Exercise 1: Expressions Solved Step by Step.

One of the exercises you had to learn in your childhood was to solve arithmetic expressions and the easiest way to learn that was to solve them step by step starting from the innermost sub-expressions and proceeding by reducing the complexity of the expression. E.g., ((4+5)*2) → (9*2) → 18.

Python, as well as many other programming languages, can solve any kind of arithmetic expression even the most complicated but it produces immediately the result without showing the intermediate steps.

You have to define i) a class `calculator` whose instances represent an expression parsed out from a string and permits all the basic operation on such an expression and ii) a function `print_reduction` that given an instance of calculator can evaluate the expression step by step (printing all the intermediate results) as in:

```python
from reduction import *

if __name__ == '__main__':
  expressions = ["+34", "+3-15", "*+34-23", "+7++34+23",
      "*++34-34/6-35", "/+-81*45*/93/52", "*+/12/14-2/32", "+2*-53/63"]
  [print_reduction(calculator(expr)) for expr in expressions]
```

```
[19:21]cazzola@surtur:~/pa/es1>python3 main-reduction.py
(3+4)
7
(3+(1-5))
(3+-4)
-1
((3+4)*(2-3))
(7*-1)
-7
(7+((3+4)+(2+3)))
(7+(7+5))
(7+12)
19
(((3*4)+(3-4))*(6/(3-5)))
((12+-1)*(6/-2))
(11*-3)
-33
(((8-1)+(4*5))/((9/3)*(5/2)))
((7+20)/(3*2))
(27/6)
4
(((1/2)+(1/4))*(2-(3/2)))
((0+0)*(2-1))
(0*1)
0
(2+((5-3)*(6/3)))
(2+(2*2))
(2+4)
6
```

The only admitted operators are `+`, `-`, `*` and `/` with the traditional meaning. The only available type is int (the '/ ' is a division among intergers). To simplify the parsing, the expressions are written in polish notation and numbers in the initial expression are only 1-figure positive integers.

Note Polish notation is a form of notation for arithmetic that places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity.

Soluzione:

```python
ops = { '+': lambda x,y: x+y,
        '-': lambda x,y: x-y,
        '*': lambda x,y: x*y,
        '/': lambda x,y: x//y }

def make_node(op, name):
  class node:
    def __init__(self, op1, op2):
      self.__operand1 = op1
      self.__operand2 = op2
    def eval(self):
      return op(self.__operand1.eval(), self.__operand2.eval())
    def combine(self):
      if isinstance(self.__operand1, leaf) and
         isinstance(self.__operand2, leaf):
        return leaf(str(self.eval()))
      else:
        self.__operand1 = self.__operand1.combine()
        self.__operand2 = self.__operand2.combine()
        return self
    def __str__(self): return "("+self.__operand1.__str__() +
                              name + self.__operand2.__str__()+")"
  return node

class leaf:
  def __init__(self, value):
    self.__value = value
  def eval(self):
    return int(self.__value)
  def combine(self): return self
  def __str__(self): return self.__value

translator = {op:make_node(fop, op) for op, fop in ops.items()}
translator.update({str(x):leaf(str(x)) for x in range(10)})

class calculator:
  def __init__(self, expr):
    self.__root, dropped = self.__convert(expr, 0)
  def is_value(self): return isinstance(self.__root,leaf)
  def __convert(self, expr, n):
    if n < len(expr):
       if expr[n] in {'+', '*', '-', '/'}:
         op1,n1 = self.__convert(expr,n+1)
         op2,n2 = self.__convert(expr,n1+1)
         return translator[expr[n]](op1,op2),n2
       else: return translator[expr[n]],n
  def combine(self):
    self.__root = self.__root.combine()
    return self
  def eval(self): return self.__root.eval()
  def __str__(self): return self.__root.__str__()

def print_reduction(e):
  print(e)
  if not e.is_value(): print_reduction(e.combine())
```

Exercise 2: Gray Code Calculator.

The reflected binary code, also known as **Gray code** after Frank Gray, is a binary numeral system where two successive values differ in only one bit. The Gray code was originally designed to prevent spurious output from electromechanical switches.

Many devices indicate position by closing and opening switches. If that device uses natural binary codes, these two positions would be right next to each other. The problem with natural binary codes is that, with real (mechanical) switches, it is very unlikely that switches will change states exactly in synchrony. In the brief period while all are changing, the switches will read some spurious position. Even without keybounce, the transition might look like 011 — 001 — 101 — 100. When the switches appear to be in position 001, the observer cannot tell if that is the "real" position 001, or a transitional state between two other positions. If the output feeds into a sequential system (possibly via combinational logic) then the sequential system may store a false value.

The reflected binary code solves this problem by changing only one switch at a time, so there is never any ambiguity of position,

Dec	Gray	Binary
0	000	000
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101
6	101	110
7	100	111

Notice that state 7 can roll over to state 0 with only one switch change. This is called the "cyclic" property of a Gray code. In the standard Gray coding the least significant bit follows a repetitive pattern of 2 on, 2 off (... 11001100 ...); the next digit a pattern of 4 on, 4 off; and so forth.

More formally, a Gray code is a code assigning to each of a contiguous set of integers, or to each member of a circular list, a word of symbols such that each two adjacent code words differ by one symbol. These codes are also known as single-distance codes, reflecting the Hamming distance of 1 between adjacent codes. There can be more than one Gray code for a given word length, but the term was first applied to a particular binary code for the non-negative integers, the binary-reflected Gray code, or BRGC, the three-bit version of which is shown above.

The scope of the exercise is to implement two functions: `gray` and `mgray` that took an integer `n` return a generator with the complete Gray code of lenght `n`. The difference between the two versions is that the second uses memoization so it stores and reuses previously calculated sequences. Both implementation should be **recursive** as well as memoization should be introduced via a decorator or a meta-class. **Note** that any solution realized without recursion, withouth decorators/meta-classes and that doesn't return a generator is considered as wrong.

The following is an example of the expected computation.

```
from gray import *

if __name__ == "__main__":
  print( "GC(1) :-", end=' ')
  for gc in gray(1): print(gc, end=' ')
  print( "\nGC(2) :-", end=' ')
  for gc in gray(2): print(gc, end=' ')
  print( "\nGC(3) :-", end=' ')
  for gc in gray(3): print(gc, end=' ')
  print( "\nGC(4) :-", end=' ')
  for gc in gray(4): print(gc, end=' ')
  print()
  print( "GC_(1) :-", end=' ')
  for gc in mgray(1): print(gc, end=' ')
  print( "\nGC_(2) :-", end=' ')
  for gc in mgray(2): print(gc, end=' ')
  print( "\nGC_(3) :-", end=' ')
  for gc in mgray(3): print(gc, end=' ')
  print( "\nGC_(4) :-", end=' ')
  for gc in mgray(4): print(gc, end=' ')
  print()
```

```
[19:20]cazzola@surtur:~/pa/es2>python3 main-gray.py
GC(1) :- 0 1
GC(2) :- 00 01 11 10
GC(3) :- 000 001 011 010 110 111 101 100
GC(4) :- 0000 0001 0011 0010 0110 0111 0101 0100
         1100 1101 1111 1110 1010 1011 1001 1000
GC_(1) :- 0 1
GC_(2) :-
### cached value for (1,) --> ['0', '1']
00 01 11 10
GC_(3) :-
### cached value for (2,) --> ['00', '01', '11', '10']
000 001 011 010 110 111 101 100
GC_(4) :-
### cached value for (3,) --> ['000', '001', '011', '010',
                               '110', '111', '101', '100']
0000 0001 0011 0010 0110 0111 0101 0100
1100 1101 1111 1110 1010 1011 1001 1000
```

Soluzione:

```python
def gray_(n):
  if n == 0: return [""]
  else:
    lower = gray_(n-1)
    return ["0"+bits for bits in lower]+["1"+bits for bits in lower[::-1]]

def gray(n):
  for g in gray_(n): yield g

def memoization(f):
  def wrapper(*args):
    if not args in wrapper.cache:
      wrapper.cache[args] = f(*args)
    else:
      print("\n### cached value for {0} --> {1}".
              format(args, wrapper.cache[args]), end='\n')
    return wrapper.cache[args]
  wrapper.cache = dict()
  return wrapper

def mgray(n):
  global gray_
  gray_ = memoization(gray_)
  return gray(n)
```