Algorithmique de base

Module : Fondement de la programmation

Douglas Teodoro

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

2019-2020

Sommaire

Objective

- Le type `string`
- Introduction à la programmation fonctionnelle
  - fonctions
  - procédures
- Mettre en œuvre quelques structures pour résoudre des problèmes simples

Sommaire

CHAÎNES DE CARACTÈRES

Introduction

Nous allons maintenant nous intéresser au type string (ou str) qui permet de définir des chaînes de caractères

Exemple

```
cours:str = "Algo I"
```

Les chaînes de caractères - Fonctions utiles

len : calculer la longueur d'une chaîne
python : `len("Algo I")`

+ : concaténer 2 chaînes
python : `"Algo" + " I"`

lower : convertir la chaîne en minuscules
python : `"Algo I".lower()`

upper : convertir la chaîne en majuscules
python : `"Algo I".upper()`

split : diviser la chaîne
python : `p1, p2 = "Algo I".split()`

strip : supprimer les espaces excédantes au début et à la fin de la chaîne
python : `" Algo I ".strip()`

Traitements courants - Formatage

- **L'opérateur `%`** après une `string` est utilisé pour combiner une chaîne des caractères avec des variables

- L'opérateur **`%`** remplacera `%s` dans une chaîne de caractères par la variable `string` qui la suit

```
var = "def" ⟶ "abc %s" % var ⟶ "abc def"
```

- Le symbole spécial **`%d`** est utilisé comme caractère de remplacement pour les valeurs entiers

```
var = 10 ⟶ "abc %d" % var ⟶ "abc 10"
```

- Le symbole spécial **`%.␣f`** est utilisé comme caractère de remplacement pour les valeurs flottantes

```
var = 3.1415 ⟶ "abc %.2f" % var ⟶ "abc 3.14"
```

Traitements courants - Comparaison

Les opérateurs de comparaison s'appliquent aussi aux chaînes de caractères

Comparaison

```
012...789ABC...WYZabc...wyz

"A" < "Z" == True
"a" < "z" == True
"0" < "9" == True
"A" < "a" == True
"0" < "A" == True
"abc" < "b" == True
```

- Python a de nombreux types d'opérateurs de comparaison dont >=, <=, >, <=, >, <, etc.
- Les comparaisons donnent des valeurs booléennes : True ou False

Traitements courants - Indexation

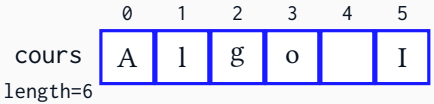
Définition

Une chaîne de caractères n'est rien d'autre qu'un tableau de caractères, où chaque caractère correspond à un indice

Exemple

Chaînes de caractères `cours` contenant 6 caractères :

```
cours: str = "Algo I"
```




```
cours[0] == 'A'
cours[1] == 'l'
cours[2] == 'g'
cours[3] == 'o'
cours[4] == ' '
cours[5] == 'I'
```

Traitements courants - Découpage

On peut utiliser l'**indice** pour accéder à une **sous-chaîne** d'une chaîne de caractères

`str[index]` : le caractère à l'indice `index`

`str[start:end]` : les caractères de l'indice `start` à l'indice `end-1`

`str[start:]` : les caractères de l'indice `start` jusqu'au reste du tableau

`str[:end]` : les caractères du début à l'indice `end-1`

Sous-chaîne

Pour la chaîne :

```
cours = "Algo I"
```

```
cours[0:2] == "Al"
cours[0:3] == "Alg"
cours[1:6] == "lgo I"
cours[4] == " "
```

Algorithme d'itération sur une chaîne de caractères

```
for i in range(len(<str>)):
    print(<str>[i])
```

```
for i in <str>:
    print(i)
```

```
var_str: str = "Algo I"
for i in range(len(var_str)):
    print("%d -> %s" % (i, var_str[i]))
```

```
0 -> A
1 -> l
2 -> g
3 -> o
4 ->
5 -> I
```

```
var_str: str = "Algo I"
for i in var_str:
    print("%s" % i)
```

```
A
l
g
o

I
```

PyCharm

exercice_[1-3].py

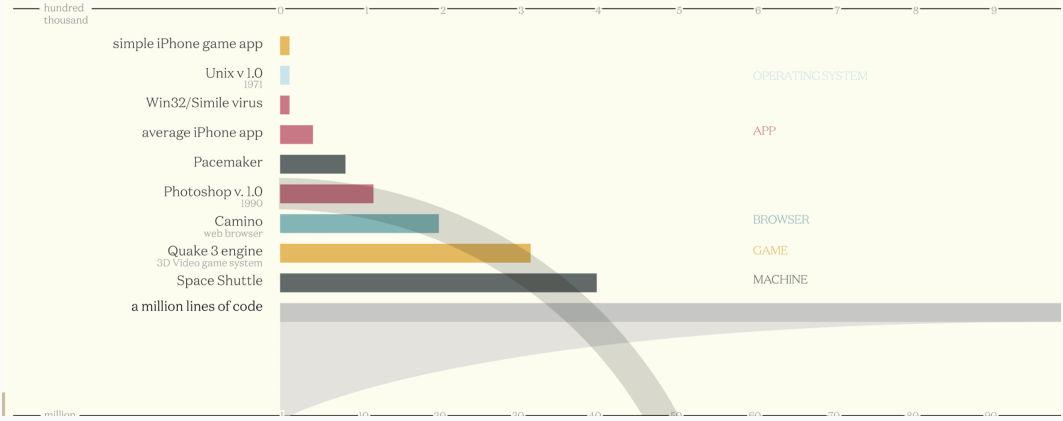
Sommaire

LES SOUS-PROGRAMME

Sous-programme - Principe

Un programme est généralement un code long et compliqué, fait à partir d'instructions rudimentaires

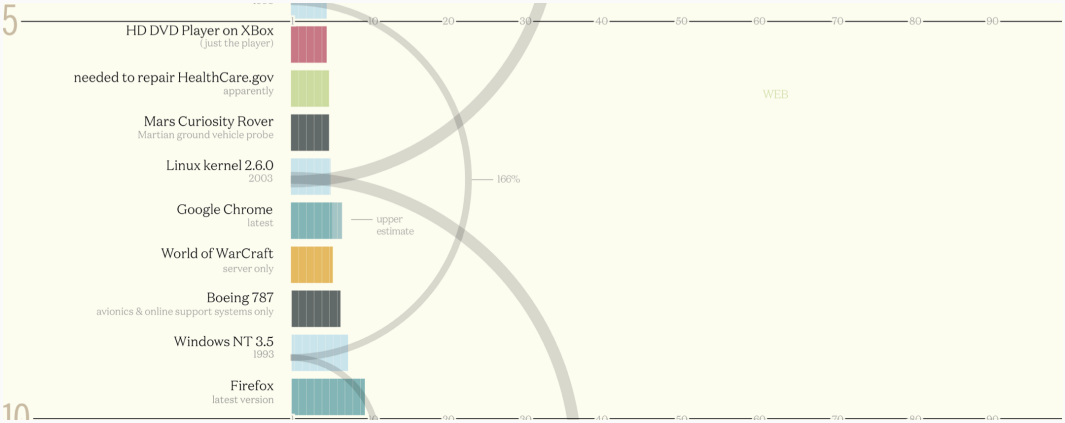
Lignes de code - Milliers



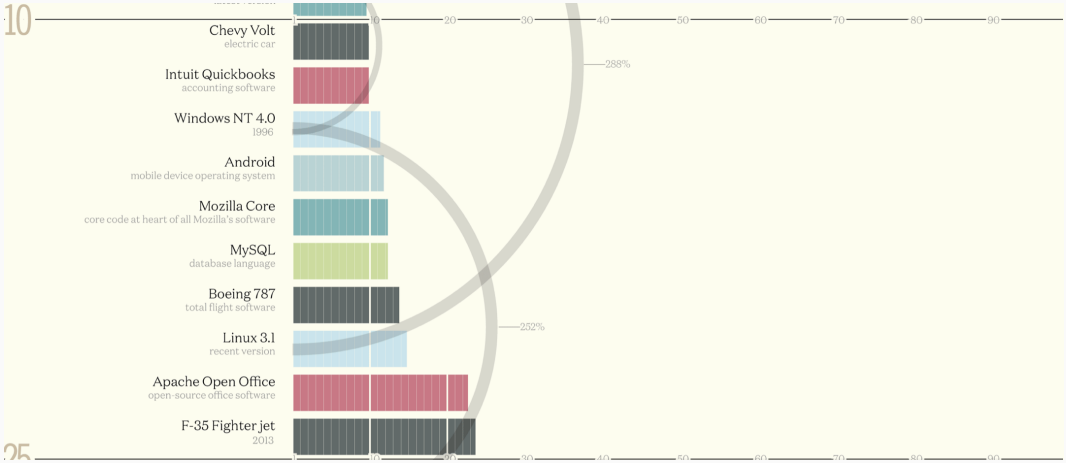
https://informationisbeautiful.net/visualizations/million-lines-of-code

Lignes de code - Millions

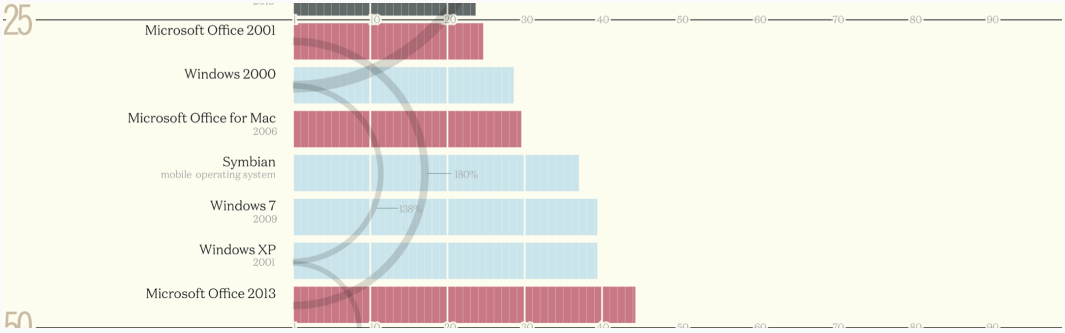
Lignes de code - Millions



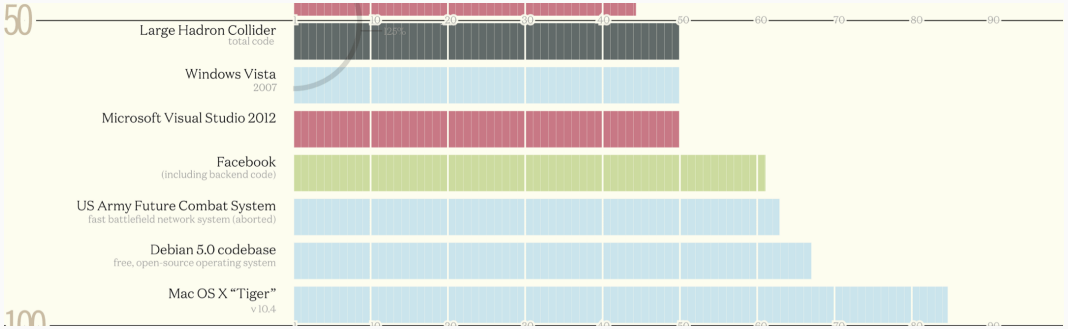
Lignes de code - Millions



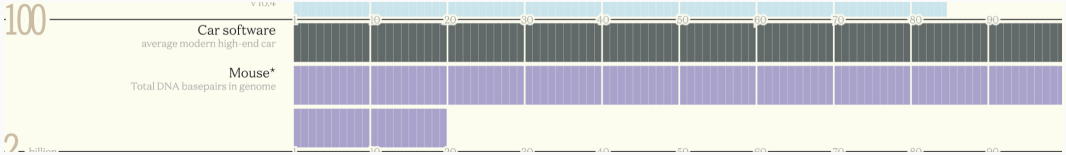
Lignes de code - Millions



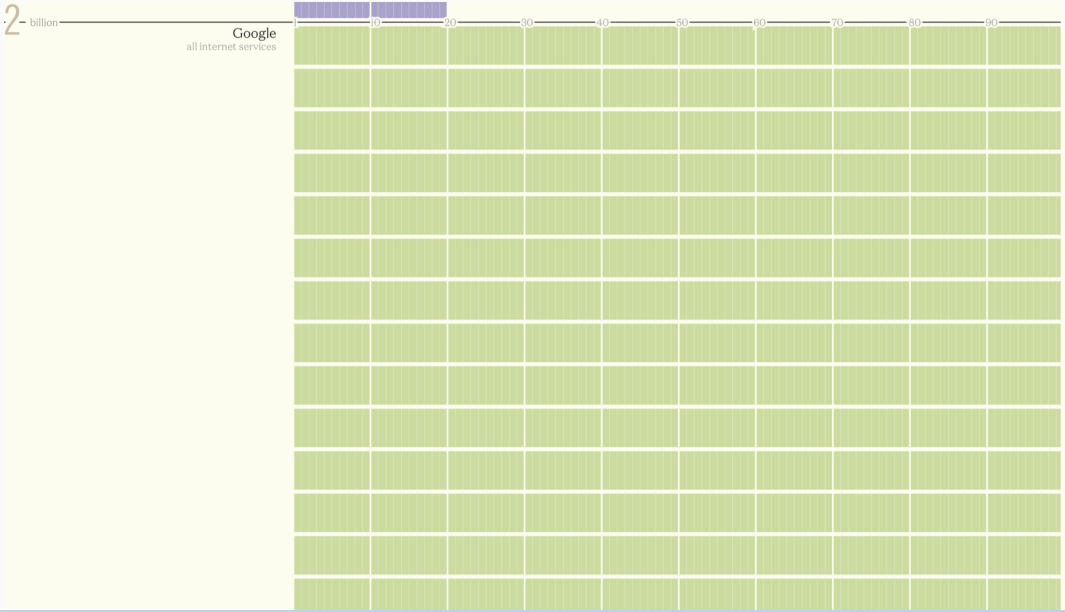
Lignes de code - Millions



Lignes de code - Millions



Lignes de code - Milliards



Sous-programme - Principe

Lisibilité et compréhension algorithmique

Une suite de 10 000 instructions rudimentaires serait **incompréhensible**

Réutilisabilité

On constate également par la pratique que les programmes on souvent besoin de **faire les mêmes choses** (ou quasiment les mêmes choses)

Réutilisabilité et correction

Il serait dommage de recopier 100 fois les mêmes lignes d'instructions :

1. c'est consommateur en **temps de développement**
2. c'est **générateur d'erreur**
3. c'est **compliqué à corriger** (il faut penser à corriger une erreur pour les 100 occurrences du même bout de programme)

Réutilisabilité des algorithmes

Problème

Comment **réutiliser** un algorithme existant sans avoir à le réécrire ?

```
"""algo: factorielle_5
données: n -> int
résultat: factorielle de n ->
    int
"""
### decl. et init.
### des variables
n: int = 5
resultat: int = 1
### séquence d'opérations
for i in range(1, n+1):
    resultat = i * resultat

print("resultat =", resultat)
```

```
"""algo: factorielle_100
données: n -> int
résultat: factorielle de n ->
    int
"""
### decl. et init.
### des variables
n: int = 100
resultat: int = 1
### séquence d'opérations
for i in range(1, n+1):
    resultat = i * resultat

print("resultat =", resultat)
```

Réutilisabilité des algorithmes

```python
### decl. et init. des variables
n: int = 5
resultat: int = 1
### séquence d'opérations
for i in range(1, n+1):
    resultat = i * resultat
print("resultat =", resultat)
```

```python
### decl. et init. des variables
n: int = 100
resultat: int = 1
### séquence d'opérations
for i in range(1, n+1):
    resultat = i * resultat
print("resultat =", resultat)
```

Elément de réponse

Encapsuler le code dans des fonctions

```
factorielle(5)
```

```
factorielle(100)
```

Structuration des algorithmes

Problème

Comment structurer un algorithme pour le rendre plus compréhensible ?

```
""" algo: puissance
données: x -> float, n -> int
résultat: puissance x^n -> float
"""
x: float = float(input("La valeur de x: "))
n: int = int(input("La valeur de n: "))
resultat: float = 1

if n != 0:           # x^0 = 1
    signe: int = 1
    if n < 0:        # teste le signal de n
        n = -n
        signe = -1       # puissance négative
    for cpt in range(1, n+1):
        resultat = resultat * x
    if signe < 0:
        resultat = 1/resultat

print("x^n =", resultat)
```

Structuration des algorithmes

Problème

Comment structurer un algorithme pour le rendre plus compréhensible ?

```
""" algo: puissance
données: x -> float, n -> int
résultat: puissance x^n -> float
"""
x: float = float(input("La valeur de x: "))
n: int = int(input("La valeur de n: "))
resultat: float = 1

if n != 0:              # x^0 = 1
    signe: int = 1
    if n < 0:           # teste le signal de n
        n = -n
        signe = -1      # puissance négative
    for cpt in range(1, n+1):
        resultat = resultat * x
    if signe < 0:
        resultat = 1/resultat

print("x^n =", resultat)
```

Structuration des algorithmes

Elément de réponse

Utiliser des **fonctions**

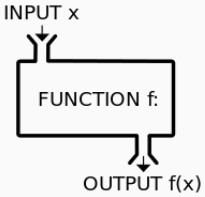
```
""" algo: puissance
données: x -> float, n -> int
résultat: puissance x^n -> float
"""
x: float = float(input("x: "))
n: int = int(input("n: "))

resultat: float = puissance(x,n)

print("x^n =", resultat)
```

```
""" algo: puissance
données: x -> float, n -> int
résultat: puissance x^n -> float
"""
x: float = float(input("x: "))
n: int = int(input("n: "))
resultat: float = 1

if n != 0:              # x^0 = 1
    signe: int = 1
    if n < 0:   # teste le signal de n
        n = -n
        signe = -1  # puissance négative
    for cpt in range(1, n+1):
        resultat = resultat * x
    if signe < 0:
        resultat = 1/resultat

print("x^n =", resultat)
```

Réutilisabilité des algorithmes

Un **sous-programme** (fonction ou procedure) est un sort de boîte noire :

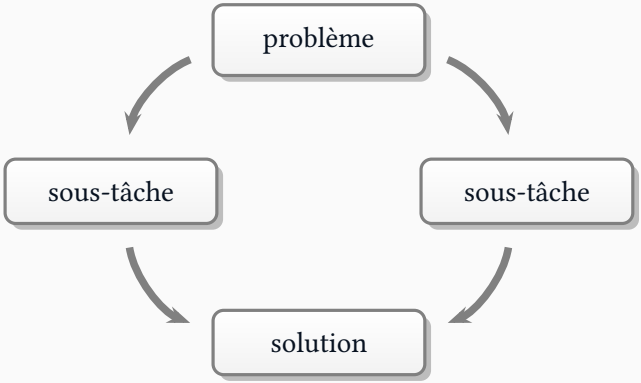
- à l'extérieure, il est vue comme une instruction qui réalise une tâche de traitement de données (peu importe comment !)
- à l'intérieur, c'est un (mini-)programme qui implémente la sous-tâche de traitement de l'information



Diviser pour régner

Structuration

Les **fonctions** et les **procédures** permettent de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés eux-mêmes en fragments plus petits, et ainsi de suite



Les fonctions - Définition

- Une **fonction** est une suite ordonnée d'instructions qui retourne une valeur
- Un bloc d'instructions nommé et paramétré

Fonction ≡ expression

Une fonction joue le rôle d'une expression
Elle enrichit le jeu des expressions possibles

Exemple

```
y = sin(x)
```

renvoie la valeur du sinus de `x`

nom	`sin`
paramètres	`x: float`
retourne	`float`

Les procédures - Définition

- Une **procédure** est une suite ordonnée d'instructions qui ne retourne pas de valeur
- Un bloc d'instructions nommé et paramétré

Procédure ≡ instruction

Une procédure joue le rôle d'une instruction
Elle enrichit le jeu des instructions existantes

Exemple

```
print(x, y, z)
```

affiche les valeurs de x, y et z

nom `print`
paramètres `x, y, z`

Déclaration et définition - dans un algorithme

Avant de pouvoir utiliser un sous-programme, il **faut le définir ou le déclarer**, c'est-à-dire indiquer au programme principal qu'il existe

- nom
- les paramètres
- la valeur de retour
- contenu (bloc d'instructions)

En algorithmique et dans les langages de programmation, les sous-programmes sont déclarés et entièrement écrits au **tout début**

- Le programme principal **ne peut pas utiliser** un sous-programme s'il ne sait pas s'il existe
- Ex. : Python

Déclaration et définition - en Python

Fonction

```
""" nom
données: ...
résultat: ...
"""
def nom_fonction(parametres) -> type_retour:
    # bloc d'instructions
    ...
    return ...

# programme principal
...
```

Procédure

```
""" nom
données: ...
résultat: ...
"""
def nom_procedure(parametres):
    # bloc d'instructions
    ...
    # pas d'instruction return

# programme principal
...
```

- `def` : mot clé réservé pour indiquer la définition d'une fonction ou procédure
- `return` : mot clé réservé pour indiquer la valeur retourné par la fonction
- `nom` : nom donné au sous-programme, même convention que pour les variables
- `parametres` : la liste de paramètres passé au sous-programme
- `type_retour` : le type retourné (str, int, float, etc.)

Déclaration et définition - dans un algorithme

```
1  """ algo: calcule puissance
2  données: n -> int, x -> float
3  résultat: puissance x^n -> float
4  """
5  def puissance(x: float, n: int) -> float:
6      resultat: float = 1
7      for cpt in range(1, n + 1):
8          resultat = resultat * x
9      return resultat
10
11 def print_res(x: float, n: int, y: float):
12     print("puissance de %.2f ^ %d est %.2f"
13         %(x, n, y))
14
15 x: float = float(input("x: "))
16 n: int = int(input("n: "))
17
18 resultat: float = puissance(x, n)
19 print_res(x, n, resultat)
```

- lignes 1 - 4 : documentation du programme

Déclaration et définition - dans un algorithme

```
1  """ algo: calcule puissance
2  données: n -> int, x -> float
3  résultat: puissance x^n -> float
4  """
5  def puissance(x: float, n: int) -> float:
6      resultat: float = 1
7      for cpt in range(1, n + 1):
8          resultat = resultat * x
9      return resultat
10
11 def print_res(x: float, n: int, y: float):
12     print("puissance de %.2f ^ %d est %.2f"
13         %(x, n, y))
14
15 x: float = float(input("x: "))
16 n: int = int(input("n: "))
17
18 resultat: float = puissance(x, n)
19 print_res(x, n, resultat)
```

- lignes 1 - 4 : documentation du programme
- lignes 15 - 19 : programme principal

Déclaration et définition - dans un algorithme

```
1  """ algo: calcule puissance
2  données: n -> int, x -> float
3  résultat: puissance x^n -> float
4  """
5  def puissance(x: float, n: int) -> float:
6      resultat: float = 1
7      for cpt in range(1, n + 1):
8          resultat = resultat * x
9      return resultat
10
11 def print_res(x: float, n: int, y: float):
12     print("puissance de %.2f ^ %d est %.2f"
13         %(x, n, y))
14
15 x: float = float(input("x: "))
16 n: int = int(input("n: "))
17
18 resultat: float = puissance(x, n)
19 print_res(x, n, resultat)
```

- lignes 1 - 4 : documentation du programme
- lignes 15 - 19 : programme principal
- lignes 5 - 9 : fonction qui calcule la puissance

Déclaration et définition - dans un algorithme

```
1  """ algo: calcule puissance
2  données: n -> int, x -> float
3  résultat: puissance x^n -> float
4  """
5  def puissance(x: float, n: int) -> float:
6      resultat: float = 1
7      for cpt in range(1, n + 1):
8          resultat = resultat * x
9      return resultat
10
11 def print_res(x: float, n: int, y: float):
12     print("puissance de %.2f ^ %d est %.2f"
13         %(x, n, y))
14
15 x: float = float(input("x: "))
16 n: int = int(input("n: "))
17
18 resultat: float = puissance(x, n)
19 print_res(x, n, resultat)
```

- lignes 1 - 4 : documentation du programme
- lignes 15 - 19 : programme principal
- lignes 5 - 9 : fonction qui calcule la puissance
- lignes 11 - 13 : procédure qui affiche le résultat

Appel

- Un sous-programme est exécuté **depuis le programme principal** ou **un autre programme**
- Le **programme fait appel** au sous-programme : l'appel au sous-programme est une instruction qui va déclencher l'exécution de celui-ci
- Cet appel peut **avoir lieu** n'importe où
- Il est d'usage d'appeler un sous-programme par son nom : pas d'**instructions particulier**

PyCharm

exercice_[4-6].py

Conclusion

Contenu vu :

- les chaînes de caractères
- introduction aux sous-programmes
  - fonction
  - procedure

Référence

Algorithmique - Techniques fondamentales de programmation
Chapitre : Les sous-programmes
Ebel et Rohaut, https://aai-logon.hes-so.ch/eni

Cyberlearn : 19_HES-SO-GE_631-1 FONDEMENT DE LA PROGRAMMATION (*welcome*)
http://cyberlearn.hes-so.ch