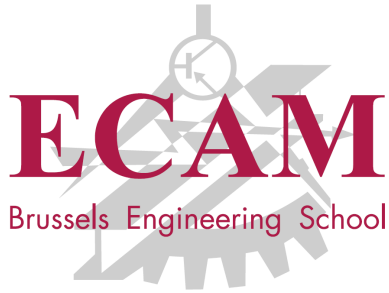


IC1T Programmation

Cours 4
Fonctions et listes

Sébastien Combéfis, Quentin Lurkin

Rappels

- Manipulations de chaînes de caractères
  - opérations sur les chaînes de caractères
  - accès aux caractères et *slicing*
  - les `string` sont non-modifiables
- Définition de fonctions
  - définition et appel de fonctions
  - paramètres et valeurs de retour
  - variables locales et globales

Objectifs

- Définition de fonctions
  - divisions de problèmes
  - fonctions récursives
  - modules
- Séquences et listes
  - modifications de listes
  - parcours de listes
  - copies de listes

Diviser les problèmes

Divisions de problèmes

- Les longs programmes ne sont pas toujours faciles à comprendre.

```
1  n = 1
2  nb = 10
3  while nb > 0:
4      divisors = 0
5      d = 1
6      while d <= n:
7          if n % d == 0:
8              divisors += 1
9          d += 1
10     if divisors == 2:
11         print(n)
12         nb -= 1
13     n += 1
```

Divisions de problèmes

- Il faut diviser le problème en sous-problèmes simples à comprendre.
- Afficher les 10 premiers nombres premiers.
  - tester si un nombre est diviseur d'un autre.
  - tester si un nombre est premier.
  - afficher les `n` premiers nombres premiers.

Divisions de problèmes

- Tester si un nombre est diviseur d'un autre.

```
1 def isDivisor(d, n):
2     return n % d == 0
```

Divisions de problèmes

- Tester si un nombre est premier.

```
1  # n strictement supérieur à 1 (1 n'est jamais premier)
2  def isPrime(n):
3      d = 2
4      while not isDivisor(d, n):
5          d += 1
6      return d == n
```

Divisions de problèmes

- Afficher les `n` premiers nombres premiers.

```
1  def printPrimes(nb):
2      n = 2
3      while nb > 0:
4          if isPrime(n):
5              print(n)
6              nb -= 1
7          n += 1
```

Divisions de problèmes

- Programme complet.

```
1  def isDivisor(d, n):
2      return n % d == 0
3
4  # n strictement supérieur à 1 (1 n'est jamais premier)
5  def isPrime(n):
6      d = 2
7      while not isDivisor(d, n):
8          d += 1
9      return d == n
10
11 def printPrimes(nb):
12     n = 2
13     while nb > 0:
14         if isPrime(n):
15             print(n)
16             nb -= 1
17         n += 1
18
19 printPrimes(10)
```

Récursion

Fonctions récursives

- Une fonction peut s'appeler elle-même.
  - Exemple : somme des `n` premiers naturels.

```
1  def sum(n):
2      result = 0
3      while n > 0:
4          result += n
5          n -= 1
6      return result
```

  - En récursif :

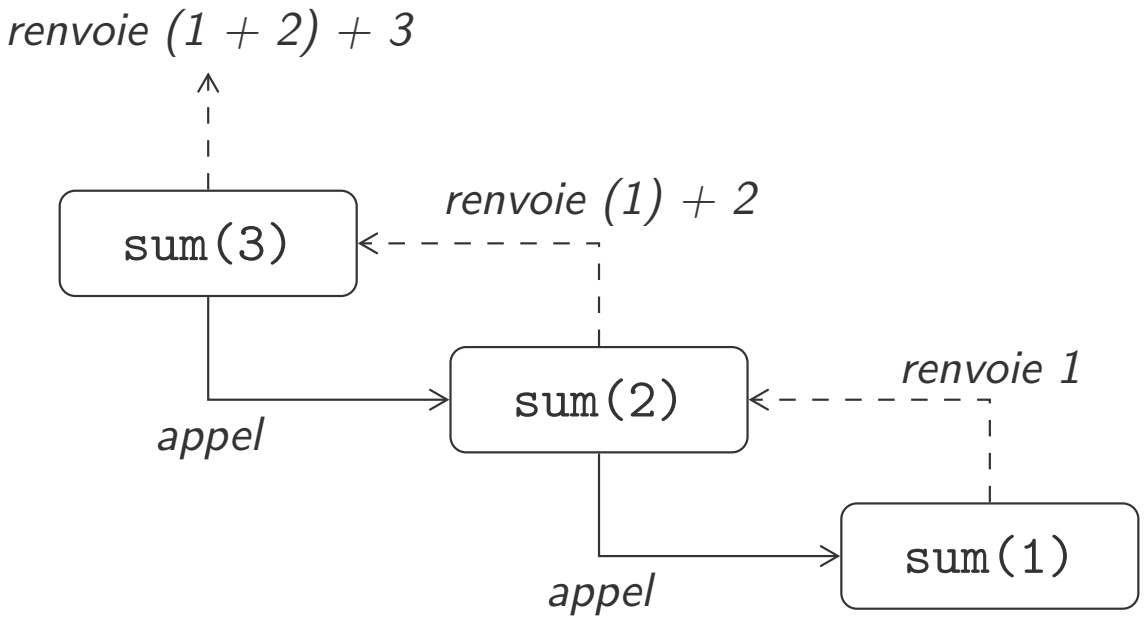
```
1  def sum(n):
2      if n == 1:
3          return 1
4      return sum(n - 1) + n
```

Fonctions récursives

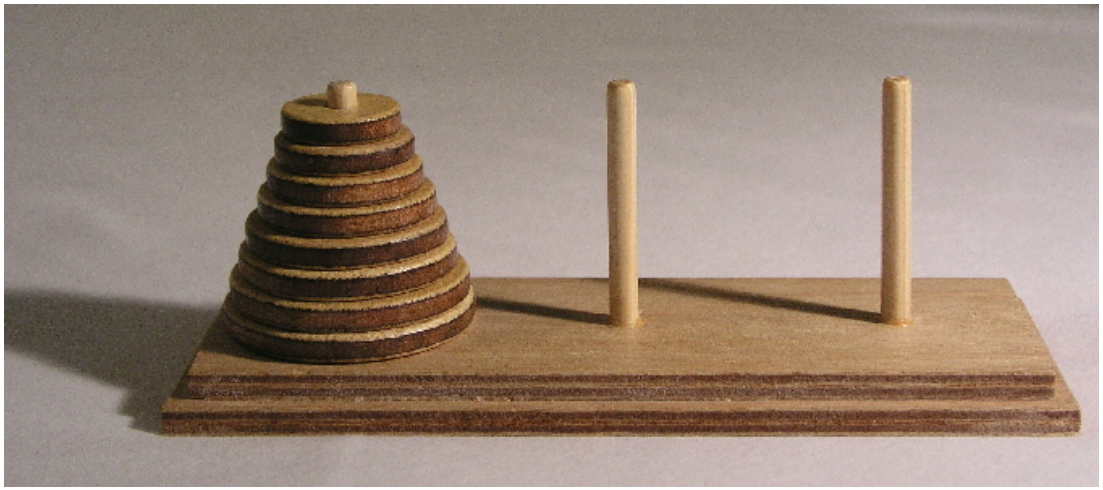
- Attention aux boucles d'appels infinies !
- Il faut toujours prévoir un cas de base.

```
1  def sum(n):
2      if n == 1:                  # cas de base
3          return 1
4      return sum(n - 1) + n       # cas récursif
```

Fonctions récursives



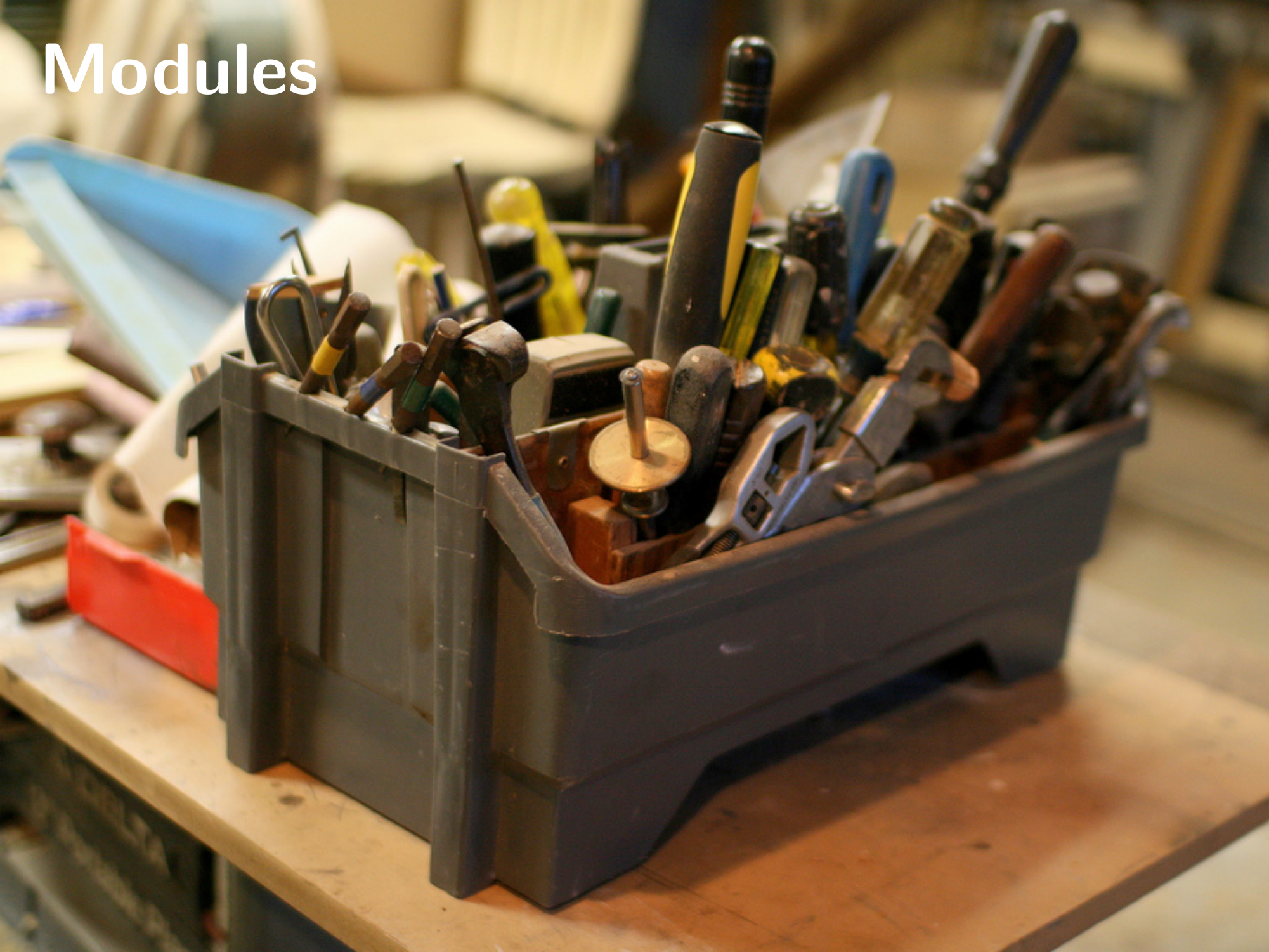
Tours de Hanoï



- Programme récursif très simple :

```
1  def hanoi(n, start, mid, end):
2      if n > 1:
3          hanoi(n-1, start, end, mid)
4          hanoi(1, start, mid, end)
5          hanoi(n-1, mid, start, end)
6      else:
7          print(start, '-->', end)
8
9  hanoi(int(input('combien de disques? ')), 'A', 'B', 'C')
```

Modules



Utilisation d'un module

- Importation d'un module :

```
1 import turtle
2
3 turtle.forward(90)
4 turtle.done()
```

- Importation des fonctions d'un module :

```
1 from turtle import forward, done
2
3 forward(90)
4 done()
```

- Pour importer toutes les fonctions, on utilise le $*$:

```
1 from turtle import *
```

Définition d'un module

- Pour définir un module, il suffit de créer un fichier `.py` contenant des définitions de fonctions.
- Exemple : définissons le module `shape` dans le fichier `shape.py`.

```
1  from turtle import *
2
3  def polygon(nbsides, side, col='black'):
4      color(col)
5      angle = 360 / nbsides
6      i = 0
7      while i < nbsides:
8          forward(side)
9          left(angle)
10         i += 1
11
12 def square(side, col='black'):
13     polygon(4, side, col)
```

- On constate ici qu'on peut importer un module dans un autre.

Définition d'un module

- On peut ensuite écrire le programme suivant :

```
1  from shape import *
2
3  square(90)
4  polygon(6, 90, 'red')
5  polygon(10, 90, 'blue')
6
7  done()
```

Séquences

Séquences et listes

- Une chaine de caractères est une séquence non-modifiable.
- Une liste est une séquence permettant de stocker tous types de valeurs.

```
1 numbers = [1, 2, 3, 4, 5]
2 words = ["I", "love", "Python"]
3 mixed = [1, "dusk-billed platypus", True]
```

- Tout ce qu'on a vu avec les `string` peut être fait avec les listes.

Listes

- L'opérateur `+` permet de concatener des listes.
- L'opérateur `*` permet de concatener plusieurs fois la même liste.
- La fonction `len()` donne la longueur de la liste.
- Le *slicing* `[m:n]` crée une nouvelle liste contenant la tranche indiquée.

Une liste est modifiable

- Contrairement au `string`, une liste est modifiable.

```
1  words = ["I", "love", "Python"]
2  words[1] = "like"
3  print(words)                  # affiche ['I', 'like', 'Python']
```

- On peut supprimer un élément d'une liste.

```
1  numbers = [1, 2, 2, 3, 4, 5]
2  del(numbers[1])
3  print(numbers)                # affiche [1, 2, 3, 4, 5]
```

Une liste est modifiable

- Les *slicing* permettent de modifier une tranche d'un coup.

```
1  numbers = [1, 2, 3, 4, 5]
2  numbers[0:0] = [0]        # ???
3  numbers[6:6] = [6]        # ???
4  numbers[3:4] = [42]       # ???
5  numbers[3:7] = [3, 4]     # ???
6  numbers[3:] = []          # ???
7  del(numbers[1:])          # ???
```

Appartenance à une liste

- Pour tester si un élément appartient à une liste, on peut utiliser l'opérateur `in`.

```
1 words = ['I', 'like', 'Python']
2 print('like' in words)       # ???
3 print('love' not in words)   # ???
```

Boucler dans une liste

- Pour parcourir les éléments d'une liste, on peut utiliser une boucle `while`.

```
1  numbers = [2, 5, 7, 2, 1]
2  sum = 0
3
4  i = 0
5  while i < len(numbers):
6      sum += numbers[i]
7
8  print(sum)
```

- On peut aussi utiliser la boucle `for ... in`.

```
1  numbers = [2, 5, 7, 2, 1]
2  sum = 0
3
4  for n in numbers:
5      sum += n
6
7  print(sum)
```

Attention aux copies !

- La copie d'une liste n'est pas aussi simple qu'il n'y parait.

```
1 words = ['I', 'like', 'Python']
2 mots = words
3 mots[1] = 'love'
4 print(words)          # ???
```

- Un *slicing* permet cependant de créer une vraie copie facilement.

```
1 words = ['I', 'like', 'Python']
2 mots = words[:]
3 mots[1] = 'love'
4 print(words)          # ???
```

Crédits

- https ://www.flickr.com/photos/biketourist/135979696/
- https ://www.flickr.com/photos/moggsterb/4538276797/
- https ://upload.wikimedia.org/wikipedia/commons/0/07/Tower_of_Hanoi.jpeg
- https ://www.flickr.com/photos/22261399@N03/2144971881/
- https ://www.flickr.com/photos/balintfoeldesi/11753707934