MOOC Python 3

Session 2018

Tous les corrigés

Table des matières

pythonid_regexp - Semaine 2 Séquence 2

```
# un identificateur commence par une lettre ou un underscore
# et peut être suivi par n'importe quel nombre de
# lettre, chiffre ou underscore, ce qui se trouve être \w
# si on ne se met pas en mode unicode
pythonid = r"[a-zA-Z_]\w*"
```

pythonid_bis - Semaine 2 Séquence 2

```
# on peut aussi bien sûr l'écrire en clair
pythonid_bis = r"[a-zA-Z_][a-zA-Z0-9_]*"
```

agenda_regexp - Semaine 2 Séquence 2

```
# l'exercice est basé sur re.match, ce qui signifie que
# le match est cherché au début de la chaine
# MAIS il nous faut bien mettre \Z à la fin de notre regexp,
# sinon par exemple avec la cinquième entrée le nom 'Du Pré'
# sera reconnu partiellement comme simplement 'Du'
# au lieu d'être rejeté à cause de l'espace
#
# du coup pensez à bien toujours définir
# vos regexps avec des raw-strings
#
# remarquez sinon l'utilisation à la fin de :? pour signifier qu'on peut
# mettre ou non un deuxième séparateur ':'
#
agenda = r"\A(?P<prenom>[-\w]*):(?P<nom>[-\w]+):?\Z"
```

phone_regexp - Semaine 2 Séquence 2

```
# idem concernant le \Z final
#
# il faut bien backslasher le + dans le +33
# car sinon cela veut dire 'un ou plusieurs'
#
phone = r"(\+33|0)(?P<number>[0-9]{9})\Z"
```

url_regexp - Semaine 2 Séquence 2

```
# en ignorant la casse on pourra ne mentionner les noms de protocoles
# qu'en minuscules
i_flag = "(?i)"

# pour élaborer la chaine (proto1|proto2|...)
protos_list = ['http', 'https', 'ftp', 'ssh', ]
protos      = "(?P<proto>" + "|".join(protos_list) + ")"

# à l'intérieur de la zone 'user/password', la partie
# password est optionnelle - mais on ne veut pas le ':' dans
# le groupe 'password' - il nous faut deux groupes
password    = r"(:(?P<password>[^:]+))?"

# la partie user-password elle-même est optionnelle
# on utilise ici un raw f-string avec le préfixe rf
# pour insérer la regexp <password> dans la regexp <user>
user        = rf"((?P<user>\w+){password}@)?"

# pour le hostname on accepte des lettres, chiffres, underscore et '.'
# attention à backslaher . car sinon ceci va matcher tout y compris /
hostname    = r"(?P<hostname>[\w\.]+)"

# le port est optionnel
port        = r"(:(?P<port>\d+))?"

# après le premier slash
path        = r"(?P<path>.*)"

# on assemble le tout
url = i_flag + protos + "://" + user + hostname + port + '/' + path
```

url_bis - Semaine 2 Séquence 2

```
# merci à sizeof qui a pointé l'utilisation de re.X
# https://docs.python.org/fr/3/library/re.html#re.X
# ce qui donne une présentation beaucoup plus compacte

protos_list = ['http', 'https', 'ftp', 'ssh', ]

url_bis = rf"""(?x)                     # verbose mode
    (?i)                                # ignore case
    (?P<proto>{"|".join(protos_list)}) # http|https|...
    ://                                 # separator
    ((?P<user>\w+){password}@)?         # optional user/password
    (?P<hostname>[\w\.]+)               # mandatory hostname
    (:(?P<port>\d+))?                   # optional port
    /(?P<path>.*)                       # mandatory path
"""
```

label - Semaine 2 Séquence 6

```
def label(prenom, note):
    if note < 10:
        return f"{prenom} est recalé"
    elif note < 16:
        return f"{prenom} est reçu"
    else:
        return f"félicitations à {prenom}"
```

label_bis - Semaine 2 Séquence 6

```
def label_bis(prenom, note):
    if note < 10:
        return f"{prenom} est recalé"
    # on n'en a pas vraiment besoin ici, mais
    # juste pour illustrer cette construction
    elif 10 <= note < 16:
        return f"{prenom} est reçu"
    else:
        return f"félicitations à {prenom}"
```

label_ter - Semaine 2 Séquence 6

```
# on n'a pas encore vu l'expression conditionnelle
# et dans ce cas précis ce n'est pas forcément une
# idée géniale, mais pour votre curiosité on peut aussi
# faire comme ceci
def label_ter(prenom, note):
    return f"{prenom} est recalé" if note < 10 \
    else f"{prenom} est reçu" if 10 <= note < 16 \
    else f"félicitations à {prenom}"
```

inconnue - Semaine 2 Séquence 6

```
# pour enlever à gauche et à droite une chaine de longueur x
# on peut faire composite[ x : -x ]
# or ici x vaut len(connue)
def inconnue(composite, connue):
    return composite[ len(connue) : -len(connue) ]
```

inconnue_bis - Semaine 2 Séquence 6

```
# ce qui peut aussi s'écrire comme ceci si on préfère
def inconnue_bis(composite, connue):
    return composite[ len(connue) : len(composite)-len(connue) ]
```

laccess - Semaine 2 Séquence 6

```
def laccess(liste):
    """
    retourne un élément de la liste selon la taille
    """
    # si la liste est vide il n'y a rien à faire
    if not liste:
        return
    # si la liste est de taille paire
    if len(liste) % 2 == 0:
        return liste[-1]
    else:
        return liste[len(liste)//2]
```

laccess_bis - Semaine 2 Séquence 6

```
# une autre version qui utilise
# un trait qu'on n'a pas encore vu
def laccess_bis(liste):
    # si la liste est vide il n'y a rien à faire
    if not liste:
        return
    # l'index à utiliser selon la taille
    index = -1 if len(liste) % 2 == 0 else len(liste) // 2
    return liste[index]
```

divisible - Semaine 2 Séquence 6

```
def divisible(a, b):
    "renvoie True si un des deux arguments divise l'autre"
    # b divise a si et seulement si le reste
    # de la division de a par b est nul
    if a % b == 0:
        return True
    # et il faut regarder aussi si a divise b
    if b % a == 0:
        return True
    return False
```

divisible_bis - Semaine 2 Séquence 6

```
def divisible_bis(a, b):
    "renvoie True si un des deux arguments divise l'autre"
    # on n'a pas encore vu les opérateurs logiques, mais
    # on peut aussi faire tout simplement comme ça
    # sans faire de if du tout
    return a % b == 0 or b % a == 0
```

morceaux - Semaine 2 Séquence 6

```
def morceaux(x):
    if x <= -5:
        return -x - 5
    elif x <= 5:
        return 0
    else:
        return x / 5 - 1
```

morceaux_bis - Semaine 2 Séquence 6

```
def morceaux_bis(x):
    if x <= -5:
        return -x - 5
    if x <= 5:
        return 0
    return x / 5 - 1
```

morceaux_ter - Semaine 2 Séquence 6

```
# on peut aussi faire des tests d'intervalle
# comme ceci  0 <= x <= 10
def morceaux_ter(x):
    if x <= -5:
        return -x - 5
    elif -5 <= x <= 5:
        return 0
    else:
        return x / 5 - 1
```

wc - Semaine 2 Séquence 6

```
def wc(string):
    """
    Compte les nombres de lignes, de mots et de caractères

    Retourne une liste de ces 3 nombres (notez qu'usuellement
    on renverrait plutôt un tuple, qu'on étudiera la semaine prochaine)
    """
    # on peut tout faire avec la bibliothèque standard
    nb_lines = string.count('\n')
    nb_words = len(string.split())
    nb_bytes = len(string)
    return [nb_lines, nb_words, nb_bytes]
```

liste_P - Semaine 2 Séquence 7

```
def P(x):
    return 2 * x**2 - 3 * x - 2

def liste_P(liste_x):
    """
    retourne la liste des valeurs de P
    sur les entrées figurant dans liste_x
    """
    return [P(x) for x in liste_x]
```

liste_P_bis - Semaine 2 Séquence 7

```
# On peut bien entendu faire aussi de manière pédestre
def liste_P_bis(liste_x):
    liste_y = []
    for x in liste_x:
        liste_y.append(P(x))
    return liste_y
```

carre - Semaine 2 Séquence 7

```
def carre(line):
    # on enlève les espaces et les tabulations
    line = line.replace(' ', '').replace('\t','')
    # la ligne suivante fait le plus gros du travail
    # d'abord on appelle split() pour découper selon les ';'
    # dans le cas où on a des ';' en trop, on obtient dans le
    #    résultat du split un 'token' vide, que l'on ignore
    #    ici avec la clause 'if token'
    # enfin on convertit tous les tokens restants en entiers avec int()
    entiers = [int(token) for token in line.split(";")
               # en éliminant les entrées vides qui correspondent
               # à des point-virgules en trop
               if token]
    # il n'y a plus qu'à mettre au carré, retraduire en strings,
    # et à recoudre le tout avec join et ':'
    return ":".join([str(entier**2) for entier in entiers])
```

carre_bis - Semaine 2 Séquence 7

```
def carre_bis(line):
    # pareil mais avec, à la place des compréhensions
    # des expressions génératrices que - rassurez-vous -
    # l'on n'a pas vues encore, on en parlera en semaine 5
    # le point que je veux illustrer ici c'est que c'est
    # exactement le même code mais avec () au lieu de []
    line = line.replace(' ', '').replace('\t','')
    entiers = (int(token) for token in line.split(";")
               if token)
    return ":".join(str(entier**2) for entier in entiers)
```

carre_ter - Semaine 2 Séquence 7

```
def carre_ter(ligne):
    # On extrait toutes les valeurs séparées par des points-
    # virgules, on les nettoie avec la méthode strip
    # et on stocke le résultat dans une liste
    liste_valeurs = [t.strip() for t in ligne.split(';')]
    # Il ne reste plus qu'à calculer les carrés pour les
    # valeurs valides (non vides) et les remettre dans une str
    return ":".join([str(int(v)**2) for v in liste_valeurs if v])
```

comptage - Semaine 3 Séquence 2

```
def comptage(in_filename, out_filename):
    """
    retranscrit le fichier in_filename dans le fichier out_filename
    en ajoutant des annotations sur les nombres de lignes, de mots
    et de caractères
    """
    # on ouvre le fichier d'entrée en lecture
    with open(in_filename, encoding='utf-8') as in_file:
        # on ouvre la sortie en écriture
        with open(out_filename, 'w', encoding='utf-8') as out_file:
            lineno = 1
            # pour toutes les lignes du fichier d'entrée
            # le numéro de ligne commence à 1
            for line in in_file:
                # autant de mots que d'éléments dans split()
                nb_words = len(line.split())
                # autant de caractères que d'éléments dans la ligne
                nb_chars = len(line)
                # on écrit la ligne de sortie; pas besoin
                # de newline (\n) car line en a déjà un
                out_file.write(f"{lineno}:{nb_words}:{nb_chars}:{line}")
                lineno += 1
```

comptage_bis - Semaine 3 Séquence 2

```
def comptage_bis(in_filename, out_filename):
    """
    un peu plus pythonique avec enumerate
    """
    with open(in_filename, encoding='utf-8') as in_file:
        with open(out_filename, 'w', encoding='utf-8') as out_file:
            # enumerate(.., 1) pour commencer avec une ligne
            # numérotée 1 et pas 0
            for lineno, line in enumerate(in_file, 1):
                # une astuce : si on met deux chaines
                # collées comme ceci elle sont concaténées
                # et on n'a pas besoin de mettre de backslash
                # puisqu'on est dans des parenthèses
                out_file.write(f"{lineno}:{len(line.split())}:"
                               f"{len(line)}:{line}")
```

comptage_ter - Semaine 3 Séquence 2

```
def comptage_ter(in_filename, out_filename):
    """
    pareil mais avec un seul with
    """
    with open(in_filename, encoding='utf-8') as in_file, \
         open(out_filename, 'w', encoding='utf-8') as out_file:
        for lineno, line in enumerate(in_file, 1):
            out_file.write(f"{lineno}:{len(line.split())}:"
                           f"{len(line)}:{line}")
```

comptage_quater - Semaine 3 Séquence 2

```
def comptage_quater(in_filename, out_filename):
    """
    si on est sûr que les séparateurs restent tous identiques,
    on peut écrire cette fonction en utilisant la méthode join
    en conjonction avec un tuple qui est un itérable
    pour ne pas répéter le séparateur
    """
    with open(in_filename, encoding="UTF-8") as in_file, \
         open(out_filename, mode='w', encoding="UTF-8") as out_file:
        for line_no, line in enumerate(in_file, 1):
            out_file.write(":".join((str(line_no), str(len(line.split())),
              str(len(line)), line)))
```

surgery - Semaine 3 Séquence 2

```
def surgery(liste):
    """
    Prend en argument une liste, et retourne la liste modifiée:
    * taille paire: on intervertit les deux premiers éléments
    * taille impaire >= 3: on fait tourner les 3 premiers éléments
    """
    # si la liste est de taille 0 ou 1, il n'y a rien à faire
    if len(liste) < 2:
        pass
    # si la liste est de taille paire
    elif len(liste) % 2 == 0:
        # on intervertit les deux premiers éléments
        liste[0], liste[1] = liste[1], liste[0]
    # si elle est de taille impaire
    else:
        liste[-2], liste[-1] = liste[-1], liste[-2]
    # et on n'oublie pas de retourner la liste dans tous les cas
    return liste
```

graph_dict - Semaine 3 Séquence 4

```
def graph_dict(filename):
    """
    construit une stucture de données de graphe
    à partir du nom du fichier d'entrée
    """
    # un dictionnaire vide normal
    graph = {}

    with open(filename) as feed:
        for line in feed:
            begin, value, end = line.split()
            # c'est cette partie qu'on économisera
            # dans la deuxième solution avec un defaultdict
            if begin not in graph:
                graph[begin] = []
            # remarquez les doubles parenthèses
            # car on appelle append avec un seul argument
            # qui est un tuple
            graph[begin].append((end, int(value)))
            # si on n'avait écrit qu'un seul niveau de parenthèses
            # graph[begin].append(end, int(value))
            # cela aurait signifié un appel à append avec deux arguments
            # ce qui n'aurait pas du tout fait ce qu'on veut
    return graph
```

graph_dict_bis - Semaine 3 Séquence 4

```
from collections import defaultdict

def graph_dict_bis(filename):
    """
    pareil mais en utilisant un defaultdict
    """
    # on déclare le defaultdict de type list
    # de cette façon si une clé manque elle
    # sera initialisée avec un appel à list()
    graph = defaultdict(list)

    with open(filename) as feed:
        for line in feed:
            # on coupe la ligne en trois parties
            begin, value, end = line.split()
            # comme c'est un defaultdict on n'a
            # pas besoin de l'initialiser
            graph[begin].append((end, int(value)))
    return graph
```

index - Semaine 3 Séquence 4

```
def index(bateaux):
    """
    Calcule sous la forme d'un dictionnaire indexé par les ids
    un index de tous les bateaux présents dans la liste en argument
    Comme les données étendues et abrégées ont toutes leur id
    en première position on peut en fait utiliser ce code
    avec les deux types de données
    """
    # c'est une simple compréhension de dictionnaire
    return {bateau[0] : bateau for bateau in bateaux}
```

index_bis - Semaine 3 Séquence 4

```
def index_bis(bateaux):
    """
    La même chose mais de manière itérative
    """
    # si on veut décortiquer
    resultat = {}
    for bateau in bateaux:
        resultat[bateau[0]] = bateau
    return resultat
```

index_ter - Semaine 3 Séquence 4

```
def index_ter(bateaux):
    """
    Encore une autre, avec un extended unpacking
    """
    # si on veut décortiquer
    resultat = {}
    for bateau in bateaux:
        # avec un extended unpacking on peut extraire
        # le premier champ; en appelant le reste _
        # on indique qu'on n'en fera en fait rien
        id, *_ = bateau
        resultat[id] = bateau
    return resultat
```

merge - Semaine 3 Séquence 4

```
def merge(extended, abbreviated):
    """
    Consolide des données étendues et des données abrégées
    comme décrit dans l'énoncé
    Le coût de cette fonction est linéaire dans la taille
    des données (longueur commune des deux listes)
    """
    # on initialise le résultat avec un dictionnaire vide
    result = {}
    # pour les données étendues
    # on affecte les 6 premiers champs
    # et on ignore les champs de rang 6 et au delà
    for id, latitude, longitude, timestamp, name, country, *_ in extended:
        # on crée une entrée dans le résultat,
        # avec la mesure correspondant aux données étendues
        result[id] = [name, country, (latitude, longitude, timestamp)]
    # maintenant on peut compléter le résultat avec les données abrégées
    for id, latitude, longitude, timestamp in abbreviated:
        # et avec les hypothèses on sait que le bateau a déjà été
        # inscrit dans le résultat, donc result[id] doit déjà exister
        # et on peut se contenter d'ajouter la mesure abrégée
        # dans l'entrée correspondante dans result
        result[id].append((latitude, longitude, timestamp))
    # et retourner le résultat
    return result
```

merge_bis - Semaine 3 Séquence 4

```
def merge_bis(extended, abbreviated):
    """
    Une deuxième version, linéaire également
    mais qui utilise les indices plutôt que l'unpacking
    """
    # on initialise le résultat avec un dictionnaire vide
    result = {}
    # on remplit d'abord à partir des données étendues
    for ship in extended:
        id = ship[0]
        # on crée la liste avec le nom et le pays
        result[id] = ship[4:6]
        # on ajoute un tuple correspondant à la position
        result[id].append(tuple(ship[1:4]))
    # pareil que pour la première solution,
    # on sait d'après les hypothèses
    # que les id trouvées dans abbreviated
    # sont déja présentes dans le résultat
    for ship in abbreviated:
        id = ship[0]
        # on ajoute un tuple correspondant à la position
        result[id].append(tuple(ship[1:4]))
    return result
```

merge_ter - Semaine 3 Séquence 4

```
def merge_ter(extended, abbreviated):
    """
    Une troisième solution
    à cause du tri que l'on fait au départ, cette
    solution n'est plus linéaire mais en O(n.log(n))
    """
    # ici on va tirer profit du fait que les id sont
    # en première position dans les deux tableaux
    # si bien que si on les trie,
    # on va mettre les deux tableaux 'en phase'
    #
    # c'est une technique qui marche dans ce cas précis
    # parce qu'on sait que les deux tableaux contiennent des données
    # pour exactement le même ensemble de bateaux
    #
    # on a deux choix, selon qu'on peut se permettre ou non de
    # modifier les données en entrée. Supposons que oui:
    extended.sort()
    abbreviated.sort()
    # si ça n'avait pas été le cas on aurait fait plutôt
    # extended = extended.sorted() et idem pour l'autre
    #
    # il ne reste plus qu'à assembler le résultat
    # en découpant des tranches
    # et en les transformant en tuples pour les positions
    # puisque c'est ce qui est demandé
    return {
        ext[0] : ext[4:6] + [ tuple(ext[1:4]), tuple(abb[1:4]) ]
        for (ext, abb) in zip (extended, abbreviated)
        }
```

read_set - Semaine 3 Séquence 5

```
# on suppose que le fichier existe
def read_set(filename):
    """
    crée un ensemble des mots-lignes trouvés dans le fichier
    """
    # on crée un ensemble vide
    result = set()

    # on parcourt le fichier
    with open(filename) as feed:
        for line in feed:
            # avec strip() on enlève la fin de ligne,
            # et les espaces au début et à la fin
            result.add(line.strip())
    return result
```

read_set_bis - Semaine 3 Séquence 5

```
# on peut aussi utiliser une compréhension d'ensemble
# (voir semaine 5); ça se présente comme
# une compréhension de liste mais on remplace
# les [] par des {}
def read_set_bis(filename):
    with open(filename) as feed:
        return {line.strip() for line in feed}
```

search_in_set - Semaine 3 Séquence 5

```
# ici aussi on suppose que les fichiers existent
def search_in_set(filename_reference, filename):
    """
    cherche les mots-lignes de filename parmi ceux
    qui sont presents dans filename_reference
    """

    # on tire profit de la fonction précédente
    reference_set = read_set(filename_reference)

    # on crée une liste vide
    result = []
    with open(filename) as feed:
        for line in feed:
            token = line.strip()
            # remarquez ici les doubles parenthèses
            # pour passer le tuple en argument
            result.append((token, token in reference_set))

    return result
```

search_in_set_bis - Semaine 3 Séquence 5

```
def search_in_set_bis(filename_reference, filename):

    # on tire profit de la fonction précédente
    reference_set = read_set(filename_reference)

    # c'est un plus clair avec une compréhension
    # mais moins efficace car on calcule strip() deux fois
    with open(filename) as feed:
        return [(line.strip(), line.strip() in reference_set)
                for line in feed]
```

diff - Semaine 3 Séquence 5

```
def diff(extended, abbreviated):
    """Calcule comme demandé dans l'exercice, et sous formes d'ensembles
    (*) les noms des bateaux seulement dans extended
    (*) les noms des bateaux présents dans les deux listes
    (*) les ids des bateaux seulement dans abbreviated
    """

    ### on n'utilise que des ensembles dans tous l'exercice

    # les ids de tous les bateaux dans extended
    # avec ce qu'on a vu jusqu'ici le moyen le plus naturel
    # consiste à calculer une compréhension de liste
    # et à la traduire en ensemble comme ceci
    extended_ids = set([ship[0] for ship in extended])

    # les ids de tous les bateaux dans abbreviated
    # je fais exprès de ne pas mettre les []
    # de la compréhension de liste, c'est pour vous introduire
    # les expressions génératrices - voir semaine 5
    abbreviated_ids = set(ship[0] for ship in abbreviated)

    # les ids des bateaux seulement dans abbreviated
    # une difference d'ensembles
    abbreviated_only_ids = abbreviated_ids - extended_ids

    # les ids des bateaux dans les deux listes
    # une intersection d'ensembles
    both_ids = abbreviated_ids & extended_ids

    # les ids des bateaux seulement dans extended
    # ditto
    extended_only_ids = extended_ids - abbreviated_ids

    # pour les deux catégories où c'est possible
    # on recalcule les noms des bateaux
    # par une compréhension d'ensemble
    both_names = \
        set([ship[4] for ship in extended if ship[0] in both_ids])
    extended_only_names = \
        set([ship[4] for ship in extended if ship[0] in extended_only_ids])
    # enfin on retourne les 3 ensembles sous forme d'un tuple
    return extended_only_names, both_names, abbreviated_only_ids
```

diff_bis - Semaine 3 Séquence 5

```
def diff_bis(extended, abbreviated):
    """
    Même code mais qui utilise les compréhensions d'ensemble
    que l'on n'a pas encore vues - à nouveau, voir semaine 5
    mais vous allez voir que c'est assez intuitif
    """
    extended_ids = {ship[0] for ship in extended}
    abbreviated_ids = {ship[0] for ship in abbreviated}

    abbreviated_only_ids = abbreviated_ids - extended_ids
    both_ids = abbreviated_ids & extended_ids
    extended_only_ids = extended_ids - abbreviated_ids

    both_names = \
          {ship[4] for ship in extended if ship[0] in both_ids}
    extended_only_names = \
          {ship[4] for ship in extended if ship[0] in extended_only_ids}

    return extended_only_names, both_names, abbreviated_only_ids
```

diff_ter - Semaine 3 Séquence 5

```
def diff_ter(extended, abbreviated):
    """
    Idem sans les calculs d'ensembles intermédiaires
    en utilisant les conditions dans les compréhensions
    """
    extended_ids =     {ship[0] for ship in extended}
    abbreviated_ids =  {ship[0] for ship in abbreviated}
    abbreviated_only = {ship[0] for ship in abbreviated
                        if ship[0] not in extended_ids}
    extended_only =    {ship[4] for ship in extended
                        if ship[0] not in abbreviated_ids}
    both =             {ship[4] for ship in extended
                        if ship[0] in abbreviated_ids}
    return extended_only, both, abbreviated_only
```

diff_quater - Semaine 3 Séquence 5

```
def diff_quater(extended, abbreviated):
    """
    Idem sans indices
    """
    extended_ids =     {id for id, *_ in extended}
    abbreviated_ids =  {id for id, *_ in abbreviated}
    abbreviated_only = {id for id, *_ in abbreviated
                        if id not in extended_ids}
    extended_only =    {name for id, _, _, _, name, *_ in extended
                        if id not in abbreviated_ids}
    both =             {name for id, _, _, _, name, *_ in extended
                        if id in abbreviated_ids}
    return extended_only, both, abbreviated_only
```

fifo - Semaine 3 Séquence 8

```
class Fifo:
    """
    Une classe FIFO implémentée avec une simple liste
    """

    # dans cette première version on utilise
    # un object 'list' standard
    # on ajoute à la fin avec    queue.append(x),
    # et on enlève au début avec queue.pop(0)
    #
    # remarquez qu'on pourrait aussi
    # ajouter au début avec queue.insert(0, x)
    # enlever à la fin avec queue.pop()

    def __init__(self):
        # l'attribut queue est un objet liste
        self.queue = []

    def __repr__(self):
        contents = ", ".join(str(item) for item in self.queue)
        return f"[Fifo {contents}]"

    def incoming(self, item):
        # on insère au début de la liste
        self.queue.append(item)

    def outgoing(self):
        # pas la peine d'utiliser un try/except dans ce cas
        if self.queue:
            return self.queue.pop(0)
        # si on utilise pylint on va avoir envie de rajouter ceci
        # qui n'est pas vraiment indispensable..
        else:
            return None
```

fifo_bis - Semaine 3 Séquence 8

```
from collections import deque

class FifoBis:
    """
    une alternative en utilisant exactement la même stratégie
    mais avec un objet de type collections.deque
    en effet, l'objet 'list' standard est optimisé pour
    ajouter/enlever **à la fin** de la liste
    et on a vu dans la première version du code qu'il nous faut
    travailler sur les deux cotés de la pile, quel que soit le sens
    qu'on choisit pour implémenter la pile
    donc si la pile a des chances d'être longue de plusieurs milliers
    d'objets, il est utile de prendre un 'deque'
    'deque' vient de 'double-entry queue', et est optimisée
    pour les accès depuis le début et/ou la fin de la liste
    """
    def __init__(self):
        self.queue = deque()

    # ici pour faire bon poids on utilise la stratégie inverse
    # de la première version de la pile, on insère au début et on
    # enlève de la fin
    # du coup on les affice dans l'autre sens
    def __repr__(self):
        contents = ", ".join(str(item) for item in reversed(self.queue))
        return f"[Fifo {contents}]"

    def incoming(self, item):
        self.queue.insert(0, item)

    def outgoing(self):
        if self.queue:
            return self.queue.pop()

```

dispatch1 - Semaine 4 Séquence 2

```
def dispatch1(a, b):
    """
    dispatch1 comme spécifié
    """
    # si les deux arguments sont pairs
    if a%2 == 0 and b%2 == 0:
        return a*a + b*b
    # si a est pair et b est impair
    elif a%2 == 0 and b%2 != 0:
        return a*(b-1)
    # si a est impair et b est pair
    elif a%2 != 0 and b%2 == 0:
        return (a-1)*b
    # sinon - c'est que a et b sont impairs
    else:
        return a*a - b*b
```

dispatch2 - Semaine 4 Séquence 2

```
def dispatch2(a, b, A, B):
    """
    dispatch2 comme spécifié
    """
    # les deux cas de la diagonale \
    if (a in A and b in B) or (a not in A and b not in B):
        return a*a + b*b
    # sinon si b n'est pas dans B
    # ce qui alors implique que a est dans A
    elif b not in B:
        return a*(b-1)
    # le dernier cas, on sait forcément que
    # b est dans B et a n'est pas dans A
    else:
        return (a-1)*b
```

libelle - Semaine 4 Séquence 2

```
def libelle(ligne):
    """
    n'oubliez pas votre docstring
    """
    # on cherche les 3 champs après avoir nettoyé
    # les éléments séparés par une virgule
    mots = [mot.strip() for mot in ligne.split(',')]
    # si on n'a pas le bon nombre de champs
    # rappelez-vous que 'return' tout court
    # est équivalent à 'return None'
    if len(mots) != 3:
        return
    # maintenant on a les trois valeurs
    nom, prenom, rang = mots
    # comment présenter le rang
    rang_ieme = "1er" if rang == "1" \
                else "2nd" if rang == "2" \
                else f"{rang}-ème"
    return f"{prenom}.{nom} ({rang_ieme})"
```

pgcd - Semaine 4 Séquence 3

```
def pgcd(a, b):
    """
    le pgcd de a et b par l'algorithme d'Euclide
    """
    # l'algorithme suppose que a >= b
    # donc si ce n'est pas le cas
    # il faut inverser les deux entrées
    if b > a:
        a, b = b, a
    if b == 0:
        return a
    # boucle sans fin
    while True:
        # on calcule le reste
        reste = a % b
        # si le reste est nul, on a terminé
        if reste == 0:
            return b
        # sinon on passe à l'itération suivante
        a, b = b, reste
```

pgcd_bis - Semaine 4 Séquence 3

```
def pgcd_bis(a, b):
    """
    Il se trouve qu'en fait la première
    inversion n'est pas nécessaire.

    En effet si a <= b, la première itération
    de la boucle while va faire:
    reste = a % b  c'est-à-dire a
    et ensuite
    a, b = b, reste = b, a
    provoque l'inversion
    """
    # si l'on des deux est nul on retourne l'autre
    if a * b == 0:
        return a or b
    # sinon on fait une boucle sans fin
    while True:
        # on calcule le reste
        reste = a % b
        # si le reste est nul, on a terminé
        if reste == 0:
            return b
        # sinon on passe à l'itération suivante
        a, b = b, reste
```

pgcd_ter - Semaine 4 Séquence 3

```
def pgcd_ter(a, b):
    """
    Une autre alternative, qui fonctionne aussi
    C'est plus court, mais on passe du temps à se
    convaincre que ça fonctionne bien comme demandé
    """
    # si on n'aime pas les boucles sans fin
    # on peut faire aussi comme ceci
    while b:
        a, b = b, a % b
    return a
```

taxes - Semaine 4 Séquence 3

```
# une solution très élégante proposée par adrienollier

# les tranches en ordre décroissant
bareme = (
    (150_000, 45),
    (50_000, 40),
    (12_500, 20),
    (0, 0),
)

def taxes(revenu):
    """
    U.K. income taxes calculator
    https://www.gov.uk/income-tax-rates
    """
    montant = 0
    for seuil, taux in bareme:
        if revenu > seuil:
            montant += (revenu - seuil) * taux // 100
            revenu = seuil
    return montant
```

taxes_bis - Semaine 4 Séquence 3

```
1
2   # cette solution est plus pataude; je la retiens
3   # parce qu'elle montre un cas de for .. else ..
4   # qui ne soit pas trop tiré par les cheveux
5   # quoique
6
7   bands = [
8       # à partir de 0. le taux est nul
9       (0, 0.),
10      # jusqu'à 12 500 où il devient de 20%
11      (12_500, 20/100),
12      # etc.
13      (50_000, 40/100),
14      (150_000, 45/100),
15  ]
16
17  def taxes_bis(income):
18      """
19      Utilise un for avec un else
20      """
21      amount = 0
22
23      # en faisant ce zip un peu étrange, on va
24      # considérer les couples de tuples consécutifs dans
25      # la liste bands
26      for (band1, rate1), (band2, _) in zip(bands, bands[1:]):
27          # le salaire est au-delà de cette tranche
28          if income >= band2:
29              amount += (band2-band1) * rate1
30          # le salaire est dans cette tranche
31          else:
32              amount += (income-band1) * rate1
33              # du coup on peut sortir du for par un break
34              # et on ne passera pas par le else du for
35              break
36      # on ne passe ici qu'avec les salaires dans la dernière tranche
37      # en effet pour les autres on est sorti du for par un break
38      else:
39          band_top, rate_top = bands[-1]
40          amount += (income - band_top) * rate_top
41      return int(amount)
```

spreadsheet - Semaine 4 Séquence 3

```
def int_to_char(n):
    """
    traduit un entier entre 1 et 26
    en un caractère entre 'A' et 'Z'
    """
    # si index était compris entre 0 et 25, on pourrait obtenir
    # la lettre comme étant chr(ord('A') + index)
    # on fait donc un changement de variable n -> n-1
    # de plus on va rendre le résultat cyclique modulo 26
    # pour pouvoir l'utiliser sur des nombres quelconques

    return chr(ord('A') + (n - 1) % 26)


def spreadsheet(index):
    """
    transforme un numéro de colonne en nom alphabétique
    dans l'ordre lexicographique
    1 -> A; 26 -> Z; 27 -> AA; 28 -> AB; etc..
    """
    # index peut être supérieur à 26
    # en remarquant que la dernière lettre s'incrémente à chaque fois
    # qu'index augmente, et repasse à 'A' de manière cyclique,
    # on voit qu'on peut utiliser notre version cyclique de `int_to_char`
    # pour calculer la lettre la plus à droite dans le résultat.
    # et pour les autres lettres, il suffit de recommencer sur le quotient

    result = int_to_char(index)
    while index > 26:
        index = (index - 1) // 26
        result = int_to_char(index) + result
    return result
```

spreadsheet_bis - Semaine 4 Séquence 3

```
def spreadsheet_bis(index):
    """
    Accessoirement on peut vérifier que la variable index fournie
    est bien un entier supérieur à 0.
    """
    if not isinstance(index, int):
        raise TypeError("index must be an integer !")
    elif index < 1:
        raise ValueError("index must be positive !")

    result = chr(ord('A') + (index - 1) % 26)
    while index > 26:
        index = (index - 1) // 26
        result = chr(ord('A') + (index - 1) % 26) + result
    return result
```

spreadsheet_ter - Semaine 4 Séquence 3

```
# la fonction int_to_char n'a pas besoin d'être exposée
#  dans l'espace de nommage du module.
# puisque c'est une fonction assistante,
#  on peut en faire une variable locale à spreadsheet_ter
#  en la déclarant à l'intérieur de la fonction
def spreadsheet_ter(index):
    """
    transforme un numéro de colonne en nom alphabétique
    dans l'ordre lexicographique
    1 -> A; 26 -> Z; 27 -> AA; 28 -> AB; etc..
    """
    def int_to_char(n):
        """
        traduit un entier entre 1 et 26
        en un caractère entre 'A' et 'Z'
        """
        return chr(ord('A') + (n - 1) % 26)

    if not isinstance(index, int):
        raise TypeError("index must be an integer!")
    elif index < 1:
        raise ValueError("index must be positive!")

    # ici int_to_char est une variable locale
    # à la fonction spreadsheet_ter
    result = int_to_char(index)
    while index > 26:
        index = (index - 1) // 26
        # idem ici bien sûr
        result = int_to_char(index) + result
    return result
```

power - Semaine 4 Séquence 3

```
def power(x, n):
    """
    mise à la puissance en O(log2(n))
    """
    # on s'astreint à ne pas utiliser ** parce que ce serait triché
    # mais bien sûr dans la pratique
    # on pourrait utiliser **2 pour traiter le cas où n est pair
    if n == 1:
        return x
    elif n % 2 == 0:
        # on met au carré power(x, n//2)
        # une petite subtilité ici, c'est que si vous écrivez
        # root = power(x, n//2) * power(x, n//2)
        # vous allez évaluez **deux fois** power()
        # et du coup vous perdez tout le bénéfice de l'exercice
        root = power(x, n//2)
        return root * root
    else:
        return x * power(x, n-1)
```

distance - Semaine 4 Séquence 6

```
import math

def distance(*args):
    """
    La racine de la somme des carrés des arguments
    """
    # avec une compréhension on calcule
    # la liste des carrés des arguments
    # on applique ensuite sum pour en faire la somme
    # vous pourrez d'ailleurs vérifier que sum ([]) = 0
    # enfin on extrait la racine avec math.sqrt
    return math.sqrt(sum([x**2 for x in args]))
```

distance_bis - Semaine 4 Séquence 6

```
def distance_bis(*args):
    """
    Idem mais avec une expression génératrice
    """
    # on n'a pas encore vu cette forme - cf Semaine 5
    # mais pour vous donner un avant-goût d'une expression
    # génératrice:
    # on peut faire aussi comme ceci
    # observez l'absence de crochets []
    # la différence c'est juste qu'on ne
    # construit pas la liste des carrés,
    # car on n'en a pas besoin
    # et donc un itérateur nous suffit
    return math.sqrt(sum(x**2 for x in args))
```

numbers - Semaine 4 Séquence 6

```
def numbers(*liste):
    """
    retourne un tuple contenant
    (*) la somme
    (*) le minimum
    (*) le maximum
    des éléments de la liste
    """

    if not liste:
        return 0, 0, 0

    return (
        # la builtin 'sum' renvoie la somme
        sum(liste),
        # les builtin 'min' et 'max' font ce qu'on veut aussi
        min(liste),
        max(liste),
    )
```

numbers_bis - Semaine 4 Séquence 6

```
# en regardant bien la documentation de sum, max et min,
# on voit qu'on peut aussi traiter le cas singulier
# (où il n'y pas d'argument) en passant
#   start à sum
#   et default à min ou max
# comme ceci
def numbers_bis(*liste):
    return (
        # attention, la signature de sum est:
        #   sum(iterable[, start])
        # du coup on ne PEUT PAS passer à sum start=0
        # parce que start n'a pas de valeur par défaut
        # on pourrait par contre faire juste sum(liste)
        # car le défaut pour start c'est 0
        # dit autrement, sum([]) retourne bien 0
        sum(liste, 0),
        # par contre avec min c'est
        #  min(iterable, *[, key, default])
        # du coup on DOIT appeler min avec default=0 qui est plus clair
        # l'étoile qui apparaît dans la signature
        # rend le paramètre default keyword-only
        min(liste, default=0),
        max(liste, default=0),
    )
```

multi_tri - Semaine 5 Séquence 2

```
def multi_tri(listes):
    """
    trie toutes les sous-listes
    et retourne listes
    """
    for liste in listes:
        # sort fait un effet de bord
        liste.sort()
    # et on retourne la liste de départ
    return listes
```

multi_tri_reverse - Semaine 5 Séquence 2

```
def multi_tri_reverse(listes, reverses):
    """
    trie toutes les sous listes, dans une direction
    précisée par le second argument
    """
    # zip() permet de faire correspondre les éléments
    # de listes avec ceux de reverses
    for liste, reverse in zip(listes, reverses):
        # on appelle sort en précisant reverse=
        liste.sort(reverse=reverse)
    # on retourne la liste de départ
    return listes
```

tri_custom - Semaine 5 Séquence 2

```
def tri_custom(liste):
    """
    trie une liste en fonction du critère de l'énoncé
    """
    # pour le critère de tri on s'appuie sur l'ordre dans les tuples
    # c'est-à-dire
    # ((1, 2) <= (1, 2, 0) <= (1, 3) <= (2, 0)) == True
    # du coup il suffit que la fonction critère renvoie
    # selon la présence de p2, un tuple de 2 ou 3 éléments
    def custom_key(item):
        if 'p2' in item:
            return (item['p'], item['n'], item['p2'])
        return (item['p'], item['n'])
    liste.sort(key=custom_key)
    return liste
```

tri_custom_bis - Semaine 5 Séquence 2

```
def tri_custom_bis(liste):
    """
    tri avec une fonction lambda et une expression conditionnelle
    """
    # la même chose avec une lambda
    # l'expression conditionnelle est nécessaire ici, car
    # dans une lambda on est limité à des expressions
    liste.sort(key=lambda d: (d['p'], d['n'], d['p2'])
                             if 'p2' in d
                             else (d['p'], d['n']))
    return liste
```

tri_custom_ter - Semaine 5 Séquence 2

```
def tri_custom_ter(liste):
    """
    tri avec une fonction lambda et une compréhension de tuple
    """
    # sous cette forme, tout devient plus simple si on devait
    # avoir d'autres colonnes à prendre en compte
    keys = ('p', 'n', 'p2')
    liste.sort(key=lambda d: tuple(d[k] for k in keys if k in d))
    return liste
```

doubler_premier - Semaine 5 Séquence 2

```
def doubler_premier(func, first, *args):
    """
    renvoie le résultat de la fonction f appliquée sur
    func(2 * first, *args)
    """
    # une fois qu'on a écrit la signature on a presque fini le travail
    # en effet on a isolé la fonction, son premier argument, et le reste
    # des arguments
    # il ne reste qu'à appeler func, en doublant first
    return func(2*first, *args)
```

doubler_premier_bis - Semaine 5 Séquence 2

```
def doubler_premier_bis(func, *args):
    """
    marche aussi mais moins élégant
    """
    first, *remains = args
    return func(2*first, *remains)
```

doubler_premier_ter - Semaine 5 Séquence 2

```
def doubler_premier_ter(func, *args):
    """
    ou encore comme ça, mais
    c'est carrément moche
    """
    first = args[0]
    remains = args[1:]
    return func(2*first, *remains)
```

doubler_premier_kwds - Semaine 5 Séquence 2

```
def doubler_premier_kwds(func, first, *args, **keywords):
    """
    équivalent à doubler_premier
    mais on peut aussi passer des arguments nommés
    """
    # c'est exactement la même chose
    return func(2*first, *args, **keywords)

# Complément - niveau avancé
# ----
# Il y a un cas qui ne fonctionne pas avec cette implémentation,
# quand le premier argument de func a une valeur par défaut
# *et* on veut pouvoir appeler doubler_premier
# en nommant ce premier argument
#
# par exemple - avec func=muln telle que définie dans l'énoncé
#def muln(x=1, y=1): return x*y

# alors ceci:
# doubler_premier_kwds(muln, x=1, y=2)
# ne marche pas car on n'a pas les deux arguments requis
# par doubler_premier_kwds
#
# et pour écrire, disons doubler_permier3, qui marcherait aussi comme cela
# il faudrait faire une hypothèse sur le nom du premier argument...
```

compare_all - Semaine 5 Séquence 2

```
def compare_all(fun1, fun2, entrees):
    """
    retourne une liste de booléens, un par entree dans entrees
    qui indique si fun1(entree) == fun2(entree)
    """
    # on vérifie pour chaque entrée si f et g retournent
    # des résultats égaux avec ==
    # et on assemble le tout avec une comprehension de liste
    return [fun1(entree) == fun2(entree) for entree in entrees]
```

compare_args - Semaine 5 Séquence 2

```
def compare_args(fun1, fun2, arg_tuples):
    """
    retourne une liste de booléens, un par entree dans entrees
    qui indique si fun1(*tuple) == fun2(*tuple)
    """
    # c'est presque exactement comme compare_all, sauf qu'on s'attend
    # à recevoir une liste de tuples d'arguments, qu'on applique
    # aux deux fonctions avec la forme * au lieu de les passer directement
    return [fun1(*arg) == fun2(*arg) for arg in arg_tuples]
```

aplatir - Semaine 5 Séquence 3

```
def aplatir(conteneurs):
    "retourne une liste des éléments des éléments de conteneurs"
    # on peut concaténer les éléments de deuxième niveau
    # par une simple imbrication de deux compréhensions de liste
    return [element for conteneur in conteneurs for element in conteneur]
```

alternat - Semaine 5 Séquence 3

```
def alternat(iter1, iter2):
    """
    renvoie une liste des éléments
    pris alternativement dans iter1 et dans iter2
    """
    # pour réaliser l'alternance on peut combiner zip avec aplatir
    # telle qu'on vient de la réaliser
    return aplatir(zip(iter1, iter2))
```

alternat_bis - Semaine 5 Séquence 3

```
def alternat_bis(iter1, iter2):
    """
    une deuxième version de alternat
    """
    # la même idée mais directement, sans utiliser aplatir
    return [element for conteneur in zip(iter1, iter2)
            for element in conteneur]
```

intersect - Semaine 5 Séquence 3

```
def intersect(tuples_a, tuples_b):
    """
    prend en entrée deux listes de tuples de la forme
    (entier, valeur)

    renvoie l'ensemble des valeurs associées, dans A ou B,
    aux entiers présents dans A et B

    il y a **plein** d'autres façons de faire, mais il faut
    juste se méfier de ne pas tout recalculer plusieurs fois
    si on veut faire trop court

    """

    # pour montrer un exemple de fonction locale:
    # une fonction qui renvoie l'ensemble des entiers
    # présents comme clé dans une liste d'entrée
    def keys(tuples):
        return {entier for entier, valeur in tuples}
    # on l'applique à A et B
    keys_a = keys(tuples_a)
    keys_b = keys(tuples_b)
    #
    # les entiers présents dans A et B
    # avec une intersection d'ensembles
    common_keys = keys_a & keys_b
    # et pour conclure on fait une union sur deux
    # compréhensions d'ensembles
    return {val_a for key, val_a in tuples_a if key in common_keys} \
         | {val_b for key, val_b in tuples_b if key in common_keys}
```

intersect_bis - Semaine 5 Séquence 3

```
def intersect_bis(A, B):
    A, B = dict(A), dict(B)
    keys = set(A) & set(B)
    return {A[k] for k in keys} | {B[k] for k in keys}
```

cesar - Semaine 5 Séquence 3

```
1
2   # pour passer des majuscules aux minuscules, il faut ajouter
3   # 97-65=32
4   import string
5
6   UPPER_TO_LOWER = ord('a') - ord('A')
7
8
9   def cesar(clear, key, encode=True):
10      """
11      retourne l'encryption du caractere <clear> par la clé <key>
12
13      le caractère <key> doit être un caractère alphabétique ASCII
14      c'est à dire que son ord() est entre ceux de 'a' et 'z' ou
15      entre ceux de 'A' et 'Z'
16      """
17
18      if clear not in string.ascii_letters:
19          return clear
20
21      # le codepoint de la clé
22      okey = ord(key)
23      # on normalise la clé pour être dans les minuscules
24      if key.isupper():
25          okey += UPPER_TO_LOWER
26
27      # la variable offset est un entier entre 1 et 26 qui indique
28      # de combien on doit décaler; dans le tout premier
29      # exemple, avec une clé qui vaut 'C' offset va valoir 3
30      offset = (okey - ord('a') + 1)
31
32      # si on encode, il faut ajouter l'offset,
33      # et si on décode, il faut le retrancher
34      if not encode:
35          offset = -offset
36
37      # ne reste plus qu'à faire le modulo
38      # sauf que les bornes ne sont pas les mêmes
39      # pour les majuscules ou pour les minuscules
40      bottom = ord('A') if clear.isupper() else ord('a')
41
42      return chr(bottom + (ord(clear) - bottom + offset) % 26)
```

cesar_bis - Semaine 5 Séquence 3

```
from itertools import chain

# une autre approche entièrement consiste à précalculer
# toutes les valeurs et les ranger dans un dictionnaire
# qui va être haché par le tuple
# (clear, key)
# ça ne demande que 4 * 26 * 26 entrées dans le dictionnaire
# c'est à dire environ 2500 entrées, ce n'est pas grand chose

# on commence par le cas où le texte et la clé sont minuscules
# on rappelle que ord('a')=97
# avec nos définitions, une clé implique un décalage
# de (ord(k)-96), car une clé A signifie un décalage de 1
# par contre pour faire les calculs modulo 26
# il faut faire (ord(c)-97) de façon à ce que A=0 et Z=25
ENCODED_LOWER_LOWER = {
    (c, k): chr((ord(c) - 97 + ord(k) - 96) % 26 + 97)
    for c in string.ascii_lowercase
    for k in string.ascii_lowercase
}

# maintenant on peut facilement en déduire la table
# pour un texte en minuscule et une clé en majuscule
# il suffit d'appliquer ENCODED_LOWER_LOWER avec la clé minuscule
ENCODED_LOWER_UPPER = {
    (c, k): ENCODED_LOWER_LOWER[(c, k.lower())]
    for c in string.ascii_lowercase
    for k in string.ascii_uppercase
}
```

cesar_bis (continued) - Semaine 5 Séquence 3

```
1
2   # enfin pour le cas où le texte est en majuscule, on
3   # va considérer l'union des deux premières tables
4   # (que l'on va balayer avec itertools.chain sur leurs items())
5   # et dire que pour encoder un caractère majuscule, on
6   # n'a qu'à prendre encoder la minuscule et mettre le résultat en majuscule
7   ENCODED_UPPER = {
8       (c.upper(), k): value.upper()
9       for (c, k), value in chain(ENCODED_LOWER_LOWER.items(),
10                                 ENCODED_LOWER_UPPER.items())
11  }
12
13  # maintenant on n'a plus qu'à construire
14  # l'union de ces 3 dictionnaires
15  ENCODE_LOOKUP = {}
16  ENCODE_LOOKUP.update(ENCODED_LOWER_LOWER)
17  ENCODE_LOOKUP.update(ENCODED_LOWER_UPPER)
18  ENCODE_LOOKUP.update(ENCODED_UPPER)
19
20  # et alors pour calculer la table inverse,
21  # c'est extrêmement simple, on dit que
22  # decode(encoded, key) == clear
23  # ssi
24  # encode(clear, key) == encoded
25  DECODE_LOOKUP = {
26      (encoded, key): clear for (clear, key), encoded
27      in ENCODE_LOOKUP.items()
28  }
29
30  # et maintenant pour faire le travail il suffit de
31  # faire exactement **UNE** recherche dans la table qui va bien
32  # ce qui est plus efficace en principe que la première approche
33  # si le couple (texte, clé) n'est pas trouvé alors on renvoie texte tel quel
34  def cesar_bis(clear, key, encode=True):
35      lookup = ENCODE_LOOKUP if encode else DECODE_LOOKUP
36      return lookup.get((clear, key), clear)
```

vigenere - Semaine 5 Séquence 3

```
from itertools import cycle

# grâce à une combinaison de zip et de itertools.cycle
# on peut itérer sur
# d'une part, le message
# et d'autre part, sur la clé, en boucle
#
# notez que
# (*) cycle ne s'arrête jamais
# (*) mais zip, lui, s'arrête au plus court de ses (ici deux)
#     ingrédients
# ce qui fait que zip(message, cycle(cle))
# fait exactement ce dont on a besoin

def vigenere(clear, key, encode=True):
    return "".join(
        cesar(c, k, encode)
        for c, k in zip(clear, cycle(key))
    )
```

produit_scalaire - Semaine 5 Séquence 4

```
def produit_scalaire(vec1, vec2):
    """
    retourne le produit scalaire
    de deux listes de même taille
    """
    # avec zip() on peut faire correspondre les
    # valeurs de vec1 avec celles de vec2 de même rang
    #
    # et on utilise la fonction builtin sum sur une itération
    # des produits x1*x2
    #
    # remarquez bien qu'on utilise ici une expression génératrice
    # et PAS une compréhension car on n'a pas du tout besoin de
    # créer la liste des produits x1*x2
    #
    return sum(x1 * x2 for x1, x2 in zip(vec1, vec2))
```

produit_scalaire_bis - Semaine 5 Séquence 4

```
# Il y a plein d'autres solutions qui marchent aussi
#
def produit_scalaire_bis(vec1, vec2):
    """
    Une autre version, où on fait la somme à la main
    """
    scalaire = 0
    for x1, x2 in zip(vec1, vec2):
        scalaire += x1 * x2
    # on retourne le résultat
    return scalaire
```

produit_scalaire_ter - Semaine 5 Séquence 4

```
# Et encore une:
# celle-ci par contre est assez peu "pythonique"
#
# considérez-la comme un exemple de
# ce qu'il faut ÉVITER DE FAIRE:
#
def produit_scalaire_ter(vec1, vec2):
    """
    Lorsque vous vous trouvez en train d'écrire:

        for i in range(len(sequence)):
            x = iterable[sequence]
            # etc...

    vous pouvez toujours écrire à la place:

        for x in sequence:
            ...

    qui en plus d'être plus facile à lire,
    marchera sur tout itérable, et sera plus rapide
    """
    scalaire = 0
    # sachez reconnaitre ce vilain idiome:
    for i in range(len(vec1)):
        scalaire += vec1[i] * vec2[i]
    return scalaire
```

decode_zen - Semaine 5 Séquence 7

```
# le module this est implémenté comme une petite énigme
#
# comme le laissent entrevoir les indices, on y trouve
# (*) dans l'attribut 's' une version encodée du manifeste
# (*) dans l'attribut 'd' le code à utiliser pour décoder
#
# ce qui veut dire qu'en première approximation, on pourrait
# énumérer les caractères du manifeste en faisant
# (this.d[c] for c in this.s)
#
# mais ce serait le cas seulement si le code agissait sur
# tous les caractères; mais ce n'est pas le cas, il faut
# laisser intacts les caractères de this.s qui ne sont pas
# dans this.d

def decode_zen(this_module):
    """
    décode le zen de python à partir du module this
    """
    # la version encodée du manifeste
    encoded = this_module.s
    # le dictionnaire qui implémente le code
    code = this_module.d
    # si un caractère est dans le code, on applique le code
    # sinon on garde le caractère tel quel
    # aussi, on appelle 'join' pour refaire une chaîne à partir
    # de la liste des caractères décodés
    return ''.join(code[c] if c in code else c for c in encoded)
```

decode_zen_bis - Semaine 5 Séquence 7

```
# une autre version un peu plus courte
#
# on utilise la méthode get d'un dictionnaire,
# qui permet de spécifier (en second argument)
# quelle valeur on veut utiliser dans les cas où la
# clé n'est pas présente dans le dictionnaire
#
# dict.get(key, default)
# retourne dict[key] si elle est présente, et default sinon

def decode_zen_bis(this_module):
    """
    une autre version, un peu plus courte
    """
    return "".join(this_module.d.get(c, c) for c in this_module.s)
```

decode_zen_ter - Semaine 5 Séquence 7

```
# une dernière version utilisant les fonctions ad hoc
# https://docs.python.org/3/library/stdtypes.html#str.translate
# et https://docs.python.org/3/library/stdtypes.html#str.maketrans

def decode_zen_ter(this_module):
    """
    cette version utilise les fonctions ad hoc de la classe str
    """
    # Le dictionnaire this_module.d n'est pas utilisable directement,
    # il faut faire la transformation fournie par str.maketrans
    # car la fonction translate attend comme clés des nombres
    # représentant la valeur Unicode des caractères.
    # Or this_module.d a comme clés les caractères à décoder
    # et non leur valeur Unicode.
    return this_module.s.translate(str.maketrans(this_module.d))
```

shipdict - Semaine 6 Séquence 4

```
1
2   # helpers - used for verbose mode only
3   # could have been implemented as static methods in Position
4   # but we had not seen that at the time
5
6
7   def d_m_s(f):
8       """
9       make a float readable; e.g. transform 2.5 into 2.30'00''
10      we avoid using the degree sign to keep things simple
11      input is assumed positive
12      """
13      d = int(f)
14      m = int((f - d) * 60)
15      s = int((f - d) * 3600 - 60 * m)
16      return "{:02d}.{:02d}'{:02d}''".format(d, m, s)
17
18
19  def lat_d_m_s(f):
20      """
21      degree-minute-second conversion on a latitude float
22      """
23      if f >= 0:
24          return "{} N".format(d_m_s(f))
25      else:
26          return "{} S".format(d_m_s(-f))
27
28
29  def lon_d_m_s(f):
30      """
31      degree-minute-second conversion on a longitude float
32      """
33      if f >= 0:
34          return "{} E".format(d_m_s(f))
35      else:
36          return "{} W".format(d_m_s(-f))
```

shipdict_suite - Semaine 6 Séquence 4

```
1
2
3   class Position(object):
4       "a position atom with timestamp attached"
5
6       def __init__(self, latitude, longitude, timestamp):
7           "constructor"
8           self.latitude = latitude
9           self.longitude = longitude
10          self.timestamp = timestamp
11
12  # all these methods are only used when merger.py runs in verbose mode
13      def lat_str(self):
14          return lat_d_m_s(self.latitude)
15
16      def lon_str(self):
17          return lon_d_m_s(self.longitude)
18
19      def __repr__(self):
20          """
21          only used when merger.py is run in verbose mode
22          """
23          return f"<{self.lat_str()} {self.lon_str()} @ {self.timestamp}>"
24
25      # required to be stored in a set
26      # see https://docs.python.org/3/reference/datamodel.html#object.__hash__
27      def __hash__(self):
28          return hash((self.latitude, self.longitude, self.timestamp))
29
30      # a hashable shall override this special method
31      def __eq__(self, other):
32          return (self.latitude == other.latitude
33                  and self.longitude == other.longitude
34                  and self.timestamp == other.timestamp)
```

shipdict_suite - Semaine 6 Séquence 4

```
1
2
3  class Ship(object):
4      """
5      a ship object, that requires a ship id,
6      and optionnally a ship name and country
7      which can also be set later on
8
9      this object also manages a list of known positions
10     """
11
12     def __init__(self, id, name=None, country=None):
13         "constructor"
14         self.id = id
15         self.name = name
16         self.country = country
17         # this is where we remember the various positions over time
18         self.positions = []
19
20     def add_position(self, position):
21         """
22         insert a position relating to this ship
23         positions are not kept in order so you need
24         to call `sort_positions` once you're done
25         """
26         self.positions.append(position)
27
28     def sort_positions(self):
29         """
30         sort of positions made unique thanks to the set by chronological order
31         for this to work, a Position must be hashable
32         """
33         self.positions = sorted(set(self.positions),
34                                 key=lambda position: position.timestamp)
```

shipdict_suite - Semaine 6 Séquence 4

```
1
2
3   class ShipDict(dict):
4       """
5       a repository for storing all ships that we know about
6       indexed by their id
7       """
8
9       def __init__(self):
10          "constructor"
11          dict.__init__(self)
12
13      def __repr__(self):
14          return f"<ShipDict instance with {len(self)} ships>"
15
16      @staticmethod
17      def is_abbreviated(chunk):
18          """
19          depending on the size of the incoming data chunk,
20          guess if it is an abbreviated or extended data
21          """
22          return len(chunk) <= 7
23
24      def add_abbreviated(self, chunk):
25          """
26          adds an abbreviated data chunk to the repository
27          """
28          id, latitude, longitude, *_, timestamp = chunk
29          if id not in self:
30              self[id] = Ship(id)
31          ship = self[id]
32          ship.add_position(Position(latitude, longitude, timestamp))
33
34      def add_extended(self, chunk):
35          """
36          adds an extended data chunk to the repository
37          """
38          id, latitude, longitude = chunk[:3]
39          timestamp, name = chunk[5:7]
40          country = chunk[10]
41          if id not in self:
42              self[id] = Ship(id)
43          ship = self[id]
44          if not ship.name:
45              ship.name = name
46              ship.country = country
47          self[id].add_position(Position(latitude, longitude, timestamp))
```

shipdict_suite - Semaine 6 Séquence 4

```
    def add_chunk(self, chunk):
        """
        chunk is a plain list coming from the JSON data
        and be either extended or abbreviated

        based on the result of is_abbreviated(),
        gets sent to add_extended or add_abbreviated
        """
        # here we retrieve the static method through the class
        # this form outlines the fact that we're calling a static method
        # note that
        # self.is_abbreviated(chunk)
        # would work fine just as well
        if ShipDict.is_abbreviated(chunk):
            self.add_abbreviated(chunk)
        else:
            self.add_extended(chunk)

    def sort(self):
        """
        makes sure all the ships have their positions
        sorted in chronological order
        """
        for id, ship in self.items():
            ship.sort_positions()

    def clean_unnamed(self):
        """
        Because we enter abbreviated and extended data
        in no particular order, and for any time period,
        we might have ship instances with no name attached
        This method removes such entries from the dict
        """
        # we cannot do all in a single loop as this would amount to
        # changing the loop subject
        # so let us collect the ids to remove first
        unnamed_ids = {id for id, ship in self.items()
                       if ship.name is None}
        # and remove them next
        for id in unnamed_ids:
            del self[id]
```

shipdict_suite - Semaine 6 Séquence 4

```
    def ships_by_name(self, name):
        """
        returns a list of all known ships with name <name>
        """
        return [ship for ship in self.values() if ship.name == name]

    def all_ships(self):
        """
        returns a list of all ships known to us
        """
        # we need to create an actual list because it
        # may need to be sorted later on, and so
        # a raw dict_values object won't be good enough
        return list(self.values())

```

two_sum - Semaine 6 Séquence 9

```
def two_sum(liste, target):
    """
    retourne un tuple de deux indices de deux nombres
    dans la liste dont la somme fait target
    """
    for i, item1 in enumerate(liste):
        for j, item2 in enumerate(liste):
            # prune the loop on j altogether once we reach i
            if j >= i:
                break
            if item1 + item2 == target:
                return j, i
```

two_sum_bis - Semaine 6 Séquence 9

```
from itertools import product


def two_sum_bis(liste, target):
    """
    pareil en utilisant itertools.product
    pour éviter les deux for imbriqués
    un tout petit peu moins efficace ici car on est dans une seule
    boucle et donc on ne peut pas avorter la boucle interne
    avec break
    """
    for (i, item1), (j, item2) in product(
        enumerate(liste), enumerate(liste)):
        if i >= j:
            continue
        if item1 + item2 == target:
            return i, j
```

two_sum_ter - Semaine 6 Séquence 9

```
def two_sum_ter(liste, target):
    """
    toujours avec product, pour illustrer l'usage de repeat=
    """
    for (i, item1), (j, item2) in product(
        enumerate(liste), repeat=2):
        if i >= j:
            continue
        if item1 + item2 == target:
            return i, j
```

longest_gap - Semaine 6 Séquence 9

```
def longest_gap(liste):
    result = 0
    begins = {}
    for index, item in enumerate(liste):
        if item not in begins:
            begins[item] = index
        else:
            result = max(result, index - begins[item])
    return result
```

meeting - Semaine 6 Séquence 9

```
def meeting(string):
    """découpage et tri"""
    persons = []
    person_strings = string.split(';')
    for person_string in person_strings:
        first, last = person_string.split(':')
        # il faut 2 niveaux de parenthèse car on insére un tuples
        persons.append((last, first))
    # on s'appuie sur le tri des tuples qui fait justement
    # ce qu'on veut
    persons.sort()
    return "".join(f"({last}, {first})" for last, first in persons)
```

meeting_bis - Semaine 6 Séquence 9

```
def meeting_bis(string):
    # on élabore une liste de [first, last]
    exploded = [ token.split(':') for token in string.split(';') ]
    # on met le nom en premier, dans des tuples
    persons = [ (last, first) for (first, last) in exploded ]
    # on trie, toujours avec le tri sur les tuples
    persons.sort()
    # on met en forme
    return "".join(f"({last}, {first})" for last, first in persons)
```

postfix_eval - Semaine 6 Séquence 9

```
def postfix_eval(chaine):
    """
    an evaluator for postfixed expressions

    all operands are integers, and division is integer division
    i.e. // i.e. quotient

    input is a string

    example:

    "5 3 + 4 2 - *" -> 16
    """
    stack = []
    # split the line into tokens
    tokens = chaine.split()
```

postfix_eval (continued) - Semaine 6 Séquence 9

```
    for token in tokens:
        operand = None
        try:
            # if it is an integer
            operand = int(token)
            # then all we need to do is push
            stack.append(operand)
        except ValueError:
            # if it's not, it's a little more complex
            operator = token
            # first our operations are all on 2 operands
            # so we can pop those, provided there's enough on the stack
            if len(stack) < 2:
                # error: not enough values to operate on
                return 'error-empty-stack'
            # first element in the stack is the rightmost operand
            right = stack.pop()
            left = stack.pop()
            # is it one of the supported operations ?
            if operator == '+':
                stack.append(left + right)
            elif operator == '-':
                stack.append(left - right)
            elif operator == '*':
                stack.append(left * right)
            elif operator == '/':
                stack.append(left // right)
            else:
                # error: unknown op
                return 'error-syntax'
    # at this point we must have **exactly one** item in the stack
    if len(stack) == 0:
        return 'error-empty-stack'
    elif len(stack) > 1:
        return 'error-unfinished'

    return stack.pop()
```

postfix_eval_bis - Semaine 6 Séquence 9

```
# exact same behaviour, but this version uses a dictionary to
# avoid the awkward part where we check for a supported operator

# use a dictionary , to map
#    each operator sign (like '+')
#    -> to a binary function (i.e. that accepts 2 parameter)
#
# we could have defined these 4 functions manually, but
# it turns out the operator module comes in handy
from operator import add, mul, sub, floordiv

operator_map = { '+' : add, '*': mul, '-': sub, '/' : floordiv }

def postfix_eval_bis(chaine):
    """
    same
    """
    stack = []
    tokens = chaine.split()
    for token in tokens:
        operand = None
        try:
            operand = int(token)
            stack.append(operand)
        except ValueError:
            operator = token
            if len(stack) < 2:
                # error: not enough values to operate on
                return 'error-empty-stack'
            right = stack.pop()
            left = stack.pop()
            # operator here is typically '+'
            # and its value in the map is a binary function
            if operator in operator_map:
                function = operator_map[operator]
                stack.append(function(left, right))
            else:
                # error: unknown op
                return 'error-syntax'
    if len(stack) == 0:
        return 'error-empty-stack'
    elif len(stack) > 1:
        return 'error-unfinished'

    return stack.pop()
```

postfix_eval_typed - Semaine 6 Séquence 9

```
def postfix_eval_typed(chaine, result_type):
    """
    a postfix evaluator, using a parametric type
    that can be either `int`, `float` or `Fraction` or similars
    """
    operators = {
        '+': lambda x, y: x+y,
        '-': lambda x, y: x-y,
        '*': lambda x, y: x*y,
        '/': lambda x, y: x//y if issubclass(result_type, int) else x/y,
    }

    stack = []
    for token in chaine.split():
        if token in operators:
            # compute operation on last 2 entries
            try:
                rhs = stack.pop()
                lhs = stack.pop()
            except:
                return "error-empty-stack"
            result = operators[token](lhs, rhs)
            stack.append(result)
        else:
            try:
                stack.append(result_type(token))
            except:
                return 'error-syntax'
            # parse as int and stack up
    if len(stack) != 1:
        return 'error-unfinished'
    return stack.pop()
```

polynomial - Semaine 6 Séquence 9

```
class Polynomial:
    """
    a class that models polynomials

    example:
        >>> f = Polynomial(3, 2, 1)
        3X^2 + 2X +1
        >>> f(10)
        321
    """


    # pretty print one monomial
    @staticmethod
    def repr_monomial(degre, coef):
        if coef == 0:
            return "0"
        elif degre == 0:
            return str(coef)
        elif degre == 1:
            return f"{coef}X" if coef != 1 else "X"
        elif coef == 1:
            return f"X^{degre}"
        else:
            return f"{coef}X^{degre}"


    def __init__(self, *high_first):
        # internal structure is a tuple of coeficients,
        # index 0 being the constant part
        # so we reverse the incoming parameters
        def skip_first_nulls(coefs):
            valid = False
            for coef in coefs:
                if coef:
                    valid = True
                if valid:
                    yield coef
        self.coefs = tuple(skip_first_nulls(high_first))[::-1]


    def __repr__(self):
        if not self.coefs:
            return '0'
        return " + ".join(reversed(
            [self.repr_monomial(d, c) for (d, c) in enumerate(self.coefs) if c]))
```

polynomial (continued) - Semaine 6 Séquence 9

```
    def _get_degree(self):
        return 0 if not self.coefs else (len(self.coefs) - 1)
    degree = property(_get_degree)


    def __eq__(self, other):
        return self.coefs == other.coefs


    def __add__(self, other):
        """add 2 Polynomial instances"""
        # this interesting thing here is the use of zip_longest
        # so that our resulting Polynomial has a degree that is the max
        # of the degrees of our operands
        # also note the use of a so-called splat operator
        # beause we need to call e.g. Polynomial(1, 2, 3) and
        # not Polynomial( [1, 2, 3])
        small_first = [c1+c2
                       for (c1, c2) in zip_longest(
                               self.coefs, other.coefs, fillvalue=0)]
        return Polynomial(*reversed(small_first))


    def __mul__(self, other):
        """multiply 2 polynomials"""
        # a rather inefficient implementation
        # - because accessing a list by index is inefficient
        # just to illustrate product() and repeat()
        result_degree = self.degree + other.degree + 1
        result_coefs = list(repeat(0, result_degree))
        for (i, c), (j, d) in product(
                enumerate(self.coefs), enumerate(other.coefs)):
            result_coefs[i+j] += c*d
        return Polynomial(*reversed(result_coefs))
```

polynomial (continued) - Semaine 6 Séquence 9

```
    def __call__(self, param):
        """make instances callable"""
        # this is an interesting idiom
        # reduce allows to apply a 2-argument function
        # on an iterable from left to right
        # that is to say for example
        # reduce(foo, [1, 2, 3, 4]) -> foo(1, foo(2, foo(3, 4))
        # in this code the function object created
        # with the lambda expression is called a closure
        # it 'captures' the 'param' parameter in a function
        # that takes 2 arguments
        return reduce(lambda a, b: a*param + b, self.coefs[::-1])


    def derivative(self):
        """
        the derivative is a polynomial as well
        """
        # 2 things are happening here
        # (*) we use the count() iterator; this never terminates
        #   except that it is embedded in a zip() that will
        #   terminate when iterating over our own coefficients expires
        # (*) here again observe the use of a splat operator

        derived_coefs = (n * c for (n, c) in zip(
                         count(1),
                         self.coefs[1:]
                        ))
        return Polynomial(*derived_coefs)
```

temperature - Semaine 6 Séquence 9

```
class Temperature:
    """
    a class that models temperatures

    example:
       >>> k = Temperature(kelvin=0); k
       0 °K
       >>> c = Temperature(celsius=0); c
       -273 °K
       >>> c.kelvin
       -273
       >>> k.celsius
       273
    """

    KELVIN = 273

    def __init__(self, *,
                 # that star sign above means that any parameter
                 # **MUST BE NAMED**, and that one cannot call
                 # e.g. Temperature(10)
                 kelvin=None, celsius=None):
        # in case no parameter is set
        if kelvin is None and celsius is None:
            kelvin = 0
        # in case both are set
        if kelvin is not None and celsius is not None:
            raise ValueError("Temperature wants only one among kelvin and celsius")
        # our unique internal data is _kelvin
        # but even from the constructor we'll
        # access it **only through properties**
        if kelvin is not None:
            # this calls _set_kelvin()
            self.kelvin = kelvin
        else:
            # this calls _set_celsius()
            self.celsius = celsius
```

temperature (continued) - Semaine 6 Séquence 9

```
    def __repr__(self):
        return f"{self._kelvin:d}°"


    def __eq__(self, other):
        return self._kelvin == other._kelvin


    def __sub__(self, other):
        return self._kelvin - other.kelvin


    # PROPERTIES

    def _get_kelvin(self):
        return self._kelvin
    def _set_kelvin(self, kelvin):
        if kelvin < 0:
            raise ValueError(f"Temperature needs a positive kelvin (got {kelvin}K)")
        self._kelvin = kelvin

    kelvin = property(_get_kelvin, _set_kelvin)


    def _get_celsius(self):
        # celsius + KELVIN = kelvin
        return self._kelvin - self.KELVIN
    def _set_celsius(self, celsius):
        self.kelvin = celsius + self.KELVIN

    celsius = property(_get_celsius, _set_celsius)

```

primes - Semaine 6 Séquence 9

```
import math
import itertools

def primes():
    """
    enumerate prime numbers
    """
    # the primes we have found so far
    previous = [2, 3]
    yield 2
    yield 3
    # consider only odd numbers
    for n in itertools.count(5, 2):
        # deemed prime until we find a divisor
        is_prime = True
        # no need to go beyond this
        root = math.sqrt(n)
        # try only primes
        for i in previous:
            # above root, no need to go on
            if i > root:
                break
            # a divisor is found
            # no need to go on either
            if n % i == 0:
                is_prime = False
                break
        # yield, and record in previous
        if is_prime:
            previous.append(n)
            yield n
```

prime_squares - Semaine 6 Séquence 9

```
def prime_squares():
    """
    iterates over the squares of prime numbers
    """
    # a generator expression is the most obvious way that springs to mind
    return (prime**2 for prime in primes())
```

prime_squares_bis - Semaine 6 Séquence 9

```
def prime_squares_bis():
    """
    same using a generator function
    """
    # a generator expression is the most obvious way that springs to mind
    for prime in primes():
        yield prime**2
```

prime_legos - Semaine 6 Séquence 9

```
import itertools

def prime_legos():
    """
    iterates over shifted primes (with a 5-items padding with 1s)
    and over primes squares
    """
    part1 = itertools.chain(itertools.repeat(1, 5), primes())
    part2 = (prime**2 for prime in primes())
    return zip(part1, part2)
```

prime_legos_bis - Semaine 6 Séquence 9

```
import itertools

def prime_legos_bis():
    """
    same behaviour
    we optimize CPU performance by creating a single instance
    of the primes() generator, and duplicate it using `itertools.tee()`
    """
    # this is where the pseudo-copy takes place
    primes1, primes2 = itertools.tee(primes(), 2)
    # the rest is of course the same as in the naive version
    part1 = itertools.chain(itertools.repeat(1, 5), primes1)
    part2 = (prime**2 for prime in primes2)
    return zip(part1, part2)
```

prime_th_primes - Semaine 6 Séquence 9

```
def prime_th_primes():
    """
    iterate the n-th prime number, with n it self being prime

    given that primes() emits 2, 3, 5
    then prime_th_primes() starts with 5 which has index 2 in that enumeration
    """
    # optimizing a bit, don't compute primes twice
    primes1, primes2 = itertools.tee(primes())

    # current will scan all prime numbers
    current = next(primes1)
    # index will scan all integers
    for index, prime in enumerate(primes2):
        # when it matches 'current' it means we have a winner
        if index == current:
            yield prime
            current = next(primes1)
```

prime_th_primes_bis - Semaine 6 Séquence 9

```
def prime_th_primes_bis():
    """
    same purpose

    this approach is a little more manual
    as we do our own calls to next()

    """
    # optimizing a bit, don't compute primes twice
    primes1, primes2 = itertools.tee(primes())

    # this start with -1 because it's a number of times we need to do next()
    # and, as opposed with usual indexing that starts at 0
    # to get item at index 0 we need to do ONE next()
    current_index = -1

    while True:
        # what's the next prime index
        next_index = next(primes1)
        # the amount of times we must iterate on primes2
        offset = next_index - current_index
        # move primes2 forward that many times
        for _ in range(offset):
            output = next(primes2)
        # we have a winner
        yield output
        # this is where we are, so we can compute the next hop
        current_index = next_index
```

redirector1 - Semaine 6 Séquence 9

```
class Redirector1:
    """
    a class that redirects any attribute as a lowercase
    dash-separated version of the attribute name
    """
    def __repr__(self):
        return "redirector"

    # desired behaviour is obtained by a simple
    # invokation of __getattr__
    # that is invoked each time an attribute is read
    # but is found missing in the local namespace
    def __getattr__(self, attribute_name):
        return attribute_name.lower().replace('_', '-')
```

redirector2 - Semaine 6 Séquence 9

```
class Redirector2:
    """
    a class that redirects any attribute as a method that returns
    a string made of (*) the redirector's id, (*) the attribute name,
    and (*) the argument passed to the method
    """

    def __init__(self, id):
        self.id = id

    def __repr__(self):
        return f"Redirector2({self.id})"

    # in this version, we rely on the same special method
    # but this time __getattr__ needs to return a method
    # that accepts one argument

    def __getattr__(self, methodname):
        # doit retourner une 'bound method'
        # du coup on ne recevra pas `self` comme premier paramètre
        def synthetic_method(argument):
            return f"{self.id} -> {methodname}({argument})"
        # optionnel, voir chapitre sur décorateurs
        synthetic_method.__name__ = methodname
        return synthetic_method
```

treescanner - Semaine 6 Séquence 9

```
def treescanner(tree):
    """
    enumerate all leaves in a tree
    """
    # a typical example where
    # the 'yield from' statement
    # is the only way to go
    if isinstance(tree, list):
        for subtree in tree:
            yield from treescanner(subtree)
    else:
        yield tree
```

roman - Semaine 6 Séquence 9

```
import functools
from math import nan, isnan


@functools.total_ordering
class Roman:
    """
    a class to implement limited arithmetics on roman numerals

    example:
        >>> r1, r2 = Roman(2020), Roman('XXII')
        >>> r1
        MMXX=2020
        >>> r2
        XXII=22
        >>> r1-r2
        MCMXCVIII=1998
    """

    def __init__(self, letters_or_integer):
        if isinstance(letters_or_integer, (int, str)):
            try:
                # pour gérer les chaînes de caractères
                # représentant un nombre entier
                # ex. : convertir '123' en l'entier 123
                integer = int(letters_or_integer)
            # si la conversion échoue, c'est qu'on a affaire à une str
            except ValueError:
                letters = letters_or_integer.upper()
                self._decimal = Roman.roman_to_decimal(letters)
                self._roman = 'N' if isnan(self._decimal) else letters
            # sinon c'est que c'est bien un entier
            else:
                self._roman = Roman.decimal_to_roman(integer)
                self._decimal = nan if self._roman == 'N' else integer
        elif isnan(letters_or_integer):
            self._decimal = nan
            self._roman = 'N'
        else:
            raise TypeError(
              f"Cannot initialize Roman from type {type(letters_or_integer)}")
```

roman (continued) - Semaine 6 Séquence 9

```
    def __repr__(self):
        return f"{self._roman}={self._decimal}"

    def __str__(self):
        return self._roman

    def __eq__(self, other):
        return self._decimal == other._decimal

    def __lt__(self, other):
        return self._decimal < other._decimal

    def __add__(self, other):
        return Roman(self._decimal + other._decimal)

    def __sub__(self, other):
        return Roman(self._decimal - other._decimal)

    def __int__(self):
        return self._decimal
```

roman (continued) - Semaine 6 Séquence 9

```
    # table de correspondance des nombres décimaux et
    # des nombres romains clés
    symbols = {
        1: 'I',
        5: 'V',
        10: 'X',
        50: 'L',
        100: 'C',
        500: 'D',
        1000: 'M'
    }

    @staticmethod
    def decimal_to_roman(decimal: int) ->str:
        """
        Conversion from decimal number to roman number.
        """
        if decimal <= 0:
            return 'N'

        # la chaîne de caractères résultante, construite étape par étape
        roman = ""
        # les puissances de 10 successives
        tens = 0

        try:
            while decimal:
                unit = decimal % 10
                if unit in (1, 2, 3):
                    # mettre unit fois le symbole de
                    # la puissance de 10 correspondante
                    roman = Roman.symbols[10 ** tens] * unit + roman
                elif 4 <= unit <= 8:
                    # mettre le symbole de 5 fois la puissance de 10
                    # correspondante précédé ou suivi du symbole de la
                    # puissance de 10 correspondante
                    roman = (Roman.symbols[10 ** tens] * (5 - unit)
                             + Roman.symbols[5 * 10 ** tens]
                             + Roman.symbols[10 ** tens] * (unit - 5)
                             + roman)
                elif unit == 9:
                    # le symbole de la puissance de 10 correspondante
                    # suivi de la puissance de 10 suivante
                    roman = (Roman.symbols[10 ** tens]
                             + Roman.symbols[10 ** (tens + 1)]
                             + roman)
                tens += 1
                decimal //= 10
        except KeyError:
            return 'N'
        else:
            return roman
```

roman (continued) - Semaine 6 Séquence 9

```
    # table de correspondance inversée
    # isymbols = inverted symbols
    isymbols = {v: k for k, v in symbols.items()}

    @staticmethod
    def roman_to_decimal(roman: str) ->int:
        """
        Conversion from roman number to decimal number
        """
        if not roman:
            return nan

        # la valeur décimale résultante, construite petit à petit
        decimal = 0
        # pour stocker le caractère précédent
        previous = None

        try:
            for r in roman:
                # Si le symbole précédent a une valeur moins grande,
                # il faut l'enlever une fois parce qu'on l'a compté
                # au coup précédent alors qu'il ne fallait pas,
                # et l'enlever une seconde fois parce qu'il faut
                # le soustraire à la valeur du symbole courant.
                # C'est ainsi que fonctionne le système numérique romain.
                if previous and Roman.isymbols[previous] < Roman.isymbols[r]:
                    if Roman.isymbols[r] // Roman.isymbols[previous] in (5, 10):
                        decimal -= 2 * Roman.isymbols[previous]
                    else:
                        return nan
                decimal += Roman.isymbols[r]
                previous = r
        except KeyError:
            return nan
        else:
            return decimal
```

quaternion - Semaine 6 Séquence 9

```
def number_str(x):
    if isinstance(x, int):
        return f"{x}"
    elif isinstance(x, float):
        return f"{x:.1f}"

class Quaternion:

    # possible enhancement: we could also have decided to
    # accept a single parameter, if int or float or complex
    def __init__(self, a, b, c, d):
        self.implem = (a, b, c, d)


    def __repr__(self):
        labels = ['', 'i', 'j', 'k']
        # on prépare des morceaux comme '3', '2i', '4j', '5k'
        # mais seulement si la dimension en question n'est pas nulle
        parts = (f"{number_str(x)}{label}"
                 for x, label in zip(self.implem, labels) if x)

        # on les assemble avec un + au milieu
        full = " + ".join(parts)

        # si c'est vide c'est que self est nul
        return full if full != "" else "0"
```

quaternion (continued) - Semaine 6 Séquence 9

```
    # possible enhancement: accept other
    # of builtin number types
    def __add__(self, other):
        """
        implements q1 + q2
        """
        return Quaternion(
            *(x+y for x, y in zip(self.implem, other.implem)))


    # ditto: possible enhancement: accept other
    # of builtin number types
    def __mul__(self, other):
        """
        implements q1 * q2
        """
        a1, b1, c1, d1 = self.implem
        a2, b2, c2, d2 = other.implem
        a = a1 * a2 - b1 * b2 - c1 * c2 - d1 * d2
        b = a1 * b2 + b1 * a2 + c1 * d2 - d1 * c2
        c = a1 * c2 + c1 * a2 + d1 * b2 - b1 * d2
        d = a1 * d2 + d1 * a2 + b1 * c2 - c1 * b2
        return Quaternion(a, b, c, d)


    def __eq__(self, other):
        """
        implements q1 == q2

        here we have decided to allow for comparison
        with a regular number
        """
        if isinstance(other, (bool, int, float)):
            return self == Quaternion(other, 0, 0, 0)
        elif isinstance(other, complex):
            return self == Quaternion(other.real, other.imag, 0, 0)
        elif isinstance(other, Quaternion):
            return self.implem == other.implem
        else:
            return False
```

checkers - Semaine 7 Séquence 05

```
def checkers(size, corner_0_0=True):
    """
    Un damier
    le coin (0, 0) vaut 1 ou 0 selon corner_0_0
    se souvenir que False == 0 et True == 1

    credits: JeF29 pour avoir suggéré une simple
    addition plutôt qu'un xor
    """
    # on peut voir le damier comme une fonction sur
    # les coordonnées, du genre (i + j) % 2
    # pour choisir le coin, on ajoute avant de faire le % 2
    I, J = np.indices((size, size))
    return (I + J + corner_0_0) % 2
```

checkers_2 - Semaine 7 Séquence 05

```
def checkers_2(size, corner_0_0=True):
    """
    sur une ligne, avec
    * sum() pour l'addition I + J

    et, pour les illustrer un petit, les opérateurs bit-wise:
    * et logique (&) pour le modulo 2
    * et xor (^) pour inverser

    credits: j4l4y
    """
    # avec sum() sur indices()
    # on peut tout faire en une ligne:
    return sum(np.indices((size, size))) & 1 ^ corner_0_0
```

checkers_3 - Semaine 7 Séquence 05

```
def checkers_3(size, corner_0_0=True):
    """
    Une autre approche complètement
    """
    # on part de zéro
    result = np.zeros(shape=(size, size), dtype=int_)
    # on remplit les cases à 1 en deux fois
    # avec un slicing astucieux; c'est le ::2 qui fait le travail
    result[1::2, 0::2] = 1
    result[0::2, 1::2] = 1
    # encore une autre façon de renverser,
    # plutôt que le xor, puisque False == 0 et True == 1
    if corner_0_0:
        result = 1 - result
    return result
```

checkers_4 - Semaine 7 Séquence 05

```
def checkers_4(size, corner_0_0=True):
    """
    Et encore une autre, sans doute pas très lisible
    mais très astucieuse

    credits: j4l4y
    """
    # une utilisation très astucieuse de resize,
    # broadcasting, décalage, bravo !
    return (np.resize((corner_0_0, 1-corner_0_0),
                      (1, size))
            ^ np.arange(size)[:, np.newaxis] & 1)
```

hundreds - Semaine 7 Séquence 05

```
def hundreds(lines, columns, offset):
    """
    Fabrique un tableau lines x columns où:

    tab[i, j] = 100 * i + 10 * j + offset
    """
    # avec indices(), on a directement
    # deux tableaux prêts à être broadcastés
    indx, indy = np.indices((lines, columns))
    return 100*indx + 10*indy + offset
```

hundreds_bis - Semaine 7 Séquence 05

```
def hundreds_bis(lines, columns, offset):
    """
    Pareil, toujours à base de broadcasting
    """
    # cette fois on se fabrique soi-même la souche
    # des lignes et des colonnes pour montrer
    # comment on peut se faire indices() à la main
    # dans du vrai code, utilisez indices()
    #
    # une colonne 0, 1, .. lines-1
    column = np.arange(lines)[:, np.newaxis]
    # une ligne 0, 1, ... columns-1
    line = np.arange(columns)
    # il n'y a plus qu'à broadcaster les deux
    # attention toutefois que c'est column qui contient
    # les indices en i
    return 100*column + 10*line + offset
```

hundreds_ter - Semaine 7 Séquence 05

```
def hundreds_ter(lines, columns, offset):
    """
    Une approche discutable
    """
    # à la Fortran; ça n'est pas forcément
    # la bonne approche ici bien sûr
    # mais si un élève a des envies de benchmarking...
    result = np.zeros(shape=(lines, columns), dtype=np.int_)
    for i in range(lines):
        for j in range(columns):
            result[i, j] = 100 * i + 10 * j + offset
    return result
```

stairs - Semaine 7 Séquence 05

```
def stairs(taille):
    """
    la pyramide en escaliers telle que décrite dans l'énoncé
    """
    # on calcule n
    total = 2 * taille + 1
    # on calcule les deux tableaux d'indices
    # tous les deux de dimension total
    I, J = np.indices((total, total))
    # on décale et déforme avec valeur absolue, pour obtenir
    # deux formes déjà plus propices
    I2, J2 = np.abs(I-taille), np.abs(J-taille)
    # si ajoute on obtient un négatif,
    # avec 0 au centre et taille aux 4 coins
    negatif = I2 + J2
    # ne retse plus qu'à renverser
    return 2 * taille - negatif
```

stairs_2 - Semaine 7 Séquence 05

```
def stairs_2(taille):
    """
    même idée, modalités légèrement différentes
    Aussi on peut inverser plus tôt
    """
    total = 2 * taille + 1
    # on peut préciser le type, mais ce n'est pas
    # réellement nécessaire ici
    I, J = np.indices((total, total), dtype=np.int8)
    # on peut inverser avant d'ajouter si c'est plus naturel
    return  (taille - np.abs(I-taille)) + (taille - np.abs(J-taille))
```

stairs_3 - Semaine 7 Séquence 05

```
def stairs_3(taille):
    """
    en fait on n'a pas vraiment besoin d'indices
    """
    # la première ligne
    line = taille - np.abs(np.arange(-taille, taille+1))
    # la première colonne est la transposée
    # comme je n'aime pas utiliser .T
    # je préfère un reshape
    # et il n'y a qu'à ajouter
    return line + line.reshape((2*taille+1, 1))
```

stairs_4 - Semaine 7 Séquence 05

```
def stairs_4(taille):
    """
    une approche par mosaique
    on construit un quart, et on le duplique avec
    * np.hstack (une fonction d'empilement)
    * np.flip (une fonction de miroir)

    credits: JeF29
    """
    a = np.arange(taille)
    b = np.hstack((a, taille, np.flip(a)))
    return b + b.reshape(-1, 1) # ou b + b[:, np.newaxis]
```

stairs_ter - Semaine 7 Séquence 05

```
def stairs_ter(taille):
    """
    Version proposée par j4l4y
    Dans la rubrique 'oneliner challenge'

    credits: j4l4y
    """
    # la forme np.abs(np.range(-n, n+1)) correspond à la forme
    # en V, par exemple pour n=3 : -3, -2, -1, 0, 1, 2, 3
    # dans cette version, on l'agrandit artificiellement en 2D
    # pour pouvoir prendre sa transposée
    return (lambda x: x + x.T)(
        taille - np.abs(range(-taille, taille+1))[:, np.newaxis]
    )
```

dice - Semaine 7 Séquence 05

```
def dice(target, nb_dice=2, nb_sides=6):
    """
    Pour un jeu où on lance `nb_dice` dés qui ont chacun `sides` faces,
    quel est le nombre de tirages dont la somme des dés fasse `target`

    Version force brute, il y a bien sûr des outils mathématiques
    pour obtenir une réponse beaucoup plus rapidement

    Toutes les solutions procèdent en deux étapes

    * calcul de l'hypercube qui énumère les tirages,
      et calcule la somme des dés pour chacun de ces tirages
    * trouver le nombre de points dans le cube où la somme des dés
      correspond à ce qu'on cherche

    les deux étapes sont indépendantes, et peuvent donc être mélangées
    entre les solutions
    """

    # pour élaborer le cube, on procède par broadcating
    # on commence avec un simple vecteur de shape (nb_sides,) - e.g. de 1 à 6
    # on lui ajoute lui-même mais avec une forme (nb_sides, 1) - en colonne donc
    # et ainsi de suite avec
    # shape=(nb_sides, 1, 1) pour la dimension 3,
    # shape=(nb_sides, 1, 1, 1) pour la dimension 4
    sides = np.arange(1, nb_sides+1)
    cube = sides
    # une liste plutôt qu'un tuple pour décrire la shape,
    # car on va y ajouter '1' à chaque tour
    shape = [nb_sides]
    # on a déjà un dé
    for _dimension in range(nb_dice - 1):
        shape.append(1)
        cube = cube + sides.reshape(shape)

    # le cube est prêt,
    # pour chercher combien de cases ont la valeur target,
    # on peut faire par exemple
    return np.sum(cube == target)

```

dice_2 - Semaine 7 Séquence 05

```
def dice_2(target, nb_dice=2, nb_sides=6):
    """
    une variante de la première forme, qui utilise
    astucieusement une matrice diagonale pour énumérer
    les 'shapes' qui entrent en jeu

    credits: aurelien
    """
    sides = np.arange(1, nb_sides+1)
    shapes = np.diag([nb_sides-1]*nb_dice) + 1
    # attention ici c'est le sum Python
    # et non pas np.sum qui ferait complètement autre chose
    cube = sum(sides.reshape(s) for s in shapes)

    # une autre façon de faire le décompte
    return np.count_nonzero(cube == target)
```

dice_3 - Semaine 7 Séquence 05

```
def dice_3(target, nb_dice=2, nb_sides=6):
    """
    même logique globalement, mais en utilisant
    np.newaxis pour changer de dimension
    """
    sides = np.arange(1, nb_sides+1)
    cube = sides
    # on a déjà un dé
    for _dimension in range(nb_dice - 1):
        sides = sides[:, np.newaxis]
        cube = cube + sides

    # une autre façon de faire le décompte
    return np.count_nonzero(cube == target)
```

dice_4 - Semaine 7 Séquence 05

```
def dice_4(target, nb_dice=2, nb_sides=6):
    """
    on peut aussi tirer profit de indices()
    qui fait déjà presque le travail
    puisqu'il construit plusieurs cubes de la bonne dimension
    qu'il ne reste plus qu'à additionner
    """
    # il faut quand même faire attention
    # car indices() commence à 0
    all_indices = np.indices(nb_dice * (nb_sides,)) + 1
    cube = sum(all_indices)

    return np.count_nonzero(cube == target)
```

dice_5 - Semaine 7 Séquence 05

```
def dice_5(target, nb_dice=2, nb_sides=6):
    """
    une très légère variante
    """
    all_indices = np.indices(nb_dice * (nb_sides,))
    # une façon plus pédante mais plus propre de faire la somme
    # si on n'a pas rectifié avant, il faut maintenant ajouter nb_dice
    cube = np.add.reduce(all_indices) + nb_dice

    return np.count_nonzero(cube == target) # ou return len(res[res == target])
```

dice_6 - Semaine 7 Séquence 05

```
# on peut aussi utiliser itertools.product qui permet
# d'itérer sans aucune mémoire sur le même hypercube
#
# de manière un peu paradoxale, cette version en Python pur,
# bien que nécessitant en théorie beaucoup moins de mémoire,
# est beaucoup moins efficace que la version numpy
# je vous renvoie à la discussion sur le forum intitulée
# "Exercice dice"
from itertools import product

def dice_6(target, nb_dice=2, nb_sides=6):
    """
    Une autre méthode complètement, qui n'alloue aucun tableau
    du coup on n'a pas besoin de numpy
    """
    # en version facile, on peut utiliser le paramètre `repeat`
    # de product qui fait exactement ce qu'on veut, puisque
    # tous les dés ont le même nombre de faces
    #
    # par exemple le cas standard (2 dés, 6 faces) se ferait avec
    # quelque chose comme
    # (for (i, j) in itertools.product(range(1, 7), repeat=2))
    #
    # le premier sum compte les occurences de True dans l'itération
    return sum(
        # ici sum(x) fait la somme des tirages des dés
        sum(x) == target
        for x in product(range(1, sides+1), repeat=nb_dice))
```

matdiag - Semaine 7 Séquence 05

```
import numpy as np

def matdiag(liste):
    """
    si les arguments sont x1, x2, .. xn
    retourne une matrice carrée n x n
    dont les éléments valent
    m[i, j] = xi si i == j
    m[i, j] = 0 sinon

    credit: JeF29
    """
    # on crée une matrice diagonale unité avec np.eye
    # (car I se prononce comme eye en anglais)
    # et on la multiplie par broadcasting avec un vecteur
    # composé de nos arguments
    # on la crée de type `int64` de façon à obtenir
    # pour le résultat final un type entier, flottant
    # ou complexe, selon les valeurs dans liste
    return np.eye(len(liste), dtype=np.int64) * liste
```

matdiag_2 - Semaine 7 Séquence 05

```
def matdiag_2(liste):
    """
    même propos mais cette fois avec du slicing
    """
    #
    # on initialise un tableau de la bonne taille n x n
    # mais tout à plat, avec des zéros
    # ici si on veut que ça marche avec des complexes,
    # il faut alors créer tout de suite le tableau de type
    # complexe, sinon on n'a pas la place
    n = len(liste)
    plat = np.zeros((n * n,), dtype=np.complex)
    #
    # dans cette représentation là, la diagonale correspond
    # à un slice qui commence à 1 avec un pas de n+1
    plat[0 : : n+1] = liste
    #
    # maintenant on peut remettre
    # dans une forme n x n avec reshape
    #
    return plat.reshape((n, n))
```

matdiag_3 - Semaine 7 Séquence 05

```
def matdiag_3(liste):
    """
    bon maintenant qu'on s'est bien creusé les méninges
    pour le faire à la main, il se trouve qu'il y a
    - bien sûr - une fonction pour ça dans numpy
    """
    return np.diag(liste)
```

xixj - Semaine 7 Séquence 05

```
import numpy as np

def xixj(*args):
    """
    si les arguments sont x1, x2, .. xn
    retourne une matrice carrée n x n
    dont les éléments valent
    m[i, j] = xi * xj

    première solution à base de produit usuel
    entre un vecteur et une colonne, en utilisant
    le broadcasting

    credits: JeF29
    """

    # une ligne qui contient x1, .. xn
    line = np.array(args)
    # habile façon de reshaper automatiquement
    column = line.reshape(-1, 1)
    # on aurait pu faire aussi
    #column = line[:, np.newaxis]
    return line * column
```

xixj_2 - Semaine 7 Séquence 05

```
def xixj_2(*args):
    """
    pareil mais on construit la colonne avec .T
    qui est la transposée - méfiance quand même
    """
    # sauf que pour pouvoir utiliser .T il faut
    # une shape qui est explicitement [1, n]
    #
    # c'est pourquoi moi j'ai tendance à éviter .T
    # voyez plutôt np.transpose() si vous avez besoin
    # de transposer une matrice
    line = np.array(args).reshape((1, -1))
    return line * line.T
```

xixj_3 - Semaine 7 Séquence 05

```
def xixj_3(*args):
    """
    on peut aussi penser à faire un produit matriciel
    """
    # on doit lui donner une dimension 2 même si c'est une ligne
    line = np.array(args).reshape((1, -1))
    column = line.reshape((-1, 1))
    return column @ line
```

xixj_4 - Semaine 7 Séquence 05

```
def xixj_4(*args):
    """
    pareil mais en utilisant .dot()
    """
    column = np.array(args).reshape((-1, 1))
    # dans cette version on fait le produit de matrice
    # en utilisant la méthode dot sur les tableaux
    return column.dot(column.T)
    # remarquez qu'on aurait pu faire aussi bien
    # return np.dot(column, column.T)
```

npsearch - Semaine 7 Séquence 05

```
import numpy as np

def npsearch(world, needle):
    """
    world est la "grande" matrice dans laquelle
    on cherche les occurrences de needle
    qui peut être une matrice 2d ou une simple ligne

    npsearch est une fonction génératrice qui énumère
    les tuples (i, j) correspondant à une occurrence de
    needle dans world
    """
    if len(needle.shape) == 1:
        needle = needle[np.newaxis, :]
    n, m = needle.shape
    # pas la peine de faire une grande boucle sur tout le tableau
    # s'il y a égalité c'est nécessairement que
    # le world[i, j] == needle[0, 0]
    for i, j in np.argwhere(world == needle[0][0]):
        # c'est ici le point délicat
        # si vous comparez les deux tableaux à base de ==
        # (même en utiisant np.all)
        # vous allez potentiellement mettre en oeuvre
        # un broadcasting non souhaitable
        if np.array_equal(world[i:i+n, j:j+m], needle):
            yield i, j
```

taylor - Semaine 7 Séquence 10

```
class Taylor:
    """
    provides an animated view of Taylor approximation
    where one can change the degree interactively

    Taylor is applied on X=0, translate as needed
    """

    def __init__(self, function, domain):
        self.function = function
        self.domain = domain

    def display(self, y_range):
        """
        create initial drawing with degree=0

        Parameters:
          y_range: a (ymin, ymax) tuple
            for the animation to run smoothly, we need to display
            all Taylor degrees with a fixed y-axis range
        """
        # create figure
        x_range = (self.domain[0], self.domain[-1])
        self.figure = figure(title=self.function.__name__,
                             x_range=x_range, y_range=y_range)

        # each of the 2 curves is a bokeh line object
        self.figure.line(self.domain, self.function(self.domain), color='green')
        # store this in an attribute so _update can do its job
        self.line_approx = self.figure.line(
            self.domain, self._approximated(0), color='red', line_width=2)

        # needed so that push_notebook can do its job down the road
        self.handle = show(self.figure, notebook_handle=True)
```

taylor (continued) - Semaine 7 Séquence 10

```
    def _approximated(self, degree):
        """
        Computes and returns the Y array, the images of the domain
        through Taylor approximation

        Parameters:
          degree: the degree for Taylor approximation
        """
        # initialize with a constant f(0)
        # 0 * self.domain allows to create an array
        # with the right length
        result = 0 * self.domain + self.function(0.)
        # f'
        derivative = autograd.grad(self.function)
        for n in range(1, degree+1):
            # the term in f(n)(x)/n!
            result += derivative(0.)/factorial(n) * self.domain**n
            # next-order derivative
            derivative = autograd.grad(derivative)
        return result

    def _update(self, degree):
        # update the second curve only, of course
        # the 2 magic lines for bokeh updates
        self.line_approx.data_source.data['y'] = self._approximated(degree)
        push_notebook(handle=self.handle)

    def interact(self, degree_widget):
        """
        Parameters:
          degree_widget: a ipywidget, typically an IntSlider
            styled at your convenience
        """
        interact(lambda degree: self._update(degree), degree=degree_widget)
```