```python
"""NUI Galway CT5132/CT5148 Programming and Tools for AI (James McDermott)

Skeleton/solution for Assignment 1: Numerical Integration

By writing my name below and submitting this file, I/we declare that
all additions to the provided skeleton file are my/our own work, and that
I/we have not seen any work on this assignment by another student/group.

Student name(s): Marcel Aguilar Garcia
Student ID(s): 20235620

"""

import numpy as np
import sympy
import itertools
import math

def numint_py(f, a, b, n):
      """Numerical integration. For a function f, calculate the definite
      integral of f from a to b by approximating with n "slices" and the
      "lb" scheme. This function must use pure Python, no Numpy.

      >>> round(numint_py(math.sin, 0, 1, 100), 5)
      0.45549
      >>> round(numint_py(lambda x: 1, 0, 1, 100), 5)
      1.0
      >>> round(numint_py(math.exp, 1, 2, 100), 5)
      4.64746

      """
      A = 0
      w = (b - a) / n # width of one slice
      # STUDENTS ADD CODE FROM HERE TO END OF FUNCTION
      for d in range(0,n): # iterating over each of the slices
            A+= w*f(a+d*w) # adding rectangle area of each slice to the total area
      return A

def numint(f, a, b, n, scheme='mp'):
      """Numerical integration. For a function f, calculate the definite
      integral of f from a to b by approximating with n "slices" and the
      given scheme. This function should use Numpy, and eg np.linspace()
      will be useful.

      >>> round(numint(np.sin, 0, 1, 100, 'lb'), 5)
      0.45549
      >>> round(numint(lambda x: np.ones_like(x), 0, 1, 100), 5)
      1.0
      >>> round(numint(np.exp, 1, 2, 100, 'lb'), 5)
      4.64746
      >>> round(numint(np.exp, 1, 2, 100, 'mp'), 5)
      4.67075
      >>> round(numint(np.exp, 1, 2, 100, 'ub'), 5)
      4.69417

      """
      # STUDENTS ADD CODE FROM HERE TO END OF FUNCTION
      w = (b - a) / n # width of one slice

      slices = list(np.linspace(a,b,n+1)) #List of n+1 points evenly spaced between a and b

      if scheme == 'lb': # if scheme is lower bound
            #returning area by always selecting xi on the left of the rectangle
            #therefore the last slice is not used
            return sum([f(x)*w for x in slices[:-1]])

      elif scheme == 'ub': # if scheme is upper bound
            #returning area by always selecting xi on the right of the rectangle
```

```python
            #therefore the first slice is not used
            return sum([f(x)*w for x in slices[1:]])

        elif scheme == 'mp': # if scheme is mid-point
            #defining a new list with midpoints
            slices = [(slices[n]+slices[n+1])/2 for n in range(0,len(slices)-1)]
            #returning area by using midpoints
            return sum([f(x)*w for x in slices])

        else:
            #in case that user enters a non valid strategy
            #this will be raised as an error
            raise ValueError("Please, introduce a valid scheme")

def true_integral(fstr, a, b):
    """Using Sympy, calculate the definite integral of f from a to b and
    return as a float. Here fstr is an expression in x, as a str. It
    should use eg "np.sin" for the sin function.

    This function is quite tricky, so you are not expected to
    understand it or change it! However, you should understand how to
    use it. See the doctest examples.

    >>> true_integral("np.sin(x)", 0, 2 * np.pi)
    0.0
    >>> true_integral("x**2", 0, 1)
    0.3333333333333333

    STUDENTS SHOULD NOT ALTER THIS FUNCTION.

    """
    x = sympy.symbols("x")
    # make fsym, a Sympy expression in x, now using eg "sympy.sin"
    fsym = eval(fstr.replace("np", "sympy"))
    A = sympy.integrate(fsym, (x, a, b)) # definite integral
    A = float(A.evalf()) # convert to float
    return A

def numint_err(fstr, a, b, n, scheme):
    """For a given function fstr and bounds a, b, evaluate the error
    achieved by numerical integration on n points with the given
    scheme. Return the true value, absolute error, and relative error
    as a tuple.

    Notice that the relative error will be infinity when the true
    value is zero. None of the examples in our assignment will have a
    true value of zero.

    >>> print("%.4f %.4f %.4f" % numint_err("x**2", 0, 1, 10, 'lb'))
    0.3333 0.0483 0.1450

    """
    f = eval("lambda x: " + fstr) # f is a Python function
    A = true_integral(fstr, a, b)
    # STUDENTS ADD CODE FROM HERE TO END OF FUNCTION
    #calculating numerical integration by using numint
    numerical_approx = numint(f, a, b, n, scheme)
    #Absolute error between the true integral an numerical integration
    absolute_error = abs(A-numerical_approx)
    #Relative error between the true integral an numerical integration
    relative_error = absolute_error / A
    return (A,absolute_error,relative_error)

def make_table(f_ab_s, ns, schemes):
    """For each function f with associated bounds (a, b), and each value
```

```python
    of n and each scheme, calculate the absolute and relative error of
    numerical integration and print out one line of a table. This
    function doesn't need to return anything, just print. Each
    function and bounds will be a tuple (f, a, b), so the argument
    f_ab_s is a list of tuples.

    Hint 1: use print() with the format string
    "%s,%.2f,%.2f,%d,%s,%.4g,%.4g,%.4g", or an equivalent f-string approach.

    Hint 2: consider itertools.

    >>> make_table([("x**2", 0, 1), ("np.sin(x)", 0, 1)], [10, 100], ['lb', 'mp'])
    x**2,0.00,1.00,10,lb,0.3333,0.04833,0.145
    x**2,0.00,1.00,10,mp,0.3333,0.0008333,0.0025
    x**2,0.00,1.00,100,lb,0.3333,0.004983,0.01495
    x**2,0.00,1.00,100,mp,0.3333,8.333e-06,2.5e-05
    np.sin(x),0.00,1.00,10,lb,0.4597,0.04246,0.09236
    np.sin(x),0.00,1.00,10,mp,0.4597,0.0001916,0.0004168
    np.sin(x),0.00,1.00,100,lb,0.4597,0.004211,0.009161
    np.sin(x),0.00,1.00,100,mp,0.4597,1.915e-06,4.167e-06

    """

    # STUDENTS ADD CODE FROM HERE TO END OF FUNCTION
    #By using itertools, this for will go through all possible combinations of
    #elements between these lists
    for p in itertools.product(f_ab_s, ns, schemes):
        #calculating the errors by using numint_err
        A,absolute_error,relative_error = numint_err(p[0][0],p[0][1],p[0][2],p[1],p[2])
        #printing results in the correct format
        print(f"{p[0][0]},{p[0][1]:.2f},{p[0][2]:.2f},{p[1]},{p[2]},{A:.4g},{absolute_error:.4g},{relative_error:.4g}")

def main():
    """Call make_table() as specified in the pdf."""
    # STUDENTS ADD CODE FROM HERE TO END OF FUNCTION
    #Defining function, values and strategy from assignment
    f_ab_s = [("np.cos(x)",0,math.pi/2),("np.sin(2*x)",0,1),("np.exp(x)",0,1)]
    ns = [10,100,1000]
    schemes = ['lb','mp']
    #Calling make_table with these variables
    make_table(f_ab_s, ns, schemes)
"""
Which scheme is better?

In all cases, "mid-point" strategy has better results than "lower bound"
strategy as the numerical integration has a lower absolute and relative error.

By how much?

In order to answer this question, I have based my answer on the relative error.
I have calculated the improvement of the results by dividing
"Relative Error lb"/"Relative Error mp", the results can be seen below:

+----------------------------------+------+---------------------------+
|            function              |  n   | interval | How much better |
+----------------------------------+------+---------------------------+
| np.cos(x)                        | 10   | [0,PI/2] |      74.324     |
| np.cos(x)                        | 100  | [0,PI/2] |      7.62E2     |
| np.cos(x)                        | 1000 | [0,PI/2] |      7.638E3    |
-----------------------------------------------------------------------
| np.sin(2*x)                      | 10   | [0,1]    |      40.46      |
| np.sin(2*x)                      | 100  | [0,1]    |      3.87E2     |
| np.sin(2*x)                      | 1000 | [0,1]    |      3.85E3     |
-----------------------------------------------------------------------
| np.exp(x)                        | 10   | [0,1]    |      118.05     |
| np.exp(x)                        | 100  | [0,1]    |      1.20E3     |
| np.exp(x)                        | 1000 | [0,1]    |      1.20E4     |
+----------------------------------+------+---------------------------+

It can be seen that as n grows, the improvement of 'mp' over 'lb' gets better
by a factor of approximately n.
```

```
Why?

In general, mp gives better results as the left side and right side
error of the rectangle cancels out each other.

"""

def numint_nd(f, a, b, n):
      """numint in any number of dimensions.

      f: a function of m arguments
      a: a tuple of m values indicating the lower bound per dimension
      b: a tuple of m values indicating the upper bound per dimension
      n: a tuple of m values indicating the number of steps per dimension

      STUDENTS ADD DOCTESTS
      >>> round(numint_nd(lambda x, y: x*np.cos(x*y), [0,0], [math.pi/2,math.pi/2],[100,100]),5)
      1.13399

      >>> round(numint_nd(lambda x, y: x**2+4*y, [11,7], [14,10],[10,10]),5)
      1718.9325

      >>> round(numint_nd(np.exp, [1], [2], [100]), 5)
      4.67075
      """

      # My implementation uses Numpy and the mid-point scheme, but you
      # are free to use pure Python and/or any other scheme if you prefer.

      # Hint: calculate w, the step-size, per dimension
      w = [(bi - ai) / ni for (ai, bi, ni) in zip(a, b, n)]

      # STUDENTS ADD CODE FROM HERE TO END OF FUNCTION
      #List containing lists of n+1 points  evenly spaced between ai and bi
      #for i = 1,...,m
      slices = [list(np.linspace(ai,bi,ni+1)) for (ai,bi,ni) in zip(a,b,n)]
      #calculating the mid-points in order to apply numerical integration
      #with mp scheme
      slices = [[(dim[n]+dim[n+1])/2 for n in range(0,len(dim)-1)] for dim in slices]
      #calculating area of the base of the hyperrectangle
      base = np.prod(w)
      #itertools.product has been used in order to create a grid between the
      #values all values from ai,bi for i=1,...,m
      return sum([f(*t)*base for t in itertools.product(*slices)])



if __name__ == "__main__":
      import doctest
      doctest.testmod()
      main()
```