Bases de la programmation impérative

Données, représentations

Ensimag 1ère année

Sommaire

Variable

```
1  #!/usr/bin/env python3
2  def main():
3      abscisse = 3
4      print(abscisse)
5      abscisse = 5
6      print(abscisse)
7      ordonee = abscisse
8      print(ordonee)
9
10 main()
```

Variable

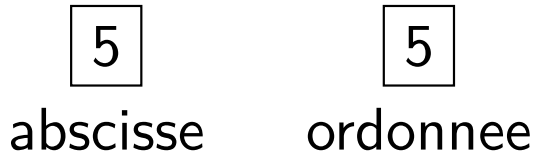
```
1  #!/usr/bin/env python3
2  def main():
3      abscisse = 3
4      print(abscisse)
5      abscisse = 5
6      print(abscisse)
7      ordonee = abscisse
8      print(ordonee)
9
10 main()
```

Variable

```
1  #!/usr/bin/env python3
2  def main():
3      abscisse = 3
4      print(abscisse)
5      abscisse = 5
6      print(abscisse)
7      ordonee = abscisse
8      print(ordonee)
9
10 main()
```

Variable

```
1  #!/usr/bin/env python3
2  def main():
3      abscisse = 3
4      print(abscisse)
5      abscisse = 5
6      print(abscisse)
7      ordonee = abscisse
8      print(ordonee)
9
10 main()
```



Types : bases

```
1  #!/usr/bin/env python3
2  def main():
3      mon_entier = 3
4      print(type(mon_entier))
5      mon_flottant = 2.5
6      print(type(mon_flottant))
7      mon_entier = mon_flottant
8      print(type(mon_entier))
9
10 main()
```

- chaque variable est typée
- certains types sont liés au matériel

Types : bases

```
1  #!/usr/bin/env python3
2  def main():
3      mon_entier = 3
4      print(type(mon_entier))
5      mon_flottant = 2.5
6      print(type(mon_flottant))
7      mon_entier = mon_flottant
8      print(type(mon_entier))
9
10 main()
```

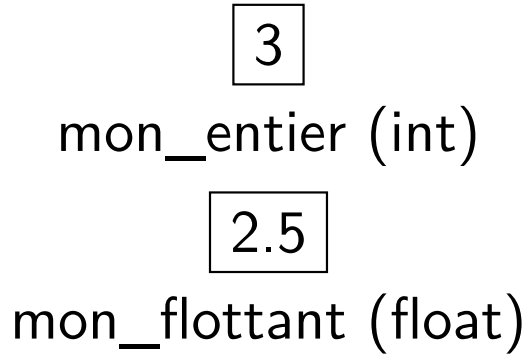
- chaque variable est typée
- certains types sont liés au matériel

mon_entier (int)

Types : bases

```
1  #!/usr/bin/env python3
2  def main():
3      mon_entier = 3
4      print(type(mon_entier))
5      mon_flottant = 2.5
6      print(type(mon_flottant))
7      mon_entier = mon_flottant
8      print(type(mon_entier))
9
10 main()
```

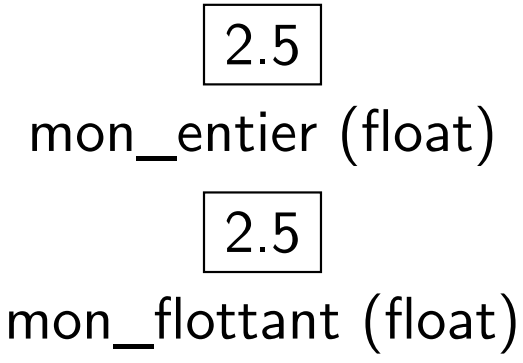
- chaque variable est typée
- certains types sont liés au matériel



Types : bases

```
1  #!/usr/bin/env python3
2  def main():
3      mon_entier = 3
4      print(type(mon_entier))
5      mon_flottant = 2.5
6      print(type(mon_flottant))
7      mon_entier = mon_flottant
8      print(type(mon_entier))
9
10 main()
```

- chaque variable est typée
- certains types sont liés au matériel



Types : bases

```
1  #!/usr/bin/env python3
2  def main():
3      chaine1 = "bonjour"
4      chaine2 = " tout le
           monde"
5      chaine3 = chaine1 +
           chaine2
6      print(chaine3)
7      chaine3 = chaine1 + 5
8      print(chaine3)
9
10 main()
```

```
bonjour tout le monde
Traceback (most recent call las
File "./var3.py", line 10, in
  <module> main()
File "./var3.py", line 7,
  in main
  chaine3 = chaine1 + 5
TypeError: Can't convert 'int'
object to str implicitly
```

Fonctions : exemple

```
1  #!/usr/bin/env python3
2  def somme(entier1, entier2):
3      return entier1 + entier2
4
5
6  def affiche(chaine):
7      print("nous affichons:", chaine)
8
9
10 def main():
11     affiche(somme(24, 18))
12
13 main()
```

Fonctions

- encapsule un bout de code : évite le code dupliqué
- permet un nommage clair d'opérations à réaliser
- arguments, passages par objets
  - copie de références
- variables locales à la fonction
- l'indentation délimite la fonction

Fonctions

- encapsule un bout de code : évite le code dupliqué
- permet un nommage clair d'opérations à réaliser
- arguments, passages par objets
  - copie de références
- variables locales à la fonction
- l'indentation délimite la fonction

Objectif du cours

Comment structurer un gros projet en fonctions ?

Variables locales

```
1  #!/usr/bin/env python3
2  def change_x():
3      x = 5
4
5  def main():
6      x = 3
7      print(x)
8      change_x()
9      print(x)
10
11 main()
```

Variables locales

```
1  #!/usr/bin/env python3
2  def change_x():
3      x = 5
4
5  def main():
6      x = 3
7      print(x)
8      change_x()
9      print(x)
10
11 main()
```

- chaque x est une variable locale à sa fonction

Variables locales

```
1  #!/usr/bin/env python3
2  def change_x():
3      x = 5
4
5  def main():
6      x = 3
7      print(x)
8      change_x()
9      print(x)
10
11 main()
```

- chaque x est une variable locale à sa fonction
- on peut éviter les confusions en procédant à un renommage en renommant localement à une fonction

Variables locales

```
1  #!/usr/bin/env python3
2  def change_x():
3      y = 5
4
5  def main():
6      x = 3
7      print(x)
8      change_x()
9      print(x)
10
11 main()
```

- chaque x est une variable locale à sa fonction
- on peut éviter les confusions en procédant à un renommage en renommant localement à une fonction

Qu'affiche ce code ?

```
1  #!/usr/bin/env python3
2  def affiche_x():
3      print(x)
4
5  def main():
6      x = 3
7      affiche_x()
8
9  main()
```

Qu'affiche ce code ?

```
1  #!/usr/bin/env python3
2  def affiche_x():
3      print(x)
4
5  def main():
6      x = 3
7      affiche_x()
8
9  main()
```

Erreur

NameError : name 'x' is not defined

Variables locales

```
1  #!/usr/bin/env python3
2  def affiche_plus(entier):
3      entier = entier + 1
4      print(entier)
5
6  def affiche_moins(entier):
7      entier = entier - 1
8      print(entier)
9
10 def main():
11     valeur = 3
12     affiche_plus(valeur)
13     affiche_moins(valeur)
14
15 main()
```

- que se passe t'il ?

Variables locales

```
1  #!/usr/bin/env python3
2  def affiche_plus(entier):
3      entier = entier + 1
4      print(entier)
5
6  def affiche_moins(entier):
7      entier = entier - 1
8      print(entier)
9
10 def main():
11     valeur = 3
12     affiche_plus(valeur)
13     affiche_moins(valeur)
14
15 main()
```

- que se passe t'il ?

Explication

- ligne 3 : l'affectation affecte un nouvel entier

Variables globales

```
1  #!/usr/bin/env python3
2  ENTIER = 5
3
4
5  def main():
6      print(ENTIER)
7
8  main()
```

- en général à éviter
  - rend le code plus difficile à lire et maintenir
  - gros problèmes pour la parallélisation

Variables globales

```
1  #!/usr/bin/env python3
2  ENTIER = 5
3
4
5  def main():
6      print(ENTIER)
7
8  main()
```

- en général à éviter
  - rend le code plus difficile à lire et maintenir
  - gros problèmes pour la parallélisation
- accepté plus facilement pour des constantes (nom en majuscule)

Écriture de variables globales

```
1  #!/usr/bin/env python3
2  ENTIER = 5
3
4
5  def main():
6      global ENTIER
7      ENTIER = 3
8
9  main()
10 print(ENTIER)
```

Sommaire

Pylint

- analyseur statique externe
- conseils sur :
  - le style
  - la documentation
  - les mauvaises pratiques (variables globales,...)
  - la simplification du code

```
>>  6 #!/usr/bin/env python3
>>  5 def main():
    4     mon_entier = 3
    3     print(type(mon_entier))
    2     mon_flottant = 2.5
    1     print(type(mon_flottant))
>>7       mon_entier = mon_flottant
    1     print(type(mon_entier))
    2
    3 main()
NORMAL  master  var2.py        python  utf-8[unix]  70% ℓN  7:  1  [Syntax: line:1 (3)]
[redefined-variable-type] Redefinition of mon_entier type from int to float
```

Pylint : utilisation

- intégrable dans les éditeurs / en ligne de commande
- une aide précieuse pour démarrer le langage
- warnings désactivables au cas par cas (à justifier)

Attention !

Il y aura une pénalité sévère lors des examens en salle machine pour les codes qui ne passent pas pylint avec une note de 10/10.

Documentation

```
1  """
2  Mon premier module. Fournit
       une fonction "increment".
3  """
4
5
6  def increment(entier):
7      """
8      Renvoie l'entier donne,
           augmente de 1.
9      """
10     return entier+1
```

- obligatoire !
- indique ce que réalise une fonction
- les pré-conditions ou post-conditions
- les éventuels effets de bord

Modules

- organisation du code en différents fichiers
- réutilisation de parties communes entre différents programmes
- exemple : importer la fonction cos du module math : `from math import cos`
- de nombreux modules open-source sont disponibles
- le partage de modules entre différents programmeurs rend nécessaire une bonne documentation

Documentation

```
1  #!/usr/bin/env python3
2  from math import cos
3
4
5  def main():
6      help(cos)
7
8  main()
```

- https://docs.python.org/3/
- accessible en ligne de commande avec pydoc3
- l'introspection permet de consulter les opérations réalisables sur des variables (help)

python 2 / 3

Attention à la version de python utilisée.

Sommaire

T-uples

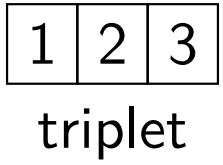
```
1  #!/usr/bin/env python3
2  """
3  Petit module de geometrie.
4  """
5  def origine():
6      """
7      Renvoie le couple de coordonnees (0, 0).
8      """
9      return (0, 0)
10
11 def main():
12     """
13     test du module.
14     """
15     print(origine())
16
17 main()
```

T-uples

- ensemble statique de plusieurs données
- initialisation : `triplet = (1, 2, 3)` ou `triplet = 1, 2, 3`
- affectation : `triplet2 = triplet`
- accès individuel aux éléments :

```
print("1er element:", triplet[0])
print("2eme element:", triplet[1])
print("3eme element:", triplet[2])
```

- lecture seule !



Affectations

```
1  #!/usr/bin/env python3
2  def main():
3      triplet = 3, 6, 9
4      print(triplet)
5      x, y, z = triplet
6      print(y, z)
7      couple = x, str(x)
8      print(couple[0], couple[1])
9
10 main()
```

Vecteurs

```
1  #!/usr/bin/env python3
2
3  def main():
4      entiers = [0, 2, 4, 6, 8]
5      print("le premier entier
           pair est", entiers[0])
6      # changeons le contenu
7      entiers[0], entiers[3] =
           1, 7
8      print(entiers)
9
10 main()
```

- list en python
- ensemble de taille variable
- éléments non statiques

1	2	4	7	8

Vecteurs

- de nombreuses opérations sont implémentées sur les vecteurs :
  - ajout d'éléments en fin,
  - fusion,
  - suppression d'éléments en fin,
  - ...
- nous les verrons plus tard, au cours du semestre

En deux dimensions

```
1  matrice = [
2        [0, 1, 2],
3        [3, 4, 5],
4        [6, 7, 8]
5  ]
```

0	1	2
3	4	5
6	7	8

En deux dimensions

```
1  matrice = [
2        [0, 1, 2],
3        [3, 4, 5],
4        [6, 7, 8]
5  ]
```

0	1	2
3	4	5
6	7	8

Quizz

Comment accéder au premier élément de la seconde colonne ?

En deux dimensions

```
1  matrice = [
2        [0, 1, 2],
3        [3, 4, 5],
4        [6, 7, 8]
5  ]
```

0	1	2
3	4	5
6	7	8

Quizz

Comment accéder au premier élément de la seconde colonne ?

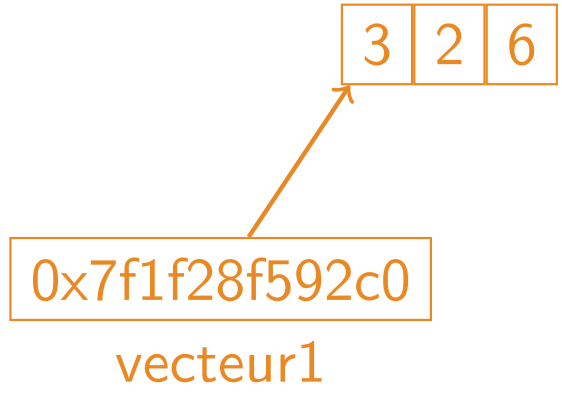
`matrice[0][1]`

En réalité

```
vecteur1 = [3, 2, 6]
vecteur2 = vecteur1
vecteur2[0] = 0
```

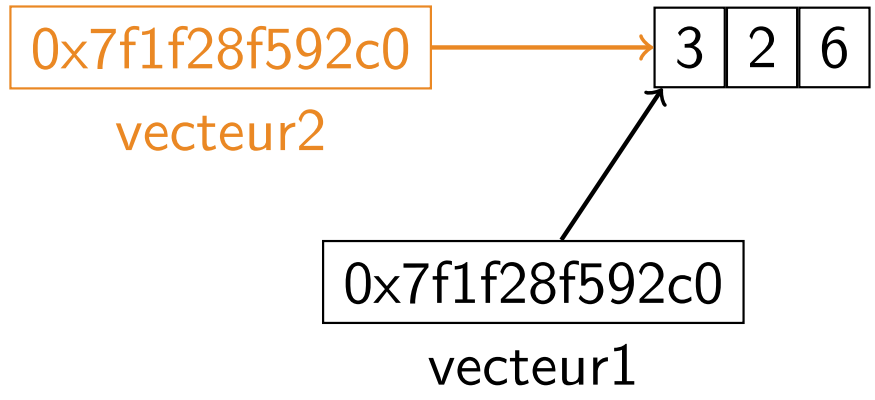
En réalité

```
vecteur1 = [3, 2, 6]
vecteur2 = vecteur1
vecteur2[0] = 0
```



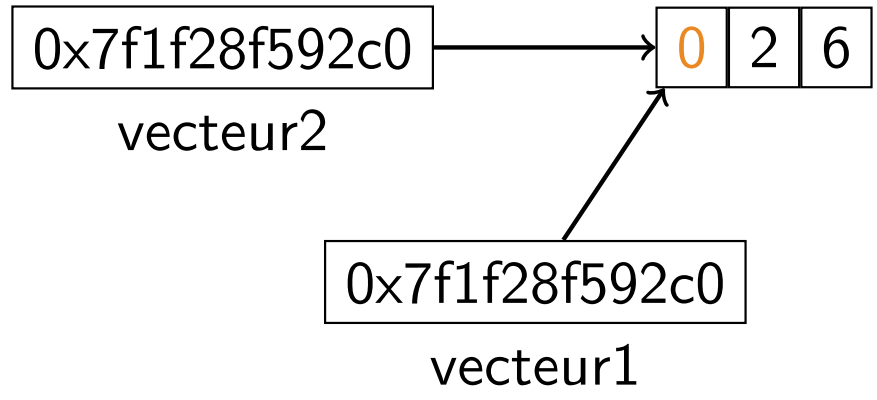
En réalité

```
vecteur1 = [3, 2, 6]
vecteur2 = vecteur1
vecteur2[0] = 0
```



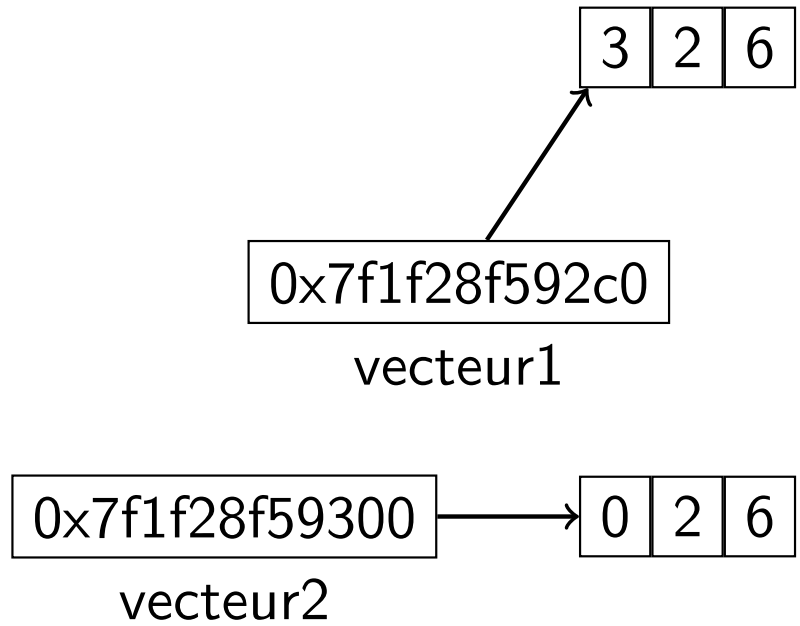
En réalité

```
vecteur1 = [3, 2, 6]
vecteur2 = vecteur1
vecteur2[0] = 0
```



En réalité

```
vecteur1 = [3, 2, 6]
vecteur2 = list(vecteur1)
vecteur2[0] = 0
```

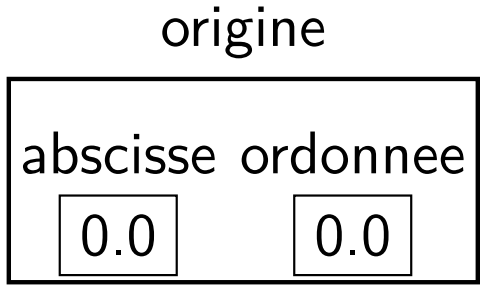


Objets

- permet de grouper des données ensemble en nommant chacune
- un type s'appelle alors une classe
- base de la programmation orientée objet au programme de 2A !
- pour nous, utilisation à minima : servira juste de conteneur pratique

Exemple : Point

```
1  #!/usr/bin/env python3
2  class Point:
3      def __init__(self,
           abscisse, ordonnee):
4          self.abscisse =
               abscisse
5          self.ordonnee =
               ordonnee
6
7  def main():
8      origine = Point(0.0, 0.0)
9      print(origine)
10
11 main()
```



Utilisation

- création d'un nouveau type d'objet (une classe) : `class Point:`
- définition du constructeur permettant la création des variables : `def __init__(self, ...)`
- création d'une variable : `mon_point = Point(1.4, 2.5)`
- accès au contenu d'un objet, par exemple pour afficher l'attribut abscisse de `mon_point` : `print(mon_point.abscisse)`

Méthodes

- sucre syntaxique sur les fonctions manipulants des objets
  - déclaration
    - comme une fonction classique
    - dans la classe
    - premier argument est `self`, l'objet lui même
    - exemple : `def affiche(self):`
  - utilisation

```
1  origine = Point(0.0, 0.0)
2  origine.affiche()
3  # au lieu de affiche(origine)
```

- possibilité de surcharger les opérateurs

Sommaire

Bilan

- différentes manières de stocker des données
  - différentes syntaxes
  - différentes représentations
  - différentes utilisations
- les variables ont une portée
  - privilégier les variables locales
  - passage par objets

Compétences à acquérir

- exécution pas à pas
  - dessiner l'état de la mémoire
  - mettre à jour sur chaque instruction
- structurer son code (début)
  - découpe en fonctions
  - documentées
  - création de modules