Bases du Codage

Introduction à Python

by Mickaël Bettinelli (Université Grenoble-Alpes)
on September 24, 2020

» Présentation du cours

Durée	12 heures
Objectif	Apprendre le langage de programmation Python et revoir les bases de l'algorithmique
Structure	Mélange de cours et de TPs
Évaluation	5 TPs à réaliser en binômes
	Coéfficients :
	TP1, 2, 3, 4 - 15%
	TP5 - 40%
Feedback	2 courtes évaluations de début et fin de module pour apprécier vos progrès (ignorées dans la moyenne)

C’est quoi Python ?

» Quelques banalités

Créateur Guido van Rossum.

Année 1991

Caractéristiques Cross-platform (utilisable sous la majorité des OS).

Un grand nombre de modules à importer

Langage de haut niveau (comme Java)

Langage interprété

Langage orienté objet

Langage open-source

Un des langages les plus utilisé

» Pourquoi utiliser Python ?

- Facilité d'apprentissage
- Langage peu verbeux et rapide à écrire
- Possède une grande communauté
- Beaucoup de modules existants pour repousser les limites du langage
- Permet de tout faire

» L'interpréteur Python

L'interpréteur de Python est accessible depuis une console. Il permet de tester des commandes et de commencer à jouer avec le langage. Pour y accéder, tapez dans votre console :

```
> python
```

Vous pouvez ensuite utiliser cet interprêteur comme calculatrice en tapant :

```
> 5 * 5
```

Vous pouvez aussi essayer d'entrer des commandes plus complexes comme :

```
> a = 7
> b = 6
> a * b
```

Variables et types de données

* Typage des variables
* Opérations
* Manipulation des variables

Variables et types de données

* Typage des variables
* Opérations
* Manipulation des variables

» Typage des variables

Types existants string, integer, float, boolean, complex

Typage dynamique les types des variables ne sont pas choisis par le développeur

Par exemple :

```
1    prenom = "Frodon"
2    taille = 120
3    age = 27.5
4    roux = False
5    test = 15+6i
```

Listing 1: Initialisation de variables

Variables et types de données

* Typage des variables
* Opérations
* Manipulation des variables

» Opérations

Opération	Notation
Addition	+
Soustraction	-
Multiplication	*
Division	/
Exponentiation	**
Division entière	//
Reste de la division entière	%

Variables et types de données

* Typage des variables
* Opérations
* Manipulation des variables

» Manipulation des variables

Affectation :

```
    a = 5
    a = 99
    print(a)
```

Résultat :

```
    99
```

Affectation composée :

```
    a = 0
    a = a + 1
    a += 1
    b = 2
    b *= 2
    print(a, b)
```

Résultat :

```
    2 4
```

Structures de contrôles

* Opérateurs de comparaison
* Conditions
* Boucles (while, for)

Structures de contrôles

* Opérateurs de comparaison
* Conditions
* Boucles (while, for)

» Opérateurs de comparaison

Opérateur	Notation
Inférieur	<
Supérieur	>
Egal	==
Inférieur ou égal	<=
Supérieur ou égal	>=
Différent	!=

Structures de contrôles

* Opérateurs de comparaison
* Conditions
* Boucles (while, for)

» if, elif, else

Les conditions permettent de modifier le flot d'exécution d'un programme en fonction du résultat d'un test.

```
    b = 0
    if b > 0:
        print("b est supérieur à 0")
    elif b == 0:
        print("b est égal à 0")
    else:
        print("b est inférieur à 0")
```

Résultat :

```
    b est égal à 0
```

Attention à l'indentation !

En Python l'indentation permet de définir ce qui appartient à un bloc d'instructions ou non contrairement à d'autres langages qui utilisent des accolades.

» Opérateur ternaire

Un opérateur ternaire est une forme contractée d'une condition sur une ligne. Il est utilisé lorsqu'une condition ne modifie le flot d'exécution du programme que d'une seule ligne et que la condition se limite à un if / else (sans elif). L'opérateur s'écrit de cette manière :

```
1    value_if_true if condition else value_if_false
```

Par exemple :

```
1    a = 0
2    b = 5
3    c = a + b if a < b else a * b
4    print(c)
```

Ce qui donnera :

```
1    5
```

Structures de contrôles

* Opérateurs de comparaison
* Conditions
* Boucles (while, for)

» while

Exécute un bloc d'instructions contenu dans le while tant qu'une condition est vrai.

```
a = 0
while a < 5:
    a += 1
print(a)
```

Résultat :

```
6
```

Attention à l'indentation sous le while !

» for

La boucle for est utilisée pour parcourir une séquence (une structure de données ou une chaine de caractères).
Attention à l'indentation sous le for !

```
    fruits = ["apple", "banana", "cherry"]
    for x in fruits:
        print(x)
```

Listing 2: Parcours d'un tableau

Résultat :

```
    apple
    banana
    cherry
```

» for

Lors de la recherche d'un élément dans une liste, "break" arrête la boucle avant la fin du parcours.

```
    fruits = ["apple", "banana", "cherry"]
    for x in fruits:
        if x == "banana":
            break
        print(x)
```

Listing 3: Parcours d'un tableau avec arrêt

Résultat :

```
    apple
```

» for

Il est aussi possible de récupérer l'indice des éléments d'un tableau en itérant dessus.

```
    fruits = ["apple", "banana", "cherry"]
    for i, fruit in enumerate(fruits):
        print(i, fruit)
```

Listing 4: Parcours d'un tableau avec indice

Résultat :

```
    0 apple
    1 banana
    2 cherry
```

» for

On peut accéder aux clés d'un dictionnaire ainsi que les valeurs associées.

```
    dictionnary = {
        "brand": "Ford",
        "model": "Mustang",
        "year": 1964
    }
    for key in dictionnary:
        print(key, dictionnary[key])
```

Listing 5: Parcourir un dictionnaire

Résultat :

```
    brand Ford
    model Mustang
    year 1964
```

» for

for permet aussi d'itérer sur un bloc un nombre défini de fois. La borne maximum n'est pas incluse.

```
1    for x in range(2, 5):
2        print(x)
```

Listing 6: Itérer N fois sur un bloc

Résultat :

```
1    2
2    3
3    4
```

Structures de données

* Liste
* Dictionnaire
* Ensemble
* Tuple

Structures de données

* Liste
* Dictionnaire
* Ensemble
* Tuple

» Liste

Définition Structure contenant en ensemble d'objets rangés dans un ordre spécifique

Utilisation Possibilité de l'utiliser comme un tableau, une pile ou une queue.

Déclaration

```
1    mylist = []
```

Initialisation

```
1    mylist = [2, 5, 9]
```

A noter !

Une liste n'est pas nécessairement une liste chainée ! Ici, une liste est un tableau dans lequel les éléments sont accessibles grâce à leur indice. Une liste chainée est une suite d'éléments dans laquelle il est nécessaire d'accéder à l'élément N-1 pour lire l'élément N.

» Liste (tableau)

Définition Collection indexable et contigüe d'éléments.

Utilisation Utiliser une liste comme un **tableau** :

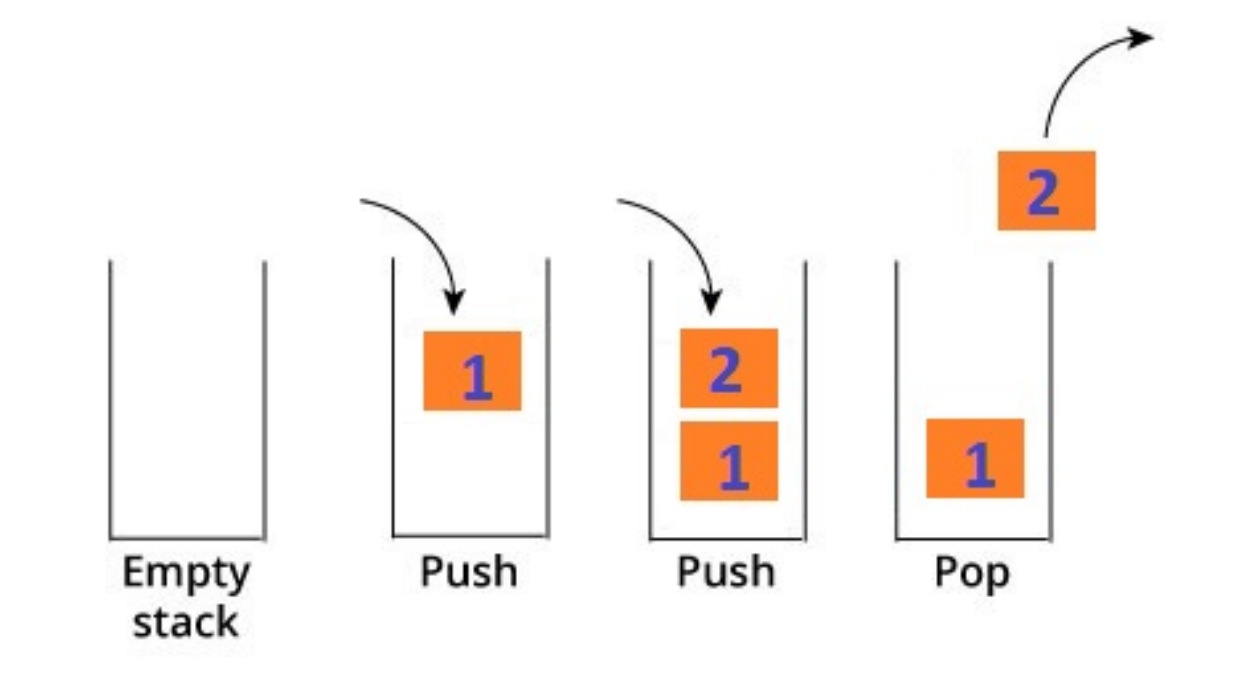
```
1    a = [15, 333, 987, 52.4, 333]
2    a.insert(2, -1)
3    print(a)
4    a.append(444)
5    a.remove(333)
6    print(a)
7    a.sort()
8    print(a)
9    print(a.index(52.4))
```

Résultat :

```
1    [15, 333, -1, 987, 52.4, 333]
2    [15, -1, 987, 52.4, 333, 444]
3    [-1, 15, 52.4, 333, 444, 987]
4    2
```

» Liste (pile)

Définition Collection d'éléments LIFO (Last In First Out)



[Image] https://javabycode.com/dsa/stack-data-structure-in-java.html

» Liste (pile)

Utilisation Utiliser une liste comme une **pile** :

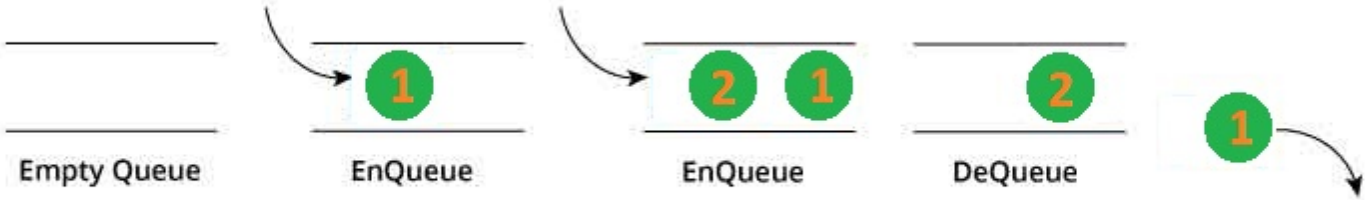
```
1    stack = [3, 4, 5]
2    stack.append(6)
3    stack.append(8)
4    print(stack)
5    stack.pop()
6    print(stack)
```

Résultat :

```
1    [3, 4, 5, 6, 8]
2    [3, 4, 5, 6]
```

» Liste (queue)

Définition **Collection d'éléments FIFO (First In First Out)**



[Image] https://javabycode.com/dsa/stack-data-structure-in-java.html

» Liste (queue)

Utilisation Utiliser une liste comme une **queue** :

```
1    from collections import deque
2    queue = deque(["Eric", "John", "Michael"])
3    queue.append("Terry")
4    queue.popleft()
5    print(queue)
```

Résultat :

```
1    ['Eric', 'John', 'Michael', 'Terry']
2    ['John', 'Michael', 'Terry']
```

Structures de données

* Liste
* Dictionnaire
* Ensemble
* Tuple

» Dictionnaire

Définition Tableau associatif dans lequel est associé une clef à une valeur

Spécificités Le dictionnaire est indexé par clef;
Une clef doit être une valeur non modifiable (integer ou string par exemple).

```
1    taille = {'antoine': 179, 'mathilde': 170}
2    tel["erwan"] = 169
3    print(taille)
4    del tel['antoine']
5    print(list(taille))
6    print(sorted(taille))
7    print("antoine" in taille)
```

Résultat :

```
1    {'antoine': 179, 'mathilde': 170, 'erwan': 169}
2    {'mathilde', 'erwan'}
3    {'erwan', 'mathilde'}
4    False
```

Structures de données

* Liste
* Dictionnaire
* Ensemble
* Tuple

» Ensemble

Définition Collection d'éléments non triée sans redondance

Spécificités Impossibilité d'avoir deux fois le même élément dans un set;

Supporte des opérations mathématique comme l'union, l'intersection, etc.

```
1     shopping_list = {'apple', 'orange', 'pear', 'banana'}
2     available_in_market = {'apple', 'pineapple', 'banana', '
      peach'}
3     shopping_list.add('apple')
4     # Eléments que je peux acheter au marché
5     print(shopping_list & available_in_market)
```

Résultat :

```
1     {'apple', 'banana'}
```

Structures de données

* Liste
* Dictionnaire
* Ensemble
* Tuple

» Tuple

Définition Collection d'éléments séparés par des virgules.

Spécificités Contrairement aux autres structures, on peut se permettre de mettre des types de données différents aux éléments du tuple;
Les tuples ne sont pas modifiables après leur création;
Il est possible d'affecter des attributs modifiables à un tuple.

```
1    physique = 'corentin', 189, 80
2    print(physique[0])
3    imc = (physique, 20.33)
4    imc[1] = 20.0
```

Résultat :

```
1    'corentin'
2    TypeError: 'tuple' object does not support item
     assignment
```

Manipuler des fichiers

* Lecture
* Ecriture
* Parser les données

Manipuler des fichiers

* Lecture
* Ecriture
* Parser les données

» Lecture

fonction Python propose la fonction "open()" built-in pour ouvrir les fichiers

paramètres
1. nom du fichier
2. r, w, : read, write

```
1    open(filename, mode)
```

Exemple

```
1    f = open("nom.txt", "r")
2    s = f.read()
3    print(s)
```

Manipuler des fichiers

* Lecture
* Ecriture
* Parser les données

» Ecriture

exemple Pour l'écriture :

```
1    f = open("nom.txt", "w")
2    f.write("Now the file has more content!")
3    s = f.read()
4    print(s)
```

Manipuler des fichiers

* Lecture
* Ecriture
* Parser les données

» Parser les données

Définition Convertir une chaine de caractères en sous séquences de différents types.

Objectif Extraire des informations de chaines de caractères

Comment Python propose la méthode built-in "split()". Split sépare une chaine de caractère en fonction du séparateur passé en paramètre et range le résultat de la séparation dans une liste.

Utilisation "".split(separator)

Exemple

```
1        mystring = "1 2 3 4 5"
2        mystring.split(" ")
3        print(mystring)
```

Résultat :

```
1        [1, 2, 3, 4, 5]
```

Procédures et fonctions

» Procédure

Définition Une procédure est un appel à un bloc d'instructions dans lequel on peut passer des paramètres. Une procédure **ne retourne aucune valeur**.

```
1    def print_max(param1, param2):
2        if param1 > param2:
3            print(param1)
4        elif param2 > param1:
5            print(param2)
6        else:
7            pass
```

Attention à l'indentation sous la définition de la procédure !

» Fonction

Définition Une fonction est un appel à un bloc d'instructions dans lequel on peut passer des paramètres. Une fonction **retourne au moins une valeur**.

Comme toutes les autres variables, les paramètres ne sont pas typés.

```
1    import math
2
3    def get_max(param1, param2):
4        maxv = math.inf
5        if param1 > param2:
6            maxv = param1
7        elif param2 > param1:
8            maxv = param2
9        return maxv
```

Attention à l'indentation sous la définition de la fonction !

» Récursivité

Définition Un algorithme récursif est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème.

Exemple Calculer une factorielle :

```
1    def facto(n, valeur):
2        if n <= 1:
3            return valeur
4        return facto(n - 1, valeur * n)
5
6    def factorielle(n):
7        return facto(n, 1)
8
9    print(factorielle(5))
```

Résultat :

```
1    120
```

Programmation orientée objet

* Syntaxe
* Encapsulation

» Programmation orientée objet

Définition Un langage orienté objet permet de créer des obets que l'on peut manipuler. Python est similaire aux autres langages sur ce point.

Spécificité La visibilité des attributs est obligatoirement public (ce qui n'empêche pas de respecter l'encapsulation)

Le constructeur est une méthode appelée __ini__

Convention Une classe, une méthode ou un attribut privé doit être précédé d'un _

Programmation orientée objet

* Syntaxe
* Encapsulation

» Syntaxe

Exemple Classe Personnage

```
1   class Personnage:
2       def __ini__(self, poids, taille, nom):
3           # Attributs privés
4           self._poids = poids
5           self._taille = taille
6           self._nom = nom
7
8       # Méthode publique
9       def getPoids(self):
10          return self._poids
11
12      # Méthode privée
13      def _grandir(self):
14          self._taille += 5
```

```
1       p = Personnage(176, 80, 'Vincent')
2       p.getPoids()
```

Attention à l'indentation !

Programmation orientée objet

* Syntaxe
* Encapsulation

» Encapsulation

Définition Regroupement de données brutes accessibles par un ensemble de routines permettant de les manipuler.

Objectifs Offrir une interface orientée service à l'utilisateur en indiquant quels sont les services offerts et quelles sont les responsabilités de la classe utilisée;

Permettre de modifier la structure d'une classe sans modifier son interface pour ne pas gêner l'utilisateur.

» Encapsulation

Exemple

```
1     class Personnage:
2         def __ini__(self, poids, taille, nom):
3             # Attributs privés
4             self._poids = poids
5             self._taille = taille
6             self._nom = nom
```

Sans encapsulation, l'utilisateur de Personnage devrait modifier le nom de son instance comme cela :

```
1     p = Personnage(176, 80, 'Vincent')
2     p._nom = 'Jocelyn'
```

Si le concepteur de la classe Personnage souhaite modifier le nom de son attribut "*_nom*", il ne peut plus le faire sans obliger l'utilisateur à modifier son code.

» Encapsulation

Exemple

```
1     class Personnage:
2         def __ini__(self, poids, taille, nom):
3             # Attributs privés
4             self._poids = poids
5             self._taille = taille
6             self._nom = nom
7
8         def setNom(self, valeur):
9             self._nom = valeur
```

Avec l'encapsulation, l'utilisateur de Personnage peut utiliser l'interface *"setNom()"*.

```
1     p = Personnage(176, 80, 'Vincent')
2     p.setNom('Jocelyn')
```

Le concepteur est libre de changer le nom de son attribut en *"_prenom"*.

» Ressources supplémentaires

Conteneurs https://docs.python.org/3.10/tutorial/datastructures.html

Cours https://python.developpez.com/tutoriels/apprendre-programmation-python/les-bases/