

B201A Informatique appliquée

Séance 2

Tuple et objet



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

■ Ensemble

- Collection non ordonnée d'éléments distincts
- Opérations ensemblistes
- Modification et ensemble non modifiable

■ Dictionnaire

- Ensemble de paires clé-valeur, avec clés distinctes
- Opérations d'accès et de modification
- Base de données et format JSON

Objectifs

- Utilisation de **tuples**
 - Définition et utilisation
 - Fonction avec plusieurs valeurs de retour
 - Tuple nommé
- Introduction aux **objets**
 - Création et utilisation d'objets
 - Appel de méthode
 - Programmation orientée objet



10/2/20

- Paul

- Sol

- Creme

- arciufamula

- perevo

- mecelivella

- battue

Tuple

Tuple

- Séquence ordonnée et **non modifiable** d'éléments

Les éléments d'un tuple ne peuvent pas être modifiés

- **Définition** d'un tuple avec ()

Parenthèses pas obligatoires lorsqu'au moins un élément

```
1 # Tuple vide
2 a = ()
3
4 # Tuples contenant un seul élément
5 b = 1,
6 c = (1,)
7
8 # Tuples contenant trois éléments
9 d = 1, 2, 3
10 e = (1, 2, 3)
```

Somme des éléments d'un tuple

- **Accès aux éléments** d'un tuple avec les crochets

En lecture seulement, l'accès en écriture est interdit

- **Utilisation** d'un tuple similaire à celle des listes

- Taille d'un tuple obtenue avec la fonction `len`
- Parcours d'un tuple avec `while` ou `for`

```
1 def sum(values): | def sum(values):
2     s = 0         |     s = 0
3     i = 0         |     for element in values:
4     while i < len(values): |         s += element
5         s += values[i]    |         return s
6         i += 1           |
7     return s      |
```

Définition avec parenthèses

- Définition **avec parenthèses** d'un tuple parfois obligatoire
 - Pour le tuple vide
 - Lors d'un appel de fonction
- L'oubli de parenthèses provoque une **ambigüité**

Elle est relevée par le compilateur par une erreur

```
1 result = sum(1, 2, 3)
2
3 # Solution correcte :
4 # result = sum((1, 2, 3))
```

```
Traceback (most recent call last):
  File "program.py", line 9, in
    r = sum(1, 2, 3)
TypeError: sum() takes 1 positional argument but 3 were given
```

Emballage et déballage

- On peut **emballer** plusieurs valeurs dans un tuple

Elles sont toutes accessibles à partir d'une seule variable

- On peut **déballer** un tuple dans plusieurs variables

Chaque variable reçoit la valeur d'un élément du tuple

```
1 t = 1, 2, 3
2 print(t)
3
4 a, b, c = t
5 print(a, b, c, sep=' / ')
```

```
(1, 2, 3)
1/2/3
```

Plusieurs valeurs de retour

- Une fonction peut **renvoyer plusieurs valeurs**

Il suffit en fait de renvoyer un tuple

- Fonction qui **cherche un élément** dans une liste
 - Si l'élément est trouvé, renvoie True et son indice
 - Sinon renvoie False et None comme indice

```
1 def find(list, element):
2     i = 0
3     while i < len(list):
4         if list[i] == element:           # L'élément est trouvé
5             return True, i
6             i += 1
7     return False, None                   # L'élément n'a pas été trouvé
```

Récupération des valeurs de retour

- Deux solutions pour récupérer les valeurs de retour
 - Sous forme d'une variable contenant un tuple
 - Avec autant de variables qu'il y a d'éléments dans le tuple

```
1 values = [1, 2, 3, 4]
2
3 result = find(values, 2)
4 found, index = find(values, 6)
5
6 print(result, found, index)
```

```
(True, 1) False None
```

Absence de valeur

- La valeur spéciale None représente l'absence de valeur

A permis d'avoir des valeurs de retour homogènes pour find

- Ou alors on vérifie le type de la valeur de retour

```
1 def find(list, element):
2     i = 0
3     while i < len(list):
4         if list[i] == element:           # L'élément est trouvé
5             return True, i
6         i += 1
7     return False                         # L'élément n'a pas été trouvé
8
9 values = [1, 2, 3, 4]
10 result = find(values, 2)                # result peut être
11                                           # de différents type
12 if type(result) is tuple:
13     print(result[0], result[1])
14 else:
15     print(result)
```

Opérateur de déballage

- Deux manières de **déballer** un tuple
 - Avec l'opérateur d'affectation
 - Lors d'un appel de fonction
- **Opérateur de déballage** (*) lors d'un appel de fonction

Déballer le tuple dans les paramètres de la fonction appelée

```
1 def max(a, b, c):
2     if a > b and a > c:
3         return a
4     elif b > c:
5         return b
6     return c
7
8 t = 1, 2, 3
9 result = max(*t)
```

Affectation multiple

- Changer la valeur de **plusieurs variables** en une fois

Combinaison d'emballage puis de déballage

- Permet d'**échanger** les valeurs de deux variables

En une seule opération, sans variable intermédiaire

```
1 x, y = "Hello", 42
2 print(x, y)
3
4 x, y = y, x
5 print(x, y)
```

```
Hello 42
42 Hello
```

Tuple nommé

- Attribuer un **nom** à chacun des éléments d'un tuple

Et permettre d'accéder aux éléments à partir du nom

- Création d'un **tuple nommé** en deux phases

1 Définition d'un nouveau type de tuple nommé

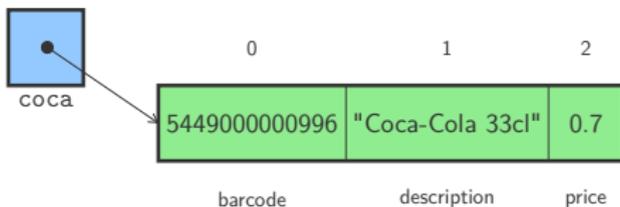
2 Création du tuple nommé

```
1 from collections import namedtuple
2
3 Item = namedtuple('Item', ['barcode', 'description', 'price'])
4
5 coca = Item(5449000000996, "Coca-Cola 33cl", 0.70)
```

Accès aux champs

- Accès aux champs avec l'opérateur d'accès (.)
- Un tuple nommé est un **tuple enrichi**

On peut utiliser les opérations applicables au tuples



```
1 print(len(coca))           # 3
2 print(coca[1])            # Coca-Cola 33cl
3 print(coca[1:3])          # ('Coca-Cola 33cl', 0.7)
4
5 print(coca.price)         # 0.7
```

Vecteur dans le plan (1)

- Définitions de fonctions qui agissent sur un vecteur

Représentation textuelle et calcul de la norme

```
1 from math import sqrt
2 from collections import namedtuple
3
4 Vector = namedtuple('Vector', ['x', 'y'])
5
6 def vectostr(v):
7     return "(" + str(v.x) + ", " + str(v.y) + ")"
8
9 def norm(v):
10    return sqrt(v.x ** 2 + v.y ** 2)
11
12 u = Vector(1, 1)
13 print(vectostr(u))
14 print(norm(u))
```

Tuple nommé vs dictionnaire

- Éléments d'un tuple nommé **ordonnés** car séquence

Un dictionnaire est un ensemble de paires clé-valeur

- Tuple nommé **non modifiable**

On peut modifier et ajouter des valeurs à un dictionnaire

- Champs d'un tuple nommé sont des **chaines de caractères**

Les clés d'un dictionnaire sont des valeurs non modifiables

Immuabilité (1)

- Un tuple est **immuable**

Son état n'est pas modifiable une fois initialisé

- Plusieurs **avantages** à l'immuabilité
 - Peut être élément d'un ensemble ou clé d'un dictionnaire
 - Peut être passé à une fonction « *sans risque* »
 - Pas besoin d'en faire des copies profondes
- Concept-clé de la **programmation orientée objet**

Immuabilité (2)

- Différence entre **liste** et tuple

Les valeurs d'une liste peuvent être modifiées

```
1 def thug(data):
2     for i in range(len(data)):
3         data[i] = 0
4
5 d = [1, 2, 3]
6 print(d)
7 thug(d)
8 print(d)
```

```
Avant appel : [1, 2, 3]
Après appel : [0, 0, 0]
```

Immuabilité (3)

- Différence entre liste et **tuple**

Les valeurs d'un tuple ne peuvent pas être modifiées

```
1 d = (1, 2, 3)
2 print(d)
3 thug(d)
4 print(d)
```

```
Avant appel : (1, 2, 3)
Traceback (most recent call last):
  File "program.py", line 7, in <module>
    thug(d)
  File "program.py", line 3, in thug
    data[i] = 0
TypeError: 'tuple' object does not support item assignment
```

Objet



Objet (1)

- Un **objet** combine des données et des fonctions

Les fonctions ont accès complet aux données de l'objet

- Permet de définir des **types de données** complexes

On a déjà rencontré les listes, chaînes de caractères, ensembles...

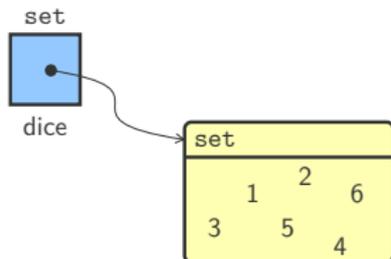
```
1 dice = {1, 2, 3, 4, 5, 6}           # Initialisation des données
2 face = dice.pop()                  # Appel d'une fonction
3
4 print("La face visible du dé est :", face)
```

```
La face visible du dé est : 1
```

Objet (2)

- **Trois éléments** existent lorsqu'on crée un objet
 - L'**objet**, avec ses attributs, se trouve en mémoire
 - Une **variable** du même type que l'objet est déclarée
 - Une **référence** vers l'objet est stockée dans la variable

```
1 dice = {1, 2, 3, 4, 5, 6}
```

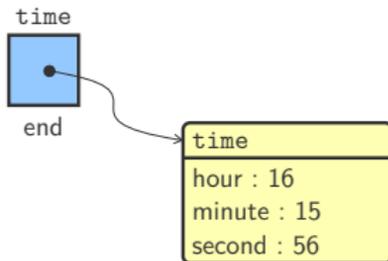
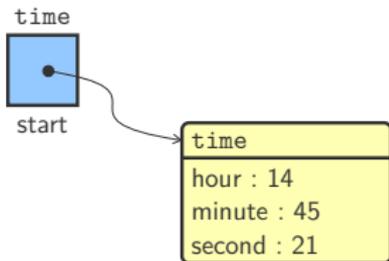


Création

- Création d'un objet en exécutant un **constructeur**

Permet d'initialiser les attributs de l'objet

```
1 from datetime import time
2
3 start = time(14, 45, 21)
4 end = time(16, 15, 56)
```



Des objets partout...

Person



Food



TransportationMean



Accès aux attributs

- **Accès aux attributs** d'un objet avec l'opérateur d'accès (.)

Exactement comme avec les tuples nommés

- L'accès peut se faire en **lecture et/ou écriture**

Certains attributs sont protégés et en lecture seule

```
1 startseconds = 3600 * start.hour + 60 * start.minute + start.second
2 endseconds = 3600 * end.hour + 60 * end.minute + end.second
3
4 difference = endseconds - startseconds
5 print("Le cours va durer :", difference, "secondes")
```

Attribut en lecture seule

- Erreur d'exécution si **modification attribut lecture seule**

Modification d'un attribut faite par affectation

```
1 from datetime import time
2
3 start = time(14, 45, 21)
4 start.minute = 15
```

```
Traceback (most recent call last):
  File "program.py", line 4, in <module>
    start.minute = 15
AttributeError: attribute 'minute' of 'datetime.time' objects is
not writable
```

Paramètre de type objet

- Une fonction peut recevoir des **paramètres de type objet**

Le paramètre reçoit une copie de la référence vers l'objet

```
1 from datetime import time
2
3 def toseconds(t):
4     return 3600 * t.hour + 60 * t.minute + t.second
5
6
7 start = time(14, 45, 21)
8 end = time(16, 15, 56)
9
10 difference = toseconds(end) - toseconds(start)
11 print("Le cours va durer :", difference, "secondes")
```

```
Le cours va durer : 5435 secondes
```

Valeur de retour de type objet

- Une fonction peut renvoyer un objet

La fonction crée l'objet et renvoie une référence vers ce dernier

```
1 from datetime import time
2
3 def theoreticalend(start, duration):
4     minute = start.minute + (duration % 60)
5     hour = start.hour + (duration // 60) + (minute // 60)
6     return time(hour, minute % 60, start.second)
7
8
9 start = time(14, 45, 21)
10 print("Le cours devrait finir à :", theoreticalend(start, 90))
```

```
Le cours devrait finir à : 16:15:21
```

Appel de méthode (1)

- Une fonction associée à un objet est appelée une **méthode**

Une méthode est appelée sur un objet cible

```
1 from calendar import TextCalendar
2
3 cal = TextCalendar()
4 cal.prmonth(2015, 9)           # Affiche le calendrier de septembre 2015
```

```
September 2015
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
  7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
```

Appel de méthode (2)

- Méthode appelée avec l'opérateur d'appel de méthode (.)

L'objet cible est précisé avant le point

```
1 from calendar import TextCalendar
2
3 cal = TextCalendar()
4 cal.setfirstweekday(6)      # Change le premier jour de la semaine
5 cal.prmonth(2015, 9)
```

```
    September 2015
Su Mo Tu We Th Fr Sa
      1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

Vecteur dans le plan (2)

- Méthode appelée directement sur l'objet cible

On ne doit plus passer le vecteur en paramètre



```
1 # u = Vector(1, 1)
2 # print(vectostr(u))
3 # print(norm(u))
4
5 u = Vector(1, 1)
6 print(u)
7 print(u.norm())
```

Objet vs tuple nommé

- **Modification** possible des attributs d'un objet (directe ou non)

Un tuple nommé est non modifiable

- Fonctions associées aux objets, appelées **méthodes**

Fonction utilisant un tuple nommé doit le recevoir en paramètre

- **Accès** aux attributs peut être restreint aux méthodes

Tous les champs d'un tuple sont toujours accessibles

Programmation orientée objet

- La **programmation orientée objet** manipule des objets
Concepts et entités représentés par des objets
- Représenter des objets **concrets ou conceptuels** du monde réel
Une personne, un moyen de transport, une date, une liste...
- Création de **nouveaux types de données**
Permet une programmation de plus haut niveau

État d'un objet

- Chaque objet est unique et possède son propre **état**
Identité propre à chaque objet, avec ses propres attributs
- L'état d'un objet est **modifiable ou non**
 - Objet immuable aura toujours le même état
 - État d'un objet non modifiable ne peut être changé

Identité d'un objet

- **Identité** d'un objet propre à chaque objet créé

Caractérisée par son emplacement en mémoire

- Fonction prédéfinie **id** renvoie l'identité

```
1 numbers = {8, 3, 1, -2, 0}
2 letters = {'A', 'P', 'Q'}
3
4 print(id(numbers))
5 print(id(letters))
```

```
4302577224
4329799752
```

Tout est objet (1)

- En Python, toute donnée est un **objet**

Même tout ce qu'on a déjà vu comme les nombres, booléens...

- Fonction prédéfinie **dir** donne la liste des méthodes

```
1 temperature = 19
2 print(dir(temperature))
```

```
['_abs_', '_add_', '_and_', '_bool_', '_ceil_', '_class_', '_delattr_', '_dir_', '_divmod_', '_doc_', '_eq_', '_float_', '_floor_', '_floordiv_', '_format_', '_ge_', '_getattr_', '_getnewargs_', '_gt_', '_hash_', '_index_', '_init_', '_int_', '_invert_', '_le_', '_lshift_', '_lt_', '_mod_', '_mul_', '_ne_', '_neg_', '_new_', '_or_', '_pos_', '_pow_', '_radd_', '_rand_', '_rdivmod_', '_reduce_', '_reduce_ex_', '_repr_', '_rfloordiv_', '_rlshift_', '_rmod_', '_rmul_', '_ror_', '_round_', '_rpow_', '_rrshift_', '_rshift_', '_rsub_', '_rtruediv_', '_rxor_', '_setattr_', '_sizeof_', '_str_', '_sub_', '_subclasshook_', ...]
```

Tout est objet (2)

- Les nombres entiers ont une **méthode `__add__`**

Permet de faire la somme de deux entiers

- Appel à la méthode `__add__` simplifié par l'**opérateur `+`**

Redéfinition d'un opérateur rend le code plus lisible

```
1 print(temperature + 11)
2 print(temperature.__add__(11))
```

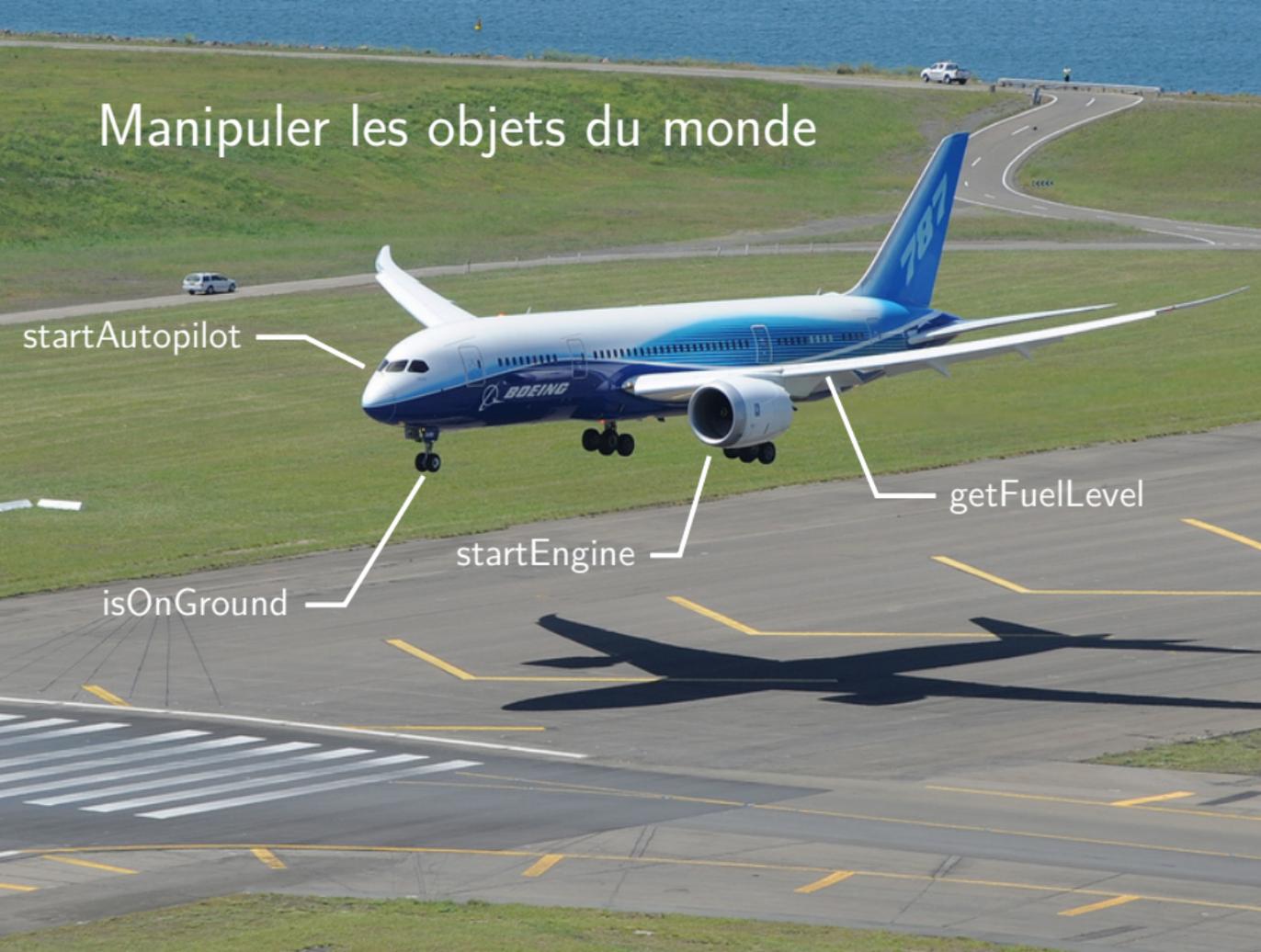
Manipuler les objets du monde

startAutopilot

isOnGround

startEngine

getFuelLevel



Exemple : les complexes

- **Nombre complexe** représenté par des objets de type `complex`

Deux façons de les créer puis on peut utiliser plusieurs méthodes

```
1 a = 2 + 3j
2 b = complex(-1, 4)
3
4 print(type(a))
5 print(a.imag)
6 print(b.conjugate())
```

```
<class 'complex'>
3.0
(-1-4j)
```

Exemple : les chaînes de caractères

- **Chaîne de caractères** représenté par des objets de type `str`

Myriade de méthodes permettant de les manipuler

```
1 s = "heLLo"  
2  
3 print(s.upper())  
4 print(s.capitalize())  
5 print(s.startswith("H"))
```

```
HELLO  
Hello  
False
```

Exemple : le navigateur web

- Module `webbrowser` permet de manipuler le **navigateur web**

Récupération du navigateur puis ouverture de pages avec `open`

```
1 import webbrowser
2
3 chrome = webbrowser.get()
4 chrome.open("http://www.ukonline.be")
```

Crédits

- <https://www.flickr.com/photos/sunshinecity/985725985>
- https://www.flickr.com/photos/madalena_pestana/2828893154
- <http://www.flickr.com/photos/jetstarairways/6769120131/>