MOOC Python 3

Session 2018

Corrigés de la semaine 6

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
```