Lycée d'enseignement général
et technologique international Victor Hugo
COLOMIERS

NSI
NUMÉRIQUE ET SCIENCES INFORMATIQUES

Langages et programmation

Cours & Activité Pratique

Nom : Prénom : Date :

Table des matières

Langages et programmation

Cours & Activité Pratique

1 Programme et donnée

Un programme est une donnée comme une autre. A l'instar des autres données, certains programmes utilisent d'autres programmes comme ils le font avec des variables par exemple.

Dans un premier temps, nous allons écrire des programmes en partant de leur algorithme. Puis nous écrirons un programme qui utilise les premiers comme des données quelconques.

1.1 Le calcul du PGCD

On se propose d'écrire un programme de calcul du Plus Grand Commun Diviseur de deux entiers a et b à partir de l'algorithme d'Euclide.

Qui était **Euclide** ?

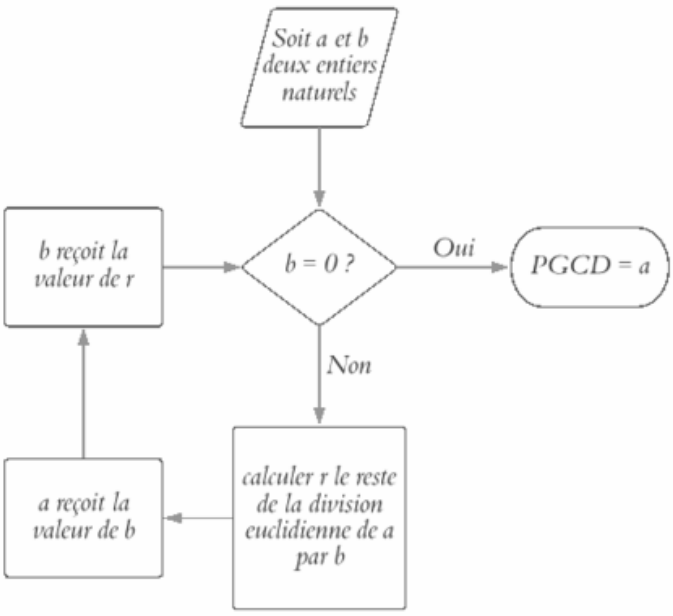
Euclide (en grec ancien Εὐκλείδης / Eukleídês) est un mathématicien de la Grèce antique, auteur d'éléments de mathématiques, qui constituent l'un des textes fondateurs de cette discipline en Occident. Aucune information fiable n'est parvenue sur la vie ou la mort d'Euclide ; il est possible qu'il ait vécu vers 300 avant notre ère.

Son ouvrage le plus célèbre, les Éléments, est un des plus anciens traités connus présentant de manière systématique, à partir d'axiomes et de postulats, un large ensemble de théorèmes accompagnés de leurs démonstrations. Il porte sur la géométrie, tant plane que solide, et l'arithmétique théorique. L'ouvrage a connu des centaines d'éditions en toutes langues et ses thèmes restent à la base de l'enseignement des mathématiques au niveau secondaire dans de nombreux pays.

Du nom d'Euclide, dérivent en particulier l'algorithme d'Euclide, la géométrie euclidienne (et non euclidienne), la division euclidienne.

A faire vous-même I

Selon l'organigramme ci-contre, on pose la question "est-ce que b = 0 ?". Si oui, le « pgcd » est égal à « a ». Sinon, on part dans la boucle « Non ». On revient à la question avec les nouvelles valeurs de a et b.

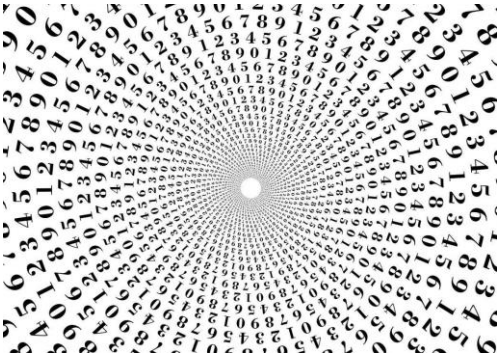


A partir de cet organigramme ci-dessous, écrire le programme de calcul du PGCD en langage Python. Vous utiliserez l'environnement de développement Python 3 (IDLE)

Consigne :

- Libellez votre programme « pgcd.py »
- Faites un en-tête de programme
- Commentez votre code
- N'utilisez pas de fonction

1.2 Le calcul du PPCM



Le PPCM ou Plus Petit Commun Multiple de 2 entiers est par définition l'entier non nul le plus petit possible qui soit à la fois multiple des 2 entiers.

A faire vous-même II

a. On se propose maintenant d'écrire un programme de calcul du Plus Petit Commun Multiple de deux entiers a et b à partir de l'algorithme suivant :

```
# Début algorithme

# Soient a et b deux entiers naturels
#Initialisation des variables et constantes
#       a = 0
#       b = 0
#       c = 0
#       d = 0
# *** code ***


# Présentation du programme
# *** code ***

# Saut de ligne
# *** code ***

# Lire a
# *** code ***

# Lire b
# *** code ***

# Copie des variables a et b dans c et d
# *** code ***

# Boucle de calcul du PGCD
# TANT_QUE a != b FAIRE
# *** code
#...
#...
#      code ***

# Saut de ligne
# *** code ***

# Ecrire le PPCM
# *** code ***

# Fin algorithme
```

Aide et Consignes :

- Libellez votre programme « ppcm.py »
- Faites un en-tête de programme (libellé et rôle)
- N'utilisez pas de fonction
- Utilisez les emplacements signalés (*** code *** et *** code … code ***) pour saisir vos instructions

b. Le PPCM est donné par le rapport du produit des 2 entiers donnés et de leur PGCD (non nul). On obtient la formule suivante PPCM (a,b) = a.b / PGCD (a,b). Exploitez cette propriété afin d'écrire une deuxième version du programme de calcul du PPCM. Vous libellerez votre programme « ppcm_rapport.py »

1.3 Le calcul de « Factorielle(n) »

En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n. Cette opération est notée avec un point d'exclamation, n!, ce qui se lit soit « factorielle de n », soit « factorielle n » soit « n factorielle ». *(Wikipédia)*

A faire vous-même III

On se propose maintenant d'écrire un programme de la factorielle d'un entier naturel n à partir de l'algorithme suivant :

```
# Début algorithme

# Soit n un entier naturel
# Rappel 0! = 1
# Initialisation des variables et constantes
#       n = 0
#       i = 0
#       factorielle = 1
#*** code ***

# Présentation du programme
#*** code ***

# Saut de ligne
#*** code ***

# Lire n
#*** code ***

# Si n non nul alors
#*** code ***

    # Boucle de calcul de factorielle(n)
    # Pour ...
    #*** code
    #....
    #....
    #    code ***

# Ecrire le résultat
#*** code ***

#Fin algorithme
```

Aide et Consignes :

- Libellez votre programme « factorielle_n.py »
- Faites un en-tête de programme (libellé et rôle)
- N'utilisez pas de fonction
- Utilisez les emplacements signalés (*** code *** et *** code … code ***) pour saisir vos instructions

1.4 Programme ou donnée ?

Imaginons que nous voulions calculer :

- Le PGCD de deux entiers naturels a et b
- Le PPCM de deux entiers naturels a et b
- La factorielle d'un entier naturel n

au sein d'un seul et même programme

Il existe plusieurs solutions à ce problème. Naturellement, nous pensons à réutiliser nos trois programmes déjà écrits. En premier lieu, nous pourrions pensez à « copier – coller » nos programmes dans un seul et même fichier. Cette solution résoudrait notre problème. Mais elle n'est pas satisfaisante car cela revient à dupliquer du code. Entre autres inconvénients liés à une telle solution, citons :

- Une maintenance compliquée
- Une gestion des versions des programmes considérablement alourdie
- L'apparition de multiples sources d'erreur

En revanche, considérons une solution qui consisterait à considérer les programmes déjà écrits (pgcd, ppcm, factorielle) comme des données que l'on pourrait importer depuis un quatrième programme.

A faire vous-même IV

On se propose maintenant d'écrire ce quatrième programme à partir de l'algorithme suivant :

```
# Début algorithme

# Présentation du programme
#*** code
#....
#....
#    code ***

# Saut de ligne
#*** code ***

# Appel du programme pgcd.py
# import pgcd

# Saut de ligne
#*** code ***

# Appel du programme ppcm.py
#*** code ***

# Saut de ligne
#*** code ***

# Appel du programme factorielle_n.py
#*** code ***

#Fin algorithme
```

Aide et Consignes :

- Libellez votre programme « programme_donnee.py »
- Faites un en-tête de programme (libellé et rôle)
- N'utilisez pas de fonction
- Utilisez les emplacements signalés (*** code *** et *** code … code ***) pour saisir vos instructions

A retenir : Certains programmes en utilisent d'autres comme des données externes avec des objectifs parfois différents. Les systèmes d'exploitation (OS) utilisent beaucoup l'appel de programme depuis un autre programme. Par exemple, le gestionnaire de paquets de « Raspbian » utilisent d'autres programmes pour :

- Etablir une liste de mises à jour disponible (commande « update »)
- Faire les mises à jour (commande « upgrade »)
- Installer un nouveau logiciel (commande « install »)
- Etc …

2 La modularité

2.1 Qu'est-ce que c'est ?

La programmation modulaire est le fait de structurer le code source d'un programme informatique en plusieurs fichiers. En effet, lorsqu'on écrit un large programme il peut s'avérer plus convenable de diviser son code source en plusieurs fichiers différents de sorte à pouvoir l'organiser en modules, c'est-à-dire en espaces regroupant des éléments définis par le programmeur (fonctions, type et structures de donnée, objets, …). Il faut noter que ce regroupement ce fait généralement en fonction de l'utilité des éléments.

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'B
lockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildPr
ocessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefuse
dError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsi
s', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFo
undError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError
', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'Inter
ruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'Lookup
Error', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADir
ectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'Overflow
Error', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError
', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', '
RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'Syn
taxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True
', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeEr
ror', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarni
ng', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '__build
_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__',
'__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'by
tearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'e
val', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinsta
nce', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max
', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'pri
nt', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'seta
ttr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 't
ype', 'vars', 'zip']
```

2.2 Retour sur les fonctions

La définition ci-dessus précise que parmi les composants modulaires, il y a les fonctions.

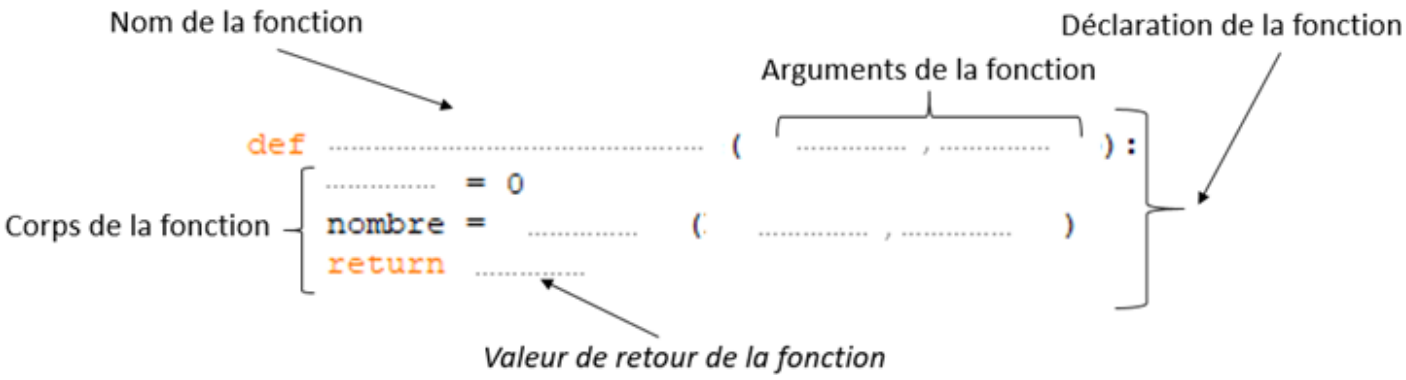
En classe de Première, nous avons appris à structurer du code à l'aide de fonctions. La modularité est donc une bonne opportunité de mobiliser et approfondir nos compétences en la matière.

Définition d'une fonction en Python

L'en-tête d'une fonction se compose de son nom et du nom de ses arguments.

Une fonction ne retourne pas forcément de valeur. Le mot clé « return » ainsi que ***la valeur de retour*** sont donc des informations optionnelles de la déclaration d'une fonction.

Déclaration d'une fonction en Python



Nature du passage d'arguments

Rappel : on peut distinguer deux modes de passage d'arguments :

- par valeur : cela signifie qu'on communique à la fonction, non pas l'entité dans l'appelant, mais seulement sa valeur ; en clair, une copie
- par référence : cela signifie qu'on passe à la fonction une référence à l'argument dans l'appelant, donc essentiellement les deux codes partagent la même mémoire.

En Python, tous les passages de paramètres se font par **référence**. Reportez-vous éventuellement au support intitulé « Tp_Découverte_Python_2019_V1 » afin de pratiquer cette notion.

La portée des variables

La portée d'une variable détermine de quel endroit du code on peut accéder à cette variable. Python utilise ce qu'on appelle la portée lexicale. Cela signifie que la portée d'une variable est déterminée en fonction de l'endroit dans le code où cette variable est définie.

Une variable référencée dans le bloc de code d'une fonction est ce qu'on appelle une variable locale. Lorsque la fonction retourne un résultat, toutes les variables locales de la fonction sont détruites.

Une variable définie en dehors de toute fonction est ce que l'on appelle une variable globale.

Nous avons donc principalement deux catégories de variables :

- les variables locales qui sont définies dans le bloc de code des fonctions
- les variables globales qui sont définies en dehors de toute fonction.

Reportez-vous éventuellement au support intitulé « Tp_Découverte_Python_2019_V1 » afin de pratiquer cette notion.

A faire vous-même V

a. En partant du programme « pgcd.py », écrire le programme de calcul du PGCD de deux entiers naturels a et b qui inclut la fonction « pgcd(x,y) ». Vous libellerez votre programme « pgcd_prog_fct.py ».

b. En partant du programme « ppcm.py », écrire le programme de calcul du PPCM de deux entiers naturels a et b qui inclut la fonction « ppcm(x,y) ». Vous libellerez votre programme « ppcm_prog_fct.py ».

c. En partant du programme « factorielle_n.py », écrire le programme de calcul de la factorielle de l'entier naturel n qui inclut la fonction « factorielle(N) ». Vous libellerez votre programme « factorielle_n_prog_fct.py ».

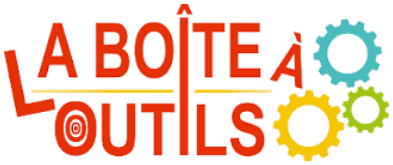
Pourquoi crée-t-on des fonctions ? Car on souhaite:

- Obtenir la meilleure organisation du programme (regrouper les tâches par blocs : lisibilité et maintenance)
- Eviter la redondance (pas de copier/coller maintenance, meilleure réutilisation du code)
- Avoir la possibilité de partager les fonctions (via des modules)

Construire un programme principal le plus simple possible

2.3 Les modules

Qu'est-ce qu'un module en Python ?



Un module en Python, c'est :

- Un fichier « .py »
- Une boite à outils dans laquelle on peut regrouper les fonctions traitant des problèmes de même nature ou manipulant le même type d'objet

Un mode d'emploi :

- Pour charger les fonctions d'un module dans un autre module / programme principal, on utilise la commande « import <nom_du_module> »
- Les fonctions importées sont chargées en mémoire. A ce moment, s'il existe une collision de noms, les plus récentes écrasent les anciennes

```
>>> from math import pi, sin
>>> print("Valeur de Pi :", pi, "sinus(pi/4) :", sin(pi/4))
Valeur de Pi : 3.14159265359 sinus(pi/4) : 0.707106781187
```

Maintenant, nous allons créer un programme qui fait appel à notre propre module.

A faire vous-même VI

a. Réalisation du module : créez un nouveau fichier que vous libellerez « pgcd_ppcm_facto_module.py ».

b. Dans ce fichier, copiez-collez les trois programmes « pgcd_prog_fct.py », « ppcm_prog_fct.py » et « factorielle_n_prog_fct.py ».

c. Modifiez le contenu de « pgcd_ppcm_facto_module.py » de façon à ne garder que les déclarations de fonctions (i.e les blocs « def … : …»)

d. Vérifiez que votre module possède une forme similaire au module suivant :

```
# nom : pgcd_ppcm_facto_module_algo.py
# rôle : Module de fonctions regroupant:
# la fonction pgcd
# la fonction ppcm
# la fonction factorielle

# Début Module

# déclaration de la fonction pgcd
# def pgcd(x,y):
#     ...
#     ...


# déclaration de la fonction ppcm
# def ppcm(x,y):
#     ...
#     ...

# déclaration de la fonction factorielle(N)
# def factorielle(N):
#     ...
#     ...

# Fin module
```

L'importation des modules

Découvrons le mécanisme d'importation des modules sur un exemple.
L'interpréteur de commandes Python est pratique. Nous pouvons également l'utiliser comme une calculatrice ! Admettons que l'on souhaite extraire la racine carrée du nombre « 272 ». La fonction python qui permet une telle opération est « sqrt() » (sq*uare* roo*t*). Elle fait partie du module python « math ».

a. Ouvrons immédiatement notre interpréteur afin de tester cette opération :

```
>>> sqrt(272)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    sqrt(272)
NameError: name 'sqrt' is not defined
```

b. Vérifions que cette fonction fait bien partie du module « math » :

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh'
, 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fm
od', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'is
inf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan'
, 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'ta
u', 'trunc']
```

c. A priori, elle en fait bien partie. Mais pour Python puisse invoquer cette fonction, il faut **importer** le module « math » ainsi :

```
>>> import math
```

d. Puis on peut alors extraire la racine de « 272 » :

```
>>> math.sqrt(272)
16.492422502470642
```

La calculatrice fonctionne bien !

A faire vous-même VII

Accéder à une fonction du module « pgcd_ppcm_facto_module.py » depuis l'interpréteur de commandes.

Si vous voulez structurer vos projets en ayant des fichiers dans différents répertoires, vous aurez parfois des problèmes au moment de l'importation des modules parce que Python ne saura pas où chercher vos modules.

a. Avant d'importer un module « maison », il faut indiquer à l'interpréteur à quel endroit (chemin) il peut le trouver. La commande suivante ajoute un chemin quoi permettra au logiciel Python de trouver le module en question :

```
>>> sys.path.append('/home/pi/Documents/Langages_et_programmation/sources')
```

La fonction « path() » fait partie du module « sys »

Pour connaitre les fonctions proposées par le module « sys », il faut exécuter la fonction native dir() avec l'argument « sys » : >>> dir(sys).
Pour connaitre les spécifications de la fonction « path » incluse dans le module « sys », il faut exécuter la fonction native help() avec l'argument « sys.path » : >>> help(sys.path)

b. Obtenez-vous un message d'erreur ? Pourquoi ? Résolvez le problème puis vérifiez que le chemin qui mène à votre module est correctement renseigné avec la commande :

```
>>> print(sys.path)
```

il faut importer le module sys
"import sys"

c. Dans quelle SDD sont stockés les chemins ?

Dans C:\\

d. Importez le module « pgcd_ppcm_facto_module.py » avec la commande :

```
>>> import pgcd_ppcm_facto_module
```

e. En utilisant la fonction native « dir() », donner la liste des fonctions proposées par ce module

```
['PGCD', 'PPCM', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__spec__',
'fact']
```

f. En utilisant la fonction native help(), affichez les spécifications de la fonction ppcm()

```
help(pgcd_ppcm_facto_module.PPCM)
```

g. Modifiez le code du module « pgcd_ppcm_facto_module.py » afin d'obtenir la spécification suivante :

h. Depuis l'interpréteur, utilisez la fonction factorielle et donnez le résultat de « 7! »

5040

i. La fonction factorielle est incluse dans le module « math » de python. Après avoir consulté la spécification de la fonction, utilisez-la afin de vérifier l'exactitude du résultat précédent (h.)

```
import math
math.factorial(7)
5040
```

A faire vous-même VIII

Nous allons maintenant réaliser le programme principal qui utilise des fonctions déclarées dans d'autres modules.

a. Créez un nouveau fichier que vous libellerez « prog_module_fct.py ».

b. Ecrire le programme principal à partir des spécifications suivantes :

```
# nom : prog_module_fct.py
# rôle : Affichage du PGCD et du PPCM de deux entiers naturels non nuls a et b
#        Affichage de la factorielle d'un entier naturel n
#        Affichage de la racine carrée d'un entier naturel n
# Utilisation du module « pgcd_ppcm_facto_module.py »
# Utilisation de la fonction sqrt du module « math »
```

L'exécution de votre programme doit produire une sortie « écran » similaire à :

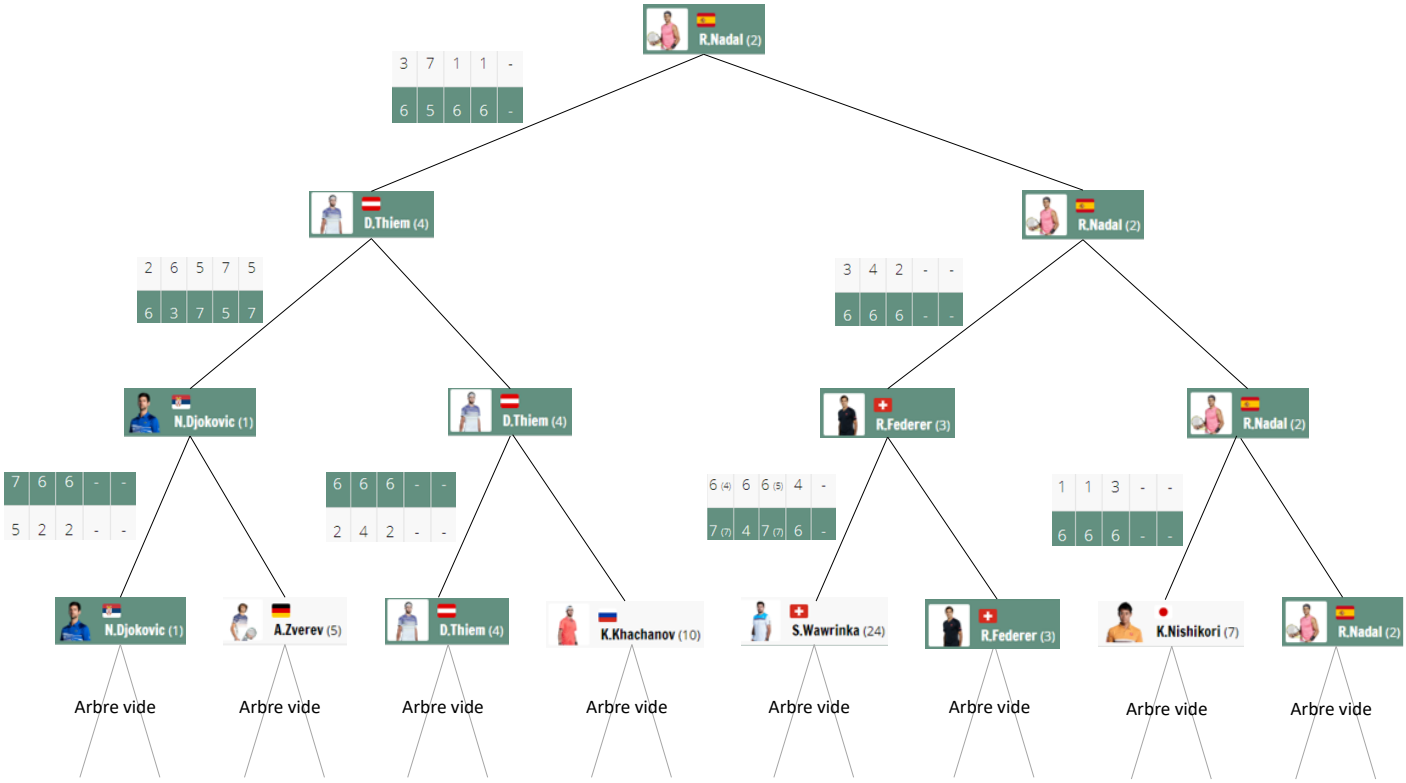
```
Calcul du PGCD(a,b), du PPCM(a,b), de n! et de racine_carrée(n)

Veuillez saisir l'entier a : 160
Veuillez saisir l'entier b : 140
Veuillez saisir l'entier n : 16

LE PGCD( 160 , 140 ) est :  20

LE PPCM( 160 , 140 ) est :  1120

factorielle(16) = 16! = 20922789888000

la racine carrée de 16 est : 4.0
```

3 La récursivité

3.1 Qu'est-ce que c'est ?

En informatique, une fonction ou plus généralement un algorithme qui contient un ou des appel(s) à lui-même est dit récursif. Ce procédé est employé dans la conception d'algorithmes basée sur le paradigme « diviser pour régner ». Deux fonctions peuvent s'appeler l'une l'autre, on parle alors de récursivité croisée.

La définition de certaines structures de données, comme les listes ou les arbres, est récursive (cf. le support intitulé « SDD_2020_V1.pdf »). Reprenons l'exemple des résultats des phases finales du tournoi de Roland Garros 2019.



Quarts de finale Rolang GARROS 2019

Cet arbre est soit fait de deux arbres binaires qu'on enracine dans un joueur (un nœud), soit est un arbre binaire vide.

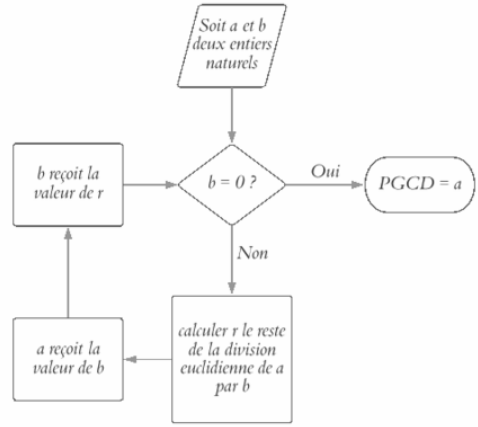
3.2 Règles d'écriture

Le concept de récursivité demande un effort d'abstraction afin d'être bien appréhendé. L'écriture d'un algorithme récursif efficace n'est une chose aisée, surtout lorsqu'on manque d'expérience.

Cependant, il existe au moins deux règles à observer afin de faciliter la réalisation d'un tel algorithme.

- Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne doit pas contenir d'appels récursifs. C'(e) est(sont) le(s) cas de base. Sinon il y a risque de boucle infinie.
- Tout appel récursif doit se faire avec des données plus proches de données satisfaisant les conditions de terminaison. Le résultat de l'appel doit converger vers le(s) cas de base.

3.3 Le calcul du PGCD



Nous pouvons commencer par chercher une formule qui exprime le calcul récursif. L'analyse de l'algorigramme nous conduit à déterminer :

- Le cas de base : le calcul s'arrête lorsque b = 0
- La formule récursive : pgcd(a,b) = pgcd(b,r)

A faire vous-même IX

a. En partant du programme « prog_prog_fct.py », écrire un programme « pgcd_prog_recursif.py » de calcul du « pgcd » de deux entiers naturels a et b. Ce programme invoque la fonction récursive « pgcd_recursif(x,y) »
b. En remarquant que « ppcm (a,b) = a.b / pgcd (a,b) », écrire le programme « ppcm_prog_recursif.py » de calcul du « ppcm » de deux entiers naturels a et b. Ce programme invoque une fonction récursive.

3.4 La factorielle d'un entier naturel

Commençons par chercher une formule qui exprime le calcul récursif :

- Les cas de base :
  - 0! = 1
  - Le calcul s'arrête lorsque n = 1
- La formule récursive :
  - n ! = n * n-1 * n-2 *…* 3 * 2 * 1
  - = n * (n – 1) !
  - (n – 1) ! = n-1 * n-2 * n-3*…* 3 * 2 * 1
  - = n-1 * (n-2) !

A faire vous-même X

En partant du programme « factorielle_n_prog_fct.py », écrire le programme « factorielle_n_prog_fct_recursif.py » du calcul de la factorielle d'un entier naturel n. Ce programme invoque la fonction récursive « factorielle_recursif(N) ».

3.5 La recherche dichotomique

Le problème : trouver un mot dans un dictionnaire.

L'idée : ouvrir le dictionnaire à la moitié puis s'interroger pour déterminer si le mot lu est avant ou après le mot recherché. Ensuite, en fonction de la moitié du dictionnaire dans laquelle peut figurer le mot recherché, on répète l'opération : on ouvre cette partie à la moitié afin de déterminer si le mot lu est avant ou après le mot recherché. Et ainsi de suite…

En classe de première, nous avons trouvé une solution à ce problème avec l'algorithme suivant :

```
# Début algorithme
# Entrée clavier du mot recherché
# *** code
#     code ***

#ouverture du fichier dictionnaire en mode lecture
# *** code ***

#Charger le ficher dans une liste de chaine
# *** code ***

# Fermeture du fichier
# *** code ***

# Initialisation des variables et constantes
# Initialisation des indices de parcours de liste : début et fin
# Initialisation de l'indice de milieu d'intervalle et du nombre de comparaisons
# *** code
#     code ***

# Boucle de comparaison
# Tant que début est strictement inférieur à fin
# *** code ***
      # Division de l'intervalle en deux intervalles
      # Indice du milieu d'intervalle <-- (début + fin)//2
      # Incrément du nombre de comparaisons
      # *** code
      #     code ***
      # Si le mot situé au milieu de l'intervalle de recherche est le mot recherché
      # *** code ***
          # début, fin <--- indice de milieu d'intervalle
          # *** code
          #     code ***
      # Autrement Si mot recherché est situé dans moitié inférieure de l'intervalle de recherche
      # *** code ***
          # fin <--- indice de milieu d'intervalle - 1
          # *** code ***
      # Autrement ( mot recherché est situé dans moitié supérieure de l'intervalle de recherche)
      # *** code ***
          # début <--- indice de milieu d'intervalle + 1
          # *** code ***

# Si le mot situé au milieu de l'intervalle de recherche est le mot recherché
# *** code ***
    # Préciser la position du mot dans le dictionnaire ainsi que le nombre de comparaisons
# *** code
#     code ***
# Autrement
# *** code ***
    # Informer l'utilisateur du résultat de la recherche
    # *** code ***

# Fin de l'algorithme
```

A faire vous-même XI

Nous allons à présent écrire un programme de recherche dichotomique récursive. Nous parviendrons à cet objectif en deux étapes :

- D'abord, nous allons écrire le programme « recherche_dichotomique » en définissant une fonction.
- Puis dans un second temps, nous écrirons le programme récursif.

a. En partant de l'algorithme ci-dessus, écrire le programme « recherche_dichotomique_fct.py » de recherche d'un mot dans le dictionnaire « liste_français.txt ». Ce programme invoque la fonction « recherche_dico(mot, dansUneListe) »

b. Recherche récursive. Après avoir déterminé le cas de base ainsi la formule récursive, écrire l'algorithme « recherche_dichotomique_fct_algo.py » de recherche d'un mot dans le dictionnaire « liste_français.txt ». Ce programme invoque la fonction « recherche_dico(mot, dansUneListe) » ainsi que la fonction récursive « recherche_mot(motRecherche, dansUneListe, debut, fin) ».

4 La Programmation Orientée Objet (POO)

4.1 Qu'est-ce qu'un objet ?

En python, tout est «objet » : les listes, les chaines de caractères, les entiers, les complexes, les fonctions,…. Leurs caractéristiques sont définies par leurs types. En classe de Première, nous avons vu et utilisé quelques types « Built-In ». Bien qu'extrêmement complets et puissants, ceux-ci ne peuvent pas répondre à des besoins très spécifiques. Pour répondre à cette problématique, on utilise le concept de « Classe ».

4.2 Qu'est-ce qu'une classe ?

En Python, une classe vous permet, entre autres choses, de définir vos propres types. Ainsi, on peut dire qu'une classe est un modèle qui permet de produire des objets possédant leurs propres caractéristiques.

Lorsque nous avons introduit la notion d'objet en Première, nous avons montré que les objets sont produits par une « usine » qu'on appelait le « type ». En fait, ce «type » et la « classe » représentent une seule et même chose !

La « classe », c'est donc « l'usine » qui va créer les objets. En POO, on appelle ces « objets » des « instances ».

Faisons une analogie avec le type « Built-In » « list » (liste) :

- le type crée les objets « listes »,
- tous les objets « listes » que l'on produit sont des instances du type list.

Prenons tout de suite un exemple pour illustrer ces concepts de classe et d'instance.

A faire vous-même XII

a. Création d'une classe. Pour ce faire, ouvrez votre interpréteur « IDLE » et créer le fichier « classe_exemple_1.py » dans votre espace de travail :

```
# coding: utf-8
# nom : classe_exemple_1.py
# rôle : création de la classe chaine

# Déclaration de la classe chaine
class Chaine:
    # je crée un attribut qui s'appelle chaine_1
    chaine_1 = "python est un langage vraiment sympa"
```

b. Exécutez votre module en cliquant sur « Run » ou en enfonçant la touche F5. Puis, directement dans l'interpréteur, créons une instance de la classe « Chaine » ainsi :

```
>>> mon_instance = Chaine()
```

Puis interrogeons la variable « mon_instance » ainsi :

```
>>> mon_instance
```

Quel résultat obtenez-vous ? Quelle est sa signification ?

```
<__main__.Chaine object at 0x0359D390>
mon instance est un objet Chaine
```

c. Créez la deuxième instance « mon_instance_2 » de la classe « Chaine ». Que référence « mon_instance_2 » ?

```
<  main  .Chaine object at 0x03AE2BD0>
```

d. Quel est son type de « mon_instance_2 » ? Utiliser la fonction « type() » pour répondre.

```
<class '  main  .Chaine'>
```

A retenir :

- La classe, c'est l'usine qui crée des instances et les instances sont les objets qui sont produits.
- Lorsqu'on appelle une classe, un objet « classe » est créé et on obtient ce qu'on peut appeler une « fabrique à instances ». A chaque appel de la classe, celle-ci crée de nouvelles instances.

4.3 Notion d'attributs et d'espace de nommage

A ce stade, on peut dire qu'un espace de nommage est un espace où l'on définit des attributs.

Une classe possède son propre espace de nommage.

Une instance possède son propre espace de nommage.

Lorsqu'on recherche un attribut dans une instance, on le recherche en fait dans l'espace de nommage de l'instance. Et si on ne le trouve pas dans l'espace de nommage de l'instance, on va le rechercher dans l'espace de nommage de la classe. On dit alors que l'on remonte l'arbre de d'héritage. L'arbre d'héritage, c'est cette recherche d'attributs dans les espaces de nommage de l'instance et de la classe.

A faire vous-même XIII

Afin d'illustrer ces concepts, reprenons l'exemple précédent :

a. Directement dans l'interpréteur, saisir l'instruction suivante :

```
>>> Chaine.chaine_1
```

Quel résultat obtenez-vous ? Interprétez-le.

```
'python est un langage vraiment sympa'
Ca affiche la variable chaine 1 de la class Chaine
```

b. Directement dans l'interpréteur, saisir l'instruction suivante :

```
>>> mon_instance2.chaine_1
```

Quel résultat obtenez-vous ? Quelle est sa signification ?

```
'python est un langage vraiment sympa'
Ca affiche la variable chaine 1 de l'objet mon instance2
```

c. Maintenant, accédons à l'espace de nommage de notre classe chaine :

```
>>> vars(Chaine)
```

Quelle est la valeur de l'attribut « chaine_1 » ?

```
'chaine 1': 'python est un langage vraiment sympa'
```

d. Maintenant, accédez à l'espace de nommage de l'instance fabriquée précédemment. Qu'obtenez-vous ?

```
{}
un dictionaire vide
```

e. Directement dans l'interpréteur, affichez l'aide sur le type built-in « str ». Lisez la définition de la fonction « split() ».

f. Créez un nouvel attribut de notre classe « Chaine » ainsi :

```
>>> Chaine.mots = Chaine.chaine_1.split()
>>> Chaine.mots
```

Quel résultat obtenez-vous ? Pourquoi ?

```
['python', 'est', 'un', 'langage', 'vraiment', 'sympa']
car la chaine 1 a ete split
```

g. Puis affichez l'espace de nommage de la classe « Chaine ». Résultat ?

```
Chaine.mots apparait
```

h. Selon vous, les instances « mon_instance1 » et « mon_instance2 » possèdent-elles un attribut « mots » ? Vérifiez-le.

```
oui car cest la class Chaine qui a ete modifier
```

A retenir :

- Une relation particulière s'instaure entre l'instance et la classe : on l'appelle « relation d'héritage ». Cela signifie que l'instance hérite de la classe et hérite de tous les attributs qui sont définis dans la classe.
- Lorsque je crée une instance, celle-ci est créée avec un espace de nommage vide.

4.4 Les méthodes

Une méthode est une fonction qui "appartient à" un objet. Elle est définie dans la classe et définit le comportement dont l'instance va hériter. En Python, le terme de méthode n'est pas unique aux instances de classes : d'autres types d'objets peuvent aussi avoir des méthodes. Par exemple, les objets listes possèdent des méthodes appelées « append », « insert », « remove », « sort » …

La liste exhaustive est accessible avec la commande native « dir(list) » que nous utilisons régulièrement.

A faire vous-même XIV

a. La méthode « __init__ ». Ouvrez votre interpréteur « IDLE » et créer le fichier « classe_exemple_2.py » dans votre espace de travail :

```
# coding: utf-8
# nom : classe_exemple_2.py
# rôle : création de la classe maPhrase
# Définition de méthodes et attributs d'instances

# Déclaration de la classe "maPhrase"
class maPhrase:
    # je crée le constructeur d'instances qui prend 2 arguments, self(instance) et "ma_phrase"
    def __init__(self,ma_phrase):
        self.mots = ma_phrase.split()
```

Analyse du programme « classe_exemple_2.py »

Nous avons défini la méthode « __init__ » qui est un « constructeur » d'instances. Dans notre exemple, cette méthode prend deux arguments « self » et « ma_phrase ».

Lorsque nous appelons la méthode « __init__() » sur la classe « maPhrase », l'instance va automatiquement être passée comme premier argument. L'argument « self » correspond donc à l'instance qui a été créée par « maPhrase ». La méthode « __init__() » est appelée lorsque je crée mon instance, et l'argument passé lors de la création de cette instance est « affecté » à « ma_phrase ».

b. Exécutez votre module en cliquant sur « Run » ou en enfonçant la touche F5. Puis, directement dans l'interpréteur, saisir les commandes suivantes :

```
>>> maphrase = maPhrase('python est un langage vraiment sympa')
>>> maphrase.mots
>>> vars(maphrase)
```

En vous appuyant sur l'analyse du programme, interprétez le résultat. Quelle est la différence avec AFVMXIII f. ?

la difference est que l'espace de nommage de maphrase contient :
{'mots': ['pthon', 'est', 'un', 'langage', 'vraiment', 'sympa']}

A retenir :

- Les attributs d'une instance appartiennent à « l'espace de nommage » de l'instance
- L'instance hérite du comportement de la classe, donc du type qui a été utilisé pour créer l'instance.

A faire vous-même XV

a. Création de nouvelles méthodes. Ouvrez votre interpréteur « IDLE » et créer le fichier « classe_exemple_3.py » dans votre espace de travail :

```
# coding: utf-8
# nom : classe_exemple_3.py
# rôle : création de la classe maPhrase
# Définition de méthodes et attributs d'instances

# Déclaration de la classe "maPhrase"
class maPhrase:
    # je crée le constructeur d'instances qui prend 2 arguments, self(instance) et "ma_phrase"
    def __init__(self,ma_phrase):
        self.mots = ma_phrase.split()

    # Méthode "titre()" qui transforme chaque de "ma_phrase" en Titre
    # Création d'un nouvel attributs "Mots" de l'instance
    def titre(self):
        self.Mots = [mot.title() for mot in self.mots]
```

b. Quelle méthode a été créée au sein de la classe « maPhrase » ? Quel(s) est(sont) son(es) argument(s) ? Que fait cette méthode ? Pour répondre à cette question, il est conseillé d'utiliser l'aide sur le type « str ».

```
la methode titre a ete créée
les arguments sont Mots
Mots contient les mots de la phrase avec une majuscule
```

c. Exécutez votre module en cliquant sur « Run » ou en enfonçant la touche F5. Puis, directement dans l'interpréteur :
   - Créez « mon_instance3 », une nouvelle instance de « maPhrase » et passez-lui en argument la phrase suivante : 'Voici une autre phrase quelconque juste par exemple'
   - Puis, appelez la méthode « titre() » sur « mon_instance3 ».

d. Selon vous, que contient l'attribut « mots » de « mon_instance3 » ? Expliquez pourquoi. Depuis l'interpréteur, saisissez la commande qui vous permettra de le vérifier.

```
 mots contient une liste de tous les mots de la phrase
>>> mon_instance3.mots
['Voici', 'une', 'autre', 'phrase', 'quelconque', 'juste', 'par', 'exemple']
```

e. Selon vous, que contient l'attribut « Mots » de « mon_instance3 » ? Expliquez pourquoi. Depuis l'interpréteur, saisissez la commande qui vous permettra de le vérifier.

```
 Mots contient une liste de tous les mots de la phrase avec uns majuscule
>>> mon_instance3.Mots
['Voici', 'Une', 'Autre', 'Phrase', 'Quelconque', 'Juste', 'Par', 'Exemple']
```

f. Selon vous, que contient l'espace de nommage de « mon_instance3 » ? Depuis l'interpréteur, saisissez la commande qui vous permettra de le vérifier.

```
 mots et mots sont dans l'espace de nammage
>>> vars(mon_instance3)
{'mots': ['Voici', 'une', 'autre', 'phrase', 'quelconque', 'juste', 'par', 'exemple']
, 'Mots': ['Voici', 'Une', 'Autre', 'Phrase', 'Quelconque', 'Juste', 'Par', 'Exemple'
]}
```

g. Maintenant, admettons que l'on souhaite afficher la longueur de l'instance « mon_instance4 » telle que :

```
>>> mon_instance4 = maPhrase('Nous souhaitons connaitre la longueur l de cette phrase')
```

Tentez d'afficher cette longueur en saisissant la commande :

```
>>> len(mon_instance4)
```

Interprétez le résultat.

```
mon instance4 n a pas dinstruction len

TypeError: object of type 'MaPhrase' has no len()
```

h. Implémentation de la méthode __len__(). Afin d'afficher la longueur de l'instance « mon_instance4 », nous allons créer une nouvelle méthode qui lorsqu'elle sera invoquée, retournera la longueur de l'instance. Pour ce faire, ajouter à « classe_exemple_3.py » le morceau de code suivant et sauvegardez le module ainsi obtenu sous « classe_exemple_4.py » :

```
# Méthode qui retourne la longueur de ma_phrase
def __len__(self):
    return len(" ".join(self.mots))
```

Exécutez votre module en cliquant sur « Run » ou en enfonçant la touche F5.
Que fait cette méthode ? Pour répondre à cette question, il est conseillé d'utiliser l'aide sur le type « str » et en particulier sur la fonction « join() ».

```
cette methode retourne la taille de la phrase
join sert a regrouper les eleents dune liste en str
```

i. Après avoir exécuté (F5) « classe_exemple_4.py » , créez une nouvelle variable « mon_instance4 ». Puis mesurer en nombre de caractères la longueur de la phrase « Nous souhaitons connaitre la longueur l de cette phrase ». Vérifiez votre résultat avec la commande :

```
>>> len(mon_instance4)
```

Quel résultat obtenez-vous ?

A retenir :

- En Python, nous manipulons des objets comme des types « built-in »
- Dans une classe, nous pouvons implémenter tous les comportements qu'adoptent les types « built-in »

A faire vous-même XVI

a. Créez une classe dont voici les spécifications :

```
# nom : classe_personne.py
# rôle : création du type "Personne"
# Attributs d'instance: le "nom", l'"age", le "sexe" et la "nationalité"
# Méthodes :
#   -le constructeur
#   -"saisie_carac()" qui permet de saisir les caractéristiques de la personne
#   -"affich_carac()" qui affiche les caractéristiques de la personne
```

b. Ecrire un programme dont voici les spécifications :

```
# nom : mesPersonnes.py
# rôle : Saisie et affichage d'un registre de personnes
# Appel du module "classe_personne.py"
```

Et dont l'exécution produit le résultat suivant :

```
Saisie des caractéristiques de la première Personne
Nom : Holm
Age : 87
Sexe (F/M) : M
Nationalité : Britannique

Saisie des caractéristiques de la deuxième Personne
Nom : Bardot
Age : 85
Sexe (F/M) : F
Nationalité : Française


Affichage des caractéristiques de la première Personne
Le nom de la personne est  Holm
Son âge :  87
Son sexe :  M
Sa nationalité :  Britannique

Affichage des caractéristiques de la deuxième Personne
Le nom de la personne est  Bardot
Son âge :  85
Son sexe :  F
Sa nationalité :  Française
```

5 Les paradigmes de programmation

5.1 Qu'est-ce que c'est ?

Il existe différentes classes de langages implémentés à partir de concepts et de schémas de pensées différents. Ainsi, le langage « C » est assez éloigné du langage « Lisp » qui lui-même ne partage guère les structures de base du langage « SmallTalk ».

On parle de **paradigme** pour désigner un ensemble de concepts particuliers.

Les langages de programmation **impératifs** représentent le premier paradigme de programmation utilisé : ce sont les langages qui héritent de l' « Algol », dont le « C » est le meilleur représentant.

Bien qu'inventé en même temps, le paradigme **orienté objet** a connu le succès avec le « C++ » et, au début du XXIème siècle grâce aux langages « Java » et « C# ».

Les autres paradigmes, comme par exemple la programmation **fonctionnelle**, sont moins utilisés. Ils comprennent le « LISP », le « Scheme » et le « Prolog ». Le « logo » est également un langage respectant le paradigme **fonctionnel**.

Certains langages sont multi-paradigmes. Ainsi « Python » offre les trois paradigmes présentés précédemment.

5.2 Le paradigme fonctionnel

Le principe général de la programmation fonctionnelle est de concevoir des programmes comme des fonctions mathématiques que l'on compose entre elles. A la différence des programmes impératifs organisés en instructions pouvant produire des « **effets de bord** », les programmes fonctionnels sont bâtis sur des **expressions** dont la valeur est le résultat du programme. En particulier dans un langage fonctionnel, il n'existe pas **d'effet de bord**.

Python est un langage multi paradigme qui supporte certains concepts de programmation fonctionnelle.

Parmi ces concepts, nous allons commencer par découvrir les « **fonctions lambda** ». Puis, pour mieux comprendre, nous verrons que tout ce nous pouvons faire avec une « fonction lambda », peut être fait à l'aide de fonctions traditionnelles.

Les « fonctions lambda »

Tout ce qu'on peut faire avec une « fonction lambda » peut être réalisé avec une fonction traditionnelle. Par conséquent, les « fonctions lambda » ne sont pas indispensables. Elles constituent simplement une facilité d'utilisation pour écrire du code, qui devient plus simple et expressif.

Une « fonction lambda » est une expression. Et c'est la principale différence entre une « fonction lambda » et une fonction traditionnelle. Par conséquent, on peut définir une « fonction lambda » partout où on peut avoir une expression. Citons par exemple, lorsque l'on écrit une liste, un dictionnaire, ou encore lorsque l'on veut passer directement un argument à une fonction.

Les fonctions lambda sont en général utilisées dans un contexte particulier, pour lequel définir une fonction à l'aide du mot clé « def » serait plus long et moins pratique.

Pour créer une telle fonction, nous utiliserons le mot clé « lambda » avec la syntaxe suivante :

lambda arg1, arg2,… : expression

A faire vous-même XVII

A l'aide de notre éditeur IDLE, découvrons maintenant les « fonctions lambda »

a. Commençons par créer une « fonction lambda avec la commande suivante :

```
>>> lambda x : x**3
<function <lambda> at 0x044EB540>
```

Nous constatons qu'un objet « fonction lambda » a bien été créé

b. Pour utiliser cet objet dans l'éditeur, il faut lui donner un nom (le « référencer » avec une variable) :

```
>>> cube = lambda x : x**3
```

Selon vous, quel sera le résultat de : `>>> cube(3)` ?

c. Alternative à la fonction lambda x : x**3.

Ecrire le programme « elever_au_cube.py ». Ce programme calcule le cube d'un entier naturel. Il contient la fonction « cube(x) ». Testez votre programme pour la valeur « x = 3 ». Est-ce le même résultat que précédemment ? Qu'a-t-on vérifié ? Conclure sur la valeur ajoutée d'une fonction lambda.

la fonction lambda est plus rapide a cree

d. Voyons maintenant ce que peut apporter une fonction « lambda » dans un contexte particulier. Créez le module « exemple_fonction_lambda.py » dans votre espace de travail selon l'algorithme :

```
# coding: utf-8
# nom : exemple_fonction_lambda_algo.py
# rôle : Définition d'un objet "fonction"
# Déclaration de la fonctions image()

# Déclaration de la fonction "image()" qui accepte une fonction comme argument
# *** code ***
    # Pour x allant de 10 à 20
    # *** code ***
        # on affiche le message "à x on associe fonction(x)"
        # *** code ***
```

e. Après avoir exécuté (F5) « exemple_fonction_lambda.py », tapez la commande :

```
>>> image(lambda x: x**3)
```

Expliquez le résultat. Dans cet exemple, quel est l'intérêt de la fonction lambda ?

l'interet de la fonction lambda est de utiliser une fonction comme argument

f. En utilisant la variable « cube », utilisez l'interpréteur afin d'exécuter la commande qui permet d'obtenir le même résultat que précédemment (e.)

```
>>> cube = lambda x:x**3
>>> image(cube)
```

g. « f » n'est plus une fonction « lambda », tentons d'obtenir le même résultat que précédemment (e. et f.) en utilisant la fonction image(f).

Ecrire le module « module_objet_fonction.py » issu de l'algorithme suivant :

```
# coding: utf-8
# nom : module_objet_fonction_algo.py
# rôle : Définition d'un objet "fonction"
# Déclaration des fonctions image() et cube()

# Déclaration de la fonction "image()" qui accepte une fonction comme argument
# *** code ...
# ...
# ... code ***

# Déclaration de la fonction "cube()" qui accepte un nombre x comme argument
# *** code ...
# ...
# ... code ***
```

Après avoir exécuté (F5) le module « module_objet_fonction.py », vérifier le type des objets « image » et « cube ».

Quelle commande entrée directement depuis l'interpréteur nous permet d'obtenir le même résultat que précédemment ? Vérifiez-le.

Par quel objet avez-vous substitué l'objet fonction « lambda x : x**3 » ?

```
image(cube)

lambda x:x**3 est subtitue par la fonction cube
```

***** Fin du document *****