Programmation système

Introduction à Python

John Samuel
CPE Lyon

Année: 2019-2020
Courriel: john(dot)samuel(at)cpe(dot)fr

1. Commentaires

En Python, si vous voulez écrire un commentaire, commencez votre phrase avec un dièse (#).

```
# Ceci est un commentaire sur une seule ligne
```

1. Commentaires

Si votre commentaire ne tient pas sur une seule ligne, vous pouvez écrire votre commentaire en le commençant avec trois guillemets (""") et en le finissant pareil comme indiqué ci-dessous.

```
"""
  Ceci est un commentaire
  sur plusieurs lignes
"""
```

2. Variables

Affichage d'une message

Afficher "Bonjour" sur l'écran.

```
# bonjour.py
print("Bonjour")
```

Exécution

```
$ python3 bonjour.py
Bonjour
```

2. Variables

Python vous permet également de stocker votre message dans une variable.

```
# une variable
message = "le monde!"
print(message)
```

L'avantage d'utiliser une variable est la possibilé d'utiliser une valeur plusieurs fois sans la réécrire.

```
# une variable
message = "le monde!"
print(message)
print(message)
```

2. Variables

Vous pouvez également changer la valeur d'une variable.

```
# une variable
message = "le monde!"
print(message)
message = "Bonjour!"
print(message)
```

2. Variables

Il est possible de stocker différents types de données. Par exemple, nous pouvons stocker les entiers dans les variables, afin de les utiliser pour un calcul mathématique.

```
a = 10
b = 20
c = a + b
print(c)
```

2. Variables

Le prochain code stocke un nombre réel dans une variable.

```
#nombres réels
pi = 3.14
print(pi)
```

2. Variables

Prochainement, nous testerons les calculs en utilisant les entiers et les nombres réels.

```
#nombres réels
pi = 3.14
rayon = 5
superficie = pi * rayon * rayon
print(superficie)
perimetre = 2 * pi * rayon
print(perimetre)
```

Testez le code au-dessus en changeant la valeur du *rayon*.

3. Le type d'une variable

Parfois, il est important de tester le type d'une variable. En Python, vous pouvez utiliser une fonction intégrée, *type* en passant la variable comme argument.

```
#type de données
message1 = "Bonjour"
a = 12
pi = 3.14
print(type(message1))
print(type(a))
print(type(pi))
```

4. Concatenation de chaînes de

Il est possible de concatener deux chaînes de caractères en utilisant l'opérateur '+'.

```
#concatenation de chaine de caractères
message = "le monde!"
print("Bonjour" + message)
```

4. Concatenation de chaînes de

Ci-dessous, vous voyez le code qui concatène deux chaînes de caractères en utilisant l'opérateur '+' et les variables.

```
#concatenation de chaine de caractères
message1 = "Bonjour "
message2 = "le monde!"
print(message1 + message2)
```

4. Concatenation de chaînes de

Testez le code suivant.

```
#concatenation de deux différents types de données
message1 = "Bonjour en Python"
a = 3
print(message1 + a)
```

Quel message d'erreur s'affiche?

4. Concatenation de chaînes de

Il est impossible de concatener une chaîne de caractères avec un autre type de données (comme, un entier dans l'exemple précédent). Python nous fournit une fonction intégrée appelée (str) qui peut convertir un entier ou un nombre réel en une chaîne de caractères.

```
#concatenation de deux différents types de données
message1 = "Bonjour en Python "
a = 3
print(message1 + str(a))
```

5. Liste

Pour stocker une ou plusieurs valeurs du même type (ou différent type) dans une variable, vous pouvez utiliser une liste. Pour créer une liste, mettez toutes les valeurs entre '[' ']' et en utilisant les virgules comme séparateur comme indiqué ci-dessous.

```
a = [10, 20, 30, 40, 50]
print(a)
```

5. Liste

L'ordre des éléments dans une liste est important. Pour accèder à un membre de la liste, nous utiliserons son index. En Python, l'index commence à 0; c'est à dire, pour accèder au premièr membre de la liste, utilisez 0 et pour le nième membre, utilisez n-1 etc.

```
a = [10, 20, 30, 40, 50]
print(a[0])
print(a[1])
print(a[2])
print(a[3])
print(a[4])
```

5. Liste

Tester le code suivant où l'on essaie d'accèder à un membre inexistant.

```
a = [10, 20, 30, 40, 50]
print(a[8])
```

5. Liste

Une chaîne de caractères est également une liste. En conséquence, vous pouvez utiliser les index pour accéder au nième caractère de la chaîne.

```
message1 = "Bonjour en Python "
print(message1[0])
print(message1[1])
print(message1[2])
print(message1[3])
print(message1[4])
print(message1[5])
print(message1[6])
print(message1[7])
```

5. Liste

Utilisez la fonction intégrée *len()* afin d'obtenir la taille d'une chaîne (ou le nombre de caractères d'une liste).

```
message1 = "Bonjour en Python "
print(len(message1))
```

5. Liste

Vous pouvez utiliser la fonction *len()* pour récupérer le nombre d'éléments d'une liste.

```
a = [10, 20, 30, 40, 50]
print(len(a))
```

5. Liste

Pour ajouter un nouveau membre dans une liste, utilisez la méthode *append*. Le nouveau membre est ajouté à la fin de la liste. Testez le code et voyez le résultat.

```
a = [10, 20, 30, 40, 50]
a.append(60)
print(a)
```

5. Liste

Il est possible de modifier la nième valeur dans une liste. Précisez le nième membre comme indiqué ci-dessous en utilisant l'index et affectez la nouvelle valeur.

```
a = [10, 20, 30, 40, 50]
a[0] = 0
print(a)
```

5. Liste

Testez le code suivant pour changer la valeur à un index inexistante. Quelle est l'erreur que vous avez eu?

```
a = [10, 20, 30, 40, 50]
a[6] = 20
print(a)
```

5. Liste

Vous avez vu au-dessus la méthode *append* qui ajoute le nouveau membre à la fin de la liste. Pourtant avec la méthode *insert*, nous pouvons insérer un nouveau membre n'importe où dans la liste. Le premier argument d'*insert* est l'index et le deuxième est la valeur. Testez le code suivant et voyez la nouvelle liste et sa taille après l'insertion.

```
a = [10, 20, 30, 40, 50]
a.insert(0, 0)
print(a)
print(len(a))
```

5. Liste

Testez le code suivant avec différentes valeurs pour l'index (premier argument d'*insert*).

```
a = [10, 20, 30, 40, 50]
a.insert(6,60)
print(a)
print(len(a))
```

6. Tuple (une liste qui ne peut plus

Un tuple est une liste qui ne peut plus être modifiée et est déclaré en mettant les membres entre (et) au lieu de symboles [et] respectivement.

```
a = (10, 20, 30, 40, 50)
print(a)
```

Nous pouvons accèder à n'importe quel membre d'un tuple en utilisant son index.

```
a = (10, 20, 30, 40, 50)
print(a[0])
```

6. Tuple (une liste qui ne peut plus

Essayez de changer les valeurs d'un tuple, comme nous avons testé changer une liste précédemment.

```
a = (10, 20, 30, 40, 50)
a[0] = 0
print(a)
```

Quel message d'erreur s'affiche?

6. Tuple (une liste qui ne peut plus

Le tuple est très performant à comparer à une liste. C'est pourquoi, les tuples sont très utilisés pour l'analyse de données. Vous pouvez convertir une liste en un tuple en utilisant la méthode *tuple*.

```
a = [10, 20, 30, 40, 50]
b = tuple(a)
print(b)
print(type(b))
```

7. Ensemble (set)

Si vous n'êtes pas interessé par l'ordre des membres dans une liste, mais plutôt interéssé par les membres distincts, utilisez un ensemble en Python. Voyez le code suivant où nous mettons plusieurs valeurs différentes, mais nous voyons beaucoup de répétition. Nous utilisons les symboles { et } pour créer un ensemble de valeurs. Regardez-bien l'affichage de ce code et voyez la variable *a* qui contient des valeurs distinctes.

```
a = {10, 20, 30, 40, 50, 10, 20, 30, 40, 50}
print(a)
```

```
set([40, 10, 20, 50, 30])
```

7. Ensemble (set)

La méthode *add* est utilisée pour ajouter un nouveau membre dans l'ensemble. Ci-dessous, nous essayons d'ajouter une valeur déjà existante dans l'ensemble.

```
a = {10, 20, 30, 40, 50, 10, 20, 30, 40, 50}
a.add(10)
print(a)
```

7. Ensemble (set)

Ci-dessous, nous essayons d'ajouter une valeur inexistante dans l'ensemble. En effet l'ordre n'est pas important dans un ensemble, la position du nouveau membre est uncertaine.

```
a = {10, 20, 30, 40, 50, 10, 20, 30, 40, 50}
a.add(60)
print(a)
```

7. Ensemble (set)

Pour supprimer une valeur d'un ensemble, utilisez la méthode *remove*.

```
a = {10, 20, 30, 40, 50, 10, 20, 30, 40, 50}
a.remove(40)
print(a)
```

7. Ensemble (set)

Enfin, nous reregardons les différentes façons de stocker une collection de valeurs.

```
#set
a = {10, 20, 30, 40, 50, 10, 20, 30, 40, 50}
print(a)
print(type(a))

#tuple
b = (10, 20, 30, 40, 50, 10, 20, 30, 40, 50)
print(b)
print(type(b))

#liste
c= [10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
print(c)
print(type(c))
```

8. Expressions conditionnelles

- Une expression conditionnelle peut retourner deux valeurs: True (vrai) ou False (faux) en dependant de la vérite de l'expression.

Le code ci-dessous vérifie si l'entier 12 (stocké dans la variable *a* est un multiple de 2 (divisible par 2). L'expression conditionnelle *a%2 == 0* retourne True (vrai) dans cet exemple. Notez bien que nous utilisons l'opérateur ==, qui est différent de =, utilisé pour l'affectation d'une variable. Testez le code suivant avec différentes valeurs de *a*.

```
a = 12
if( a%2 == 0):
  print(a, " est divisible par 2")
else:
  print(a, " n'est pas divisible par 2")
```

8. Expressions conditionnelles

Il est également possible de tester si deux chaînes de caractères seraient égales (ou contiendraient les mêmes valeurs).

```
lang = "Français"
if (lang =="Français"):
  print("Bonjour le monde!")
else:
  print("Hello World!")
```

9. Boucles for

Vous pouvez utiliser des boucles *for* pour répéter une tâche plusieurs fois. Au-dessus, nous avons utilisé print pour afficher le contenu d'une liste ou un tuple.

Si vous voulez afficher chaque membre de la liste sur une ligne, vous pouvez utiliser par exemple une boucle for comme indiqué ci-dessous.

```
for i in [10, 20, 30, 40, 50, 10, 20, 30, 40, 50]:
  print(i)
```

9. Boucles for

La deuxième ligne de code au-dessus affiche la valeur i sur une ligne. L'affichage s'arrete quand on a parcouru tous les membres de la liste.

```
for i in [10, 20, 30, 40, 50, 10, 20, 30, 40, 50]:
  print(i)
```

9. Boucles for

Prochainement, nous testerons le même code au-dessus, mais en utilisant un tuple.

```
for i in (10, 20, 30, 40, 50, 10, 20, 30, 40, 50):
  print(i)
```

9. Boucles for

Prochainement, nous testerons le même code au-dessus, mais en utilisant un ensemble.

```
for i in {10, 20, 30, 40, 50, 10, 20, 30, 40, 50}:
  print(i)
```

10. range()

Vous pouvez utiliser la méthode *range* qui prend deux (ou trois) arguments: le numéro de départ et le dernier numéro et qui retourne une liste de nombres en ordre croissant (par défaut) de numéro de départ jusqu'au dernier numéro non inclus.

```
for i in range(0,10):
  print(i)
```

10. range()

Par défaut, range retourne des nombres en ordre croissant avec un pas égal à 1. Pourtant, il est possible de changer ce pas en passant le troisème argument, un entier. Testez le code suivant en utilisant différentes valeurs pour le pas.

```
for i in range(0,10,2):
  print(i)
```

10. range()

Il est possible de changer l'affichage sur chaque ligne de print, en modifiant l'argument end de print. Ci-dessous, nous demanderons à *print* d'utiliser une espace au lieu d'un saut de ligne. Voyez le résultat. Testez le code en utilisant différentes valeurs pour end, par exemple, une virgule.

```
for i in range(0,10,2):
  print(i, end=' ')
```

10. range()

Afin d'obtenir les nombres en ordre décroissant, vous pouvez utiliser un numéro de départ supérieur au dernier numéro et avec pas négatif.

```
for i in range(10,0,-2):
  print(i)
```

11. split()

Testez le code suivant.

```
for i in "Bonjour le monde!":
  print(i)
```

11. split()

Comparez la différence entre le code ci-dessous et le code au-dessus.

```
for i in "Bonjour le monde!".split():
  print(i)
```

11. split()

La méthode *split* est très utile pour extraire des mots dans un texte. split (par défaut) extraire des mots d'une texte en utilisant l'espace, le caractère de tabulation ou le caractère de rétour chariot comme séparateur.

```
for i in "Bonjour,le,monde!".split():
  print(i)
```

11. split()

Vous pouvez changer le comportement de cette méthode en passant le caractère de séparation desirée comme un argument. Ci-dessous, nous passerons le caractère virgule.

```
for i in "Bonjour,le,monde!".split(","):
  print(i)
```

12. Tri

Lorque nous travaillons avec les chiffres, les fonctions plus démandées sont les possibilités de trier les valeurs en ordre croissant ou décroissant, obtenir les valeurs maximum et minimum dans une liste, les n prémières/dernières valeurs etc.

La méthode *sort* est capable de trier une liste en ordre croissant (par défaut).

```
num = [10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
num.sort()
print(num)
```

12. Tri

Tri en ordre décroissant

Afin de trier une liste en ordre décroissant, utilisez la méthode *sort* en passant la valeur pour l'argument reverse. Si reverse est égale à True, la méthode sort trie la liste en ordre décroissant.

```
num = [10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
num.sort(reverse=True)
print(num)
```

12. Tri

Valeur minimum

Pour avoir la valeur minimum d'une liste, utilisez la fonction intégrée *min*.

```
num = [10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
print(min(num))
```

12. Tri

Valeur maximum

Pour avoir la valeur maximum d'une liste, utilisez la fonction intégrée *max*.

```
num = [10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
print(max(num))
```

12. Tri

n grandes/petites valeurs

Pour récupérer les n grandes/petites valeurs d'une liste, nous pouvons réutiliser la méthode sort et les index comme indiqué suivant où l'on affiche les 2 grandes et les 2 petites valeurs de la liste.

```
num = [10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
num.sort(reverse=True)
print(num[:2])
print(num[-2:])
```

12. Tri

sorted()

Un inconvenient de la méthode sort est qu'elle écrasse la liste initiale en triant les valeurs de la liste et remplacant la liste avec la liste triée. La fonction sorted évite cet inconvenient en créant et retournant une nouvelle liste. Testez le code suivant et vérifiez la différence entre les deux approches.

```
num = [70, 20, 30, 10, 50, 60, 20, 80, 70, 50]
numtri = sorted(num,reverse=True)
print(num)
print(numtri)
```

12. Tri

Vous pouvez également utiliser la fonction sorted pour travailler avec des chaînes de caractères. Le code ci-dessous extrait les mots d'une chaine de caractères et ensuite, trie les mots par ordre alphabétique inverse. Rémarquez bien l'argument key qui assure le tri des mots en considérant et convertissant les caractères en miniscule.

```
print(sorted("Bonjour le monde!".split(), key=str.lower,
reverse=True))
```

13. Dictionnaires

Parfois, nous aimerons bien associer une text à une valeur au lieu d'un index; par exemple, associer une adresse à un nom. Les dictionnaires en Python nous donnent cette possibilité d'associer une clé à une valeur. Voyez le code ci-dessous et regardez bien comment nous avons associé 3 clés différents aux 3 valeurs différentes.

```
a = {"contente": 12, "content": 12, "triste": 2}
print(a)
print(type(a))
```

13. Dictionnaires

Notre prochain objectif est de traverser le dictionnaire, c'est à dire afficher tous les clés-valeurs. Pour faire ça, nous utiliserons une boucle for, comme nous avons vu précédemment. Regardez bien comment nous utilisons le cle au-lieu du index.

```
a = {"contente": 12, "content": 12, "triste": 2}
for cle in a:
  print("la phrase ", cle, " apparait ", a[cle], " fois")
```

13. Dictionnaires

Voici une autre façon d'accéder aux couples clés-valeurs d'un dictionnaire. Cette fois, nous utiliserons la méthode *items* ainsi qu'une boucle for.

```
a = {"contente": 12, "content": 12, "triste": 2}
for cle,valeur in a.items():
print("la phrase ", cle, " apparait ", valeur, " fois")
```

13. Dictionnaires

Pour ajouter ou modifier une clé-valeur, il faut ajouter au dictionnaire la nouvelle couleure.

```
a = {"contente": 12, "content": 12, "triste": 2}
a["joie"] = 10
print(a)
```

Pour supprimer une clé-valeur d'un dictionnaire, utilisez la fonction del comme indiqué ci-dessous.

```
a = {"contente": 12, "content": 12, "triste": 2}
del a["triste"]
print(a)
```

13. Dictionnaires

Attention dans l'exemple on a convertit un dictionnaire en une liste de Tupes de taille 2.

```
mots = {"contente": 12, "content": 12, "triste": 2, "joie" : 10}
mots_tuple = [(cle, valeur) for cle,valeur in mots.items()]
print(mots_tuple)
```

13. Dictionnaires

itemgetter

Voyez le code ci-dessous pour trier les cles-valeurs du dictionnaire.

```
mots = {"contente": 12, "content": 12, "triste": 2, "joie" : 10}
print(sorted(mots))
```

13. Dictionnaires

Si vous voulez trier le dictionnaire en utilisant les valeurs, vous pouvez utilisez la méthode itemgetter. L'idée derrière le code ci-dessous est d'utiliser le deuxième membre du tuple pour le tri.

```
from operator import itemgetter

mots = {"contente": 12, "content": 12, "triste": 2, "joie" : 10}
mots_tuple = [(cle, valeur) for cle,valeur in mots.items()]
print(sorted(mots_tuple, key=itemgetter(1)))
```

14. Interaction avec l'utilisateur

Jusque-là, nous avons affiché les résultats et les messages et jamais interagit avec l'utilisateur. Notre prochain objectif est d'interagir avec les utilisateurs afin de leur demander les informations nécessaires. Nous commençons en demandant le nom de l'utilisateur.

```
nom = input("Quel est votre nom?")
print(nom)
```

14. Interaction avec l'utilisateur

Maintenant, nous demanderons son âge. Voyez le code ci-dessous et vous verrez que le type de la valeur stockée dans la variable âge est une chaine de caractères (str), même si l'utilisateur a saisi un entier.

```
age = input("Quel est votre âge? ")
print(age)
print(type(age))
```

14. Interaction avec l'utilisateur

Donc, avant d'utiliser la valeur age, nous convertisserons l'age en entier en utilisant la fonction *int*.

```
age = input("Quel est votre âge? ")
age = int(age)
print(age)
print(type(age))
```

15. Manipulation d'un fichier

L'objectif du code ci-dessous est d'ouvrir un fichier en mode écriture et écrire un message et enfin fermez le fichier.

```
message = "Bonjour le monde"
with open("bonjour.txt", "w") as file:
  file.write(message)
file.close()
```

15. Manipulation d'un fichier

L'objectif du code ci-dessous est d'ouvrir un fichier en mode lecture et lire le fichier entier et le firmer à la fin.

```
with open("bonjour.txt", "r") as file:
  text = file.read()
  print(text)
file.close()
```

15. Manipulation d'un fichier

Ci-dessous sont deux codes qui écrivent deux messages dans un fichier. Comparez les deux codes et les résultats.

Approche 1

```
message1 = "Bonjour le monde"
message2 = "Programmation en Python"
with open("bonjour.txt", "w") as file:
  file.write(message1)
  file.write(message2)
file.close()

with open("bonjour.txt", "r") as file:
  text = file.read()
  print(text)
file.close()
```

15. Manipulation d'un fichier

Approche 2

```
message1 = "Bonjour le monde\n"
message2 = "Programmation en Python"
with open("bonjour.txt", "w") as file:
  file.write(message1)
  file.write(message2)
file.close()

with open("bonjour.txt", "r") as file:
  text = file.read()
  print(text)
file.close()
```

15. Manipulation d'un fichier

readline()

Il est intéressant de lire un fichier texte ligne par ligne surtout quand le fichier est très volumineux. Vous pouvez utiliser la méthode *readline* ou juste une boucle for.

15. Manipulation d'un fichier

readline()

Approche 1

```
message1 = "Bonjour le monde\n"
message2 = "Programmation en Python"
with open("bonjour.txt", "w") as file:
  file.write(message1)
  file.write(message2)
file.close()

with open("bonjour.txt", "r") as file:
  text = file.readline()
  print(text)
file.close()
```

15. Manipulation d'un fichier

Approche 2

```
message1 = "Bonjour le monde\n"
message2 = "Programmation en Python\n"
with open("bonjour.txt", "w") as file:
  file.write(message1)
  file.write(message2)
file.close()

with open("bonjour.txt", "r") as file:
  for line in file:
    print(line)
file.close()
```

16. Lecture d'un dossier

Vous pouvez utiliser la fonction *listdir* pour lire toutes les entrées d'un répertoire, comme tous les fichiers, sous-répertoires, etc.

```
from os import listdir

for f in listdir("./"):
print(f)
```

17. L'interface en ligne de commande

Vous pouvez également passer des arguments à votre programme à partir de l'interface de ligne de commande.

```
import argparse

parser = argparse.ArgumentParser(description='opérations
mathématiques.')
parser.add_argument('entiers', metavar='N', type=int, nargs=2,
    help='entier')
parser.add_argument('--operation', choices=['addition',
'soustraction'])
args = parser.parse_args()

print(args.operation)
print(args.entiers)
```

17. L'interface en ligne de commande

Grâce au programme précédent, vous pouvez afficher un message pour comprendre les objectifs de votre programme.

```
$ python3 commandline.py --help
usage: commandline.py [-h] [--operation {addition,soustraction}] N N

opérations mathématiques.

positional arguments:
N entier

optional arguments:
-h, --help show this help message and exit
--operation {addition,soustraction}
```

17. L'interface en ligne de commande

Si l'utilisateur n'entre pas le nombre correct d'arguments, le programme affiche un message d'erreur et indique également le style d'utilisation de votre programme.

```
$ python3 commandline.py
usage: commandline.py [-h] [--operation {addition,soustraction}] N N
commandline.py: error: the following arguments are required: N
```

17. L'interface en ligne de commande

Lorsque l'utilisateur entre les arguments correctement, nous pouvons facilement accéder à ces valeurs grâce au 'parser'.

```
$ python3 commandline.py 12 23 --operation soustraction
soustraction
[12, 23]
```

18. Fonctions

Vous pouvez grouper un certain nombre d'instructions et créer une fonction avec zéro ou plusieurs arguments.

```
def addition(num1, num2):
  return num1 + num2

def soustraction(num1, num2):
  return num1 - num2

print(addition(23, 34))
print(soustraction(23, 34))
```

18. Fonctions

Vous pouvez également spécifier des paramètres optionnels pour une fonction

```
def increment(num1, compteur=1):
  return num1+compteur

print(increment(12))
print(increment(12,2))
```

19. Classes

Vous pouvez regrouper un certain nombre de fonctions en créant une classe

```
class Calcul:
   def addition(num1, num2):
     return num1 + num2

   def soustraction(num1, num2):
     return num1 - num2

print(Calcul.addition(23, 34))
print(Calcul.soustraction(23, 34))
```

19. Classes

Vous pouvez importer certaines fonctions de vos classes.

```
from Calcul import addition,soustraction

print(addition(23, 34))
print(soustraction(23, 34))
```

19. Classes

Les classes sont généralement créées pour regrouper les données et les fonctions associées.
Par exemple, nous pouvons créer une classe d'étudiants

```
class Etudiant:
   def __init__(self, nom, adresse, notes):
     self.nom = nom
     self.adresse = adresse
     self.notes = notes

   def affiche(self):
     print("Nom: " + self.nom)
     print("Adresse: " + self.nom)
     print("Notes: " + str(self.notes))

pierre = Etudiant("Pierre", "Lyon", [18, 19])
pierre.affiche()
```

19. Classes

Python permet l'héritage de classe. Contrairement à Java, Python n'autorise pas les membres privés et protégés. Pour plus de lisibilité, vous pouvez nommer vos variables privées en commençant par __.

```
class EtudiantIRC(Etudiant):
   def __init__(self, nom, adresse, notes):
     Etudiant.__init__(self,nom, adresse, notes)

   def affiche(self):
     Etudiant.affiche(self)


pierre = EtudiantIRC("Pierre", "Lyon", [18, 19])
pierre.affiche()
print(type(pierre))
```

19. Classes

Python permet l'héritage de classe.

```
class Salarié:
  def __init__(self, salaire):
     self.salaire = salaire

   def affichage(self):
     print("Salaire: " + str(self.salaire))
```

19. Classes

Python permet également l'héritage multiple.

```
class EtudiantIRC(Etudiant,Salarié):
   def __init__(self, nom, adresse, notes, salaire):
     Etudiant.__init__(self,nom, adresse, notes)
     Salarié.__init__(self,salaire)

   def affichage(self):
     Etudiant.affichage(self)
     Salarié.affichage(self)


pierre = EtudiantIRC("Pierre", "Lyon", [18, 19],1)
pierre.affichage()
print(type(pierre))
```

Références

1. https://www.python.org/
2. https://fr.wikipedia.org/wiki/Python_(langage)
3. https://docs.python.org/3/