PYTHON 3.8

#!/usr/bin/python3.8 *-> en debut de fichier*
-*-coding:Utf-8 -*

« « « commentaire *#commentaire inline*
sur plusieurs lignes » » »

FROM module/package **IMPORT** fonction **AS** ft

import **math** / **random** ->operation math.degrees/radians -> convertion
from **fractions** import **Fraction**
-> permet de manipuler des fractions : qd les floats sont mal représentés par ex

import **time / datetime** -> exprimer le temps
time.time() .localtime() .mktime() .sleep(3.5) .strftime(%Y %d %M)

import **re** -> re.**match**(regex, chaine) re.**search**(regex, chaine) re.**sub**(match, replace, chaine) re.**compile**(regex_to_save)

import **pickle** -> pour écrire/lire objet ds les fichiers
import **hashlib** -> hashage hashlib.algorithms_available
hashlib.sha1(b'chaine').hexidigest
import **getpass** **->** getpass(«tapez votre mdp : »)

import **sys** -> sys.stdin/stdout/stderr sys.**argv**
import **os** -> gérer fichier write open close
os.system(« ls ») **os.popen**(« ls »)
import **signal** -> intercepter des signaux dans le code

import **django**

— — OPERATOR — —

==, <, >, >=, <=, !=
+
+=
-
-=
/
/=
// return partie entière
//=
%
%=
*****
***=**
**** puissance
****=**
@ matrix ???

— BIT OPERATOR — —

&
&=
<<
<<=
>>
>>=
^
^=
|
|=
~

— — — — — — KEYWORDS — — — — — —

FOR ... IN: -> boucle (FOR index, value IN enumerate(list))
IF ... IN ... **-> boucle**

I**F ELIF ELSE**:
OR AND NOT
IS compare les adresses
True / False / NONE

With ... as
```
with open(file.txt, a) as fd:
    fd.write(''je rajoute cette ligne'')
```
WHILE -> boucle
break / continue

try / except (essaye l'instruction ds **try** et execute celle ds **except** si l'exception est interceptée)

```
try:
    var = str(input(''comment ça va?''))
except TypeError :
    print(''ce n'est pas une réponse ça valable'')
except Exception :
    pass
```

Raise TypeError (lève une exception)

finally (execute dans tout les cas)
Return (on peut return plusieurs valeurs séparées dune virgule -> renvoie un tuple)
await -> suspend l'execution
pass (ne fait rien, lorsque l'on attends une instruction)
del -> supprime une variable

class -> class declaration
def -> fonction declaration
async def -> coroutine definition
yield -> pour définir une fonction generator
lambda -> fonction declaration
ft_mult = lambda x : x * 10 if x % 5 == 0
ft_mult(5) = 50

global précède une var ds le corps d'une fonction :
précise que celle-ci est globale, en dehors du corps de la fonction
et permet son accès en lecture et écriture
/ nonlocal ...
assert -> vérifie une condition sinon stop l'execution
async for/with -> ???

LIST

Les listes sont des séquences « mutables »
LIST = [a, b]

new_list = list() ou []
enumerate(list) = prend chaque élément de list et lui donne un index de 0 a ++ -> renvoie des **tuples**

del new_list[:] -> supprime tout les éléments de la liste
list(variable) -> transform variable en list
list[2] = value a l'index 2
list[:2] = values de l'index 0 a 2 exclu
list[-1] = dernier element
liste[start:stop:step]
-> liste[0:11:3] = [du 1er au 10e truc dans liste, de 3 en 3]
-> liste[::-2] = du dernier au 1er, de 2 en 2

list.**index**(''c'') = trouve l'index de ''c''

list.**append**(c)   = ajoute c à la fin de la liste
list.**insert**(1, ''c'') = insert c à l'index 1 de la liste
list**.extend**(list2)   = append toutes les valuers de list2 à la fin de la list

zip(liste1, liste2)   = prend 2 à 2 les éléments de liste1 et liste2 et fait une new liste
(marche pour plus que 2 listes)

list.**remove**(item) = supprime l'item
list.**pop**(index) = return l'item à l'index désigné et le super de list
del(list[1]) = suppr l'item à l'index 1 dans list
del(a) = suppr le 1er element rencontré dont la valeur est a

list.**sort**() || **sorted**(list) qui return une copy triée
sorted(list, **key**=lambda elem: elem[2]) —> trie une liste de liste/tuple en fonction de la 3e case
sorted(list, **key**=lambda elem: elem.attribut) —> trie une liste de objet en fonction d'un attribut
sorted(list, **key**=fonction, **reverse**=True) param optionnel attendant un bool

plus rapide: le module **operator**
attrgetter peut prendre plusieurs paramètres:

```
from operator import itemgetter
sorted(etudiants, key=itemgetter(2))
```

```
from operator import attrgetter
sorted(etudiants, key=attrgetter("moyenne"))
```

list.**reverse**()
filter(lambda x: x%3 == 0, list) = filtre les nombres divisible par 3 dans list
map(fontcion ou lambda, liste/chaine/dictionnaire) = applique la fonction sur chaque element du 2e param
reduce(fonction ou lambda, liste/chaine/dictionnaire) =
la fonction reçoit au premier appel les deux 1er elem de l'itérable,
puis l'élément suivant (le 3e) et son propre retour l'appel précédent

— —

COMPRÉHENSION DE LISTE
Return une liste modifiée/filtrée :

pycon

```
>>> liste_origine = [0, 1, 2, 3, 4, 5]
>>> [nb * nb for nb in liste_origine]
[0, 1, 4, 9, 16, 25]
>>>
```

```
# On change le sens de l'inventaire, la quantité avant le nom
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit,qtt in inventaire]
# On trie l'inventaire inversé dans l'ordre décroissant
inventaire_inverse.sort(reverse=True)
# Et on reconstitue l'inventaire
inventaire = [(nom_fruit, qtt) for qtt,nom_fruit in inventaire_inverse]
```

pycon

```
>>> liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [nb for nb in liste_origine if nb % 2 == 0]
[2, 4, 6, 8, 10]
>>>
```

DICTIONNAIRE

new_dict = { key: value, key2:value2, ...}
new_dict_vide = dict() ou {}
new_dict[« key »] = value >>> ajoute une pair key/value ou écrase la value si la key existe deja.

.**items**() = return les key/value sous forme d'une liste de tuples (dans un ordre aléatoire)
.**keys** = return une liste des keys
.**values** = return une liste des values
.**pop**(key) = suppr et return la value **del** dictionnaire[key] = delete the key/value

tuple *fonction*
on peut mettre n'importe quel **type** de donnée en **key** et **valeur.** ex -> dict[(''Arthur'', 26)] = ft_putchar

for key in dict parcours des keys
for key in dict.values() parcours des values
for key, valeur in dict les deux
if 21 in dict.values dans une condition

POINTEUR SUR FONCTION:

pycon

```
>>> def fete():
...     print("C'est la fête.")
...
>>> def oiseau():
...     print("Fais comme l'oiseau...")
...
>>> fonctions = {}
>>> fonctions["fete"] = fete # on ne met pas les parenthèses
>>> fonctions["oiseau"] = oiseau
>>> fonctions["oiseau"]
<function oiseau at 0x00BA5198>
>>> fonctions["oiseau"]() # on essaye de l'appeler
Fais comme l'oiseau...
>>>
```

RECUPERER les paramètres nommés

pycon

```
>>> def fonction_inconnue(**parametres_nommes):
...     """Fonction permettant de voir comment récupérer les paramètres nommés
...     dans un dictionnaire"""
...
...
...     print("J'ai reçu en paramètres nommés : {}.".format(parametres_nommes))
...
>>> fonction_inconnue() # Aucun paramètre
J'ai reçu en paramètres nommés : {}
>>> fonction_inconnue(p=4, j=8)
J'ai reçu en paramètres nommés : {'p': 4, 'j': 8}
>>>
```

Pour capturer tous les paramètres nommés non précisés dans un dictionnaire, il faut mettre deux étoiles **** avant le nom du paramètre.

Si vous passez des paramètres non nommés à cette fonction, Python lèvera une exception.

Ainsi, pour avoir une fonction qui accepte n'importe quel type de paramètres, nommés ou non, dans n'importe quel ordre, dans n'importe quelle quantité, il faut la déclarer de cette manière :

python

```
def fonction_inconnue(*en_liste, **en_dictionnaire):
```

TUPLE

les tuples sont des séquences « imutables »

TUPLE = (a, b)

a, b = tuple -> l'opération inverse est possible

new_tuple_vide = () ou **tuple**()

new_tuple_rempli = (1,)

other_tuple_rempli = (1, 2, 3)

STRINGS

on enferme une chaine de caractère entre simple/double/triple quote : ‘ ou ‘‘ ou ‘‘‘‘‘
str() transforme en string

.**split**(‘‘ ‘‘) -> renvoie une list avec les mots
‘‘ ‘‘.**join**(liste) = return une str composée des éléments de liste join avec ‘‘ ‘‘
‘‘je m’appelle {0} et j’ai {1} ans’’.**format**(‘‘Arthur’’, age)
chaine2 += str(42) + « fois le tour du{}\n».format(« monde.»)
.**replace**
‘chaine’.encode() ou b’’chaine’’
‘chaine’.**decode**
.**upper**() .**lower**()
.**capitalize**() -> 1ère lettre en Maj
.isalpha() .isalnum() .isdigit() .isdecimal()...
.**center**(20)
.rjust()
...
len() longueur

raw_input(‘‘) = return une string

FONCTION

def fonction(arg_a, arg_b): ou (*a) quand fonction variadique (en liste)
‘‘ ‘‘ ‘‘ description ‘‘ ‘‘ ‘‘ ou (**a) param nommés -> en dictionnaire
  conditions:
    instructions

lambda x: condition >>> return x sous condition
lambda x: x = ‘‘a’’
return true si x vaut ‘‘a’’

— ex

—

type() return type
tuple list str int float dict class
float() int() tuple() list() dict() str()
dir() renvoie tout ce qui est utilisé
bin() prend un integer et rend une string en nbr binaire. ex = bin(2)= 0b10
input(''Comment vous appelez vous?'') -> return l'input de l'utilisateur

abs() valeur absolue **round**(x , n) = arrondit x au n chiffre après la virgule **exp**() exponetielle
sum() fait la somme **sqrt**( ) racine carré
pow() puissance
max() trouve le nbr max **min**() trouve le nbr min
math.**floor**(3.6) renvoie l'entier sup math.**ceil**(3.6) renvoie l'entier inf math.**trunc**(3.6) renvoie la partie entière

from **random import ***
randrange(10) random nbr ds le range :

range(stop) =	génère une liste de 0 à stop exclu
range(start, stop) =	génère une liste de start à stop exclu
range(start, stop, step) =	idem + valeur de la step qui par default 1
randint(1, 6)	-> random en tee 1 et 6
choice(list)	-> return une des valeurs de la liste
shuffle(list)	-> mélange la liste
sample(list, 5)	-> prend 5 éléments aléatoires ds list
random()	-> return float random

— ex
—

— — — — — FICHIER — — — — —

IMPORT **os** (*module pour gérer les fichiers : créer suppr copy...*)
os.**getcwd**() -> return pwd

fd = **open**('path/file.txt', r) fd.**close**() buffer = fd.**read**()
r -> read || rb *en binaire* buffer.split('\n')
w -> écrase || wb *en binaire* fd.closed est true if fichier bien close

a -> ajoute || ab *en binaire* fd.**write**(«ça écrit»)

import **pickle** (module pour encode des objets)
with open(fichier.py, ‘wb’) as fd:
encode = pickle.**Pickler**(fd)
encode.**dump**(dictionnaire) -> ecrire le dict

decode = pickle.**Unpickler**(fd)
récupérer = decode.**load**() -> récupérer l’objet

```
if os.path.exists(nom_fichier_scores): # Le fichier existe
    # On le récupère
    fichier_scores = open(nom_fichier_scores, "rb")
    mon_depickler = pickle.Unpickler(fichier_scores)
    scores = mon_depickler.load()
    fichier_scores.close()
else: # Le fichier n'existe pas
    scores = {}
return scores
```

CLASS

class MaClasseFille(MaClasseMere) -> a(b, c) a hérite de b et de c
l’orde de l’héritage compte et donne l’ordre de recherche des méthodes
issubclass(subclass, class) return true/false
insistance(instance, class) return true/false

class MaClass:
’’’’’’ceci est ma classe’’’’’’
attribut_de_classe = 0
def **__init__(self,** nom, age): —> méthode spéciales : __method__
self . nom = nom
self . age = age
self . _attribut_private —> _ devant le nom : convention -> faire des geter/seter
MaClasse.attribut_de_classe += age

def Ch_Des(self, description): —> méthode d’instance : self
self . description = description

def Age_Total(**cls**): —> méthode de classe : cls
return cls . attribut_de_classe

a_private = **property(**_get_a_private, _set_a_private**)**

__dict__ —> attribut spécial , dictionnaire par défaut ds chaque objet contenant nom_attribut : valeur_attribut
new = MaClasse(''arthur'', 34) -> instancie un objet
new . __dict__[« nom »] = ''hortense'' == new.nom = ''hortense''
new.Age_Total() == MaClasse.Age_Total() on peut appeler une méthode de classe depuis une instance
hasattr(new, "nom") # Renvoie True si l'attribut "nom" existe dans l'instance new

méthode special
— — — —— — — —— — — —— — — —— — — — viennent de la classe mère objet
__init__ __del__
def **__repr__**(self) pour redefinir la representation de l'instance
return «MaClass: nom '{}' age '{}'».format(self.nom, self.age)
def **__str__**(self)
return ''la chaine a return qd on veut faire appel a **print**() sur notre objet''
def **__getattr__**(self, nom) ->> fonction appelée qd on essaye d'accéder a un attribut qui n'existe pas
print(''pas d'atribut {} ici''.format(nom))
def **__setattr__**(self, nom_attr, val_attr):
"""Méthode appelée quand un attribut est set. On se charge d'enregistrer l'objet"""
object . __setattr__(self, nom_attr, val_attr) objet —> classe mère
self.enregistrer()
def **__delattr__**(self, nom) ->> fonction appelée qd on essaye de suppr un attribut
objet . __delattr__(self, nom)
print(''suppression de l'atribut {} OK''.format(nom))
__getitem__ **__setitem__** **__delitem__** **__contains__** pour redefinir qd on fait
objet[a] / objet[a]= / del objet[a] / value in objet

__len__() quand on fait len(conteneur)

__add__ __sub__ __truediv__ __floordiv__ __mod__ __mul__ __pow__ . . . pour redefinir les operations sur l'objet
(objet + ... , objet - ... , objet / ... , objet // ... , objet % ... , objet * ... , objet ** ...)
attention au sens de l'operation sinon définir **__r**add__ pour : ... + objet
__iadd__ pour +=
__iter__ et __next__ qui redéfinissent quand on fait un **for in** sur l'objet ou iterator = **iter**(objet) puis des **next**(iterator)
__getstate__ et **__setstate__** pour agir avant la sérialisation / après la désérialisation de l'objet

```
def __setattr__(self, nom_attribut, valeur_attribut):
        """Méthode appelée quand on fait objet.attribut = valeur"""
        print("Attention, on modifie l'attribut {0} de l'objet !".format(nom_attribut))
        object.__setattr__(self, nom_attribut, valeur_attribut)
```

Opérateur	Méthode spéciale	Résumé
`==`	`def __eq__(self, objet_a_comparer):`	Opérateur d'égalité (*equal*). Renvoie `True` si `self` et `objet_a_comparer` sont égaux, `False` sinon.
`!=`	`def __ne__(self, objet_a_comparer):`	Différent de (*non equal*). Renvoie `True` si `self` et `objet_a_comparer` sont différents, `False` sinon.
`>`	`def __gt__(self, objet_a_comparer):`	Teste si `self` est strictement supérieur (*greater than*) à `objet_a_comparer`.
`>=`	`def __ge__(self, objet_a_comparer):`	Teste si `self` est supérieur ou égal (*greater or equal*) à `objet_a_comparer`.
`<`	`def __lt__(self, objet_a_comparer):`	Teste si `self` est strictement inférieur (*lower than*) à `objet_a_comparer`.
`<=`	`def __le__(self, objet_a_comparer):`	Teste si `self` est inférieur ou égal (*lower or equal*) à `objet_a_comparer`.

— — — ——— — — ——— — — ——— — — ——— — — ——— — — ——— — — —

PROPRIETES(property)

new.a_private = 8 —> appel _set_a_private() via la propriété a_private
les 4 méthodes optionnelles a définir dans property(...)
1 -> qui return la value
2 -> qui set objet.attribut
3 -> qui est appelée quand ont fait del objet.attribut

4 -> qui print un message quand on fait help(objet.attribut)

- la méthode donnant accès à l'attribut ;
- la méthode modifiant l'attribut ;
- la méthode appelée quand on souhaite supprimer l'attribut ;
- la méthode appelée quand on demande de l'aide sur l'attribut.

Je vous propose d'essayer de créer une classe dynamiquement, sans passer par le mot-clé `class` mais par la classe `type` directement.

La classe `type` prend trois arguments pour se construire :

- le nom de la classe à créer ;
- un **tuple** contenant les classes dont notre nouvelle classe va hériter ;
- un dictionnaire contenant les attributs et méthodes de notre classe.

pycon

```
1 >>> Personne = type("Personne", (), {})
```

SOCKETS

```
import socket
```

import socketserver

SERVEUR :

`socket.AF_INET` : la famille d'adresses, ici ce sont des adresses Internet ;
`socket.SOCK_STREAM` : le type du socket, `SOCK_STREAM` pour le protocole TCP.

```
>>> connexion_principale = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>>
```

```
1 >>> connexion_principale.bind(('', 12800))
```

nom_hote, port -> ici '' , 12800

```
>>> connexion_principale.listen(5)
```

Il est important de noter que la méthode `accept` renvoie deux informations :

- le socket connecté qui vient de se créer, celui qui va nous permettre de dialoguer avec notre client tout juste connecté ;
- un tuple représentant l'adresse IP et le port de connexion du client.

```
>>> connexion_avec_client, infos_connexion = connexion_principale.accept()
```

```
connexion_avec_client.send(b"Je viens d'accepter la connexion")
```

CLIENT:

```
>>> connexion_avec_serveur.connect(('localhost', 12800))
```

```
>>> connexion_avec_serveur.connect(('localhost', 12800))
```

```
msg_recu = connexion_avec_serveur.recv(1024)
```

connexion.close()

```
import socket

hote = ''
port = 12800

connexion_principale = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connexion_principale.bind((hote, port))
connexion_principale.listen(5)
print("Le serveur écoute à présent sur le port {}".format(port))

connexion_avec_client, infos_connexion = connexion_principale.accept()

msg_recu = b""
while msg_recu != b"fin":
    msg_recu = connexion_avec_client.recv(1024)
    # L'instruction ci-dessous peut lever une exception si le message
    # Réceptionné comporte des accents
    print(msg_recu.decode())
    connexion_avec_client.send(b"5 / 5")

print("Fermeture de la connexion")
connexion_avec_client.close()
connexion_principale.close()
```

import **select**

```
rlist, wlist, xlist = select.select(clients_connectes, [], [], 2)
```

rlist -> list de connexion client attendant d'etre read

wlist -> list de connexion client attendant d’etre write
xlist -> list de connexion client attendant une erreur
4e param : timeout -> temps avant que select.**select**() renvoie ces contenus