

Informatique 1
Introduction à la programmation

Mission 5 : restructuration

Kim Mens Charles Pecheur Cristel Pelsser

Agenda

- Restructuration
  - Spécifications et Tests
  - Tuples
  - Algorithmes de recherche

Tuples

Tuples

Tuples ne sont **pas** mutables

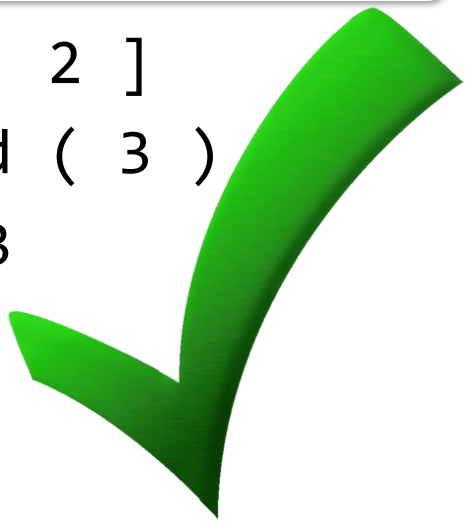
```
t = ( 1, 2 )
t.append ( 3 )
t[0] = 3
```



Listes

Listes sont mutables

```
l = [ 1, 2 ]
l.append ( 3 )
l[0] = 3
```

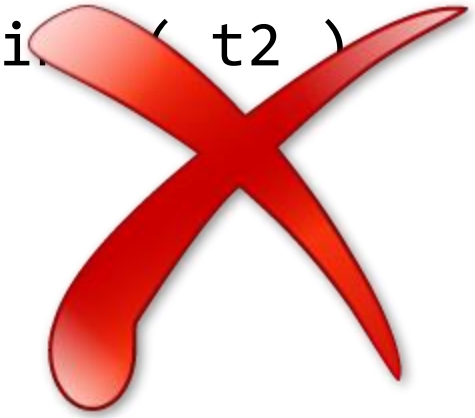


Tuples

```
t = ( 2, 4 )
t2 = ( 2, 3 ) + t
print ( t2 )
```

Tuples

```
t = ( 2 )
t2 = ( 2, 3 ) + t
pri   ( t2 )
```



TypeError: can only concatenate tuple (not "int") to tuple

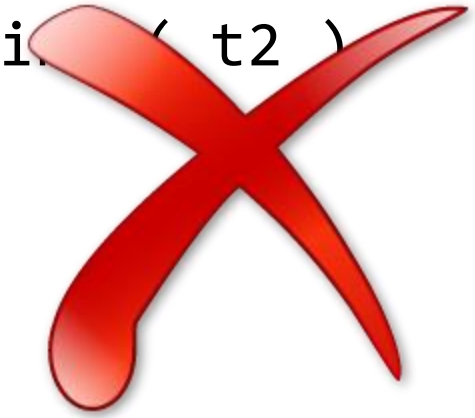
Tuples

```
t = ( 2, )
t2 = ( 2, 3 ) + t
print ( t2 )

(2,3,2)
```

Tuples

```
t = [ 2 ]
t2 = ( 2, 3 ) + t
pri   ( t2 )
```



TypeError: can only concatenate tuple (not "list") to tuple

Tuples

```
t = [ 2 ]
t2 = ( 2, 3 ) + tuple(t)
print ( t2 )
```

Tuples

```
l = [(1,2),(3,4),(5,6)]
for t in l:
    print (t[0],t[1])
```

```
l = [(1,2),(3,4),(5,6)]
for x,y in l:
    print (x,y)
```

Qu'est-ce qu'un **test unitaire** ?

- ***Comment tester son code?***
- Le principe :
  - Chaque test unitaire vérifie automatiquement qu'une petite partie du programme (fonction, méthode) se comporte comme prévu.
- Idée clé :
  1. On appelle la function/méthode avec des valeurs connues.
  2. On compare le résultat attendu et le résultat obtenu.
  3. Si ça ne correspond pas → le test échoue.
- Pourquoi faire ?
  - Détecter les erreurs tôt.
  - Éviter de casser du code déjà correct.
  - Avoir confiance dans la qualitédu programme.

Tests unitaires avec `assert`

- Même si des frameworks spécialisés pour les tests unitaires existent :
  - untittest
  - pytest
- Pour faire simple nous utiliserons que des simples **`assert`**
- Un test unitaire par “unité” dans votre programme (p.ex. une fonction)
- Chaque test unitaire est implémenté par un ensemble de **`assert`**
- Pour vérifier le comportement correct de la fonction dans plusieurs cas de figure

Exemple simple avec `assert`

- Exemple d'un test unitaire pour une fonction addition :

```
def ajout(a, b):
        return a + b
```

- Test unitaire = un ensemble d'assertions pour cette fonction :

```
assert ajout(2,3)==5, "Échec: 2+3  doit donner 5"
assert ajout(-1,1)==0,"Échec: -1+1 doit donner 0"
assert ajout(0,0)==0, "Échec: 0+0  doit donner 0"
print("Tous les tests passés avec succès!")
```

- Si tous les `assert` sont vrais → le programme continue.
- Si un test échoue → Python affiche le message d'erreur correspondant.
- `assert` signifie : Je m'attends à ce que cette condition soit vraie.

Raisonner comme un testeur

- Tester, c’est prévoir :
  - Les cas normaux → usage attendu.
  - Les cas limites → valeurs extrêmes, 0, négatifs, etc.
  - Les cas erronés → entrées invalides si pertinent.
- Bon réflexe :
  - Relancer les tests après chaque modification du code.
  - Les tests servent de filet de sécurité contre les régressions.



Spécifications et Tests

```
def ordre ( l ):
  """ Pre:
         l: une liste de nombres entiers
      Post:
         Un booléen
         True si la liste est ordonnée
         False sinon

      Une liste est ordonnée
      - si l[i] <= l[i+1] pour tout i
      - ou si l[i] >= l[i+1] pour tout i
      Par exemple, [0,1,1,2] est ordonnée,
      [2,1,1,0] est ordonnée, mais
      [2,0,1,1] n'est pas ordonnée.
  """
```

Spécifications et Tests

```
def test_ordre ():

  assert ordre ( [ 0, 1, 1, 2 ] ) == True
  assert ordre ( [ 2, 1, 1, 0 ] ) == True
  assert ordre ( [ 2, 0, 1, 1 ] ) == False
```

Spécifications et Tests

```
def test_ordre ():

  assert ordre ( [ 0, 1, 1, 2 ] ) == True
  assert ordre ( [ 2, 1, 1, 0 ] ) == True
  assert ordre ( [ 2, 0, 1, 1 ] ) == False
  assert ordre ( [ 1, 1 ] ) == True
```

Spécifications et Tests

- "Pseudo-code": un algorithme en langage *presque naturel*, sans référence à un langage de programmation en particulier.

```
fonction ordre ( l )
  Si l'ordre de l est croissant, retourner Vrai
  Si l'ordre de l est décroissant, retourner Vrai
  retourner Faux
Fin de fonction
```

```
fonction est croissant ( l )
  pour tout index i dans l
    si l[i] > l[i+1], retourner Faux
  Fin pour
  retourner Vrai
Fin de fonction
```

Spécifications et Tests

```
def est_croissant ( l ):
    for i in range(len(l)):
        if l[i] > l[i+1]:
            return False
    return True

def est_decroissant ( l ):
    for i in range(len(l)):
        if l[i] < l[i+1]:
            return False
    return True

def ordre ( l ):
    if est_croissant ( l ):
        return True
    if est_decroissant ( l ):
        return True
    return False
```

(1) séquence (indice)

(2) Condition de traitement

(3) Corps de la boucle

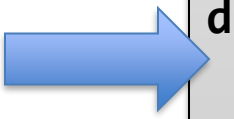
Spécifications et Tests

```
def est_croissant ( l ):
    for i in range(len(l)-1):
        if l[i] > l[i+1]:
            return False
    return True

def est_decroissant ( l ):
    for i in range(len(l)-1):
        if l[i] < l[i+1]:
            return False
    return True

def ordre ( l ):
    if est_croissant ( l ):
        return True
    if est_decroissant ( l ):
        return True
    return False
```

Spécifications et Tests



```
def est_croissant ( l ):
    while len(l) >= 2 and l[0] <= l[1]:
      del l[0]
    return len(l) <= 1

def est_decroissant ( l ):
    while len(l) >= 2 and l[0] >= l[1]:
      del l[0]
    return len(l) <= 1

def ordre ( l ):
    if est_croissant ( l ):
        return True
    if est_decroissant ( l ):
        return True
    return False
```

```
assert ordre ( [ 0, 1, 1, 2 ] ) == True
```

Spécifications et Tests

```
def est_croissant ( l ):
    while len(l) >= 2 and l[0] <= l[1]:
      del l[0]
    return len(l) <= 1
```

```
l = [ 0, 1, 1, 2 ]

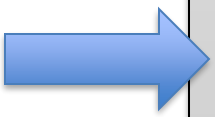
l = [ 1, 1, 2 ]

l = [ 1, 2 ]

l = [ 2 ]

return True
```

Spécifications et Tests



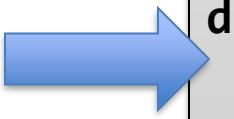
```
def est_croissant ( l ):
    while len(l) >= 2 and l[0] <= l[1]:
      del l[0]
    return len(l) <= 1

def est_decroissant ( l ):
    while len(l) >= 2 and l[0] >= l[1]:
      del l[0]
    return len(l) <= 1

def ordre ( l ):
    if est_croissant ( l ):
        return True
    if est_decroissant ( l ):
        return True
    return False
```

```
assert ordre ( [ 0, 1, 1, 2 ] ) == True
```

Spécifications et Tests



```
def est_croissant ( l ):
    while len(l) >= 2 and l[0] <= l[1]:
      del l[0]
    return len(l) <= 1

def est_decroissant ( l ):
    while len(l) >= 2 and l[0] >= l[1]:
      del l[0]
    return len(l) <= 1

def ordre ( l ):
    if est_croissant ( l ):
        return True
    if est_decroissant ( l ):
        return True
    return False
```

```
assert ordre ( [ 0, 1, 2, 3, 2 ] ) == False
```

Spécifications et Tests

```
def est_croissant ( l ):
    while len(l) >= 2 and l[0] <= l[1]:
      del l[0]
    return len(l) <= 1
```

```
l = [ 0, 1, 2, 3, 2 ]

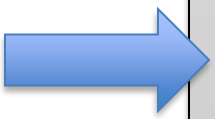
l = [ 1, 2, 3, 2 ]

l = [ 2, 3, 2 ]

l = [ 3, 2 ]

return False
```

Spécifications et Tests



```
def est_croissant ( l ):
    while len(l) >= 2 and l[0] <= l[1]:
      del l[0]
    return len(l) <= 1

def est_decroissant ( l ):
    while len(l) >= 2 and l[0] >= l[1]:
      del l[0]
    return len(l) <= 1

def ordre ( l ):
    if est_croissant ( l ):
        return True
    if est_decroissant ( l ):
        return True
    return False
```

```
assert ordre ( [ 0, 1, 2, 3, 2 ] ) == False
```

Spécifications et Tests

```
def est_croissant ( l ):
    while len(l) >= 2 and l[0] <= l[1]:
      del l[0]
    return len(l) <= 1

def est_decroissant ( l ):
    while len(l) >= 2 and l[0] >= l[1]:
      del l[0]
    return len(l) <= 1

def ordre ( l ):
    if est_croissant ( l ):
        return True
    if est_decroissant ( l ):
        return True
    return False
```

l = [3,2]

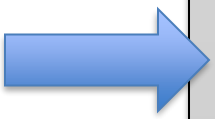
```
assert ordre ( [ 0, 1, 2, 3, 2 ] ) == False
```

Spécifications et Tests

```
def est_croissant ( l ):
    while len(l) >= 2 and l[0] <= l[1]:
      del l[0]
    return len(l) <= 1

def est_decroissant ( l ):
    while len(l) >= 2 and l[0] >= l[1]:
      del l[0]
    return len(l) <= 1

def ordre ( l ):
    if est_croissant ( l ):
        return True
    if est_decroissant ( l ):
        return True
    return False
```



```
assert ordre ( [ 0, 1, 2, 3, 2 ] ) == False
```

AssertionError

Spécifications et Tests

Sauf si explicitement spécifié, une fonction NE DOIT JAMAIS modifier une liste passée en argument

Autres Critères de Qualité

Pas de longues fonctions

N'appelez pas une variable

- `'list'`
- `'tuple'`
- `'str'`
- `'int'`
- `'float'`

Recherche Linéaire

```
def est_croissant ( l ):
    for i in range(len(l)-1):
        if l[i] > l[i+1]:
            return False
    return True
```

```
def est_dans( c, l ):
    for car in l:
        if car == c:
            return True
    return False
```

```
    [Boucle for]
        if [Condition]:
            return [Résultat]
    return [Pas de résultat]
```

Recherche Linéaire

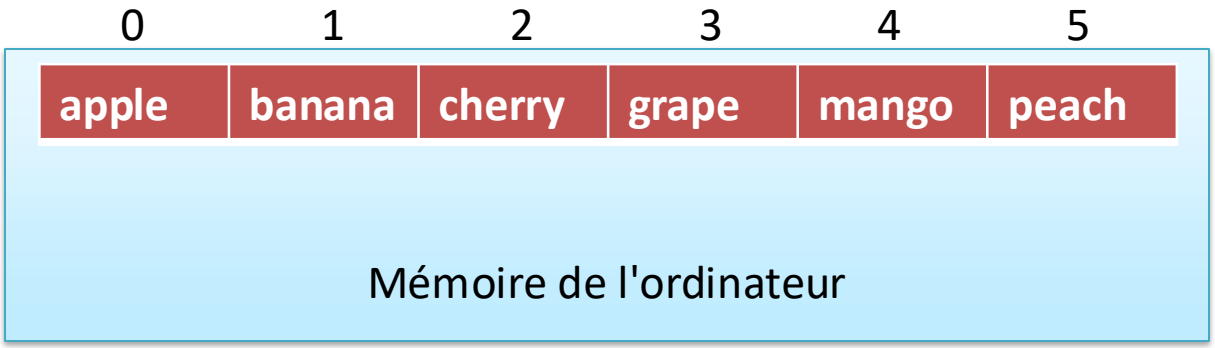
```
def est_dans( c, l ):
    for car in l:
        if car == c:
            return True
        else:
            return False
    return False
```



Recherche dichotomique

```
def binary_search ( name, list_of_names ):
  first = 0
  last = len(list_of_names)-1
  found = False
  while first<=last and not found:
    middle = (first + last)//2
    if list_of_names[middle] == name:
      found = True
    else:
      if name < list_of_names[middle]:
        last = middle-1
      else:
        first = middle+1
  return found
```

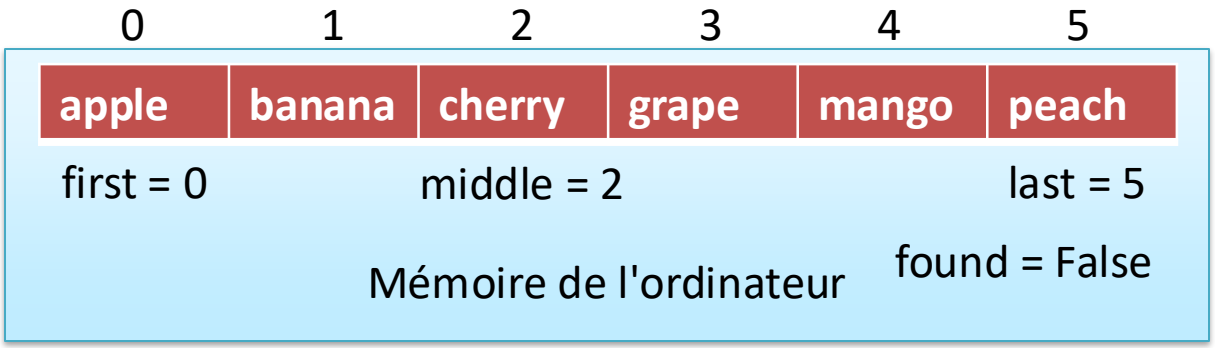
Recherche dichotomique



Cherchons "banana"

```
def binary_search ( name, list_of_names ):
  first = 0
  last = len(list_of_names)-1
  found = False
  while first<=last and not found:
    middle = (first + last)//2
    if list_of_names[middle] == name:
      found = True
    else:
      if name < list_of_names[middle]:
        last = middle-1
      else:
        first = middle+1
  return found
```

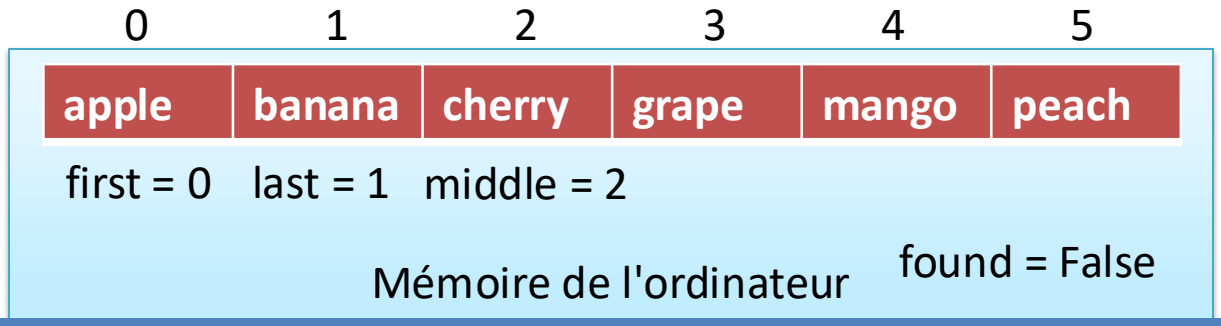
Recherche dichotomique



Cherchons "banana"

```
def binary_search ( name, list_of_names ):
  first = 0
  last = len(list_of_names)-1
  found = False
  while first<=last and not found:
    middle = (first + last)//2
    if list_of_names[middle] == name:
      found = True
    else:
      if name < list_of_names[middle]:
        last = middle-1
      else:
        first = middle+1
  return found
```

Recherche dichotomique



Cherchons "banana"

Utiliser un debugger pour suivre l'évolution des variables

```
def binary_search ( name, list_of_names ):
  first = 0
  last = len(list_of_names)-1
  found = False
  while first<=last and not found:
    middle = (first + last)//2
    if list_of_names[middle] == name:
      found = True
    else:
      if name < list_of_names[middle]:
        last = middle-1
      else:
        first = middle+1
  return found
```