# Algorithmique et Programmation 2

## 1. Compléments sur les types Python

Lionel.Moisan@parisdescartes.fr

Université Paris Descartes
http://www.mi.parisdescartes.fr/∽moisan/

# Compléments sur les types Python

1. Nombres (int, float, complex)
2. Listes
3. Tuples (n-uplets)
4. Ensembles
5. Tableaux associatifs (dictionnaires)

# Nombres (int, float, complex)

# Les nombres entiers

## Le type `int`

En Python, les entiers sont codés dans le type `int`. Contrairement à la plupart des langages, des entiers arbitrairement grands peuvent être représentés, il n'y a pas de limitation intrinsèque au codage.

---

```
>>> N = 3**1000; N
13220708194808066368904552597521443659654220327521481676649203682268285973467048995407783138506080619639097776968725823559509545821006189118653427252579536740276202251983208038780147742289648412743904001175886180411289478156230944380615661730540866744905061781254803444055470543970388958174653682549161362208302685637785822902284163983078878969185564040848989376093732421718463599386955167650189405881090604260896714388641028143503856487471658320106143661321731027689028552200 01
```

# Les nombres entiers

## Le type `int`

En Python, les entiers sont codés dans le type `int`. Contrairement à la plupart des langages, des entiers arbitrairement grands peuvent être représentés, il n'y a pas de limitation intrinsèque au codage.

```
>>> N = 3**1000; N
1322070819480806636890455259752144365965422032752148167664920368226828597346704899540778313850608061963909777696872582355950954582100618911865342725257953674027620225198320803878014774228964841274390400117588618041128947815623094438061566173054086674490506178125480344405547054397038895817465368254916136220830268563778582290228416398307887896918556404084898937609373242171846359938695516765018940588109060426089671438864102814350385648747165832010614366132173102768902855220001
>>> import math
>>> N = math.factorial(200000) # factorielle de 200000
>>> len(str(N)) # nombre de chiffres de N
973351
```

# Utilisation d'autres bases

Si l'on souhaite définir un entier dans une base autre que la base 10, on peut utiliser un préfixe :

# Utilisation d'autres bases

Si l'on souhaite définir un entier dans une base autre que la base 10, on peut utiliser un préfixe :

- '0b' pour la base 2 (représentation binaire)

# Utilisation d'autres bases

Si l'on souhaite définir un entier dans une base autre que la base 10, on peut utiliser un préfixe :

- '0b' pour la base 2 (représentation binaire)
- '0o' pour la base 8 (représentation octale)

# Utilisation d'autres bases

Si l'on souhaite définir un entier dans une base autre que la base 10, on peut utiliser un préfixe :

- '0b' pour la base 2 (représentation binaire)
- '0o' pour la base 8 (représentation octale)
- '0x' pour la base 16 (représentation hexadécimale)

# Utilisation d'autres bases

Si l'on souhaite définir un entier dans une base autre que la base 10, on peut utiliser un préfixe :

- '0b' pour la base 2 (représentation binaire)
- '0o' pour la base 8 (représentation octale)
- '0x' pour la base 16 (représentation hexadécimale)

Inversement, les fonctions `str()`, `bin()`, `oct()` et `hex()` renvoient l'écriture d'un nombre en base 10, 2, 8 et 16 respectivement.

---

```
>>> 0b1111 # 2^3+2^2+2^1+2^0
15
```

# Utilisation d'autres bases

Si l'on souhaite définir un entier dans une base autre que la base 10, on peut utiliser un préfixe :

- '0b' pour la base 2 (représentation binaire)
- '0o' pour la base 8 (représentation octale)
- '0x' pour la base 16 (représentation hexadécimale)

Inversement, les fonctions `str()`, `bin()`, `oct()` et `hex()` renvoient l'écriture d'un nombre en base 10, 2, 8 et 16 respectivement.

```
>>> 0b1111 # 2^3+2^2+2^1+2^0
15
>>> 0o60 # 6*8+0
48
```

# Utilisation d'autres bases

Si l'on souhaite définir un entier dans une base autre que la base 10, on peut utiliser un préfixe :

- '0b' pour la base 2 (représentation binaire)
- '0o' pour la base 8 (représentation octale)
- '0x' pour la base 16 (représentation hexadécimale)

Inversement, les fonctions `str()`, `bin()`, `oct()` et `hex()` renvoient l'écriture d'un nombre en base 10, 2, 8 et 16 respectivement.

```
>>> 0b1111 # 2^3+2^2+2^1+2^0
15
>>> 0o60 # 6*8+0
48
>>> 0xff # 15*16+15
255
```

# Utilisation d'autres bases

Si l'on souhaite définir un entier dans une base autre que la base 10, on peut utiliser un préfixe :

- '0b' pour la base 2 (représentation binaire)
- '0o' pour la base 8 (représentation octale)
- '0x' pour la base 16 (représentation hexadécimale)

Inversement, les fonctions `str()`, `bin()`, `oct()` et `hex()` renvoient l'écriture d'un nombre en base 10, 2, 8 et 16 respectivement.

```
>>> 0b1111 # 2^3+2^2+2^1+2^0
15
>>> 0o60 # 6*8+0
48
>>> 0xff # 15*16+15
255
>>> str(15),bin(15),oct(48),hex(255)
('15', '0b1111', '0o60', '0xff')
```

# Les nombres réels

**Le type `float`**

En Python, les nombres réels sont codés dans le type `float`. Sur la plupart des architectures, la représentation est fait sur 4 octets (64 chiffres binaires) selon le standard IEEE 754.

# Les nombres réels

**Le type `float`**

En Python, les nombres réels sont codés dans le type `float`. Sur la plupart des architectures, la représentation est fait sur 4 octets (64 chiffres binaires) selon le standard IEEE 754.

Ce format fixe ne représente qu'un nombre fini de nombre réels. Les calculs sont donc limités et effectués avec une précision donnée (environ 16 chiffres significatifs).

# Les nombres réels

**Le type `float`**

En Python, les nombres réels sont codés dans le type `float`. Sur la plupart des architectures, la représentation est fait sur 4 octets (64 chiffres binaires) selon le standard IEEE 754.

Ce format fixe ne représente qu'un nombre fini de nombre réels. Les calculs sont donc limités et effectués avec une précision donnée (environ 16 chiffres significatifs).

```
>>> 1.3, 1., -.027, 13.2e18 # quelques nombres réels (float)
(1.3, 1.0, -0.027, 1.32e+19)
```

# Les nombres réels

**Le type `float`**

En Python, les nombres réels sont codés dans le type `float`. Sur la plupart des architectures, la représentation est fait sur 4 octets (64 chiffres binaires) selon le standard IEEE 754.

Ce format fixe ne représente qu'un nombre fini de nombre réels. Les calculs sont donc limités et effectués avec une précision donnée (environ 16 chiffres significatifs).

```
>>> 1.3, 1., -.027, 13.2e18 # quelques nombres réels (float)
(1.3, 1.0, -0.027, 1.32e+19)
>>> a = 1e307; a,a*10,a*100 # existence d'un plus grand réel
(1e+307, 1e+308, inf)
```

# Les nombres réels

**Le type `float`**

En Python, les nombres réels sont codés dans le type `float`. Sur la plupart des architectures, la représentation est fait sur 4 octets (64 chiffres binaires) selon le standard IEEE 754.

Ce format fixe ne représente qu'un nombre fini de nombre réels. Les calculs sont donc limités et effectués avec une précision donnée (environ 16 chiffres significatifs).

```
>>> 1.3, 1., -.027, 13.2e18 # quelques nombres réels (float)
(1.3, 1.0, -0.027, 1.32e+19)
>>> a = 1e307; a,a*10,a*100 # existence d'un plus grand réel
(1e+307, 1e+308, inf)
>>> a = 1e-322; a,a/10,a/100 # et d'un plus petit réel positif
(1e-322, 1e-323, 0.0)
```

# Les nombres réels

**Le type `float`**

En Python, les nombres réels sont codés dans le type `float`. Sur la plupart des architectures, la représentation est fait sur 4 octets (64 chiffres binaires) selon le standard IEEE 754.

Ce format fixe ne représente qu'un nombre fini de nombre réels. Les calculs sont donc limités et effectués avec une précision donnée (environ 16 chiffres significatifs).

```
>>> 1.3, 1., -.027, 13.2e18 # quelques nombres réels (float)
(1.3, 1.0, -0.027, 1.32e+19)
>>> a = 1e307; a,a*10,a*100 # existence d'un plus grand réel
(1e+307, 1e+308, inf)
>>> a = 1e-322; a,a/10,a/100 # et d'un plus petit réel positif
(1e-322, 1e-323, 0.0)
>>> format(1+1e-15,'.20f') # affichage avec 20 décimales
'1.00000000000000111022'
```

# Les nombres réels

## Le type `float`

En Python, les nombres réels sont codés dans le type `float`. Sur la plupart des architectures, la représentation est fait sur 4 octets (64 chiffres binaires) selon le standard IEEE 754.

Ce format fixe ne représente qu'un nombre fini de nombre réels. Les calculs sont donc limités et effectués avec une précision donnée (environ 16 chiffres significatifs).

```
>>> 1.3, 1., -.027, 13.2e18 # quelques nombres réels (float)
(1.3, 1.0, -0.027, 1.32e+19)
>>> a = 1e307; a,a*10,a*100 # existence d'un plus grand réel
(1e+307, 1e+308, inf)
>>> a = 1e-322; a,a/10,a/100 # et d'un plus petit réel positif
(1e-322, 1e-323, 0.0)
>>> format(1+1e-15,'.20f') # affichage avec 20 décimales
'1.00000000000000111022'
>>> 1+1e-16-1 # l'addition n'est pas associative !
0.0
```

# Les nombres réels

- On peut convertir un nombre entier en réel avec la fonction `float()`.

# Les nombres réels

- On peut convertir un nombre entier en réel avec la fonction `float()`.

```
>>> float(3**100)
5.15377520732011e+47
```

# Les nombres réels

- On peut convertir un nombre entier en réel avec la fonction `float()`.

```
>>> float(3**100)
5.15377520732011e+47
>>> float(3**1000) # valeur trop grande pour le type float
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

# Les nombres réels

- On peut convertir un nombre entier en réel avec la fonction `float()`.

```
>>> float(3**100)
5.15377520732011e+47
>>> float(3**1000) # valeur trop grande pour le type float
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

- Inversement, on peut arrondir un réel avec `int()` et `round()`.

# Les nombres réels

- On peut convertir un nombre entier en réel avec la fonction `float()`.

```
>>> float(3**100)
5.15377520732011e+47
>>> float(3**1000) # valeur trop grande pour le type float
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

- Inversement, on peut arrondir un réel avec `int()` et `round()`.

```
>>> int(3.1), int(3.7), int(-3.7) # arrondi sans partie décimale
(3, 3, -3)
```

# Les nombres réels

- On peut convertir un nombre entier en réel avec la fonction `float()`.

```
>>> float(3**100)
5.15377520732011e+47
>>> float(3**1000) # valeur trop grande pour le type float
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

- Inversement, on peut arrondir un réel avec `int()` et `round()`.

```
>>> int(3.1), int(3.7), int(-3.7) # arrondi sans partie décimale
(3, 3, -3)
>>> round(3.1), round(3.7), round(-3.7) # entier le plus proche
(3, 4, -4)
```

# Les nombres réels

• On peut convertir un nombre entier en réel avec la fonction `float()`.

```
>>> float(3**100)
5.15377520732011e+47
>>> float(3**1000) # valeur trop grande pour le type float
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

• Inversement, on peut arrondir un réel avec `int()` et `round()`.

```
>>> int(3.1), int(3.7), int(-3.7) # arrondi sans partie décimale
(3, 3, -3)
>>> round(3.1), round(3.7), round(-3.7) # entier le plus proche
(3, 4, -4)
```

• La partie entière (fonction $E()$ en mathématiques) s'obtient avec `math.floor()`.

# Les nombres réels

- On peut convertir un nombre entier en réel avec la fonction `float()`.

```
>>> float(3**100)
5.15377520732011e+47
>>> float(3**1000) # valeur trop grande pour le type float
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

- Inversement, on peut arrondir un réel avec `int()` et `round()`.

```
>>> int(3.1), int(3.7), int(-3.7) # arrondi sans partie décimale
(3, 3, -3)
>>> round(3.1), round(3.7), round(-3.7) # entier le plus proche
(3, 4, -4)
```

- La partie entière (fonction $E()$ en mathématiques) s'obtient avec `math.floor()`.

```
>>> import math; math.floor(-3.7)
-4
```

# Constantes et fonctions mathématiques

Le module `math` contient les constantes et fonctions mathématiques usuelles : trigonométriques (`cos, sin, tan`) et réciproques, (`acos, asin, atan`), hyperboliques (`cosh, sinh, tanh`) et réciproques, (`acosh, asinh, atanh`), logarithme (`log`), exponentielle (`exp`), fonction gamma (`gamma`), etc.

# Constantes et fonctions mathématiques

Le module `math` contient les constantes et fonctions mathématiques usuelles : trigonométriques (`cos`, `sin`, `tan`) et réciproques, (`acos`, `asin`, `atan`), hyperboliques (`cosh`, `sinh`, `tanh`) et réciproques, (`acosh`, `asinh`, `atanh`), logarithme (`log`), exponentielle (`exp`), fonction gamma (`gamma`), etc.

---

```
>>> import math
>>> math.pi
3.141592653589793
```

# Constantes et fonctions mathématiques

Le module `math` contient les constantes et fonctions mathématiques usuelles : trigonométriques (`cos`, `sin`, `tan`) et réciproques, (`acos`, `asin`, `atan`), hyperboliques (`cosh`, `sinh`, `tanh`) et réciproques, (`acosh`, `asinh`, `atanh`), logarithme (`log`), exponentielle (`exp`), fonction gamma (`gamma`), etc.

---

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

# Constantes et fonctions mathématiques

Le module `math` contient les constantes et fonctions mathématiques usuelles : trigonométriques (`cos`, `sin`, `tan`) et réciproques, (`acos`, `asin`, `atan`), hyperboliques (`cosh`, `sinh`, `tanh`) et réciproques, (`acosh`, `asinh`, `atanh`), logarithme (`log`), exponentielle (`exp`), fonction gamma (`gamma`), etc.

---

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.acos(0) # pi/2
1.5707963267948966
```

# Constantes et fonctions mathématiques

Le module `math` contient les constantes et fonctions mathématiques usuelles : trigonométriques (`cos`, `sin`, `tan`) et réciproques, (`acos`, `asin`, `atan`), hyperboliques (`cosh`, `sinh`, `tanh`) et réciproques, (`acosh`, `asinh`, `atanh`), logarithme (`log`), exponentielle (`exp`), fonction gamma (`gamma`), etc.

---

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.acos(0) # pi/2
1.5707963267948966
>>> math.asinh(0)
0.0
```

# Constantes et fonctions mathématiques

Le module `math` contient les constantes et fonctions mathématiques usuelles : trigonométriques (`cos`, `sin`, `tan`) et réciproques, (`acos`, `asin`, `atan`), hyperboliques (`cosh`, `sinh`, `tanh`) et réciproques, (`acosh`, `asinh`, `atanh`), logarithme (`log`), exponentielle (`exp`), fonction gamma (`gamma`), etc.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.acos(0) # pi/2
1.5707963267948966
>>> math.asinh(0)
0.0
>>> math.gamma(5) # factorielle de 6
24.0
```

# Erreurs

Certaines opérations illicites conduisent à des erreurs, d'autres renvoient des valeurs correspondant à des limites.

# Erreurs

Certaines opérations illicites conduisent à des erreurs, d'autres renvoient des valeurs correspondant à des limites.

```
>>> I = float('inf'); I # +infini
inf
```

# Erreurs

Certaines opérations illicites conduisent à des erreurs, d'autres renvoient des valeurs correspondant à des limites.

```
>>> I = float('inf'); I # +infini
inf
>>> 2*math.atan(I) # limite de Arctan en +infini
3.141592653589793
```

# Erreurs

Certaines opérations illicites conduisent à des erreurs, d'autres renvoient des valeurs correspondant à des limites.

```
>>> I = float('inf'); I # +infini
inf
>>> 2*math.atan(I) # limite de Arctan en +infini
3.141592653589793
>>> math.log(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

# Erreurs

Certaines opérations illicites conduisent à des erreurs, d'autres renvoient des valeurs correspondant à des limites.

```
>>> I = float('inf'); I # +infini
inf
>>> 2*math.atan(I) # limite de Arctan en +infini
3.141592653589793
>>> math.log(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> math.log(I)
inf
```

# Erreurs

Certaines opérations illicites conduisent à des erreurs, d'autres renvoient des valeurs correspondant à des limites.

```
>>> I = float('inf'); I # +infini
inf
>>> 2*math.atan(I) # limite de Arctan en +infini
3.141592653589793
>>> math.log(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> math.log(I)
inf
>>> 1/I
0.0
```

# Erreurs

Certaines opérations illicites conduisent à des erreurs, d'autres renvoient des valeurs correspondant à des limites.

```
>>> I = float('inf'); I # +infini
inf
>>> 2*math.atan(I) # limite de Arctan en +infini
3.141592653589793
>>> math.log(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> math.log(I)
inf
>>> 1/I
0.0
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

# les nombres complexes

**Le type `complex`**

En Python, les nombres complexes sont codés dans le type `complex`.

Les nombres imaginaires se notent avec le suffixe 'j'.

# les nombres complexes

**Le type `complex`**

En Python, les nombres complexes sont codés dans le type `complex`.

Les nombres imaginaires se notent avec le suffixe 'j'.

---

```
>>> z = 3+4j; z
(3+4j)
```

# les nombres complexes

**Le type `complex`**

En Python, les nombres complexes sont codés dans le type `complex`.
Les nombres imaginaires se notent avec le suffixe 'j'.

---

```
>>> z = 3+4j; z
(3+4j)
>>> z.real # partie réelle
3.0
```

# les nombres complexes

## Le type `complex`

En Python, les nombres complexes sont codés dans le type `complex`.
Les nombres imaginaires se notent avec le suffixe 'j'.

---

```
>>> z = 3+4j; z
(3+4j)
>>> z.real # partie réelle
3.0
>>> z.imag # partie imaginaire
4.0
```

# les nombres complexes

## Le type `complex`

En Python, les nombres complexes sont codés dans le type `complex`.
Les nombres imaginaires se notent avec le suffixe 'j'.

---

```
>>> z = 3+4j; z
(3+4j)
>>> z.real # partie réelle
3.0
>>> z.imag # partie imaginaire
4.0
>>> abs(z) # module de z
5.0
```

# les nombres complexes

## Le type `complex`

En Python, les nombres complexes sont codés dans le type `complex`.
Les nombres imaginaires se notent avec le suffixe 'j'.

---

```
>>> z = 3+4j; z
(3+4j)
>>> z.real # partie réelle
3.0
>>> z.imag # partie imaginaire
4.0
>>> abs(z) # module de z
5.0
>>> import cmath; cmath.phase(1+1j) # argument principal
0.7853981633974483
```

# les nombres complexes

## Le type `complex`

En Python, les nombres complexes sont codés dans le type `complex`.
Les nombres imaginaires se notent avec le suffixe 'j'.

---

```
>>> z = 3+4j; z
(3+4j)
>>> z.real # partie réelle
3.0
>>> z.imag # partie imaginaire
4.0
>>> abs(z) # module de z
5.0
>>> import cmath; cmath.phase(1+1j) # argument principal
0.7853981633974483
>>> z = -8-6j
>>> z**0.5 # une racine carrée de z
(1-3j)
```

# les nombres complexes

## Le type `complex`

En Python, les nombres complexes sont codés dans le type `complex`.
Les nombres imaginaires se notent avec le suffixe 'j'.

```
>>> z = 3+4j; z
(3+4j)
>>> z.real # partie réelle
3.0
>>> z.imag # partie imaginaire
4.0
>>> abs(z) # module de z
5.0
>>> import cmath; cmath.phase(1+1j) # argument principal
0.7853981633974483
>>> z = -8-6j
>>> z**0.5 # une racine carrée de z
(1-3j)
>>> cmath.exp(1j*math.pi) # identité d'Euler (numérique)
(-1+1.2246467991473532e-16j)
```

# Listes

# Les listes

**Le type `list`**

Rappel : une liste de type `list` est une collection ordonnée et mutable (i.e. modifiable) d'éléments éventuellement hétérogènes.

Définition d'une liste : éléments entre crochets, séparés par des virgules

```
>>> L = [3,1e-10,'Paris',abs]
```

# Les listes

**Le type `list`**

Rappel : une liste de type `list` est une collection ordonnée et mutable (i.e. modifiable) d'éléments éventuellement hétérogènes.

Définition d'une liste : éléments entre crochets, séparés par des virgules

```
>>> L = [3,1e-10,'Paris',abs]
```

Opérations sur les listes :

- extraction/suppression/remplacement (`L[a:b:c]`)

# Les listes

**Le type `list`**

Rappel : une liste de type `list` est une collection ordonnée et mutable (i.e. modifiable) d'éléments éventuellement hétérogènes.

Définition d'une liste : éléments entre crochets, séparés par des virgules

```
>>> L = [3,1e-10,'Paris',abs]
```

Opérations sur les listes :

- extraction/suppression/remplacement (`L[a:b:c]`)
- concaténation ('+')

# Les listes

**Le type `list`**

Rappel : une liste de type `list` est une collection ordonnée et mutable (i.e. modifiable) d'éléments éventuellement hétérogènes.

Définition d'une liste : éléments entre crochets, séparés par des virgules

```
>>> L = [3,1e-10,'Paris',abs]
```

Opérations sur les listes :

- extraction/suppression/remplacement (`L[a:b:c]`)
- concaténation ('+')
- multiplication ('*')

# Les listes

**Le type `list`**

Rappel : une liste de type `list` est une collection ordonnée et mutable (i.e. modifiable) d'éléments éventuellement hétérogènes.

Définition d'une liste : éléments entre crochets, séparés par des virgules

```
>>> L = [3,1e-10,'Paris',abs]
```

Opérations sur les listes :

- extraction/suppression/remplacement (`L[a:b:c]`)
- concaténation ('+')
- multiplication ('*')
- supression de l'élément d'indice i : `del L[i]`

# Les listes

**Le type `list`**

Rappel : une liste de type `list` est une collection ordonnée et mutable (i.e. modifiable) d'éléments éventuellement hétérogènes.

Définition d'une liste : éléments entre crochets, séparés par des virgules

```
>>> L = [3,1e-10,'Paris',abs]
```

Opérations sur les listes :

- extraction/suppression/remplacement (`L[a:b:c]`)
- concaténation ('+')
- multiplication ('*')
- supression de l'élément d'indice i : `del L[i]`
- supression des élément d'indices i (inclus) à j (exclu) : `del L[i:j]`

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L
- `L.index(v,i,j)` : idem, entre i (inclus) et j (exclu)

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L
- `L.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- `L.remove(v)` : supprime la 1ère occurence de v dans L

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L
- `L.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- `L.remove(v)` : supprime la 1ère occurence de v dans L
- `L.insert(i,v)` : insère v juste avant L[i]

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L
- `L.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- `L.remove(v)` : supprime la 1ère occurence de v dans L
- `L.insert(i,v)` : insère v juste avant L[i]
- `L.sort()` : trie L par valeurs croissantes

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L
- `L.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- `L.remove(v)` : supprime la 1ère occurence de v dans L
- `L.insert(i,v)` : insère v juste avant L[i]
- `L.sort()` : trie L par valeurs croissantes
- `L.sort(reverse=True)` : trie L par valeurs décroissantes

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L
- `L.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- `L.remove(v)` : supprime la 1ère occurence de v dans L
- `L.insert(i,v)` : insère v juste avant L[i]
- `L.sort()` : trie L par valeurs croissantes
- `L.sort(reverse=True)` : trie L par valeurs décroissantes
- `len(L)` : nombre d'éléments dans la liste L

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L
- `L.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- `L.remove(v)` : supprime la 1ère occurence de v dans L
- `L.insert(i,v)` : insère v juste avant L[i]
- `L.sort()` : trie L par valeurs croissantes
- `L.sort(reverse=True)` : trie L par valeurs décroissantes
- `len(L)` : nombre d'éléments dans la liste L
- `sum(L)` : somme des éléments de L (si défini)

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L
- `L.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- `L.remove(v)` : supprime la 1ère occurence de v dans L
- `L.insert(i,v)` : insère v juste avant L[i]
- `L.sort()` : trie L par valeurs croissantes
- `L.sort(reverse=True)` : trie L par valeurs décroissantes
- `len(L)` : nombre d'éléments dans la liste L
- `sum(L)` : somme des éléments de L (si défini)
- `min(L) / max(L)` : min/max des éléments de L (si défini)

# Méthodes et fonctions associées aux listes

- `L.append(e)` : ajoute l'élément e à la fin de L
- `L.pop()` : supprime (et renvoie) le dernier élément de L
- `L.count(v)` : nombre d'occurences de v dans L
- `L.index(v)` : indice de la 1ère occurence de v dans L
- `L.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- `L.remove(v)` : supprime la 1ère occurence de v dans L
- `L.insert(i,v)` : insère v juste avant L[i]
- `L.sort()` : trie L par valeurs croissantes
- `L.sort(reverse=True)` : trie L par valeurs décroissantes
- `len(L)` : nombre d'éléments dans la liste L
- `sum(L)` : somme des éléments de L (si défini)
- `min(L) / max(L)` : min/max des éléments de L (si défini)
- `sorted(L)` : renvoie la liste L triée (sans la modifier)

# Copie d'une liste

Attention, une affectation entre listes ne copie pas le contenu mais seulement une référence.

```
>>> A = [1,2,3]
>>> B = A; B
[1, 2, 3]
>>> A[1] = 10 # on modifie A
>>> A
[1, 10, 3]
>>> B # ce qui impacte B !
[1, 10, 3]
```

# Copie d'une liste

Pour recopier une liste A en une nouvelle liste B, on peut utiliser

```
B = A[:]
```

# Copie d'une liste

Pour recopier une liste A en une nouvelle liste B, on peut utiliser

`B = A[:]` ou `B = A.copy()`

# Copie d'une liste

Pour recopier une liste A en une nouvelle liste B, on peut utiliser

`B = A[:]` ou `B = A.copy()` ou `B = list(A)`

# Copie d'une liste

Pour recopier une liste A en une nouvelle liste B, on peut utiliser

`B = A[:]` ou `B = A.copy()` ou `B = list(A)`

à condition qu'aucun élément de `A` ne soit de type composé.

# Copie d'une liste

Pour recopier une liste A en une nouvelle liste B, on peut utiliser

`B = A[:]` ou `B = A.copy()` ou `B = list(A)`

à condition qu'aucun élément de `A` ne soit de type composé.

---

```
>>> A = [1,2,3]
```

# Copie d'une liste

Pour recopier une liste A en une nouvelle liste B, on peut utiliser

`B = A[:]` ou `B = A.copy()` ou `B = list(A)`

à condition qu'aucun élément de `A` ne soit de type composé.

---

```
>>> A = [1,2,3]
>>> B = A[:] # copie de A dans B
```

# Copie d'une liste

Pour recopier une liste A en une nouvelle liste B, on peut utiliser

`B = A[:]` ou `B = A.copy()` ou `B = list(A)`

à condition qu'aucun élément de `A` ne soit de type composé.

---

```
>>> A = [1,2,3]
>>> B = A[:] # copie de A dans B
>>> B
[1, 2, 3]
```

# Copie d'une liste

Pour recopier une liste A en une nouvelle liste B, on peut utiliser

`B = A[:]` ou `B = A.copy()` ou `B = list(A)`

à condition qu'aucun élément de `A` ne soit de type composé.

---

```
>>> A = [1,2,3]
>>> B = A[:] # copie de A dans B
>>> B
[1, 2, 3]
>>> A[1] = 10; A # on modifie A
[1, 10, 3]
```

# Copie d'une liste

Pour recopier une liste A en une nouvelle liste B, on peut utiliser

`B = A[:]` ou `B = A.copy()` ou `B = list(A)`

à condition qu'aucun élément de `A` ne soit de type composé.

```
>>> A = [1,2,3]
>>> B = A[:] # copie de A dans B
>>> B
[1, 2, 3]
>>> A[1] = 10; A # on modifie A
[1, 10, 3]
>>> B # B n'est pas affecté
[1, 2, 3]
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
>>> A[0][0] = 'nouveau' # modification de A
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
>>> A[0][0] = 'nouveau' # modification de A
>>> B # B n'est pas une copie de A indépendante
[['nouveau', 2], [3, 4]]
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
>>> A[0][0] = 'nouveau' # modification de A
>>> B # B n'est pas une copie de A indépendante
[['nouveau', 2], [3, 4]]
>>> import copy; C = copy.deepcopy(B)
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
>>> A[0][0] = 'nouveau' # modification de A
>>> B # B n'est pas une copie de A indépendante
[['nouveau', 2], [3, 4]]
>>> import copy; C = copy.deepcopy(B)
>>> A[0][0] = 'encore'
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
>>> A[0][0] = 'nouveau' # modification de A
>>> B # B n'est pas une copie de A indépendante
[['nouveau', 2], [3, 4]]
>>> import copy; C = copy.deepcopy(B)
>>> A[0][0] = 'encore'
>>> C # C n'est pas modifiée
[['nouveau', 2], [3, 4]]
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
>>> A[0][0] = 'nouveau' # modification de A
>>> B # B n'est pas une copie de A indépendante
[['nouveau', 2], [3, 4]]
>>> import copy; C = copy.deepcopy(B)
>>> A[0][0] = 'encore'
>>> C # C n'est pas modifiée
[['nouveau', 2], [3, 4]]
```

Dans le cas d'une liste de listes, on peut aussi utiliser une compréhension de liste :

```
>>> A = [[1,2],[3,4]] # liste de listes
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
>>> A[0][0] = 'nouveau' # modification de A
>>> B # B n'est pas une copie de A indépendante
[['nouveau', 2], [3, 4]]
>>> import copy; C = copy.deepcopy(B)
>>> A[0][0] = 'encore'
>>> C # C n'est pas modifiée
[['nouveau', 2], [3, 4]]
```

Dans le cas d'une liste de listes, on peut aussi utiliser une compréhension de liste :

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> C = [L[:] for L in A] # vraie copie de A
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
>>> A[0][0] = 'nouveau' # modification de A
>>> B # B n'est pas une copie de A indépendante
[['nouveau', 2], [3, 4]]
>>> import copy; C = copy.deepcopy(B)
>>> A[0][0] = 'encore'
>>> C # C n'est pas modifiée
[['nouveau', 2], [3, 4]]
```

Dans le cas d'une liste de listes, on peut aussi utiliser une compréhension de liste :

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> C = [L[:] for L in A] # vraie copie de A
>>> A[0][0] = 'nouveau' # modification de A
```

# Copie d'une liste

Si A contient des types composés, (par exemple A est une liste de listes), on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> B = A[:]
>>> A[0][0] = 'nouveau' # modification de A
>>> B # B n'est pas une copie de A indépendante
[['nouveau', 2], [3, 4]]
>>> import copy; C = copy.deepcopy(B)
>>> A[0][0] = 'encore'
>>> C # C n'est pas modifiée
[['nouveau', 2], [3, 4]]
```

Dans le cas d'une liste de listes, on peut aussi utiliser une compréhension de liste :

```
>>> A = [[1,2],[3,4]] # liste de listes
>>> C = [L[:] for L in A] # vraie copie de A
>>> A[0][0] = 'nouveau' # modification de A
>>> C # C n'est pas modifiée
[[1, 2], [3, 4]]
```

# Tuples (n-uplets)

# Les tuples

## Le type `tuple`

Un tuple (ou n-uplet) de type `tuple` est une collection ordonnée et non mutable d'éléments éventuellement hétérogènes.

# Les tuples

**Le type `tuple`**

Un tuple (ou n-uplet) de type `tuple` est une collection ordonnée et non mutable d'éléments éventuellement hétérogènes.

Un tuple est donc une sort de liste non modifiable.

# Les tuples

**Le type `tuple`**

Un tuple (ou n-uplet) de type `tuple` est une collection ordonnée et non mutable d'éléments éventuellement hétérogènes.

Un tuple est donc une sort de liste non modifiable.

Définition d'un tuple : éléments séparés par des virgules (souvent entre parenthèses pour plus de lisibilité)

# Les tuples

**Le type `tuple`**

Un tuple (ou n-uplet) de type `tuple` est une collection ordonnée et non mutable d'éléments éventuellement hétérogènes.

Un tuple est donc une sort de liste non modifiable.

Définition d'un tuple : éléments séparés par des virgules (souvent entre parenthèses pour plus de lisibilité)

```
>>> T = (3,1e-10,'Paris',abs) # définition d'un tuple
>>> T = 3,1e-10,'Paris',abs # définition équivalente
```

# Les tuples

**Le type `tuple`**

Un tuple (ou n-uplet) de type `tuple` est une collection ordonnée et non mutable d'éléments éventuellement hétérogènes.

Un tuple est donc une sort de liste non modifiable.

Définition d'un tuple : éléments séparés par des virgules (souvent entre parenthèses pour plus de lisibilité)

```
>>> T = (3,1e-10,'Paris',abs) # définition d'un tuple
>>> T = 3,1e-10,'Paris',abs # définition équivalente
>>> type(T)
<class 'tuple'>
```

# Les tuples

**Le type `tuple`**

Un tuple (ou n-uplet) de type `tuple` est une collection ordonnée et non mutable d'éléments éventuellement hétérogènes.

Un tuple est donc une sort de liste non modifiable.

Définition d'un tuple : éléments séparés par des virgules (souvent entre parenthèses pour plus de lisibilité)

```
>>> T = (3,1e-10,'Paris',abs) # définition d'un tuple
>>> T = 3,1e-10,'Paris',abs # définition équivalente
>>> type(T)
<class 'tuple'>
>>> T = 1
>>> type(T)
<class 'int'>
```

# Les tuples

## Le type `tuple`

Un tuple (ou n-uplet) de type `tuple` est une collection ordonnée et non mutable d'éléments éventuellement hétérogènes.

Un tuple est donc une sort de liste non modifiable.

Définition d'un tuple : éléments séparés par des virgules (souvent entre parenthèses pour plus de lisibilité)

```
>>> T = (3,1e-10,'Paris',abs) # définition d'un tuple
>>> T = 3,1e-10,'Paris',abs # définition équivalente
>>> type(T)
<class 'tuple'>
>>> T = 1
>>> type(T)
<class 'int'>
>>> T = 1, # définition d'un tuple à un seul élément
>>> type(T)
<class 'tuple'>
```

# Opérations sur les tuples

Par rapport aux listes, seules les opérations qui ne modifient pas le tuple sont possibles :

- extraction (`T[a:b:c]`)

# Opérations sur les tuples

Par rapport aux listes, seules les opérations qui ne modifient pas le tuple sont possibles :

- extraction (`T[a:b:c]`)
- concaténation ('+')

# Opérations sur les tuples

Par rapport aux listes, seules les opérations qui ne modifient pas le tuple sont possibles :

- extraction (`T[a:b:c]`)
- concaténation ('+')
- multiplication ('*')

# Opérations sur les tuples

Par rapport aux listes, seules les opérations qui ne modifient pas le tuple sont possibles :

- extraction (`T[a:b:c]`)
- concaténation ('+')
- multiplication ('*')
- `T.count(v)` : nombre d'occurences de v dans T

# Opérations sur les tuples

Par rapport aux listes, seules les opérations qui ne modifient pas le tuple sont possibles :

- extraction (`T[a:b:c]`)
- concaténation ('+')
- multiplication ('*')
- `T.count(v)` : nombre d'occurences de v dans T
- `T.index(v)` : indice de la 1ère occurence de v dans T

# Opérations sur les tuples

Par rapport aux listes, seules les opérations qui ne modifient pas le tuple sont possibles :

- extraction (`T[a:b:c]`)
- concaténation ('+')
- multiplication ('*')
- `T.count(v)` : nombre d'occurences de v dans T
- `T.index(v)` : indice de la 1ère occurence de v dans T
- `T.index(v,i,j)` : idem, entre i (inclus) et j (exclu)

# Opérations sur les tuples

Par rapport aux listes, seules les opérations qui ne modifient pas le tuple sont possibles :

- extraction (`T[a:b:c]`)
- concaténation ('+')
- multiplication ('*')
- `T.count(v)` : nombre d'occurences de v dans T
- `T.index(v)` : indice de la 1ère occurence de v dans T
- `T.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- fonctions `len(T)`, `sum(T)`, `min(T)`, `max(T)`

# Opérations sur les tuples

Par rapport aux listes, seules les opérations qui ne modifient pas le tuple sont possibles :

- extraction (`T[a:b:c]`)
- concaténation ('+')
- multiplication ('*')
- `T.count(v)` : nombre d'occurences de v dans T
- `T.index(v)` : indice de la 1ère occurence de v dans T
- `T.index(v,i,j)` : idem, entre i (inclus) et j (exclu)
- fonctions `len(T)`, `sum(T)`, `min(T)`, `max(T)`
- `sorted(T)` : renvoie un tuple trié

# Conversions tuple / list

Le passage d'un type à l'autre se fait avec les fonctions `tuple` et `list`.

# Conversions tuple / list

Le passage d'un type à l'autre se fait avec les fonctions `tuple` et `list`.

```
>>> T = (1,2,'a',True)
>>> T
(1, 2, 'a', True)
```

# Conversions tuple / list

Le passage d'un type à l'autre se fait avec les fonctions `tuple` et `list`.

```
>>> T = (1,2,'a',True)
>>> T
(1, 2, 'a', True)
>>> type(T)
<class 'tuple'>
```

# Conversions tuple / list

Le passage d'un type à l'autre se fait avec les fonctions `tuple` et `list`.

```
>>> T = (1,2,'a',True)
>>> T
(1, 2, 'a', True)
>>> type(T)
<class 'tuple'>
>>> L = list(T); L # conversion tuple -> list
[1, 2, 'a', True]
```

# Conversions tuple / list

Le passage d'un type à l'autre se fait avec les fonctions `tuple` et `list`.

```
>>> T = (1,2,'a',True)
>>> T
(1, 2, 'a', True)
>>> type(T)
<class 'tuple'>
>>> L = list(T); L # conversion tuple -> list
[1, 2, 'a', True]
>>> type(L)
<class 'list'>
```

# Conversions tuple / list

Le passage d'un type à l'autre se fait avec les fonctions `tuple` et `list`.

```
>>> T = (1,2,'a',True)
>>> T
(1, 2, 'a', True)
>>> type(T)
<class 'tuple'>
>>> L = list(T); L # conversion tuple -> list
[1, 2, 'a', True]
>>> type(L)
<class 'list'>
>>> L==T # attention, pas dégalité car types différents
False
```

# Conversions tuple / list

Le passage d'un type à l'autre se fait avec les fonctions `tuple` et `list`.

```
>>> T = (1,2,'a',True)
>>> T
(1, 2, 'a', True)
>>> type(T)
<class 'tuple'>
>>> L = list(T); L # conversion tuple -> list
[1, 2, 'a', True]
>>> type(L)
<class 'list'>
>>> L==T # attention, pas dégalité car types différents
False
>>> S = tuple(L); S # conversion list -> tuple
(1, 2, 'a', True)
```

# Conversions tuple / list

Le passage d'un type à l'autre se fait avec les fonctions `tuple` et `list`.

```
>>> T = (1,2,'a',True)
>>> T
(1, 2, 'a', True)
>>> type(T)
<class 'tuple'>
>>> L = list(T); L # conversion tuple -> list
[1, 2, 'a', True]
>>> type(L)
<class 'list'>
>>> L==T # attention, pas dégalité car types différents
False
>>> S = tuple(L); S # conversion list -> tuple
(1, 2, 'a', True)
>>> S==T
True
```

# Intérêt des tuples

- L'immutabilité évite les modifications accidentelles

# Intérêt des tuples

- L'immutabilité évite les modifications accidentelles
- Le traitement des tuples est plus rapide et moins gourmand en mémoire que celui des listes

# Intérêt des tuples

- L'immutabilité évite les modifications accidentelles
- Le traitement des tuples est plus rapide et moins gourmand en mémoire que celui des listes
- Les tuples sont, comme tous les types immutables de Python, hachables (on verra l'intérêt de cette propriété pour les dictionnaires)

# Intérêt des tuples

- L'immutabilité évite les modifications accidentelles
- Le traitement des tuples est plus rapide et moins gourmand en mémoire que celui des listes
- Les tuples sont, comme tous les types immutables de Python, hachables (on verra l'intérêt de cette propriété pour les dictionnaires)

Exemple : définition d'une fonction retournant plusieurs valeurs (un tuple)

```
def mv(L):
    """renvoie la moyenne et la variance de L"""
    m,v = 0,0
    for x in L:
        m,v = m+x,v+x**2
    m,v = m/len(L), v/len(L)
    return m, v-m**2

>>> moyenne, variance = mv([1,2,3,4,5])
>>> print(moyenne,variance)
3.0 2.0
```

# Les fonction `zip` et `enumerate`

La fonction `zip` crée, à partir de deux listes (ou tuples) A et B de même taille, la séquence des couples formés d'un élément de $A$ et de l'élément de même rang de B.

# Les fonction `zip` et `enumerate`

La fonction `zip` crée, à partir de deux listes (ou tuples) A et B de même taille, la séquence des couples formés d'un élément de $A$ et de l'élément de même rang de B.

Par exemple, au lieu de

```
produit_scalaire = sum( [u[i]*v[i] for i in range(len(u))] )
```

# Les fonction `zip` et `enumerate`

La fonction `zip` crée, à partir de deux listes (ou tuples) A et B de même taille, la séquence des couples formés d'un élément de $A$ et de l'élément de même rang de B.

Par exemple, au lieu de

```
produit_scalaire = sum( [u[i]*v[i] for i in range(len(u))] )
```

on peut écrire

```
produit_scalaire = sum( [a*b for a,b in zip(u,v)] )
```

# Les fonction `zip` et `enumerate`

La fonction `zip` crée, à partir de deux listes (ou tuples) A et B de même taille, la séquence des couples formés d'un élément de $A$ et de l'élément de même rang de B.

Par exemple, au lieu de

```
produit_scalaire = sum( [u[i]*v[i] for i in range(len(u))] )
```

on peut écrire

```
produit_scalaire = sum( [a*b for a,b in zip(u,v)] )
```

Un cas particulier est donné par la fonction `enumerate`, qui, à partir d'une liste (ou tuple) L, enumère la séquence des couples (i,L(i)) :

```
>>> L = ['Paris','Lyon','Marseille']
>>> print(list(enumerate(L))
[(0, 'Paris'), (1, 'Lyon'), (2, 'Marseille')]
```

# Les fonction `zip` et `enumerate`

La fonction `zip` crée, à partir de deux listes (ou tuples) A et B de même taille, la séquence des couples formés d'un élément de $A$ et de l'élément de même rang de B.

Par exemple, au lieu de

```
produit_scalaire = sum( [u[i]*v[i] for i in range(len(u))] )
```

on peut écrire

```
produit_scalaire = sum( [a*b for a,b in zip(u,v)] )
```

Un cas particulier est donné par la fonction `enumerate`, qui, à partir d'une liste (ou tuple) L, enumère la séquence des couples (i,L(i)) :

```
>>> L = ['Paris','Lyon','Marseille']
>>> print(list(enumerate(L))
[(0, 'Paris'), (1, 'Lyon'), (2, 'Marseille')]
```

est équivalent à

```
>>> print([(i,v) for i,v in zip(range(len(L)),L)])
[(0, 'Paris'), (1, 'Lyon'), (2, 'Marseille')]
```

# Ensembles

# Les ensembles

## Le type `set`

Rappel : un ensemble de type `set` est une collection non ordonnée et mutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

# Les ensembles

**Le type `set`**

Rappel : un ensemble de type `set` est une collection non ordonnée et mutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Opérations sur les ensembles :

- appartenance ($a \in A$) : `a in A`

# Les ensembles

**Le type `set`**

Rappel : un ensemble de type `set` est une collection non ordonnée et mutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Opérations sur les ensembles :

- appartenance ($a \in A$) : `a in A`
- non-appartenance ($a \notin A$) : `a not in A`

# Les ensembles

**Le type `set`**

Rappel : un ensemble de type `set` est une collection non ordonnée et mutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Opérations sur les ensembles :

- appartenance ($a \in A$) : `a in A`
- non-appartenance ($a \notin A$) : `a not in A`
- inclusion large ($A \subset B$) : `A <= B`

# Les ensembles

**Le type `set`**

Rappel : un ensemble de type `set` est une collection non ordonnée et mutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Opérations sur les ensembles :

- appartenance ($a \in A$) : `a in A`
- non-appartenance ($a \notin A$) : `a not in A`
- inclusion large ($A \subset B$) : `A <= B`
- inclusion stricte ($A \subsetneq B$) : `A < B`

# Les ensembles

**Le type `set`**

Rappel : un ensemble de type `set` est une collection non ordonnée et mutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Opérations sur les ensembles :

- appartenance ($a \in A$) : `a in A`
- non-appartenance ($a \notin A$) : `a not in A`
- inclusion large ($A \subset B$) : `A <= B`
- inclusion stricte ($A \subsetneq B$) : `A < B`
- intersection ($A \cap B$) : `A & B`

# Les ensembles

**Le type `set`**

Rappel : un ensemble de type `set` est une collection non ordonnée et mutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Opérations sur les ensembles :

- appartenance ($a \in A$) : `a in A`
- non-appartenance ($a \notin A$) : `a not in A`
- inclusion large ($A \subset B$) : `A <= B`
- inclusion stricte ($A \subsetneq B$) : `A < B`
- intersection ($A \cap B$) : `A & B`
- union ($A \cup B$) : `A | B`

# Les ensembles

**Le type `set`**

Rappel : un ensemble de type `set` est une collection non ordonnée et mutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Opérations sur les ensembles :

- appartenance ($a \in A$) : `a in A`
- non-appartenance ($a \notin A$) : `a not in A`
- inclusion large ($A \subset B$) : `A <= B`
- inclusion stricte ($A \subsetneq B$) : `A < B`
- intersection ($A \cap B$) : `A & B`
- union ($A \cup B$) : `A | B`
- différence ($A \setminus B$) : `A - B`

# Version immutable

## Le type `frozenset`

Un ensemble de type `frozenset` est une collection non ordonnée et immutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

# Version immutable

**Le type `frozenset`**

Un ensemble de type `frozenset` est une collection non ordonnée et immutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Le type `frozenset` est donc une sorte de type `set`, mais immutable. Il est à peu près au type `set` ce que le type `tuple` est au type `list`.

# Version immutable

**Le type `frozenset`**

Un ensemble de type `frozenset` est une collection non ordonnée et immutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Le type `frozenset` est donc une sorte de type `set`, mais immutable. Il est à peu près au type `set` ce que le type `tuple` est au type `list`.

Les conversions se font avec les fonctions `set()` et `frozenset()`.

# Version immutable

**Le type `frozenset`**

Un ensemble de type `frozenset` est une collection non ordonnée et immutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Le type `frozenset` est donc une sorte de type `set`, mais immutable. Il est à peu près au type `set` ce que le type `tuple` est au type `list`.

Les conversions se font avec les fonctions `set()` et `frozenset()`.

Les fonctions associées au type `set` qui n'apportent pas de modifications sont également définies pour le type `frozenset`.

# Version immutable

**Le type `frozenset`**

Un ensemble de type `frozenset` est une collection non ordonnée et immutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Le type `frozenset` est donc une sorte de type `set`, mais immutable. Il est à peu près au type `set` ce que le type `tuple` est au type `list`.

Les conversions se font avec les fonctions `set()` et `frozenset()`.

Les fonctions associées au type `set` qui n'apportent pas de modifications sont également définies pour le type `frozenset`.

Si $A$ est de type `set` et $B$ de type `frozenset`, alors :

- `A==B` est vrai si $A$ et $B$ contiennnent les mêmes éléments

# Version immutable

**Le type `frozenset`**

Un ensemble de type `frozenset` est une collection non ordonnée et immutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Le type `frozenset` est donc une sorte de type `set`, mais immutable. Il est à peu près au type `set` ce que le type `tuple` est au type `list`.

Les conversions se font avec les fonctions `set()` et `frozenset()`.

Les fonctions associées au type `set` qui n'apportent pas de modifications sont également définies pour le type `frozenset`.

Si $A$ est de type `set` et $B$ de type `frozenset`, alors :

- `A==B` est vrai si $A$ et $B$ contiennnent les mêmes éléments
- `A&B`, `A|B` et `A-B` renvoient un objet de type `set`

# Version immutable

**Le type `frozenset`**

Un ensemble de type `frozenset` est une collection non ordonnée et immutable d'éléments uniques (i.e. non répétés) et potentiellement hétérogènes.

Le type `frozenset` est donc une sorte de type `set`, mais immutable. Il est à peu près au type `set` ce que le type `tuple` est au type `list`.

Les conversions se font avec les fonctions `set()` et `frozenset()`.

Les fonctions associées au type `set` qui n'apportent pas de modifications sont également définies pour le type `frozenset`.

Si $A$ est de type `set` et $B$ de type `frozenset`, alors :

- `A==B` est vrai si $A$ et $B$ contiennnent les mêmes éléments
- `A&B`, `A|B` et `A-B` renvoient un objet de type `set`

Les éléments d'un `frozenset` doivent être immutables (en fait hachables, cf. suite), alors qu'un `tuple` peut contenir des `list`)

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
```

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
>>> B = frozenset(range(11,21,2)); B
frozenset({17, 11, 19, 13, 15})
```

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
>>> B = frozenset(range(11,21,2)); B
frozenset({17, 11, 19, 13, 15})
>>> 11 in B # test d'appartenance
True
```

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
>>> B = frozenset(range(11,21,2)); B
frozenset({17, 11, 19, 13, 15})
>>> 11 in B # test d'appartenance
True
>>> A <= B # test d'inclusion
False
```

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
>>> B = frozenset(range(11,21,2)); B
frozenset({17, 11, 19, 13, 15})
>>> 11 in B # test d'appartenance
True
>>> A <= B # test d'inclusion
False
>>> A&B #intersection
frozenset({19})
```

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
>>> B = frozenset(range(11,21,2)); B
frozenset({17, 11, 19, 13, 15})
>>> 11 in B # test d'appartenance
True
>>> A <= B # test d'inclusion
False
>>> A&B #intersection
frozenset({19})
>>> A|B # union
frozenset({17, 19, 11, 12, 13, 15})
```

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
>>> B = frozenset(range(11,21,2)); B
frozenset({17, 11, 19, 13, 15})
>>> 11 in B # test d'appartenance
True
>>> A <= B # test d'inclusion
False
>>> A&B #intersection
frozenset({19})
>>> A|B # union
frozenset({17, 19, 11, 12, 13, 15})
>>> A.add(10) # incorrect (les frozenset sont immutables)
AttributeError: 'frozenset' object has no attribute 'add'
```

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
>>> B = frozenset(range(11,21,2)); B
frozenset({17, 11, 19, 13, 15})
>>> 11 in B # test d'appartenance
True
>>> A <= B # test d'inclusion
False
>>> A&B #intersection
frozenset({19})
>>> A|B # union
frozenset({17, 19, 11, 12, 13, 15})
>>> A.add(10) # incorrect (les frozenset sont immutables)
AttributeError: 'frozenset' object has no attribute 'add'
>>> A = A|frozenset({10}); A # correct
frozenset({10, 19, 12})
```

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
>>> B = frozenset(range(11,21,2)); B
frozenset({17, 11, 19, 13, 15})
>>> 11 in B # test d'appartenance
True
>>> A <= B # test d'inclusion
False
>>> A&B #intersection
frozenset({19})
>>> A|B # union
frozenset({17, 19, 11, 12, 13, 15})
>>> A.add(10) # incorrect (les frozenset sont immutables)
AttributeError: 'frozenset' object has no attribute 'add'
>>> A = A|frozenset({10}); A # correct
frozenset({10, 19, 12})
>>> C = frozenset({{1,2},{3,4}}) # incorrect
TypeError: unhashable type: 'set'
```

# Exemple d'utilisation des `frozenset`

```
>>> A = frozenset([12,19]); A
frozenset({19, 12})
>>> B = frozenset(range(11,21,2)); B
frozenset({17, 11, 19, 13, 15})
>>> 11 in B # test d'appartenance
True
>>> A <= B # test d'inclusion
False
>>> A&B #intersection
frozenset({19})
>>> A|B # union
frozenset({17, 19, 11, 12, 13, 15})
>>> A.add(10) # incorrect (les frozenset sont immutables)
AttributeError: 'frozenset' object has no attribute 'add'
>>> A = A|frozenset({10}); A # correct
frozenset({10, 19, 12})
>>> C = frozenset({{1,2},{3,4}}) # incorrect
TypeError: unhashable type: 'set'
>>> C = frozenset({frozenset({1,2}),frozenset({3,4})}); C # ok
frozenset({frozenset({3, 4}), frozenset({1, 2})})
```

# Tableaux associatifs (dictionnaires)

# Les dictionnaires

## Le type `dict`

Un tableau associatif (ou dictionnaire) de type `dict` est une collection non ordonnée et mutable de paires (clé, valeur), d'éléments éventuellement hétérogènes. Chaque clé doit être de type immutable.

# Les dictionnaires

## Le type `dict`

Un tableau associatif (ou dictionnaire) de type `dict` est une collection non ordonnée et mutable de paires (clé, valeur), d'éléments éventuellement hétérogènes. Chaque clé doit être de type immutable.

- définition directe avec la syntaxe {*clé1 :valeur1, clé2 :valeur2, ...*} :

```
>>> d = {'veste':2, 'pantalon':4, 'chaussures':4}
```

# Les dictionnaires

## Le type `dict`

Un tableau associatif (ou dictionnaire) de type `dict` est une collection non ordonnée et mutable de paires (clé, valeur), d'éléments éventuellement hétérogènes. Chaque clé doit être de type immutable.

- définition directe avec la syntaxe {*clé1 :valeur1, clé2 :valeur2, ...*} :

```
>>> d = {'veste':2, 'pantalon':4, 'chaussures':4}
>>> d # ordre quelconque (avant Python v. 3.7)
{'pantalon': 4, 'chaussures': 4, 'veste': 2}
```

# Les dictionnaires

## Le type `dict`

Un tableau associatif (ou dictionnaire) de type `dict` est une collection non ordonnée et mutable de paires (clé, valeur), d'éléments éventuellement hétérogènes. Chaque clé doit être de type immutable.

- définition directe avec la syntaxe {*clé1 :valeur1, clé2 :valeur2, ...*} :

```
>>> d = {'veste':2, 'pantalon':4, 'chaussures':4}
>>> d # ordre quelconque (avant Python v. 3.7)
{'pantalon': 4, 'chaussures': 4, 'veste': 2}
```

- définition incrémentale

# Les dictionnaires

## Le type `dict`

Un tableau associatif (ou dictionnaire) de type `dict` est une collection non ordonnée et mutable de paires (clé, valeur), d'éléments éventuellement hétérogènes. Chaque clé doit être de type immutable.

- définition directe avec la syntaxe {*clé1 :valeur1, clé2 :valeur2, ...*} :

```
>>> d = {'veste':2, 'pantalon':4, 'chaussures':4}
>>> d # ordre quelconque (avant Python v. 3.7)
{'pantalon': 4, 'chaussures': 4, 'veste': 2}
```

- définition incrémentale

```
>>> d = {} # ou bien d = dict()
```

# Les dictionnaires

## Le type `dict`

Un tableau associatif (ou dictionnaire) de type `dict` est une collection non ordonnée et mutable de paires (clé, valeur), d'éléments éventuellement hétérogènes. Chaque clé doit être de type immutable.

- définition directe avec la syntaxe {*clé1 :valeur1, clé2 :valeur2, ...*} :

```
>>> d = {'veste':2, 'pantalon':4, 'chaussures':4}
>>> d # ordre quelconque (avant Python v. 3.7)
{'pantalon': 4, 'chaussures': 4, 'veste': 2}
```

- définition incrémentale

```
>>> d = {} # ou bien d = dict()
>>> d['veste'] = 2
```

# Les dictionnaires

## Le type `dict`

Un tableau associatif (ou dictionnaire) de type `dict` est une collection non ordonnée et mutable de paires (clé, valeur), d'éléments éventuellement hétérogènes. Chaque clé doit être de type immutable.

- définition directe avec la syntaxe {*clé1 :valeur1, clé2 :valeur2, ...*} :

```
>>> d = {'veste':2, 'pantalon':4, 'chaussures':4}
>>> d # ordre quelconque (avant Python v. 3.7)
{'pantalon': 4, 'chaussures': 4, 'veste': 2}
```

- définition incrémentale

```
>>> d = {} # ou bien d = dict()
>>> d['veste'] = 2
>>> d['pantalon'] = 4
```

# Les dictionnaires

## Le type `dict`

Un tableau associatif (ou dictionnaire) de type `dict` est une collection non ordonnée et mutable de paires (clé, valeur), d'éléments éventuellement hétérogènes. Chaque clé doit être de type immutable.

- définition directe avec la syntaxe {*clé1 :valeur1, clé2 :valeur2, ...*} :

```
>>> d = {'veste':2, 'pantalon':4, 'chaussures':4}
>>> d # ordre quelconque (avant Python v. 3.7)
{'pantalon': 4, 'chaussures': 4, 'veste': 2}
```

- définition incrémentale

```
>>> d = {} # ou bien d = dict()
>>> d['veste'] = 2
>>> d['pantalon'] = 4
>>> d['chaussures'] = 4
```

# Les dictionnaires

## Le type `dict`

Un tableau associatif (ou dictionnaire) de type `dict` est une collection non ordonnée et mutable de paires (clé, valeur), d'éléments éventuellement hétérogènes. Chaque clé doit être de type immutable.

- définition directe avec la syntaxe {*clé1 :valeur1, clé2 :valeur2, ...*} :

```
>>> d = {'veste':2, 'pantalon':4, 'chaussures':4}
>>> d # ordre quelconque (avant Python v. 3.7)
{'pantalon': 4, 'chaussures': 4, 'veste': 2}
```

- définition incrémentale

```
>>> d = {} # ou bien d = dict()
>>> d['veste'] = 2
>>> d['pantalon'] = 4
>>> d['chaussures'] = 4
>>> d
{'pantalon': 4, 'chaussures': 4, 'veste': 2}
```

# Cas particulier : clés de type `str`

Lorsque les clés sont des chaînes de caractères simples, il existe une syntaxe particulière, de type `clé=valeur` :

# Cas particulier : clés de type `str`

Lorsque les clés sont des chaînes de caractères simples, il existe une syntaxe particulière, de type `clé=valeur` :

```
# bien noter l'absence de guillemets autour des chaînes clés !
d = dict(veste=2, pantalon=4, chaussures=4)
```

# Cas particulier : clés de type `str`

Lorsque les clés sont des chaînes de caractères simples, il existe une syntaxe particulière, de type `clé=valeur` :

```
# bien noter l'absence de guillemets autour des chaînes clés !
d = dict(veste=2, pantalon=4, chaussures=4)
```

est équivalent à

```
d = {'veste':2, 'pantalon':4, 'chaussures':4}
```

# Cas particulier : clés de type `str`

Lorsque les clés sont des chaînes de caractères simples, il existe une syntaxe particulière, de type `clé=valeur` :

```
# bien noter l'absence de guillemets autour des chaînes clés !
d = dict(veste=2, pantalon=4, chaussures=4)
```

est équivalent à

```
d = {'veste':2, 'pantalon':4, 'chaussures':4}
```

Attention, ceci ne fonctionne que si les chaînes de caractères définissant les clés sont des noms de variables valides :

# Cas particulier : clés de type `str`

Lorsque les clés sont des chaînes de caractères simples, il existe une syntaxe particulière, de type `clé=valeur` :

```
# bien noter l'absence de guillemets autour des chaînes clés !
d = dict(veste=2, pantalon=4, chaussures=4)
```

est équivalent à

```
d = {'veste':2, 'pantalon':4, 'chaussures':4}
```

Attention, ceci ne fonctionne que si les chaînes de caractères définissant les clés sont des noms de variables valides :

```
>>> d = dict(veste=2, 1pantalon=4, chaussures=4)
SyntaxError: invalid syntax
```

# Cas particulier : clés de type `str`

Lorsque les clés sont des chaînes de caractères simples, il existe une syntaxe particulière, de type `clé=valeur` :

```
# bien noter l'absence de guillemets autour des chaînes clés !
d = dict(veste=2, pantalon=4, chaussures=4)
```

est équivalent à

```
d = {'veste':2, 'pantalon':4, 'chaussures':4}
```

Attention, ceci ne fonctionne que si les chaînes de caractères définissant les clés sont des noms de variables valides :

```
>>> d = dict(veste=2, 1pantalon=4, chaussures=4)
SyntaxError: invalid syntax
>>> d = dict(veste=2, pantalon=4, chaussures fermées=4)
SyntaxError: invalid syntax
```

# Cas particulier : clés de type `str`

Lorsque les clés sont des chaînes de caractères simples, il existe une syntaxe particulière, de type `clé=valeur` :

```
# bien noter l'absence de guillemets autour des chaînes clés !
d = dict(veste=2, pantalon=4, chaussures=4)
```

est équivalent à

```
d = {'veste':2, 'pantalon':4, 'chaussures':4}
```

Attention, ceci ne fonctionne que si les chaînes de caractères définissant les clés sont des noms de variables valides :

```
>>> d = dict(veste=2, 1pantalon=4, chaussures=4)
SyntaxError: invalid syntax
>>> d = dict(veste=2, pantalon=4, chaussures fermées=4)
SyntaxError: invalid syntax
>>> d = dict(veste=2, pantalon=4, chaussures_fermées=4) # ok
```

# Cas particulier : clés de type `str`

Lorsque les clés sont des chaînes de caractères simples, il existe une syntaxe particulière, de type `clé=valeur` :

```
# bien noter l'absence de guillemets autour des chaînes clés !
d = dict(veste=2, pantalon=4, chaussures=4)
```

est équivalent à

```
d = {'veste':2, 'pantalon':4, 'chaussures':4}
```

Attention, ceci ne fonctionne que si les chaînes de caractères définissant les clés sont des noms de variables valides :

```
>>> d = dict(veste=2, 1pantalon=4, chaussures=4)
SyntaxError: invalid syntax
>>> d = dict(veste=2, pantalon=4, chaussures fermées=4)
SyntaxError: invalid syntax
>>> d = dict(veste=2, pantalon=4, chaussures_fermées=4) # ok
```

Cette syntaxe particulière prendra tout son sens lorsque nous étudierons les fonctions avec arguments optionnels

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
```

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
>>> d[1789] = 'Révolution'
```

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
>>> d[1789] = 'Révolution'
>>> d[[1495,1610]] = 'Renaissance' # incorrect (list)
```

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
>>> d[1789] = 'Révolution'
>>> d[[1495,1610]] = 'Renaissance' # incorrect (list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
>>> d[1789] = 'Révolution'
>>> d[[1495,1610]] = 'Renaissance' # incorrect (list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d[(1495,1610)] = 'Renaissance' # correct (tuple)
```

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
>>> d[1789] = 'Révolution'
>>> d[[1495,1610]] = 'Renaissance' # incorrect (list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d[(1495,1610)] = 'Renaissance' # correct (tuple)
>>> d
{1515:'Marignan', 1789:'Révolution', (1495,1610):'Renaissance'}
```

**Conséquence : pour les clés d'un dictionnaire,**

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
>>> d[1789] = 'Révolution'
>>> d[[1495,1610]] = 'Renaissance' # incorrect (list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d[(1495,1610)] = 'Renaissance' # correct (tuple)
>>> d
{1515:'Marignan', 1789:'Révolution', (1495,1610):'Renaissance'}
```

**Conséquence : pour les clés d'un dictionnaire,**

- les types `int`, `float`, `complex`, `bool`, `str` sont acceptés tels quels

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
>>> d[1789] = 'Révolution'
>>> d[[1495,1610]] = 'Renaissance' # incorrect (list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d[(1495,1610)] = 'Renaissance' # correct (tuple)
>>> d
{1515:'Marignan', 1789:'Révolution', (1495,1610):'Renaissance'}
```

**Conséquence : pour les clés d'un dictionnaire,**

- les types `int`, `float`, `complex`, `bool`, `str` sont acceptés tels quels
- Le type `list` doit être remplacé par le type `tuple`*

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
>>> d[1789] = 'Révolution'
>>> d[[1495,1610]] = 'Renaissance' # incorrect (list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d[(1495,1610)] = 'Renaissance' # correct (tuple)
>>> d
{1515:'Marignan', 1789:'Révolution', (1495,1610):'Renaissance'}
```

**Conséquence : pour les clés d'un dictionnaire,**

- les types `int`, `float`, `complex`, `bool`, `str` sont acceptés tels quels
- Le type `list` doit être remplacé par le type `tuple`*
- Le type `set` doit être remplacé par le type `frozenset`

# La clé doit être immutable

```
>>> d = dict()
>>> d[1515] = 'Marignan'
>>> d[1789] = 'Révolution'
>>> d[[1495,1610]] = 'Renaissance' # incorrect (list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d[(1495,1610)] = 'Renaissance' # correct (tuple)
>>> d
{1515:'Marignan', 1789:'Révolution', (1495,1610):'Renaissance'}
```

**Conséquence : pour les clés d'un dictionnaire,**

- les types `int`, `float`, `complex`, `bool`, `str` sont acceptés tels quels
- Le type `list` doit être remplacé par le type `tuple`*
- Le type `set` doit être remplacé par le type `frozenset`

* avec des éléments eux-mêmes immutables

# Pourquoi les clés doivent-elles être immutables ?

Lorsque l'on demande la valeur `D[k]`, Python doit chercher `k` parmi les clés du dictionnaire `D`

# Pourquoi les clés doivent-elles être immutables ?

Lorsque l'on demande la valeur `D[k]`, Python doit chercher `k` parmi les clés du dictionnaire `D`

La recherche systématique consistant à balayer toutes les clés de `D` jusqu'à trouver `k` est très inefficace : complexité en temps $O(N)$, où $N$ est le nombre de clés

# Pourquoi les clés doivent-elles être immutables ?

Lorsque l'on demande la valeur `D[k]`, Python doit chercher `k` parmi les clés du dictionnaire `D`

La recherche systématique consistant à balayer toutes les clés de `D` jusqu'à trouver `k` est très inefficace : complexité en temps $O(N)$, où $N$ est le nombre de clés

$\rightarrow$ l'idée est de maintenir une liste de clés triée, afin de faire la recherche en temps $O(\log N)$

# Pourquoi les clés doivent-elles être immutables ?

Lorsque l'on demande la valeur `D[k]`, Python doit chercher `k` parmi les clés du dictionnaire `D`

La recherche systématique consistant à balayer toutes les clés de `D` jusqu'à trouver `k` est très inefficace : complexité en temps $O(N)$, où $N$ est le nombre de clés

$\rightarrow$ l'idée est de maintenir une liste de clés triée, afin de faire la recherche en temps $O(\log N)$

Comme les clés peuvent être de natures très différentes, Python utilise une fonction de hachage (`hash()`), qui fabrique un entier (appelé empreinte) à partir d'une clé quelconque.

# Pourquoi les clés doivent-elles être immutables ?

```
>>> hash(1515) # pour un entier assez petit, identité
1515
```

# Pourquoi les clés doivent-elles être immutables ?

```
>>> hash(1515) # pour un entier assez petit, identité
1515
>>> hash([1495,1610]) # incorrect (list n'est pas hachable)
TypeError: unhashable type: 'list'
```

# Pourquoi les clés doivent-elles être immutables ?

```
>>> hash(1515) # pour un entier assez petit, identité
1515
>>> hash([1495,1610]) # incorrect (list n'est pas hachable)
TypeError: unhashable type: 'list'
>>> hash((1495,1610)) # correct (tuple est hachable)
3712308706137777606
```

# Pourquoi les clés doivent-elles être immutables ?

```
>>> hash(1515) # pour un entier assez petit, identité
1515
>>> hash([1495,1610]) # incorrect (list n'est pas hachable)
TypeError: unhashable type: 'list'
>>> hash((1495,1610)) # correct (tuple est hachable)
3712308706137777606
>>> hash('Python 3') # fonction aussi définie pour des chaînes
1842537825426745422
```

# Pourquoi les clés doivent-elles être immutables ?

```
>>> hash(1515) # pour un entier assez petit, identité
1515
>>> hash([1495,1610]) # incorrect (list n'est pas hachable)
TypeError: unhashable type: 'list'
>>> hash((1495,1610)) # correct (tuple est hachable)
3712308706137777606
>>> hash('Python 3') # fonction aussi définie pour des chaînes
1842537825426745422
>>> hash(3.1415) # ou des réels
32627678580373 8115
```

# Pourquoi les clés doivent-elles être immutables ?

```
>>> hash(1515) # pour un entier assez petit, identité
1515
>>> hash([1495,1610]) # incorrect (list n'est pas hachable)
TypeError: unhashable type: 'list'
>>> hash((1495,1610)) # correct (tuple est hachable)
3712308706137777606
>>> hash('Python 3') # fonction aussi définie pour des chaînes
1842537825426745422
>>> hash(3.1415) # ou des réels
32627678580373 8115
>>> hash(32627678580373 8115) # pas injective
32627678580373 8115
```

# Pourquoi les clés doivent-elles être immutables ?

```
>>> hash(1515) # pour un entier assez petit, identité
1515
>>> hash([1495,1610]) # incorrect (list n'est pas hachable)
TypeError: unhashable type: 'list'
>>> hash((1495,1610)) # correct (tuple est hachable)
3712308706137777606
>>> hash('Python 3') # fonction aussi définie pour des chaînes
1842537825426745422
>>> hash(3.1415) # ou des réels
32627678580373 8115
>>> hash(32627678580373 8115) # pas injective
32627678580373 8115
```

Python utilise ces valeurs entières pour trier les clés.

# Pourquoi les clés doivent-elles être immutables ?

```
>>> hash(1515) # pour un entier assez petit, identité
1515
>>> hash([1495,1610]) # incorrect (list n'est pas hachable)
TypeError: unhashable type: 'list'
>>> hash((1495,1610)) # correct (tuple est hachable)
3712308706137777606
>>> hash('Python 3') # fonction aussi définie pour des chaînes
1842537825426745422
>>> hash(3.1415) # ou des réels
32627678580373 8115
>>> hash(32627678580373 8115) # pas injective
32627678580373 8115
```

Python utilise ces valeurs entières pour trier les clés.

En Python, les types immutables (simples ou composés d'éléments immutables) sont hachables.

# Pourquoi les clés doivent-elles être immutables ?

```
>>> hash(1515) # pour un entier assez petit, identité
1515
>>> hash([1495,1610]) # incorrect (list n'est pas hachable)
TypeError: unhashable type: 'list'
>>> hash((1495,1610)) # correct (tuple est hachable)
3712308706137777606
>>> hash('Python 3') # fonction aussi définie pour des chaînes
1842537825426745422
>>> hash(3.1415) # ou des réels
32627678580373 8115
>>> hash(32627678580373 8115) # pas injective
32627678580373 8115
```

Python utilise ces valeurs entières pour trier les clés.

En Python, les types immutables (simples ou composés d'éléments immutables) sont hachables.

**Remarque :** La fonction `hash()` n'est pas injective, mais les *collisions* (deux arguments différents qui conduisent à la même valeur retournée par `hash()`) sont rares.

# Parcours d'un dictionnaire

Si `d` est un dictionnaire :

- `d.keys()` est une séquence décrivant l'ensemble des clés

# Parcours d'un dictionnaire

Si `d` est un dictionnaire :

- `d.keys()` est une séquence décrivant l'ensemble des clés
- si `k` est une clé, `d[k]` est la valeur correspondante dans `d`

# Parcours d'un dictionnaire

Si `d` est un dictionnaire :

- `d.keys()` est une séquence décrivant l'ensemble des clés
- si `k` est une clé, `d[k]` est la valeur correspondante dans `d`

---

```
>>> d = {'arbre':'tree', 'route':'road', 'hérisson':'hedgehog'}
```

# Parcours d'un dictionnaire

Si `d` est un dictionnaire :

- `d.keys()` est une séquence décrivant l'ensemble des clés
- si `k` est une clé, `d[k]` est la valeur correspondante dans `d`

---

```
>>> d = {'arbre':'tree', 'route':'road', 'hérisson':'hedgehog'}
>>> print(d.keys()) # la séquence des clés
dict_keys(['arbre', 'road', 'hérisson'])
```

# Parcours d'un dictionnaire

Si `d` est un dictionnaire :

- `d.keys()` est une séquence décrivant l'ensemble des clés
- si `k` est une clé, `d[k]` est la valeur correspondante dans `d`

---

```
>>> d = {'arbre':'tree', 'route':'road', 'hérisson':'hedgehog'}
>>> print(d.keys()) # la séquence des clés
dict_keys(['arbre', 'road', 'hérisson'])
>>> print(list(d.keys())) # peut être convertie en liste
['arbre', 'road', 'hérisson']
```

# Parcours d'un dictionnaire

Si `d` est un dictionnaire :

- `d.keys()` est une séquence décrivant l'ensemble des clés
- si `k` est une clé, `d[k]` est la valeur correspondante dans `d`

---

```
>>> d = {'arbre':'tree', 'route':'road', 'hérisson':'hedgehog'}
>>> print(d.keys()) # la séquence des clés
dict_keys(['arbre', 'road', 'hérisson'])
>>> print(list(d.keys())) # peut être convertie en liste
['arbre', 'road', 'hérisson']
>>> print(tuple(d.keys())) # ou en tuple
('arbre', 'route', 'hérisson')
```

# Parcours d'un dictionnaire

Si `d` est un dictionnaire :

- `d.keys()` est une séquence décrivant l'ensemble des clés
- si `k` est une clé, `d[k]` est la valeur correspondante dans `d`

---

```
>>> d = {'arbre':'tree', 'route':'road', 'hérisson':'hedgehog'}
>>> print(d.keys()) # la séquence des clés
dict_keys(['arbre', 'road', 'hérisson'])
>>> print(list(d.keys())) # peut être convertie en liste
['arbre', 'road', 'hérisson']
>>> print(tuple(d.keys())) # ou en tuple
('arbre', 'route', 'hérisson')
>>> for k in d.keys(): # parcours de l'ensemble des clés
...     print(k,'se traduit en anglais par',d[k])
```

# Parcours d'un dictionnaire

Si `d` est un dictionnaire :

- `d.keys()` est une séquence décrivant l'ensemble des clés
- si `k` est une clé, `d[k]` est la valeur correspondante dans `d`

```
>>> d = {'arbre':'tree', 'route':'road', 'hérisson':'hedgehog'}
>>> print(d.keys()) # la séquence des clés
dict_keys(['arbre', 'road', 'hérisson'])
>>> print(list(d.keys())) # peut être convertie en liste
['arbre', 'road', 'hérisson']
>>> print(tuple(d.keys())) # ou en tuple
('arbre', 'route', 'hérisson')
>>> for k in d.keys(): # parcours de l'ensemble des clés
...     print(k,'se traduit en anglais par',d[k])
...
arbre se traduit en anglais par tree
route se traduit en anglais par road
hérisson se traduit en anglais par hedgehog
```

**Remarque :** on peut écrire `k in d` à la place de `k in d.keys()`

# Manipulation des dictionnaires

- remplacement d'un élément par redéfinition

```
>>> D = {1:'a',2:'c',3:'c',4:'d'}
>>> D[2] = 'b'; D # redéfinition de D[2]
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

# Manipulation des dictionnaires

- remplacement d'un élément par redéfinition

```
>>> D = {1:'a',2:'c',3:'c',4:'d'}
>>> D[2] = 'b'; D # redéfinition de D[2]
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

- suppression d'un élément avec `del`

```
>>> del D[1]; D
{2: 'b', 3: 'c', 4: 'd'}
```

# Manipulation des dictionnaires

- remplacement d'un élément par redéfinition

```
>>> D = {1:'a',2:'c',3:'c',4:'d'}
>>> D[2] = 'b'; D # redéfinition de D[2]
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

- suppression d'un élément avec `del`

```
>>> del D[1]; D
{2: 'b', 3: 'c', 4: 'd'}
```

- attention, pas de "tranches" possibles avec les dictionnaires :

```
>>> D[2:4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'slice'
```

# Manipulation des dictionnaires

- remplacement d'un élément par redéfinition

```
>>> D = {1:'a',2:'c',3:'c',4:'d'}
>>> D[2] = 'b'; D # redéfinition de D[2]
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

- suppression d'un élément avec `del`

```
>>> del D[1]; D
{2: 'b', 3: 'c', 4: 'd'}
```

- attention, pas de "tranches" possibles avec les dictionnaires :

```
>>> D[2:4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'slice'
```

- de même, pas d'extraction multiple possible

```
>>> D[2,3,4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: (2, 3, 4)
```

# Clés potentiellement inexistantes : la méthode `get()`

Si `k` n'est pas une clé du dictionnaire `D`, alors `D[k]` produit une erreur

$\rightarrow$ à chaque appel de type `D[k]`, il faut s'assurer que `k` est une clé valide

# Clés potentiellement inexistantes : la méthode `get()`

Si `k` n'est pas une clé du dictionnaire `D`, alors `D[k]` produit une erreur

$\rightarrow$ à chaque appel de type `D[k]`, il faut s'assurer que `k` est une clé valide

**Exemple :** dénombrement du contenu d'une liste `L`

```
D = dict()
for k in L:
    if k in D: # équivalent à if k in D.keys():
        D[k] = D[k] + 1
    else:
        D[k] = 1
```

# Clés potentiellement inexistantes : la méthode `get()`

Si `k` n'est pas une clé du dictionnaire `D`, alors `D[k]` produit une erreur

$\rightarrow$ à chaque appel de type `D[k]`, il faut s'assurer que `k` est une clé valide

**Exemple :** dénombrement du contenu d'une liste `L`

```
D = dict()
for k in L:
    if k in D: # équivalent à if k in D.keys():
        D[k] = D[k] + 1
    else:
        D[k] = 1
```

On peut éviter ce test systématique au moyen de la méthode `get()`, qui permet de spécifier une valeur par défaut :

`D.get(k,v)` renvoie `D[k]` si `k` est une clé de `D`, et `v` sinon

# Clés potentiellement inexistantes : la méthode `get()`

Si `k` n'est pas une clé du dictionnaire `D`, alors `D[k]` produit une erreur

$\rightarrow$ à chaque appel de type `D[k]`, il faut s'assurer que `k` est une clé valide

**Exemple :** dénombrement du contenu d'une liste `L`

```
D = dict()
for k in L:
    if k in D: # équivalent à if k in D.keys():
        D[k] = D[k] + 1
    else:
        D[k] = 1
```

On peut éviter ce test systématique au moyen de la méthode `get()`, qui permet de spécifier une valeur par défaut :

> `D.get(k,v)` renvoie `D[k]` si `k` est une clé de `D`, et `v` sinon

Le code précédent devient donc

```
D = dict()
for k in L:
    D[k] = D.get(k,0) + 1
```

# Autres fonctions et méthodes sur les dictionnaires

Si `D` est un dictionnaire :

- `D.clear()` efface toutes les paires (clé,valeur) de `D`

# Autres fonctions et méthodes sur les dictionnaires

Si `D` est un dictionnaire :

- `D.clear()` efface toutes les paires (clé,valeur) de `D`
- `D.copy()` renvoie une copie de `D`
  (attention, même remarque que pour les listes et les tuples)

# Autres fonctions et méthodes sur les dictionnaires

Si `D` est un dictionnaire :

- `D.clear()` efface toutes les paires (clé,valeur) de `D`
- `D.copy()` renvoie une copie de `D`
  (attention, même remarque que pour les listes et les tuples)
- `len(D)` renvoie le nombre de clés de `D`

# Autres fonctions et méthodes sur les dictionnaires

Si `D` est un dictionnaire :

- `D.clear()` efface toutes les paires (clé,valeur) de `D`
- `D.copy()` renvoie une copie de `D`
  (attention, même remarque que pour les listes et les tuples)
- `len(D)` renvoie le nombre de clés de `D`
- `D.items()` renvoie toutes les paires (clé,valeur) de `D`

# Autres fonctions et méthodes sur les dictionnaires

Si `D` est un dictionnaire :

- `D.clear()` efface toutes les paires (clé,valeur) de `D`
- `D.copy()` renvoie une copie de `D`
  (attention, même remarque que pour les listes et les tuples)
- `len(D)` renvoie le nombre de clés de `D`
- `D.items()` renvoie toutes les paires (clé,valeur) de `D`
- `D.popitem()` renvoie un couple (clé,valeur) de `D`, et le supprime dans `D`

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

On suppose déjà codée la fonction `is_prime(p)`, qui teste si l'argument `p` est un nombre premier (et renvoie un booléen)

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

On suppose déjà codée la fonction `is_prime(p)`, qui teste si l'argument `p` est un nombre premier (et renvoie un booléen)

---

```
>>> N = 3300
```

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

On suppose déjà codée la fonction `is_prime(p)`, qui teste si l'argument `p` est un nombre premier (et renvoie un booléen)

```
>>> N = 3300
>>> D = dict()
```

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

On suppose déjà codée la fonction `is_prime(p)`, qui teste si l'argument `p` est un nombre premier (et renvoie un booléen)

```
>>> N = 3300
>>> D = dict()
>>> for p in range(2,N+1): # diviseurs premiers potentiels
```

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

On suppose déjà codée la fonction `is_prime(p)`, qui teste si l'argument p est un nombre premier (et renvoie un booléen)

```
>>> N = 3300
>>> D = dict()
>>> for p in range(2,N+1): # diviseurs premiers potentiels
...     if is_prime(p): # on se limite à p premier
```

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

On suppose déjà codée la fonction `is_prime(p)`, qui teste si l'argument p est un nombre premier (et renvoie un booléen)

```
>>> N = 3300
>>> D = dict()
>>> for p in range(2,N+1): # diviseurs premiers potentiels
...     if is_prime(p): # on se limite à p premier
...         while N%p==0:
```

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

On suppose déjà codée la fonction `is_prime(p)`, qui teste si l'argument p est un nombre premier (et renvoie un booléen)

```
>>> N = 3300
>>> D = dict()
>>> for p in range(2,N+1): # diviseurs premiers potentiels
...     if is_prime(p): # on se limite à p premier
...         while N%p==0:
...             N = N//p # division de N par p
```

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

On suppose déjà codée la fonction `is_prime(p)`, qui teste si l'argument p est un nombre premier (et renvoie un booléen)

```
>>> N = 3300
>>> D = dict()
>>> for p in range(2,N+1): # diviseurs premiers potentiels
...     if is_prime(p): # on se limite à p premier
...         while N%p==0:
...             N = N//p # division de N par p
...             D[p] = D.get(p,0) + 1 # incrémentation de D[p]
```

# Exemple d'utilisation : décomposition en facteurs premiers

Tout entier $n$ s'écrit sous la forme $\boxed{n = \prod_{p \in P_n} p^{k_p}}$,

où $P_n$ désigne l'ensemble des diviseurs premiers de $n$, et $k_p \in \mathbb{N}^*$

**Exemple :** $3300 = 33 \times 100 = 2^2 \times 3 \times 5^2 \times 11$

On souhaite coder, pour un entier $N$, cette décomposition dans un dictionnaire, sous la forme $D[p] = k_p$ pour tout $p \in P_n$

On suppose déjà codée la fonction `is_prime(p)`, qui teste si l'argument p est un nombre premier (et renvoie un booléen)

```
>>> N = 3300
>>> D = dict()
>>> for p in range(2,N+1): # diviseurs premiers potentiels
...     if is_prime(p): # on se limite à p premier
...         while N%p==0:
...             N = N//p # division de N par p
...             D[p] = D.get(p,0) + 1 # incrémentation de D[p]
>>> D # code la décomposition en facteurs premiers de N
{11: 1, 2: 2, 3: 1, 5: 2}
```

# Exemple d'utilisation : décomposition en facteurs premiers

## Vérification :

```
def prod(L):
    """ retourne le produit des éléments de L """
    p = 1
    for x in L:
        p = p*x
    return p
```

# Exemple d'utilisation : décomposition en facteurs premiers

## Vérification :

```
def prod(L):
    """ retourne le produit des éléments de L """
    p = 1
    for x in L:
        p = p*x
    return p

>>> D
{11: 1, 2: 2, 3: 1, 5: 2}
```

# Exemple d'utilisation : décomposition en facteurs premiers

## Vérification :

```
def prod(L):
    """ retourne le produit des éléments de L """
    p = 1
    for x in L:
        p = p*x
    return p

>>> D
{11: 1, 2: 2, 3: 1, 5: 2}
>>> prod([p**D[p] for p in D])
3300
```