Scientific Computing with Python Lab

13th Session(April 23rd)

Kaheon Kim : https://github.com/kaheonkim/Scientific-computing-with-python-lab-material

Things to do today

HW7 : Due April 26(Fri)

- Going through the homework

- Extra functions : isinstance, enumerate

Plans for the rest of the semester

Next week

- Deal with the numpy shortly
- Review things we have done
- Talking briefly about the final exam

Problem 1

1. In this problem, we will implement a function to carry out Euler's method to solve an ordinary differential equation (ODE) for an arbitrary real-valued function $f(x, y)$ as the right hand side over the interval $[0, b]$. We wish to approximate the solution to the ordinary differential equation:

$$\begin{aligned} y' &= f(x, y) \\ y(0) &= y_0 \end{aligned}$$

using Euler's method and N subintervals.

In a file named ode.py, write the definition of a function named eulers_method. Your eulers_method function should take four parameters: a function f (assumed to take two parameters, x and y), a float representing y_0, a float representing the endpoint b, and an integer N for the number of subintervals. The function should return a list of tuples of float values (x_i, y_i) for $i = 0, \ldots, N$.

I have provided a script that imports the euler.py module and calls the eulers_method function and then writes output to a file. I have provided sample output for the script on Canvas.

See a later page of the assignment for an explanation of Euler's method.

Problem 1

1 Euler's Method

To approximate the solution of the differential equation

$$y' = f(x, y)$$
$$y(0) = y_0$$

over the interval $[0, b]$, divide the interval $[0, b]$ into N distinct subintervals, each of equal length $h = b/N$. The subintervals will have endpoints $[x_{i-1}, x_i]$, where $x_0 = 0$, $x_N = b$ and $x_i = b/N$. Let y_i denote the approximate solution at the point x_i, i.e. $y_i \approx y(x_i)$. To compute the values y_i, we use an iterative procedure.
Take y_0 to be the initial value $y(0)$. For $i = 0, \ldots, N-1$, define iteratively

$$y_{i+1} = y_i + f(x_i, y_i)h$$

This procedure is just repeated use of the approximation

$$y(x+h) \approx y(x) + y'(x)h = y(x) + f(x, y(x))h$$

using values at the former iteration for x_i and y_i.

Problem 2

2. In this problem, we will implement a function to approximate a real root of a continuous real-valued function of a single variable. In a file named bisection_method.py, write a function named bisection_method. The function takes three parameters: a function f (assumed to take one float parameter), a 2-tuple of endpoints a and b, and an error tolerance.

   The error tolerance should take a default value of 10^{-10}. The function returns the location of a suspected root.

   If $f(a)$ and $f(b)$ have the same (non-zero) sign, raise a ValueError with the argument "No suspected root: f(a) and f(b) have the same sign". If a is a root, return a, and otherwise, if b is a root, return b (when both are roots, a should be returned). If neither is a root, perform the bisection procedure until the search interval has length less than the error tolerance, or until an exact root is found. We will adopt the convention that if no exact root is located, the endpoint of the final search interval with the smallest absolute value of the function is returned (in the event of a tie, return the left endpoint).

   I have provided a script that imports the bisection_method module and calls the bisection_method function on several examples, and then writes the output to a file. I have also provided sample output for the script on Canvas.

2 Bisection Method

For a continuous function $f(x)$ on the interval $[a, b]$, if $f(a)$ and $f(b)$ have opposite signs, then the intermediate value theorem guarantees that f has a root in the interval $[a, b]$. We can approximate the location of the root through an iterative procedure: first, compute the midpoint of the interval, $c = a + \frac{b-a}{2}$. Next, evaluate the $f(c)$. It may happen that c is a root, in which case we are done. Otherwise, compare the sign of $f(c)$ with $f(a)$. If the two have opposite signs, repeat this procedure, except with the new endpoints a and c. If the two have the same sign, then $f(c)$ and $f(b)$ have opposite signs, so instead repeat this procedure with the new endpoints c and b. This procedure continues until either a root is found, or the length of interval considered is (strictly) less than a specified tolerance.

Problem 3

3. In this problem, we will implement the methods for a Rectangle class. This class is intended to represent a rectangle with axis-aligned sides, having vertices $(a, c), (a, d), (b, c)$ and (b, d). Your task is to complete the method definitions (and definition of the congruent function) in the file rectangle.py so that the associated test script runs and produces the indicated output. (This problem is intended as a warm-up exercise for dealing with classes in preparation for the next problem.)

Problem 4

4. In this problem, we will implement the methods for a Polynomial class to permit us to perform some arithmetic and basic operations with polynomials (with real coefficients). A file named polynomial.py appears on Canvas, with the basic outline of some methods for a Polynomial class. Your task is to complete these methods so that the class functions appropriately.

   For the `__init__` method, the data used to initialize a polynomial may be in one of three forms: another Polynomial object, a Sequence object (such as a tuple or list), or a single int or float. (You may check if an object is a Sequence by checking if it is an instance of collections.abc.Sequence after importing the collections.abc module.) The coefficients should be stored as an instance attribute named `_coefs`. `_coefs` should be maintained as a **tuple of floats** representing the coefficients of the polynomial from least degree to greatest, with the entry in slot i corresponding to the coefficient of x^i. The zero polynomial should be represented by an empty tuple, and any non-zero polynomial should be represented with the property that the last entry of `_coefs` is a non-zero value.

   We will support three basic binary arithmetic operations: +, - and *. All three operations should allow for one of the operands to be a scalar (float or int value). (For simplicity, we will not implement division, even by a scalar, for this problem.) We will also support the pow/** operations for non-negative integer exponents. Some additional methods, such as for obtaining the degree or computing the derivative of a polynomial, will also be supported (see the polynomial.py file for a full list of methods that must be implemented).

   A test script that makes use of the Polynomial class and a file containing the expected output have been provided to help verify whether or not your implementation is working correctly.

Problem 4

- isinstance : When you have to check the data type

You can use it to check the class where the variable belongs to

- enumerate : pairs the index and value of some text type

Let’s go into the details