutôt set()

Les ensembles

- *a - b* : élément de a absents de b
- *a | b* : éléments dans ou dans b ou dans les deux
- *a & b* : éléments dans a et dans b
- *a ^ b* : éléments dans a ou dans b mais pas dans les deux

```
>>> panier = {'pomme', 'orange', 'pomme', 'poire', 'orange', 'banane'}
>>> print(panier)
{'poire', 'orange', 'pomme', 'banane'}
>>> 'orange' in panier
True
>>> 'raisin' in panier
False
>>>
```

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'d', 'a', 'r', 'c', 'b'}
>>> a - b
{'d', 'b', 'r'}
>>> a | b
{'l', 'd', 'a', 'z', 'r', 'm', 'c', 'b'}
>>> a & b
{'c', 'a'}
>>> a ^ b
{'l', 'd', 'z', 'r', 'm', 'b'}
>>>
```

Les ensembles

- Tout comme les compréhensions de listes, il est possible d'écrire des compréhensions d'ensemble

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'d', 'r'}
>>>
```

Les dictionnaires

- Un autre type de donnée très utile, natif dans Python, est le dictionnaire
- Ces dictionnaires sont parfois présents dans d'autres langages sous le nom de « mémoires associatives » ou de « tableaux associatifs »
- À la différence des séquences, qui sont indexées par des nombres, les dictionnaires sont indexés par des clés, qui peuvent être de n'importe quel type immutable
- Les chaînes de caractères et les nombres peuvent toujours être des clés
- Des tuples peuvent être utilisés comme clés s'ils ne contiennent que des chaînes, des nombres ou des tuples
- Si un tuple contient un objet mutable, de façon directe ou indirecte, il ne peut pas être utilisé comme une clé
- Vous ne pouvez pas utiliser des listes comme clés, car les listes peuvent être modifiées en place en utilisant des affectations par position, par tranches ou via des méthodes comme ***append()*** ou ***extend()***

Les dictionnaires

- Le plus simple est de considérer les dictionnaires comme des ensembles non ordonnés de paires clé: valeur, les clés devant être uniques (au sein d'un dictionnaire)
- Une paire d'accolades crée un dictionnaire vide : {}
- Placer une liste de paires ***clé:valeur*** séparées par des virgules à l'intérieur des accolades ajoute les valeurs correspondantes au dictionnaire
- C'est également de cette façon que les dictionnaires sont affichés en sortie
- Les principales opérations effectuées sur un dictionnaire consistent à stocker une valeur pour une clé et à extraire la valeur correspondant à une clé
- Il est également possible de supprimer une paire ***clé:valeur*** avec ***del***
- Si vous stockez une valeur pour une clé qui est déjà utilisée, l'ancienne valeur associée à cette clé est perdue
- Si vous tentez d'extraire une valeur associée à une clé qui n'existe pas, une exception est levée

Les dictionnaires

- Exécuter ***list(d.keys())*** sur un dictionnaire d renvoie une liste de toutes les clés utilisées dans le dictionnaire, dans un ordre arbitraire (si vous voulez qu'elles soient triées, utilisez ***sorted(d.keys())***)
- Pour tester si une clé est dans le dictionnaire, utilisez le mot-clé ***in***

```
>>> telephones = {'Thierry': 5001, 'Jack': 5010, 'Joe': 5020}
>>> telephones['Jack']
5010
>>> del telephones['Jack']
>>> telephones
{'Thierry': 5001, 'Joe': 5020}
>>> telephones['Joe'] = 5010
>>> telephones
{'Thierry': 5001, 'Joe': 5010}
>>> list(telephones.keys())
['Thierry', 'Joe']
>>> sorted(telephones.keys())
['Joe', 'Thierry']
>>> 'Thierry' in telephones
True
>>> 'Jack' in telephones
False
>>>
```

Les dictionnaires

- Le constructeur ***dict()*** fabrique un dictionnaire directement à partir d'une liste de paires clé:valeur stockées sous la forme de tuples
- De plus, il est possible de créer des dictionnaires par compréhension depuis un jeu de clef et valeurs
- Lorsque les clés sont de simples chaînes de caractères, il est parfois plus facile de spécifier les paires en utilisant des paramètres nommés

```
>>> dict( [('Thierry',5010),('Joe',5020),('Jack',5030)] )
{'Thierry': 5010, 'Joe': 5020, 'Jack': 5030}
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
>>> dict(Thierry=5010, Joe=5020, jack=5030)
{'Thierry': 5010, 'Joe': 5020, 'jack': 5030}
>>>
```

Techniques de boucles

- Lorsque vous faites une boucle sur un dictionnaire, les clés et leurs valeurs peuvent être récupérées en même temps en utilisant la méthode ***items()***
- Lorsque vous itérez sur une séquence, la position et la valeur correspondante peuvent être récupérées en même temps en utilisant la fonction ***enumerate()***

```
>>> chevalier = { 'Gallahad':'Le pur', 'Robin':'Le brave', 'Lancelot':'Du lac' }
>>> for k, v in chevalier.items():
...     print(k,v)
...
Gallahad Le pur
Robin Le brave
Lancelot Du lac
>>> for i, v in enumerate(['Tic','Tac','Toe']):
...     print(i, v)
...
0 Tic
1 Tac
2 Toe
>>>
```

Techniques de boucles

- Pour faire des boucles sur deux séquences ou plus en même temps, les éléments peuvent être associés par la fonction ***zip()***

```
>>> questions = ['Nom', 'Quete', 'Couleur favorite']
>>> reponses = ['Lancelot', 'Le saint-graal', 'Bleue']
>>> for q, a in zip(questions, reponses):
...     print("Quel est votre {} ? C'est {}.".format(q, a))
...
Quel est votre Nom ? C'est Lancelot.
Quel est votre Quete ? C'est Le saint-graal.
Quel est votre Couleur favorite ? C'est Bleue.
>>>
```

Techniques de boucles

- Pour faire une boucle sur une séquence inversée, commencez par créer la séquence dans son ordre normal, puis appliquez la fonction ***reversed()***
- Pour faire une boucle sur une séquence triée, utilisez la fonction ***sorted()***, qui renvoie une nouvelle liste triée sans altérer la source

```
>>> for i in reversed(range(1,10,2)):
...     print(i)
...
9
7
5
3
1
>>>
```

```
>>> panier = ['Pomme', 'Poire', 'Banane', 'Orange', 'Abricot', 'Banane']
>>> for f in sorted(set(panier)):
...     print(f)
...
Abricot
Banane
Orange
Poire
Pomme
>>>
```

Techniques de boucles

- Il est parfois tentant de changer une liste pendant son itération, cependant, c'est souvent plus simple et plus sûr de créer une nouvelle liste à la place

```
>>> donnees_brutes = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> donnees_brutes
[56.2, nan, 51.7, 55.3, 52.5, nan, 47.8]
>>> donnees_filtrees = []
>>> for valeur in donnees_brutes:
...     if not math.isnan(valeur):
...             donnees_filtrees.append(valeur)
...
>>> donnees_filtrees
[56.2, 51.7, 55.3, 52.5, 47.8]
>>>
```

- Ou bien

```
>>> donnes_filtrees = [valeur for valeur in donnees_brutes if not math.isnan(valeur)]
>>> donnees_filtrees
[56.2, 51.7, 55.3, 52.5, 47.8]
>>>
```

Plus d'informations sur les conditions

- Les conditions utilisées dans une instruction ***while*** ou ***if*** peuvent contenir n'importe quel opérateur, pas seulement des comparaisons
- Les opérateurs de comparaison ***in*** et ***not*** in testent si une valeur est présente ou non dans une séquence
- Les opérateurs ***is*** et ***is not*** testent si deux objets sont vraiment le même objet
- Ceci n'est important que pour des objets mutables comme des listes
- Tous les opérateurs de comparaison ont la même priorité, qui est plus faible que celle des opérateurs numériques
- Les comparaison peuvent être enchaînées
- Par exemple, ***a < b == c*** teste si a est inférieur ou égal à b et par ailleurs si b est égal à c

Plus d'informations sur les conditions

- Les comparaisons peuvent être combinées en utilisant les opérateurs booléens ***and*** et ***or***, le résultat d'une comparaison (ou de toute expression booléenne) pouvant être inversé avec ***not***
- Ces opérateurs ont une priorité inférieure à celle des opérateurs de comparaison
- Entre eux, not a la priorité la plus élevée et or la plus faible, de telle sorte que ***A*** and ***not B*** or ***C*** est équivalent à ***(A and (not B)) or C***
- Comme toujours, des parenthèses peuvent être utilisées pour exprimer l'instruction désirée

Plus d'informations sur les conditions

- Les opérateurs booléens ***and*** et ***or*** sont appelés opérateurs en circuit court
- Leurs arguments sont évalués de la gauche vers la droite, et l'évaluation s'arrête dès que le résultat est déterminé
- Par exemple, si ***A*** et ***C*** sont vrais et ***B*** est faux, ***A and B and C*** n'évalue pas l'expression ***C***
- Lorsqu'elle est utilisée en tant que valeur et non en tant que booléen, la valeur de retour d'un opérateur en circuit court est celle du dernier argument évalué
- Il est possible d'affecter le résultat d'une comparaison ou d'une autre expression booléenne à une variable

```
>>> stringa, stringb, stringc = '', 'Thierry', 'Decker'
>>> non_nul = stringa or stringb or stringc
>>> non_nul
'Thierry'
>>>
```

Comparer des séquences avec d'autres types

- Des séquences peuvent être comparées avec d'autres séquences du même type
- La comparaison utilise un ordre lexicographique : les deux premiers éléments de chaque séquence sont comparés, et s'ils diffèrent cela détermine le résultat de la comparaison ; s'ils sont égaux, les deux éléments suivants sont comparés à leur tour, et ainsi de suite jusqu'à ce que l'une des séquences soit épuisée
- Si deux éléments à comparer sont eux-mêmes des séquences du même type, alors la comparaison lexicographique est effectuée récursivement
- Si tous les éléments des deux séquences sont égaux, les deux séquences sont alors considérées comme égales
- Si une séquence est une sous-séquence de l'autre, la séquence la plus courte est celle dont la valeur est inférieure
- La comparaison lexicographique des chaînes de caractères utilise le code Unicode des caractères

Comparer des séquences avec d'autres types

- Quelques exemples
- Comparer des objets de type différents avec < ou > est autorisé si les objets ont des méthodes de comparaison appropriées
- Par exemple, les types numériques sont comparés via leur valeur numérique, donc 0 est égal à 0,0, etc.
- Dans les autres cas, au lieu de donner un ordre imprévisible, l'interpréteur lancera une exception ***TypeError***

```
>>> (1, 2, 3) < (1,2, 4)
True
>>> ['a', 'b','c'] > ['a', 'b', 'b']
True
>>> 'ABC' < 'C' < 'Pascal' < 'Java'
False
>>> 'ABC' < 'C'
True
>>> 'ABC' < 'C' < 'Pascal'
True
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>>
```

Modules

- Lorsque vous quittez et entrez à nouveau dans l'interpréteur Python, tout ce que vous avez déclaré dans la session précédente est perdu
- Afin de rédiger des programmes plus longs, vous devriez utiliser un éditeur de texte, préparer votre code dans un fichier, et exécuter Python avec ce fichier en paramètre
- Ça s'appelle créé un script
- Lorsque votre programme deviendra plus long encore, vous pourrez séparer votre code dans plusieurs fichiers, et vous trouverez aussi pratique de réutiliser des fonctions écrites pour un programme dans un autre sans avoir à les copier

Modules

- Pour gérer ça, Python à un moyen de rédiger des définitions dans un fichier et les utiliser dans un script ou une session interactive
- Un tel fichier est appelé un module, et les définitions d'un module peuvent être importées dans un autre module ou dans le module main (qui est le module qui contiens vos variables et définitions lors de l'exécution d'un script ou en mode interactif)
- Un module est un fichier contenant des définitions et des instructions
- Son nom de fichier est le même que son nom, suffixé de ***.py***
- À l'intérieur d'un module, son propre nom est accessible dans la variable __name__

Modules

- Créez un fichier nommé fibonacci.py ()
- Insérer le contenu du lien ci-dessus et enregistrez votre fichier
- Maintenant, en étant dans le même dossier, ouvrez un interpréteur et importez le module en tapant :

```
>>> import fibonacci
>>>
```

- Cela n'importe pas les noms des fonctions définies dans fibonacci directement dans la table des symboles courante, mais y ajoute simplement fibonacci
- Vous pouvez donc appeler les fonctions via le nom du module

```
>>> fibonacci.fib(100)
1 1 2 3 5 8 13 21 34 55 89
>>> fibonacci.fib2(1000)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
>>>
```

Les modules en détail

- Un module peut contenir aussi bien des instructions que des déclarations de fonctions
- Ces instructions permettent d'initialiser le module, et ne sont donc exécutées que la première fois que le nom d'un module est trouvé dans un import
- Elles sont aussi exécutées lorsque le fichier est exécuté en temps que script
- Chaque module a sa propre table de symboles, utilisée comme table de symboles globaux par toutes les fonctions définies par le module
- Ainsi l'auteur d'un module peut utiliser des variables globales dans un module sans se soucier de collisions de noms avec des globales définies par l'utilisateur du module
- D'un autre côté, si vous savez ce que vous faites, vous pouvez modifier une variable globale d'un module avec la même notation que pour accéder aux fonctions : modname.itemname

Les modules en détail

- Des modules peuvent importer d'autres modules
- Il est habituel mais pas obligatoire de ranger tous les import au début du module (ou du script). Les noms des module importés sont insérés dans la table des symboles globaux du module qui importe
- Il existe une variation à l'instruction import qui importe les noms d'un module directement dans la table de symboles du module qui l'importe

```
>>> from fibonacci import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>>
```

Les modules en détail

- Cela n'insère pas le nom du module depuis lequel les définitions sont récupérées dans la table locale de symboles (dans cet exemple, fibonacci n'est pas défini)
- Il existe même une variation permettant d'importer tous les noms qu'un module définit

```
>>> from fibonacci import *
>>> fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>>
```

- Tous les noms ne commençant pas par un tiret bas (_) sont importés
- Dans la grande majorité des cas, les développeurs n'utilisent pas cette syntaxe puisqu'en important un ensemble indéfini de noms, des noms déjà définis peuvent se retrouver cachés
- Notez qu'en général, importer * d'un module ou d'un paquet est déconseillé, en général ça engendre du code difficilement lisible
- Cependant, c'est acceptable de l'utiliser pour gagner quelques secondes en mode interactif

Les modules en détail

- Note :
  - Pour des raisons de performance, chaque module n'est importé qu'une fois par session
  - Si vous changez le code d'un module vous devez donc redémarrer l'interpréteur afin d'en voir l'impact
  - Ou le réimporter explicitement en utilisant ***importlib.reload()***, par exemple :
  - ***import importlib***
  - ***importlib.reload(modulename)***

Exécuter les modules comme des scripts

- Lorsque vous exécutez un module Python avec :
  - ***Python fibonacci.py <arguments>***
- le code du module sera exécuté comme si vous l'aviez importé, mais son __name__ vaudra "__main__"
- Donc en ajoutant ces lignes à la fin du module :
  - ***if __name__ == "__main__":***
  - ***import sys***
  - ***fib(int(sys.argv[1]))***
- vous pouvez rendre le fichier utilisable comme script aussi bien que comme module importable, car le code qui parse la ligne de commande n'est lancé que si le module est exécuté comme fichier "main"
- Si le fichier est importé, le code n'est pas exécuté
- C'est typiquement utilisé soit pour proposer une interface utilisateur pour un module, soit pour lancer les tests sur le module (où exécuter le module en temps que script lance les tests)

Dossiers de recherche des modules

- Lorsqu'un module nommé par exemple ***spam*** est importé, il est d'abord recherché parmi les modules natifs, puis s'il n'y est pas trouvé, l'interpréteur va chercher un fichier nommé ***spam.py*** dans une liste de dossiers donnés par la variable ***sys.path***
- ***sys.path*** est initialisée par défaut à :
  - Le dossier contenant le script courant (ou le dossier courant si aucun script n'est donné)
  - ***PYTHONPATH*** (une liste de dossiers, utilisant la même syntaxe que la variable shell ***PATH***).
  - La valeur par défaut, dépendante de l'installation
- Après leur initialisation, les programmes Python peuvent modifier leur sys.path
- Le dossier contenant le script courant est placé au début de la liste des dossiers à rechercher, avant les dossiers de bibliothèques
- Cela signifie qu'un module dans ce dossier, ayant le même nom qu'un module, sera chargé à sa place
- C'est une erreur typique, à moins que ce soit voulu. Voir Modules standards pour plus d'informations

Modules standards

- Python est accompagné d'une bibliothèque de modules standards, décrits dans la documentation de la Bibliothèque Python
- Certains modules sont intégrés dans l'interpréteur, ils exposent des outils qui ne font pas partie du langage, mais qui font tout de même partie de l'interpréteur, soit pour le côté pratique, soit pour exposer des outils essentiels tels que l'accès aux appels système
- La composition de ces modules est configurable à la compilation, et dépend aussi de la plateforme ciblée
- Par exemple, le module ***winreg*** n'est proposé que sur les systèmes Windows

Modules standards

- Un module mérite une attention particulière, le module ***sys***, qui est présent dans tous les interpréteurs Python
- Les variables ***sys.ps1*** et ***sys.ps2*** définissent les chaînes d'invites principales et secondaires
- Ces deux variables ne sont définies que si l'interpréteur est en mode interactif
- La variable ***sys.path*** est une liste de chaînes qui détermine les chemins de recherche de modules pour l'interpréteur
- Il est initialisé à un chemin par défaut pris de la variable d'environnement ***PYTHONPATH***, ou d'une valeur par défaut interne si ***PYTHONPATH*** n'est pas définie
- ***sys.path*** est modifiable en utilisant les opérations habituelles des listes

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = '--- '
---
---
```

```
>>> import sys
>>> for path in sys.path:
...     print(path)
...

C:\python\python36\python36.zip
C:\python\python36\DLLs
C:\python\python36\lib
C:\python\python36
C:\python\python36\lib\site-packages
>>>
```

La fonction *dir()*

- La fonction interne ***dir()*** est utilisée pour trouver quels noms sont définies par un module
- Elle donne une liste triée de chaînes
- Sans paramètres, ***dir()*** listes les noms actuellement définis
- ***dir()*** ne liste ni les fonctions primitives ni les variables internes. Si vous voulez les lister, ils sont définis dans le module ***builtins***

```
>>> import fibonacci, sys
>>> for name in dir(fibonacci):
...     print(name)
...
__builtins__
__cached__
__doc__
__file__
__loader__
__name__
__package__
__spec__
fib
fib2
>>> for name in dir():
...     print(name)
...
__annotations__
__builtins__
__doc__
__loader__
__name__
__package__
__spec__
fibonacci
name
sys
>>>
```

Les paquets

- Les paquets sont un moyen de structurer les espaces de noms des modules Python en utilisant une notations « pointée »
- Par exemple, le nom de module A.B désigne le sous-module B du paquet A
- De la même manière que l'utilisation des modules évite aux auteurs de différents modules d'avoir à se soucier des noms de variables globales des autres, l'utilisation des noms de modules avec des points évite aux auteurs de paquets contenant plusieurs modules tel que **NumPy** ou « Python Image Library » d'avoir à se soucier des noms des modules des autres

Les paquets

- Imaginez que vous voulez construire une collections de modules (un « paquet ») pour gérer uniformément les fichiers contenant du son et des données sonores
- Il existe un grand nombre de formats de fichiers pour stocker du son (généralement repérés par leur extension, par exemple .wav, .aiff, .au), vous aurez donc envie de créer et maintenir un nombre croissant de modules pour gérer la conversion entre tous ces formats
- Il existe aussi tout une flopée d'opérations que vous voudriez pouvoir faire sur du son (mixer, ajouter de l'écho, égaliser, ajouter un effet stéréo artificiel), donc, en plus des modules de conversion, vous allez écrire un nombre illimité de modules permettant d'effectuer ces opérations
- Voici une structure possible pour votre paquet (exprimée comme un système de fichier, hiérarchiquement)

Les paquets

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Les paquets

- Lorsqu'il importe des paquets, Python cherche dans chaque dossiers de sys.path, à la recherche du dossier du paquet
- Les fichiers ***__init__.py*** sont nécessaires pour que Python considère les dossiers comme contenant des paquets, ça évite des dossiers ayant des noms courants comme string de cacher des modules qui auraient été trouvés plus loin dans les dossiers de recherche
- Dans le plus simple des cas, ***__init__.py*** peut être vide, mais il peut exécuter du code d'initialisation pour son paquet ou configurer la variable ***__all__*** (documentée plus loin)
- Les utilisateurs d'un module peuvent importer ses modules individuellement, par exemple

```
import sound.effects.echo
```

- Chargera le sous-module ***sound.effects.echo***
- Il dit être référencé par son nom complet

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Les paquets

- Une autre manière d'importer des sous-modules est :

```
from sound.effects import echo
```

- Chargera aussi le sous-module echo, et le rendra disponible dans le préfixe du paquet, il peut donc être utilisé comme ceci :

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

- Une autre méthode consisterait à importer la fonction ou variable désirée directement

```
from sound.effects.echo import echofilter
```

- Le sous-module echo est toujours chargé, mais ici la fonction ***echofilter()*** est disponible directement

```
echofilter(input, output, delay=0.7, atten=4)
```

Les paquets

- Notez que lorsque vous utilisez ***from package import item***, item peut aussi bien être un sous-module, un sous-paquet, ou simplement un nom déclaré dans le paquet (une variable, une fonction ou une classe)
- L'instruction ***import*** cherche en premier si ***item*** est définit dans le paquet, s'il ne l'est pas, elle cherche à charger un module, et si elle n'y arrive pas, une exception ***ImportError*** est levée
- Au contraire, en utilisant la syntaxe ***import item.item.subitement.subsubitem***, chaque item sauf le dernier doivent être des paquets
- Le dernier item peut être un module ou un paquet, mais ne peut être ni une fonction, ni une classe, ni une variable défini dans l'élément précédent

Importer * depuis un paquet

- Qu'arrive-il lorsqu'un utilisateur écrit ***from sound.effects import ** ?
- Dans l'idéal on pourrait espérer que ça irait chercher tous les sous-modules du paquet sur le système de fichiers, et qu'ils seraient tous importés
- Ça pourrait être long, et importer certains sous-modules pourrait avoir des effets secondaires indésirables, du moins, désirés seulement lorsque le sous module est importé explicitement

Importer * depuis un paquet

- La seule solution, pour l'auteur du paquet, est de fournir un index explicite du contenu du paquet
- L'instruction import utilise la convention suivante :
- Si le fichier ***__init__.py*** du paquet définit une liste nommée ***__all__***, cette liste sera utilisée comme liste des noms de modules devant être importés lorsque from package import * est utilisé
- C'est le rôle de l'auteur du paquet de maintenir cette liste à jour lorsque de nouvelles version du paquet sont publiées
- Un auteur de paquet peut aussi décider de ne pas autoriser d'importer * de leur paquet. Par exemple, le fichier ***sound/effects/__init__.py*** peut contenir le code suivant :

```
__all__ = ["echo", "surround", "reverse"]
```

Importer * depuis un paquet

- Cela signifierai que ***from sound.effects import ** importait les trois sous-modules du paquet ***sound***
- Si ***__all__*** n'est pas défini, l'instruction ***from sound.effects import **** n'importera pas tous les sous-modules du paquet ***sound.effects*** dans l'espace de nom courant, mais s'assurera seulement que le paquet ***sound.effects*** à été importé (que tout le code du fichier ***__init__.py*** à été exécuté) et importe ensuite n'importe quels noms définis dans le paquet
- Cela inclut tous les noms définis (et sous modules chargés explicitement) par ***__init__.py***
- Elle inclut aussi tous les sous-modules du paquet ayant été chargés explicitement par une instruction import

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

Importer * depuis un paquet

- Dans cet exemple, les modules ***echo*** et ***surround*** sont importés dans l'espace de noms courant lorsque ***from...import*** est exécuté, parce qu'ils sont définis dans le paquet ***sound.effects***. (Cela fonctionne lorsque ***__all__*** est défini.)
- Bien que certains modules ont été pensés pour n'exporter que les noms respectant une certaine structure lorsque ***import *** est utilisé, ***import *** reste considéré comme une mauvaise pratique dans du code à destination d'un environnement de production
- Rappelez-vous qu'il n'y a rien de mauvais à utiliser ***from Package import specifique_submodule*** !
- C'est d'ailleurs la manière recommandée à moins que le module qui fait les imports ai besoin de sous-modules ayant le même nom mais provenant de paquets différents

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

Références internes dans un paquet

- Lorsque les paquets sont organisés en sous-paquets (comme le paquet sound par exemple), vous pouvez utiliser des imports absolus pour cibler des paquets voisins
- Par exemple, si le module ***sound.filters.vocoder*** a besoin du module echo du paquet sound.effects, il peut utiliser ***from sound.effects import echo***
- Il est aussi possible d'écrire des imports relatifs de la forme ***from module import name***
- Ces imports sont préfixés par des points pour indiquer leur origine (paquet courant ou parent). Depuis le module surround, par exemple vous pourriez faire :

```
from . import echo
from .. import formats
from ..filters import equalizer
```

- Notez que les imports relatifs se fient au nom du module actuel
- Puisque le nom du module principal est toujours "***__main__***", les modules utilisés par le module principal d'une application ne peuvent être importées que par des imports absolus

Paquets dans plusieurs dossiers

- Les paquets exposent un attribut supplémentaire, ***__path__***, contenant une liste, initialisée avant l'exécution du fichier ***__init__.py***, contenant le nom de son dossier dans le système de fichier
- Cette liste peut être modifiée, altérant ainsi les futures recherches de modules et sous-paquets contenus dans le paquet
- Bien que cette fonctionnalité ne soit que rarement utile, elle peut servir à élargir la liste des modules trouvés dans un paquet.

Références

- Python.org : https://www.python.org/
- Learning Python : https://github.com/thierrydecker/learning-python
- …

Outils

- IDE Pycharm Community : https://www.jetbrains.com/pycharm/
- Analyse en ligne de code Python : http://www.pythontutor.com/
- …