ASTR 119: Session 10
Numerical Integration
Ordinary Differential Equations

Outline

1) New homework due 11/9 at 8:00am
2) Visualization of the Day
3) Numerical Integration
4) Ordinary Differential Equations
5) Save your work to GitHub

Homework, due Nov 9, 8:00am

1) Write a jupyter notebook to numerically integrate the following function:

$$f(x) = e^{-2x}\cos(10x)$$

Over the range [0, pi].

2) Use the trapezoid, Simpson's method, and Romberg integration. When using Romberg integration, specify a tolerance of 1.0e-6.

3) How many iterations does Romberg integration take to reach the specified accuracy?

4) How many intervals does the trapezoid method need to reach the specified accuracy?

5) How many intervals does Simpson's method need to reach the specified accuracy?

6) Your TA will clone your code and email you commented version of the code and a grade. To get the full grade possible, all the notebooks will need to run to completion without errors and produce the requested plots.

7) Call the repository "astr-119-hw-5" and the notebook "hw-5.ipynb".

Computing/Programming Check-In

1) Create a jupyter notebook, and name it astr-119-code-check-in.ipynb.

2) Add a markdown line that describes each cell in the notebook. Each of the following instructions should be its own cell, in order.

3) Import numpy and matplotlib.pyplot as usual.

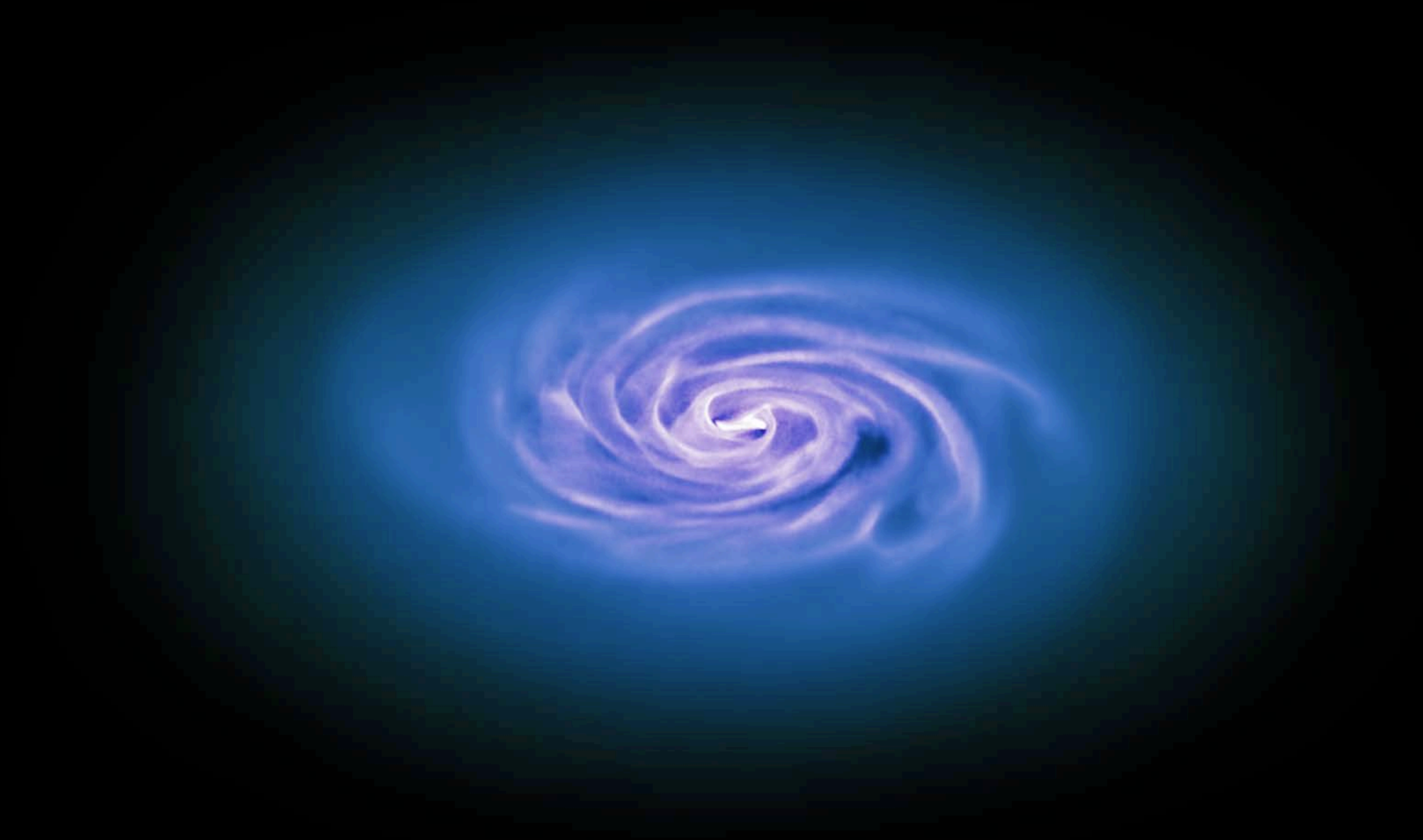
4) Declare integer i, set equal to zero. Declare a float x, set equal to 119.

5) Use a for loop, iterate i from 0 to 119 inclusive. For each even value (including 0) of i, add 3 to x. For each odd value of i, subtract 5 from x.

6) Print the final value of x in scientific notation using 2 decimal places.

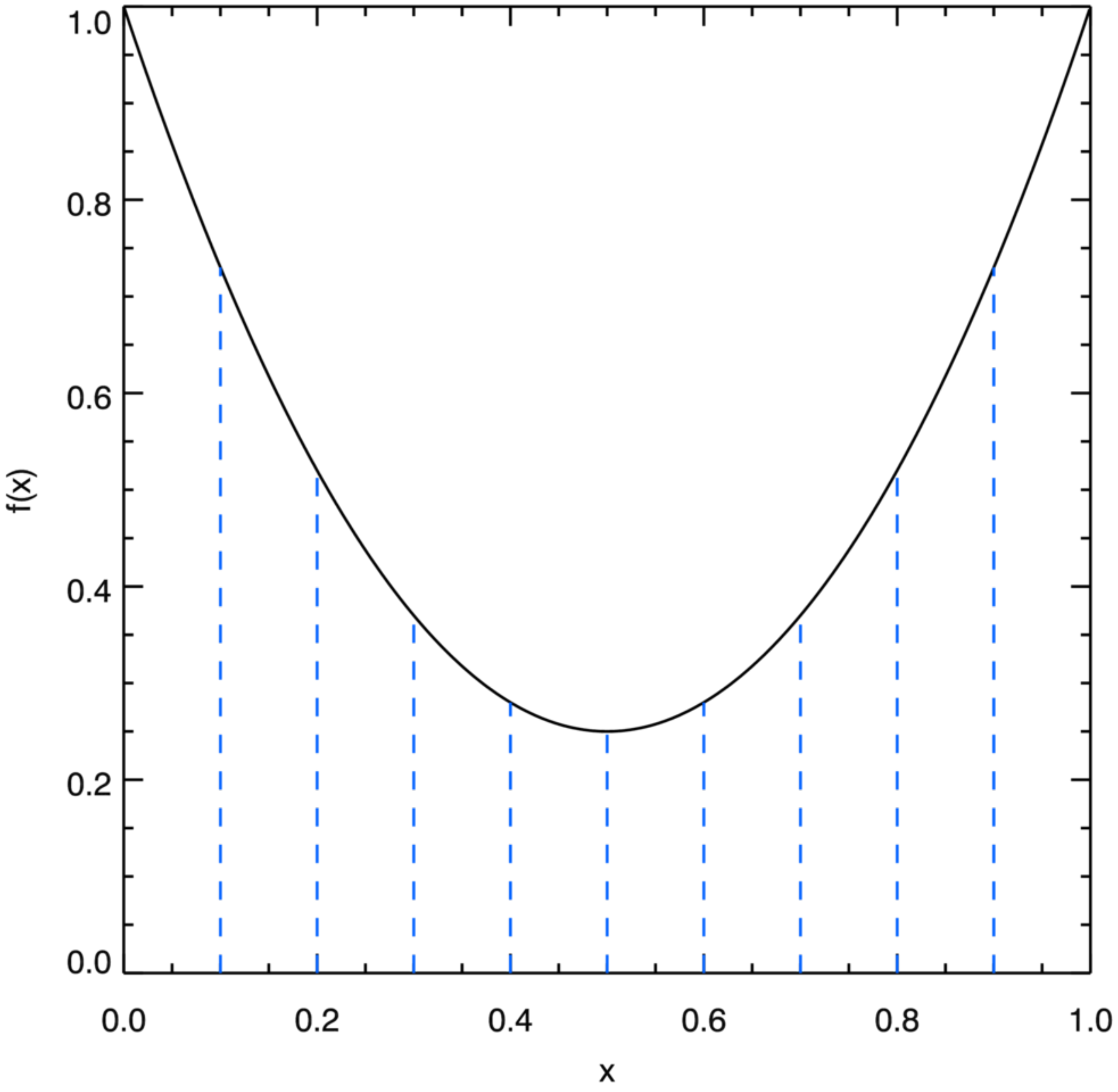
You will have ~ 5 minutes.

TAs will be sending sign-up information, starting in section next week.



Numerical Integration

As I said before, we can use even higher order approximations to the function f(x) over the interval [x_{i+1}, x_i] to obtain very accurate approximations to the integral F(a,b).



Trapezoid Method

The trapezoid method treats each subsection of the integral as a trapezoid:

$$\int_{x_i}^{x_{i+1}} f(x)dx = \frac{x_{i+1} - x_i}{2}[f(x_{i+1}) + f(x_i)]$$

Each interval has an error that is 3rd order accurate.

Simpson’s Rule

We won’t bother with the derivation, but Simpson’s rule benefits from a nice cancellation that fortuitously provides a very accurate method:

$$\int_{x_i}^{x_{i+2}} f(x)dx = h\left[\frac{1}{3}f(x_i) + \frac{4}{3}f(x_{i+1}) + \frac{1}{3}f(x_{i+2})\right] + O(h^5 f^{(4)})$$

Simpson’s rule is formally a fifth order method, with the integral converging with a factor of 32 better accuracy for every factor of two improvement in the interval *h*.

Romberg Integration

If we double the number of grid points over our interval, it's important to realize that the amount of work we perform scales linearly if we re-use the results from the lower refinement grid.

There is a method that allows us to refine our grid while reusing the previous calculations with a less refined grid. Consider our Trapezoidal rule over the interval [0,1]:

$$I_0 = \frac{1}{2}[f(1) + f(0)]$$

Now, refine by a factor of 2:

$$I_1 = \frac{1}{4}[f(0.5) + f(0)] + \frac{1}{4}[f(1) + f(0.5)]$$

Romberg Integration

Of course, this:

$$I_1 = \frac{1}{4}[f(0.5) + f(0)] + \frac{1}{4}[f(1) + f(0.5)]$$

Can be re-written as:

$$I_1 = \frac{I_0}{2} + \frac{1}{2}f(0.5)$$

We expect I_2 to be four times more accurate than I_1. What about more accurate refinements?

Romberg Integration

Another factor of two:

$$I_2 = \frac{I_1}{2} + \frac{1}{4}[f(0.25) + f(0.75)]$$

And another:

$$I_3 = \frac{I_2}{2} + \frac{1}{8}[f(0.125) + f(0.375) + f(0.625) + f(0.875)]$$

This enables us to specify a desired accuracy, as we can use this recursion relation to decide when to stop refining by checking when I_{2N} and I_N differ by some specified amount.

Romberg Integration

The Romberg Integration method just applies the trapezoidal method iteratively until a specified tolerance is reached:

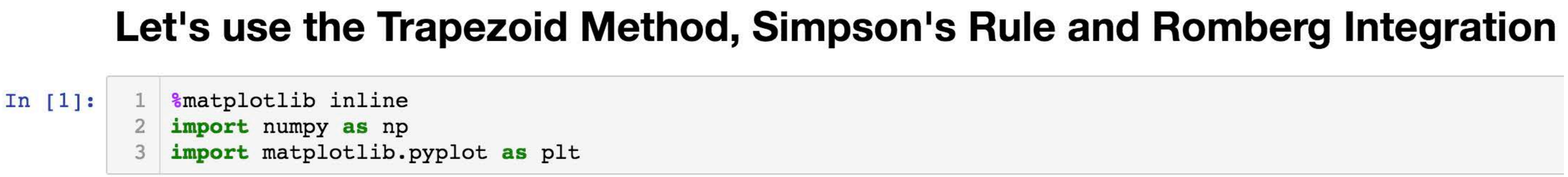
$$I_0 = \frac{h}{2}[f(x+h) + f(x)]$$

$$\Delta h = \frac{h}{2^i}$$

$$I_i(a, \Delta h) = \frac{1}{2} I_{i-1} + \frac{1}{2^{(i+1)}} \sum_{j=0}^{2^i - 1} f(a + (j + \frac{1}{2})\Delta h)$$

$$Error = |(I_i - I_{i-1})/I_{i-1}|$$

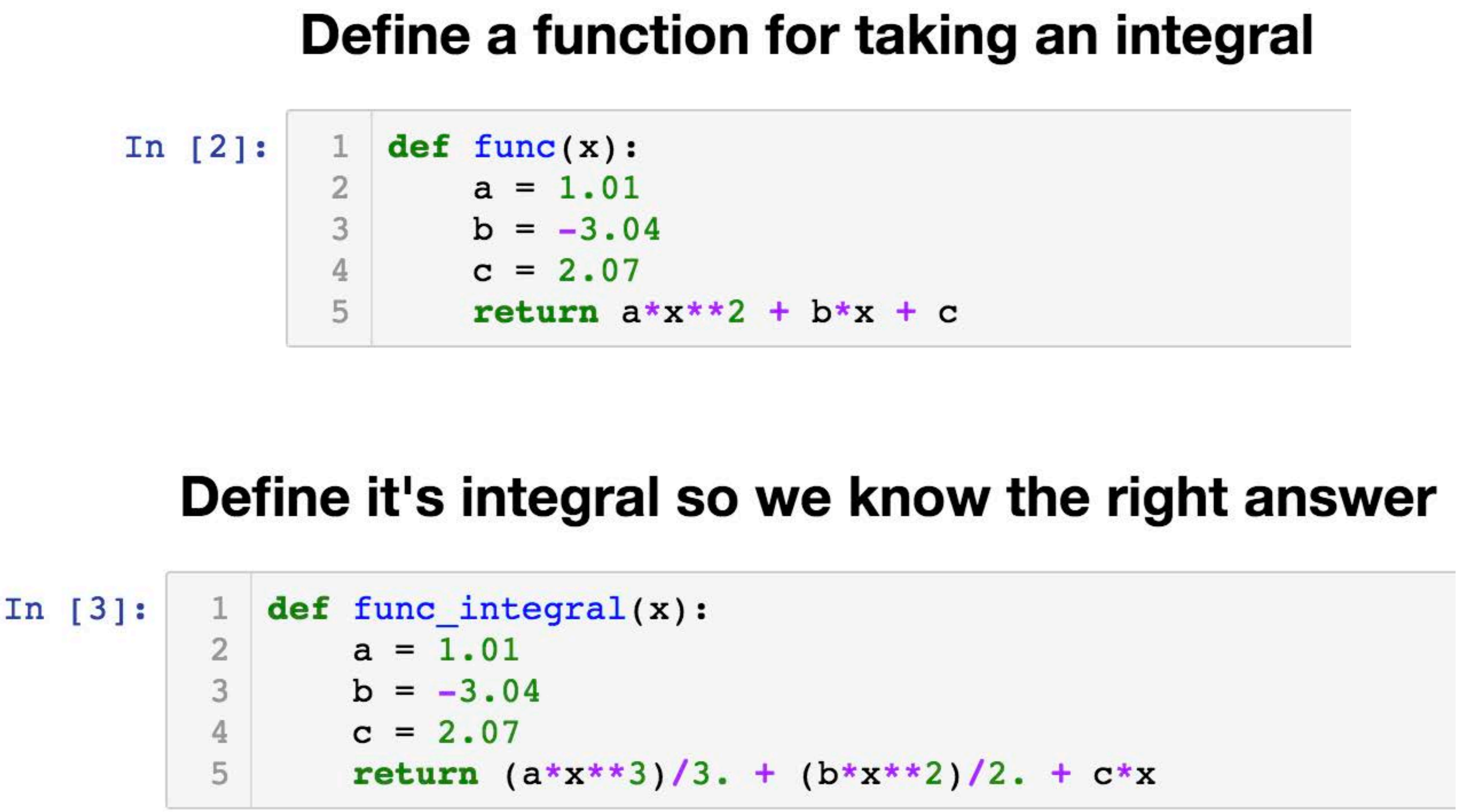
Integration Example



Let's use the Trapezoid Method, Simpson's Rule and Romberg Integration

```python
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

Integration Example



Define a function for taking an integral

In [2]:

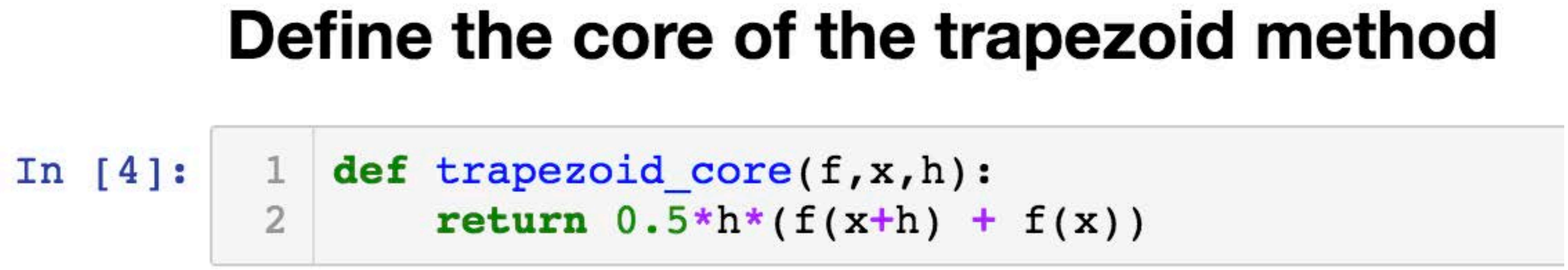
```python
def func(x):
    a = 1.01
    b = -3.04
    c = 2.07
    return a*x**2 + b*x + c
```

Define it's integral so we know the right answer

In [3]:

```python
def func_integral(x):
    a = 1.01
    b = -3.04
    c = 2.07
    return (a*x**3)/3. + (b*x**2)/2. + c*x
```

Integration Example



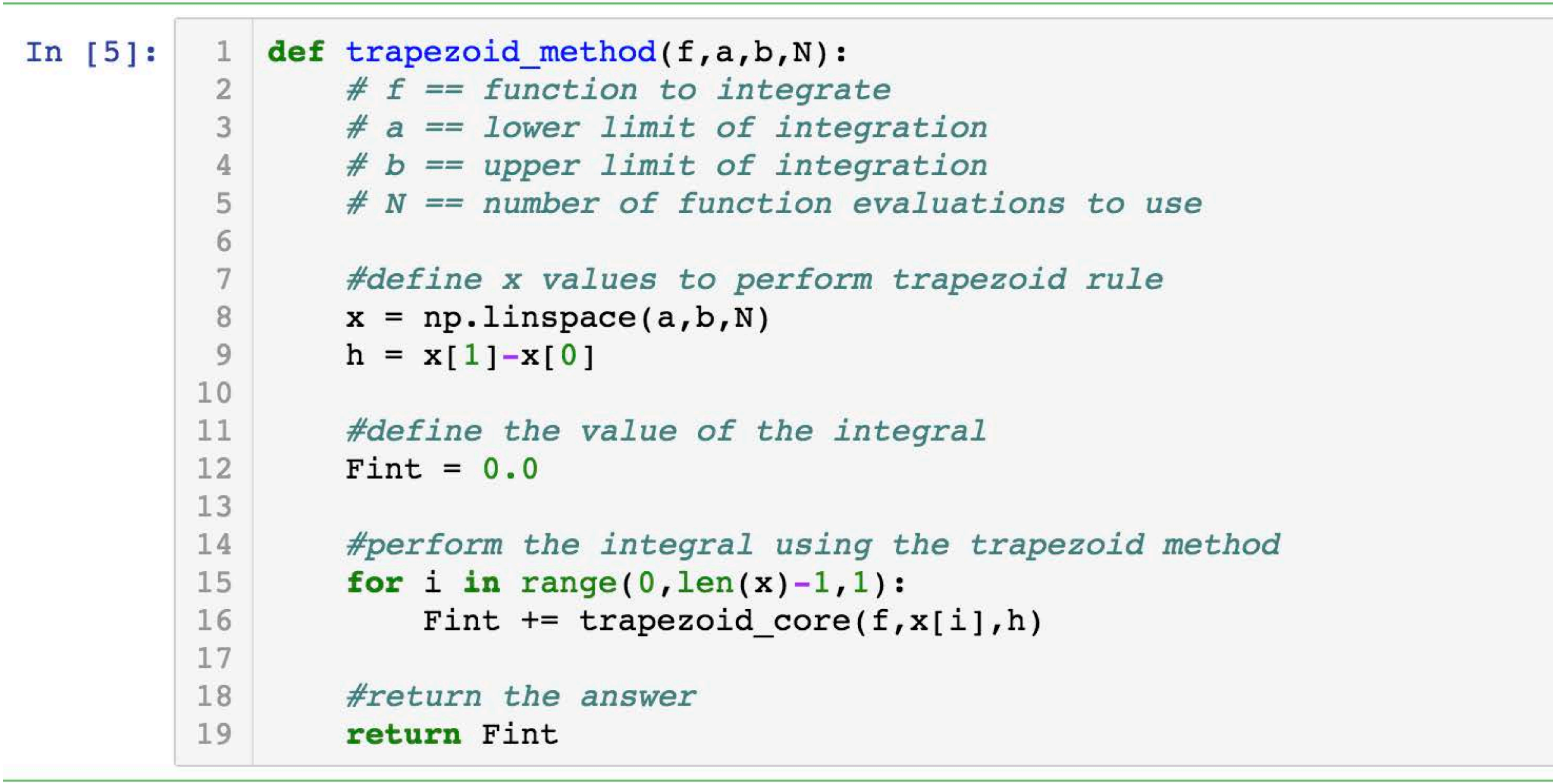
Define the core of the trapezoid method

In [4]:

```python
def trapezoid_core(f,x,h):
    return 0.5*h*(f(x+h) + f(x))
```

Integration Example

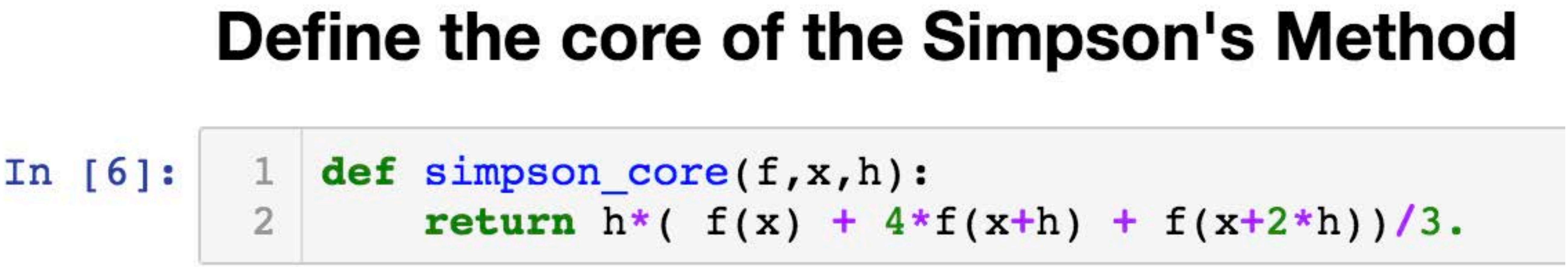
Define a wrapper function to perform trapezoid method



In [5]:

```python
def trapezoid_method(f,a,b,N):
    # f == function to integrate
    # a == lower limit of integration
    # b == upper limit of integration
    # N == number of function evaluations to use

    #define x values to perform trapezoid rule
    x = np.linspace(a,b,N)
    h = x[1]-x[0]

    #define the value of the integral
    Fint = 0.0

    #perform the integral using the trapezoid method
    for i in range(0,len(x)-1,1):
        Fint += trapezoid_core(f,x[i],h)

    #return the answer
    return Fint
```

Integration Example



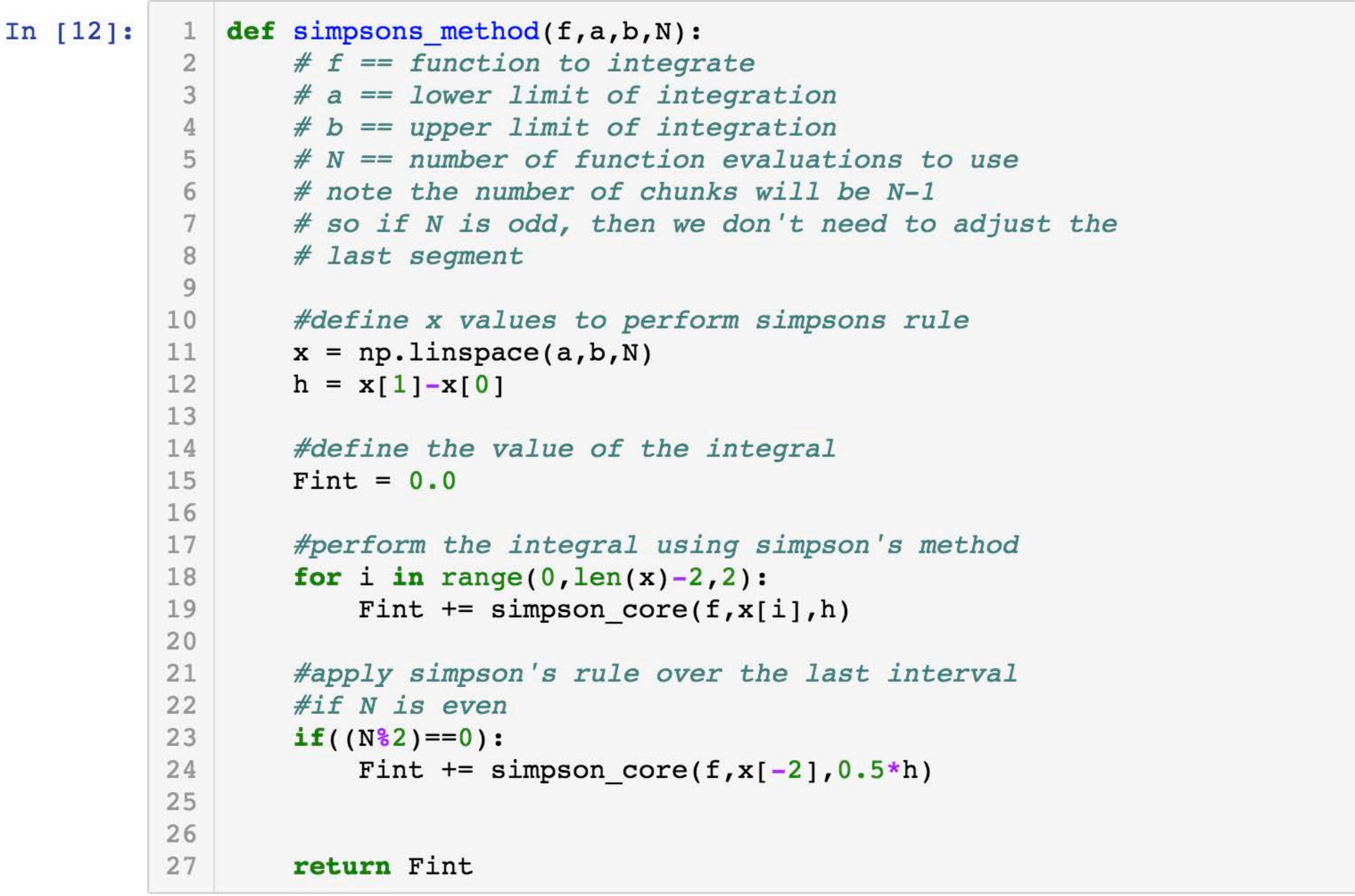
Define the core of the Simpson's Method

In [6]:

```
def simpson_core(f,x,h):
    return h*( f(x) + 4*f(x+h) + f(x+2*h))/3.
```

Integration Example

Define a wrapper function to perform Simpson's Method

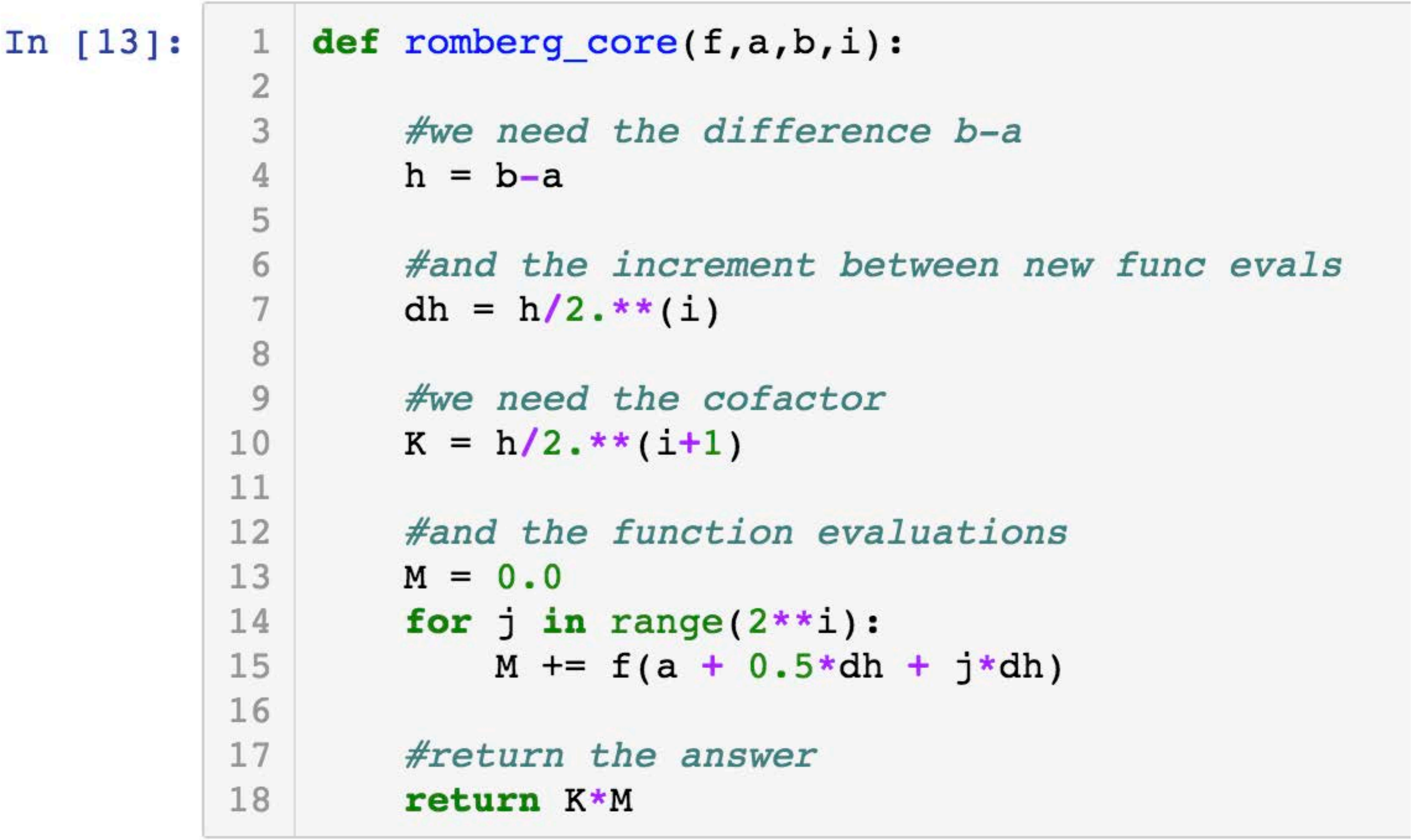


In [12]:

```
def simpsons_method(f,a,b,N):
    # f == function to integrate
    # a == lower limit of integration
    # b == upper limit of integration
    # N == number of function evaluations to use
    # note the number of chunks will be N-1
    # so if N is odd, then we don't need to adjust the
    # last segment

    #define x values to perform simpsons rule
    x = np.linspace(a,b,N)
    h = x[1]-x[0]

    #define the value of the integral
    Fint = 0.0

    #perform the integral using simpson's method
    for i in range(0,len(x)-2,2):
        Fint += simpson_core(f,x[i],h)

    #apply simpson's rule over the last interval
    #if N is even
    if((N%2)==0):
        Fint += simpson_core(f,x[-2],0.5*h)


    return Fint
```

Integration Example

Define the Romberg Integration core

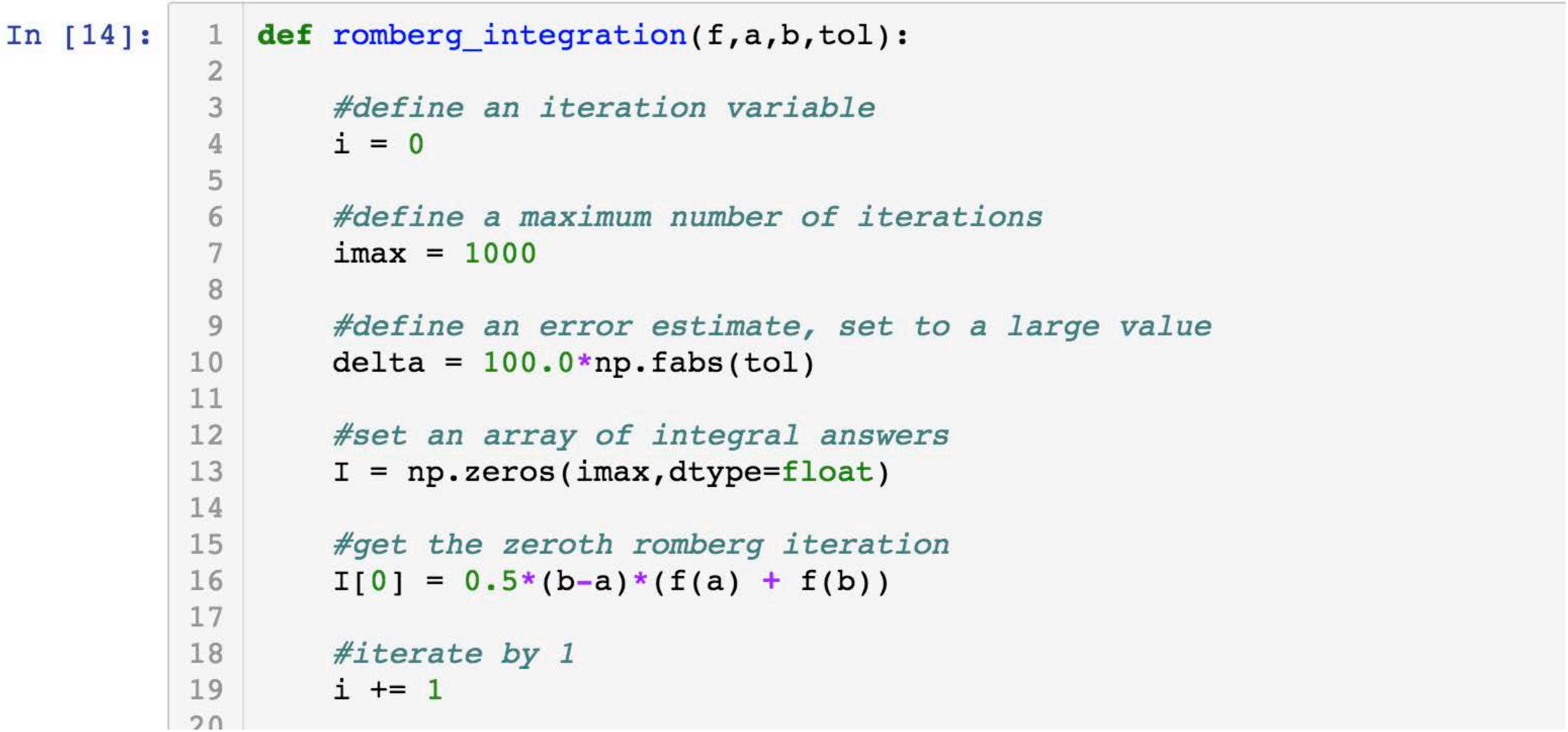


In [13]:

```python
def romberg_core(f,a,b,i):

    #we need the difference b-a
    h = b-a

    #and the increment between new func evals
    dh = h/2.**(i)

    #we need the cofactor
    K = h/2.**(i+1)

    #and the function evaluations
    M = 0.0
    for j in range(2**i):
        M += f(a + 0.5*dh + j*dh)

    #return the answer
    return K*M
```

Integration Example

Define a wrapper function to perform Romberg Integration

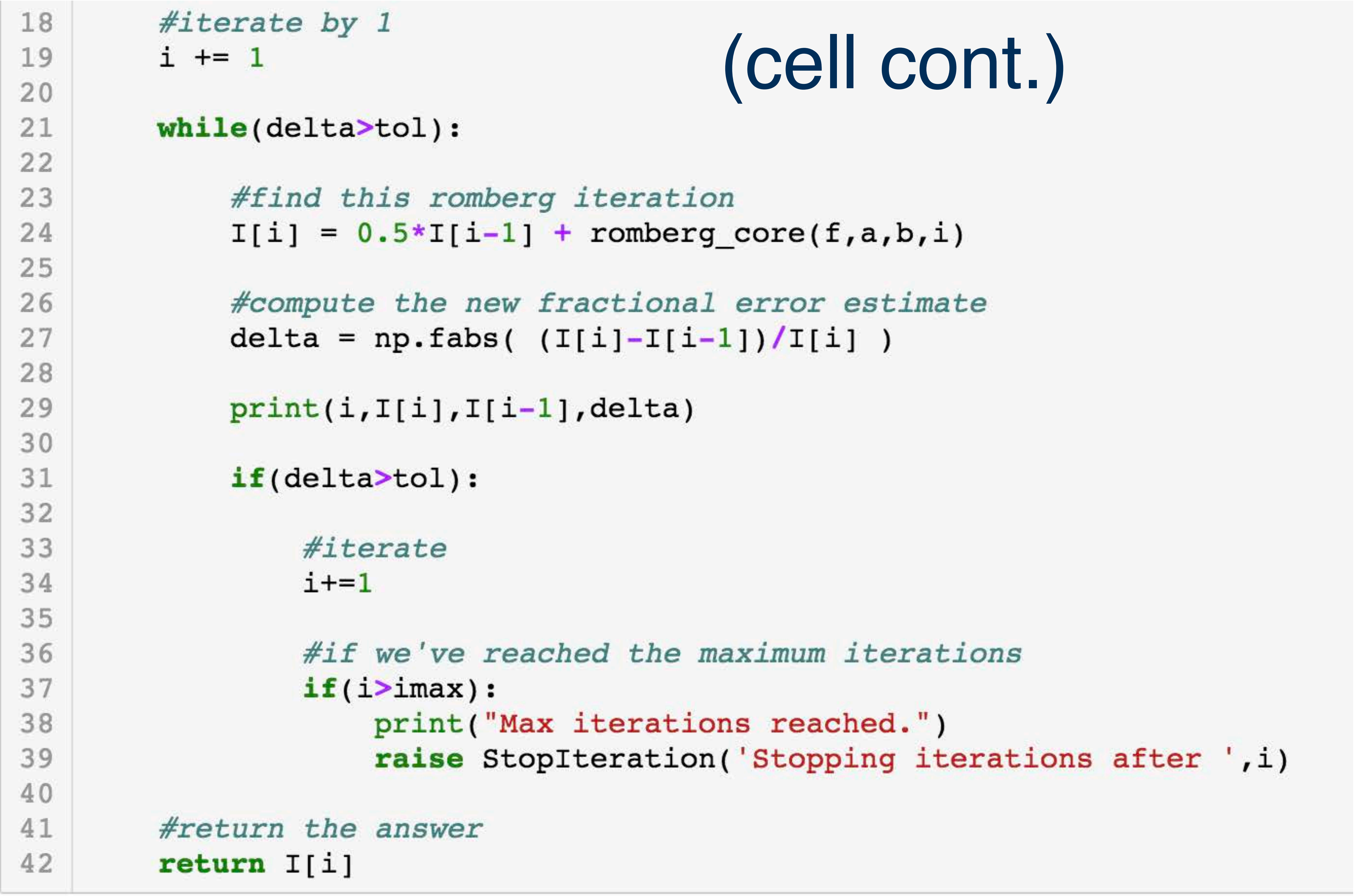


In [14]:

```python
def romberg_integration(f,a,b,tol):

    #define an iteration variable
    i = 0

    #define a maximum number of iterations
    imax = 1000

    #define an error estimate, set to a large value
    delta = 100.0*np.fabs(tol)

    #set an array of integral answers
    I = np.zeros(imax,dtype=float)

    #get the zeroth romberg iteration
    I[0] = 0.5*(b-a)*(f(a) + f(b))

    #iterate by 1
    i += 1
```

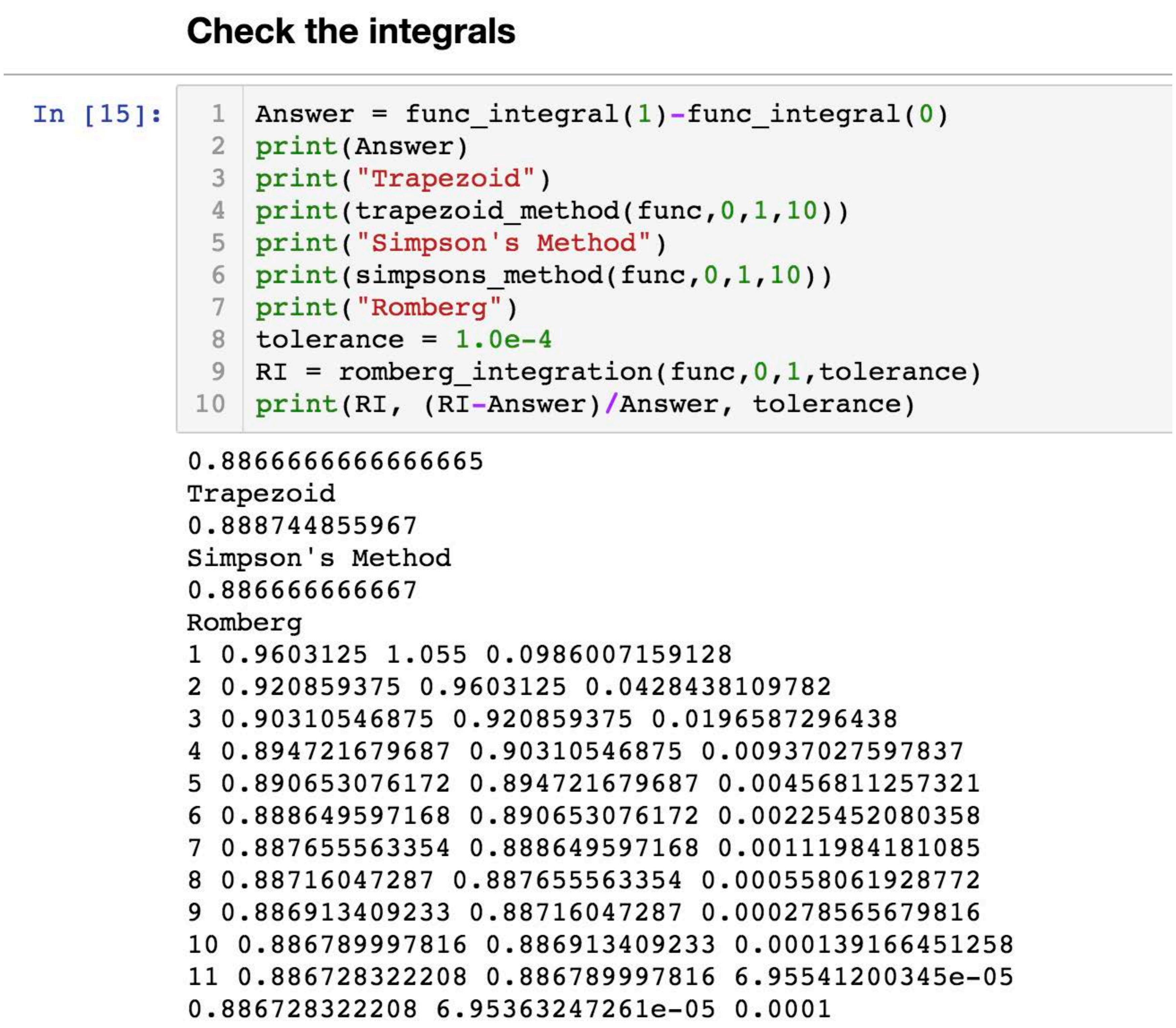
Integration Example

(cell cont.)



```
    #iterate by 1
    i += 1

    while(delta>tol):

        #find this romberg iteration
        I[i] = 0.5*I[i-1] + romberg_core(f,a,b,i)

        #compute the new fractional error estimate
        delta = np.fabs( (I[i]-I[i-1])/I[i] )

        print(i,I[i],I[i-1],delta)

        if(delta>tol):

            #iterate
            i+=1

            #if we've reached the maximum iterations
            if(i>imax):
                print("Max iterations reached.")
                raise StopIteration('Stopping iterations after ',i)

    #return the answer
    return I[i]
```

Integration Example



Check the integrals

In [15]:

```
Answer = func_integral(1)-func_integral(0)
print(Answer)
print("Trapezoid")
print(trapezoid_method(func,0,1,10))
print("Simpson's Method")
print(simpsons_method(func,0,1,10))
print("Romberg")
tolerance = 1.0e-4
RI = romberg_integration(func,0,1,tolerance)
print(RI, (RI-Answer)/Answer, tolerance)
```

```
0.8866666666666665
Trapezoid
0.888744855967
Simpson's Method
0.886666666667
Romberg
1 0.9603125 1.055 0.0986007159128
2 0.920859375 0.9603125 0.0428438109782
3 0.90310546875 0.920859375 0.0196587296438
4 0.894721679687 0.90310546875 0.00937027597837
5 0.890653076172 0.894721679687 0.00456811257321
6 0.888649597168 0.890653076172 0.00225452080358
7 0.887655563354 0.888649597168 0.00111984181085
8 0.88716047287 0.887655563354 0.000558061928772
9 0.886913409233 0.88716047287 0.000278565679816
10 0.886789997816 0.886913409233 0.000139166451258
11 0.886728322208 0.886789997816 6.95541200345e-05
0.886728322208 6.95363247261e-05 0.0001
```

ORDINARY DIFFERENTIAL EQUATIONS

What is an ODE? **Consider a function f(t) that is depends only on a single variable t. A differential equation is an equality involving one or more derivatives of the function along with any combination of the function itself and the independent variable. This equality holds for any value of the independent variable. Here are some examples:**

$$\frac{df}{dt} + f^3 = ft^2$$

$$\frac{d^2 f}{dt^2} + t^3 \frac{df}{dt} = ft^2$$

To solve the ODE: find f(t) for all t that satisfies the equality.

ODEs: BOUNDARY CONDITIONS

Consider the following ODE:

$$\frac{df}{dt} = t^2$$

ODEs: BOUNDARY CONDITIONS

Consider the following ODE:

$$\frac{df}{dt} = t^2$$

You know the solution:

$$f(t) = \frac{1}{3}t^3 + C$$

ODEs: BOUNDARY CONDITIONS

Consider the following ODE:

$$\frac{df}{dt} = t^2$$

You know the solution:

$$f(t) = \frac{1}{3}t^3 + C$$

To determine C, we must have a boundary condition that specifies f(t_0) at some time t_0. Since there are infinitely many solutions involving infinitely many choices of C, when we numerically solve ODEs we must always adopt some boundary conditions as part of the solution process.

ORDINARY DIFFERENTIAL EQUATIONS

In physics, we often describe the state of a system as a function of some unknown variables. For instance, the vertical motion of a projectile under the influence of gravity can be written:

$$y(t) = y_0 + v_0 t + \frac{1}{2} g t^2$$

However, the same physical system can be described in terms of differential equations:

$$\dot{y} = v(t)$$
$$\dot{v} = g$$

with the boundary conditions:

$$v(t = 0) = v_0$$
$$y(t = 0) = y_0$$

Finite Difference

Consider the differential equation:

$$f' \equiv \frac{df}{dx} = g(x, f), \quad f(x_0) = f_0$$

We want to solve for values f(x_i) on a grid x_i, i=[1,n].

We will notate f(x_i) as f_i and f(x_{i+1}) as f_{i+1}. Symbolically, we then want

$$f_i, x_i, g(x_i, f_i) \rightarrow f_{i+1}$$

How NOT to Evolve an ODE

Consider the discretized equation:

$$f_{i+1} = f_i + hg(x_i, f_i) + \frac{h^2}{2!} f''(\xi)$$

While this initially seems sensible, note that the error term for each step is quadratic in *h*. After many steps, the overall error will only improve as *h*, and that is not good! We need to search out better methods.

SECOND-ORDER RUNGE-KUTTA METHOD

We can produce a more accurate method by taking a half-step, f$_{i+1/2}$:

$$f_{i+1/2} = f_i + \frac{h}{2} f_i' + O(h^2 f'')$$

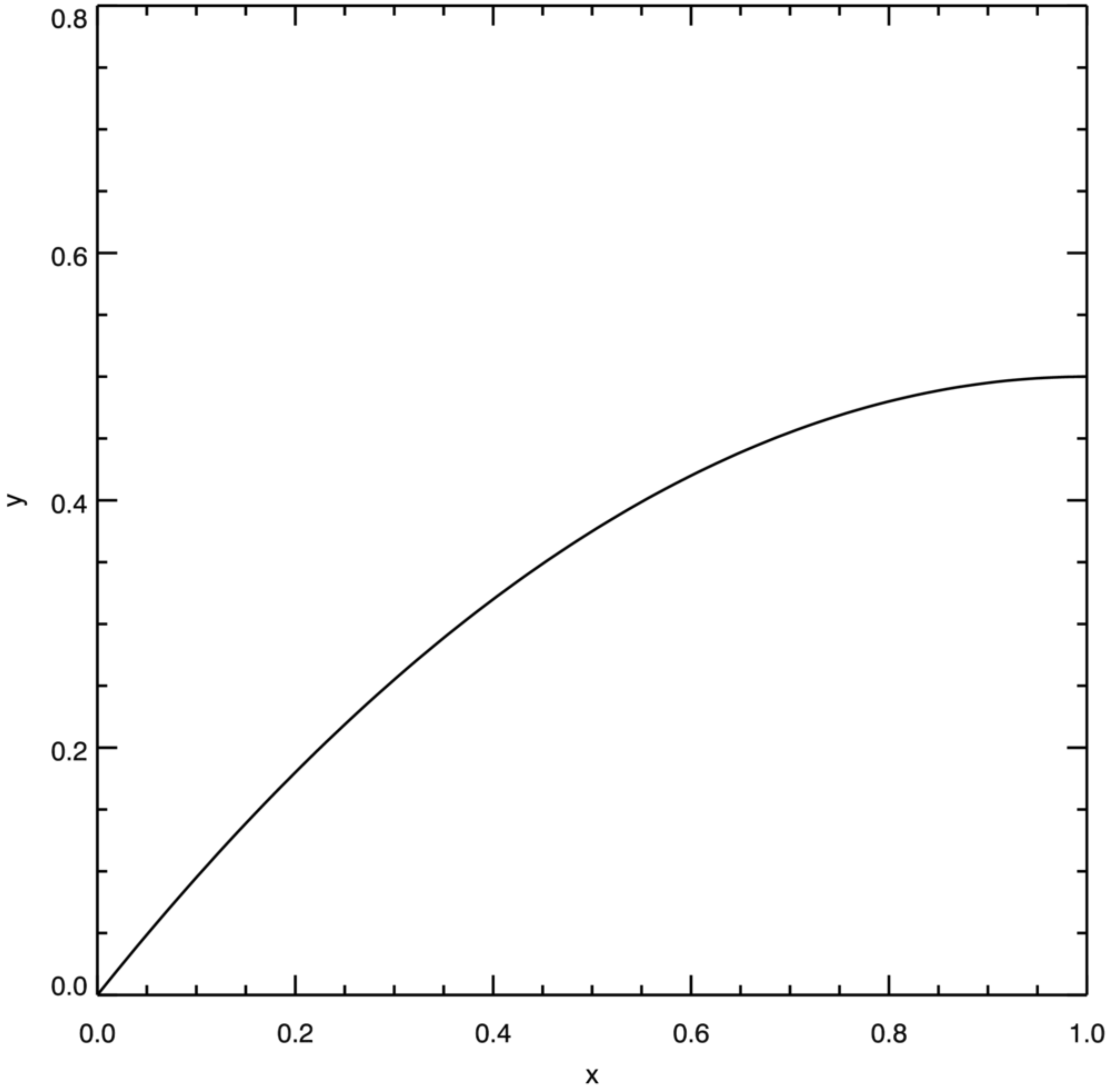
All terms involving f$_{i+1/2}$ in our approximation to f$_i$ are multiplied by h and we preserve an error O(h^3). Here are the equations that we need to evaluate:

$$f_{i+1/2} = f_i + \frac{h}{2} g(x_i, f_i)$$

$$f_{i+1} = f_i + hg(x_{i+1/2}, f_{i+1/2}) + O(h^3)$$

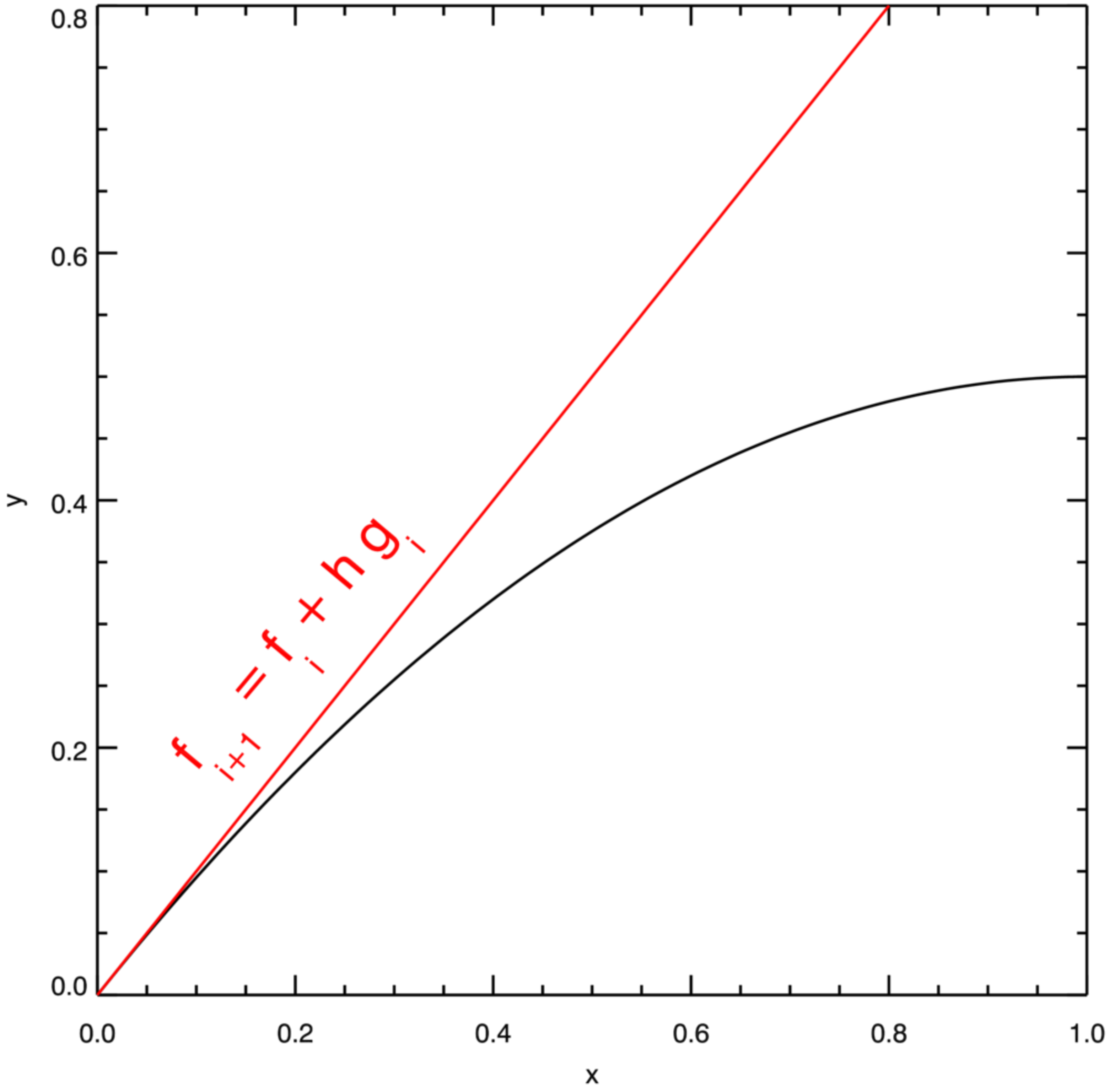
This is called the second-order Runge-Kutta method or the midpoint method.

SECOND-ORDER RUNGE-KUTTA METHOD



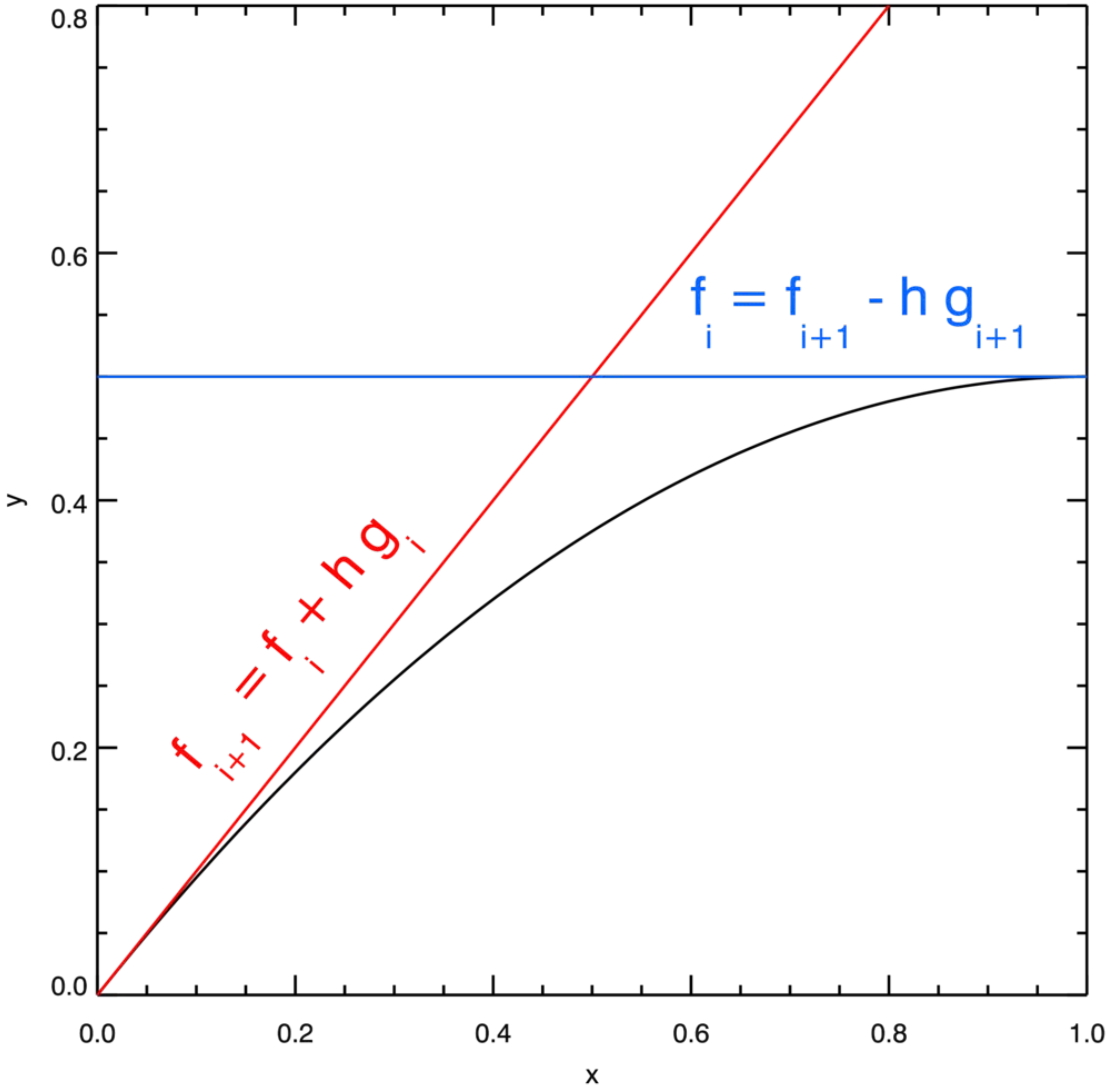
This is called the second-order Runge-Kutta method or the midpoint method.

SECOND-ORDER RUNGE-KUTTA METHOD



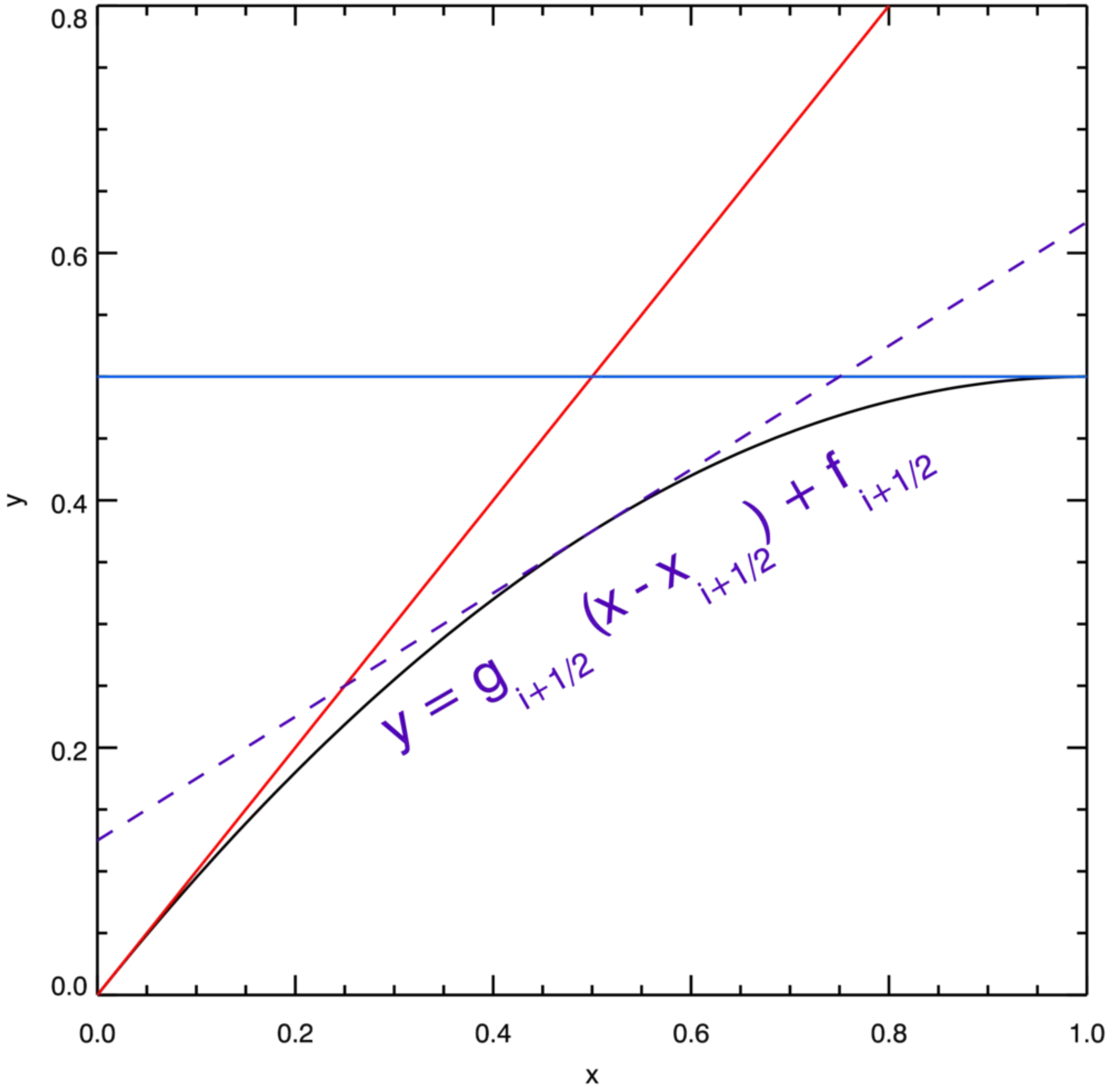
This is called the second-order Runge-Kutta method or the midpoint method.

SECOND-ORDER RUNGE-KUTTA METHOD



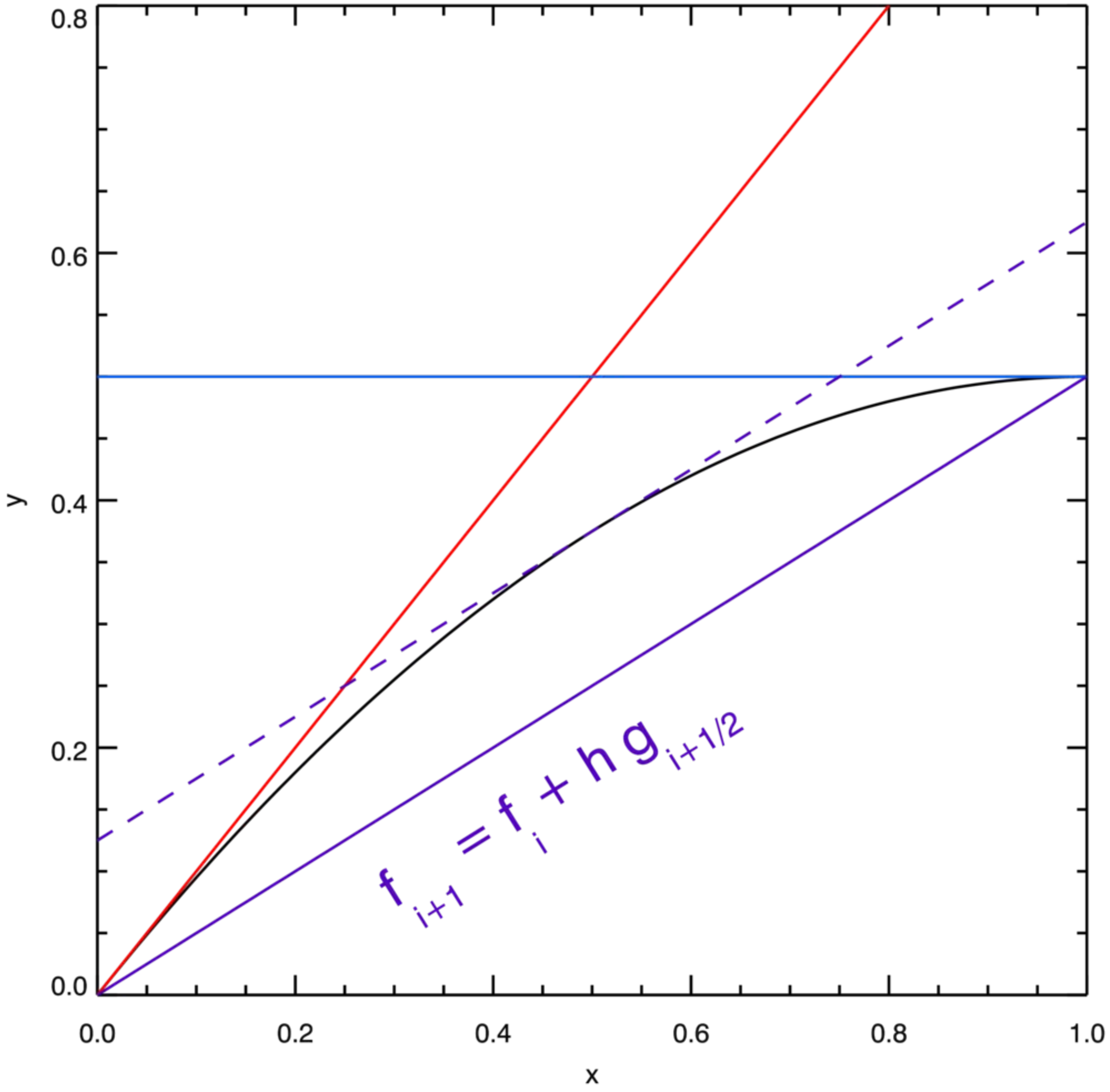
This is called the **second-order Runge-Kutta** **method or the** **midpoint** **method.**

SECOND-ORDER RUNGE-KUTTA METHOD



This is called the second-order Runge-Kutta method or the midpoint method.

SECOND-ORDER RUNGE-KUTTA METHOD



This is called the second-order Runge-Kutta method or the midpoint method.

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

$$k_2 = hg\left(x_{i+1/2}, f_i + \frac{k_1}{2}\right)$$

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

$$k_2 = hg\left(x_{i+1/2}, f_i + \frac{k_1}{2}\right)$$

$$k_3 = hg\left(x_{i+1/2}, f_i + \frac{k_2}{2}\right)$$

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

$$k_2 = hg\left(x_{i+1/2}, f_i + \frac{k_1}{2}\right)$$

$$k_3 = hg\left(x_{i+1/2}, f_i + \frac{k_2}{2}\right)$$

$$k_4 = hg\left(x_{i+1}, f_i + k_3\right)$$

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

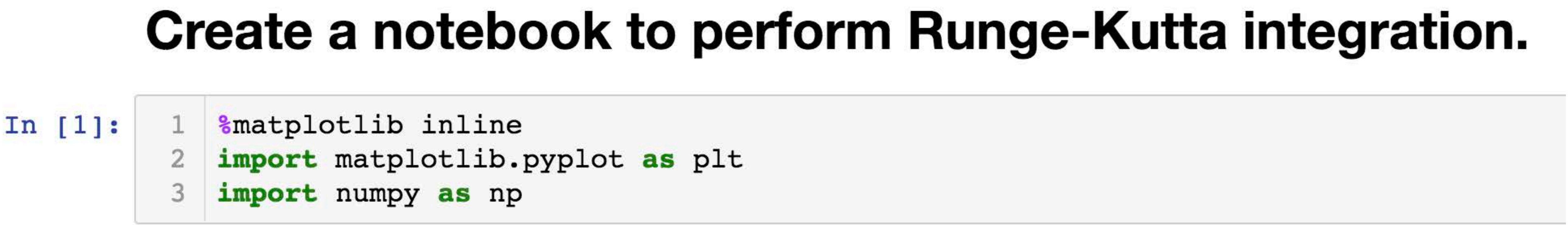
$$k_2 = hg\left(x_{i+1/2}, f_i + \frac{k_1}{2}\right)$$

$$k_3 = hg\left(x_{i+1/2}, f_i + \frac{k_2}{2}\right)$$

$$k_4 = hg\left(x_{i+1}, f_i + k_3\right)$$

$$f_{i+1} = f_i + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)$$

Runge-Kutta Implementation

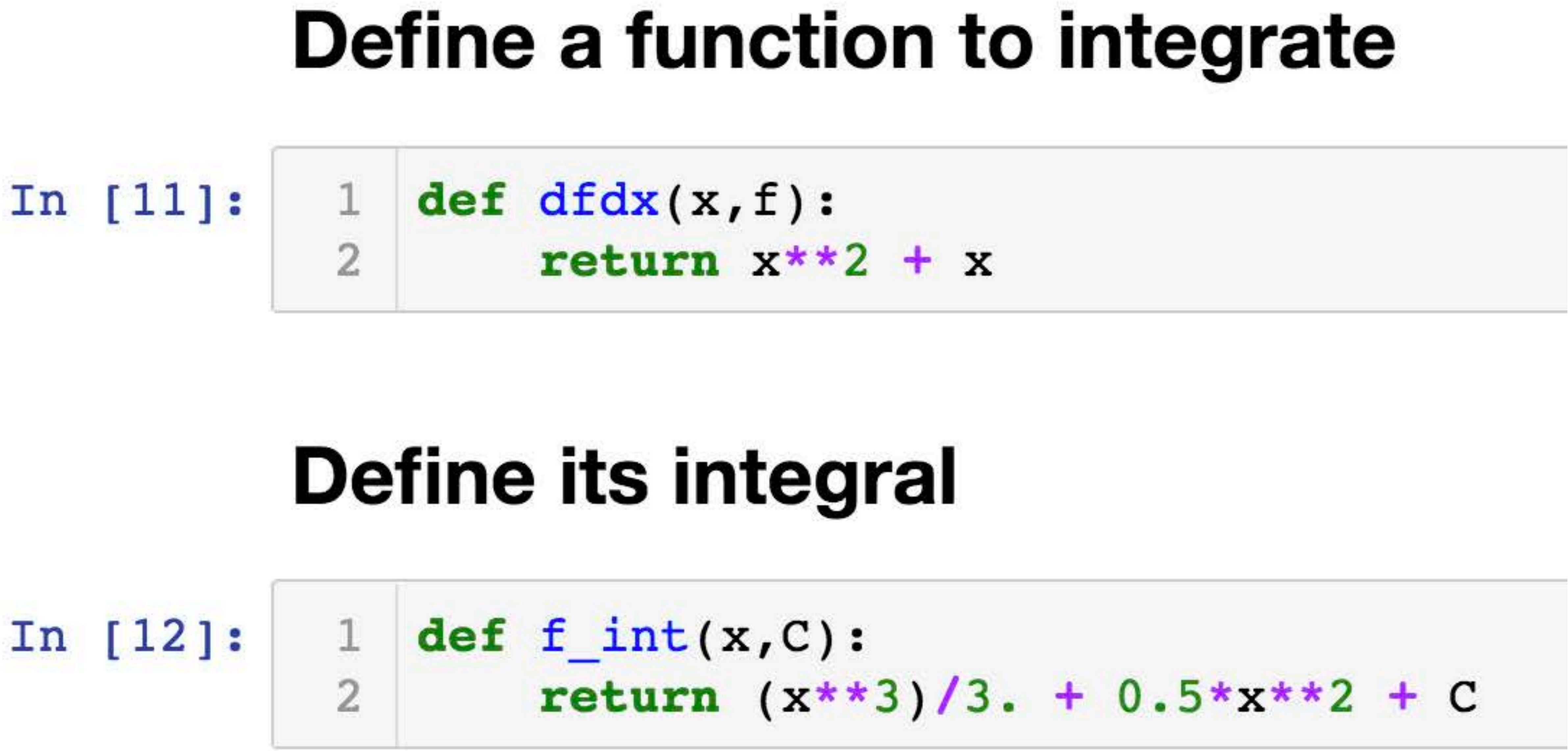


Create a notebook to perform Runge-Kutta integration.

In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

Runge-Kutta Implementation



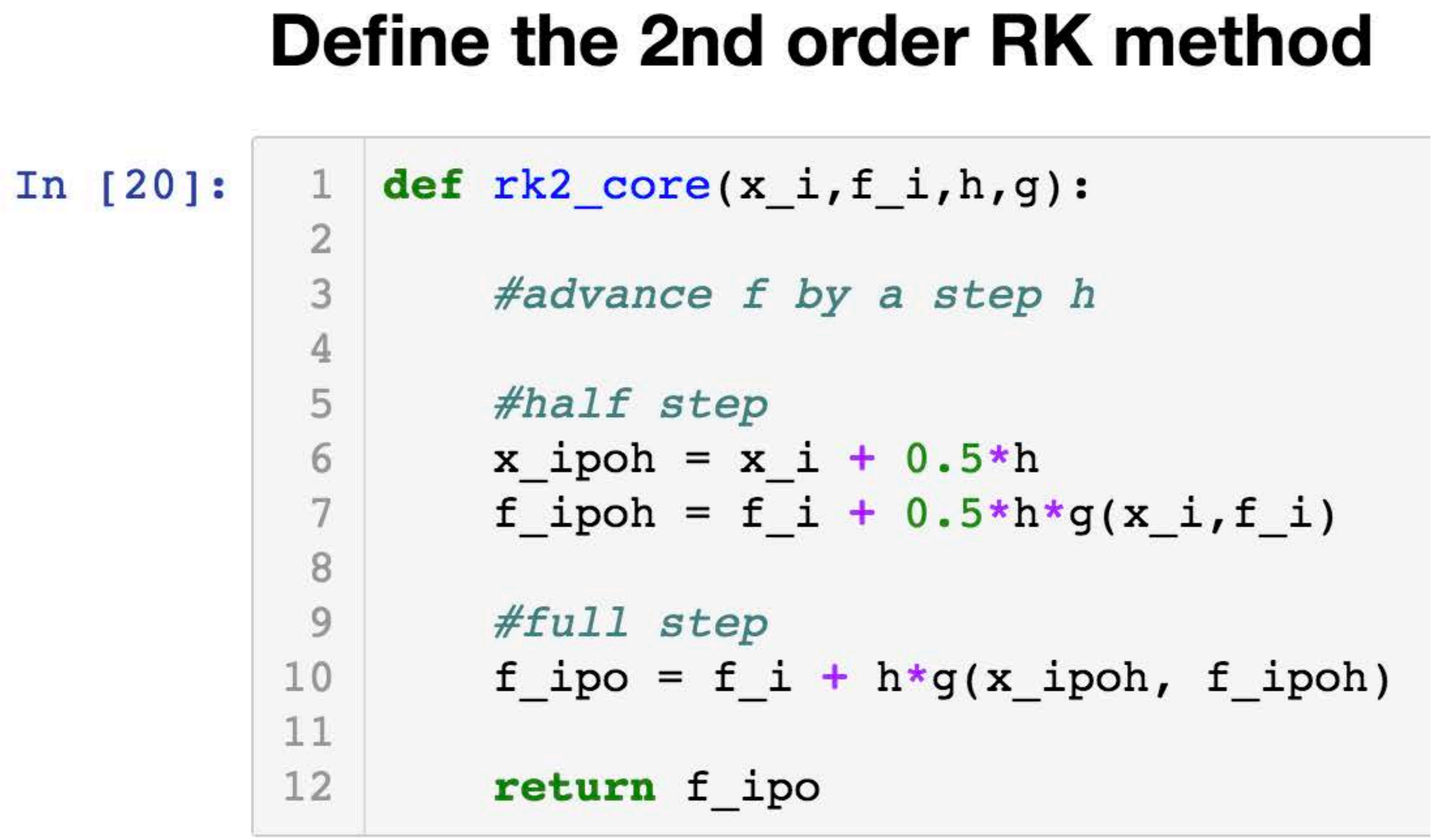
Define a function to integrate

```
In [11]:
def dfdx(x,f):
    return x**2 + x
```

Define its integral

```
In [12]:
def f_int(x,C):
    return (x**3)/3. + 0.5*x**2 + C
```

Runge-Kutta Implementation



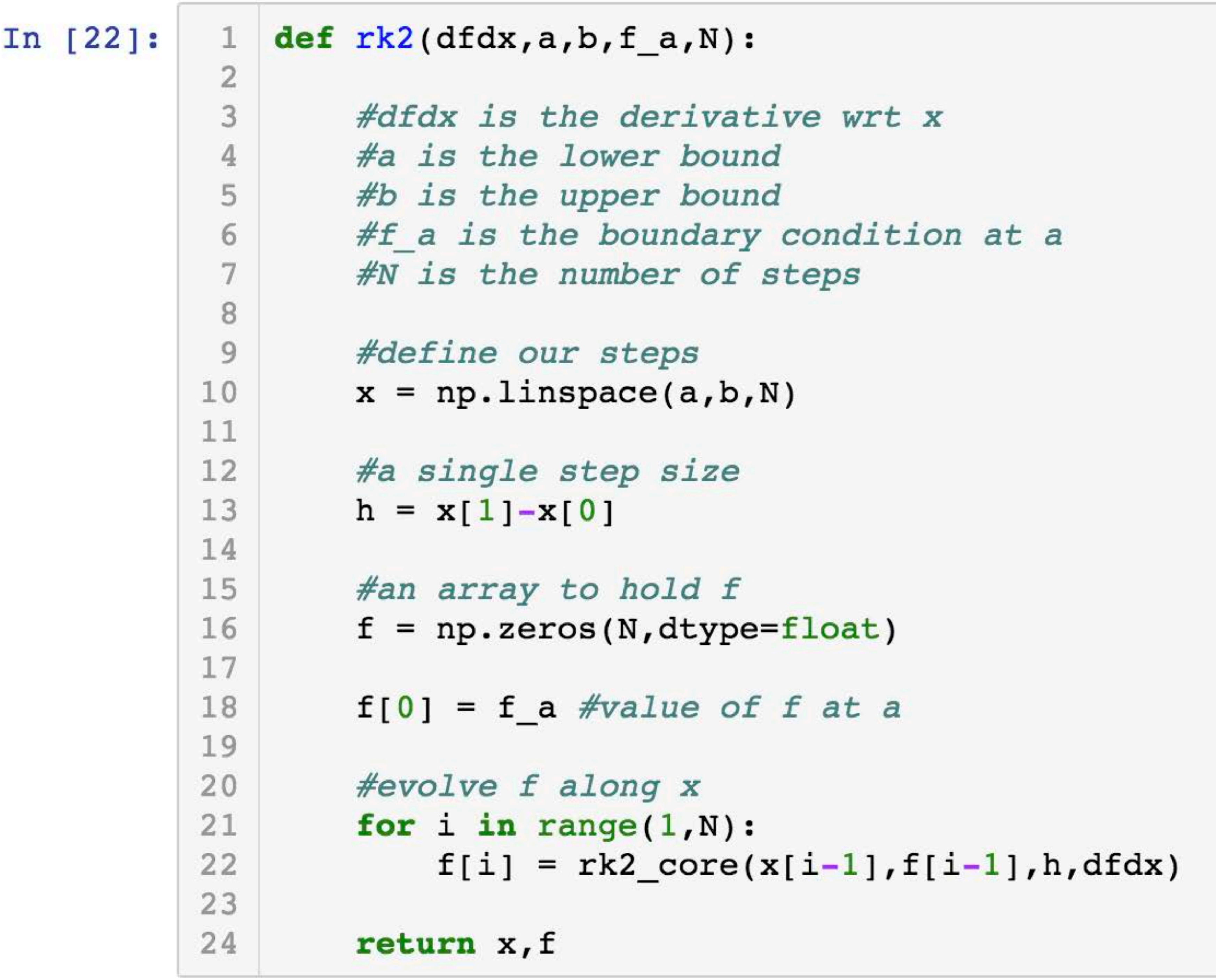
Define the 2nd order RK method

In [20]:

```python
def rk2_core(x_i,f_i,h,g):

    #advance f by a step h

    #half step
    x_ipoh = x_i + 0.5*h
    f_ipoh = f_i + 0.5*h*g(x_i,f_i)

    #full step
    f_ipo = f_i + h*g(x_ipoh, f_ipoh)

    return f_ipo
```

Runge-Kutta Implementation

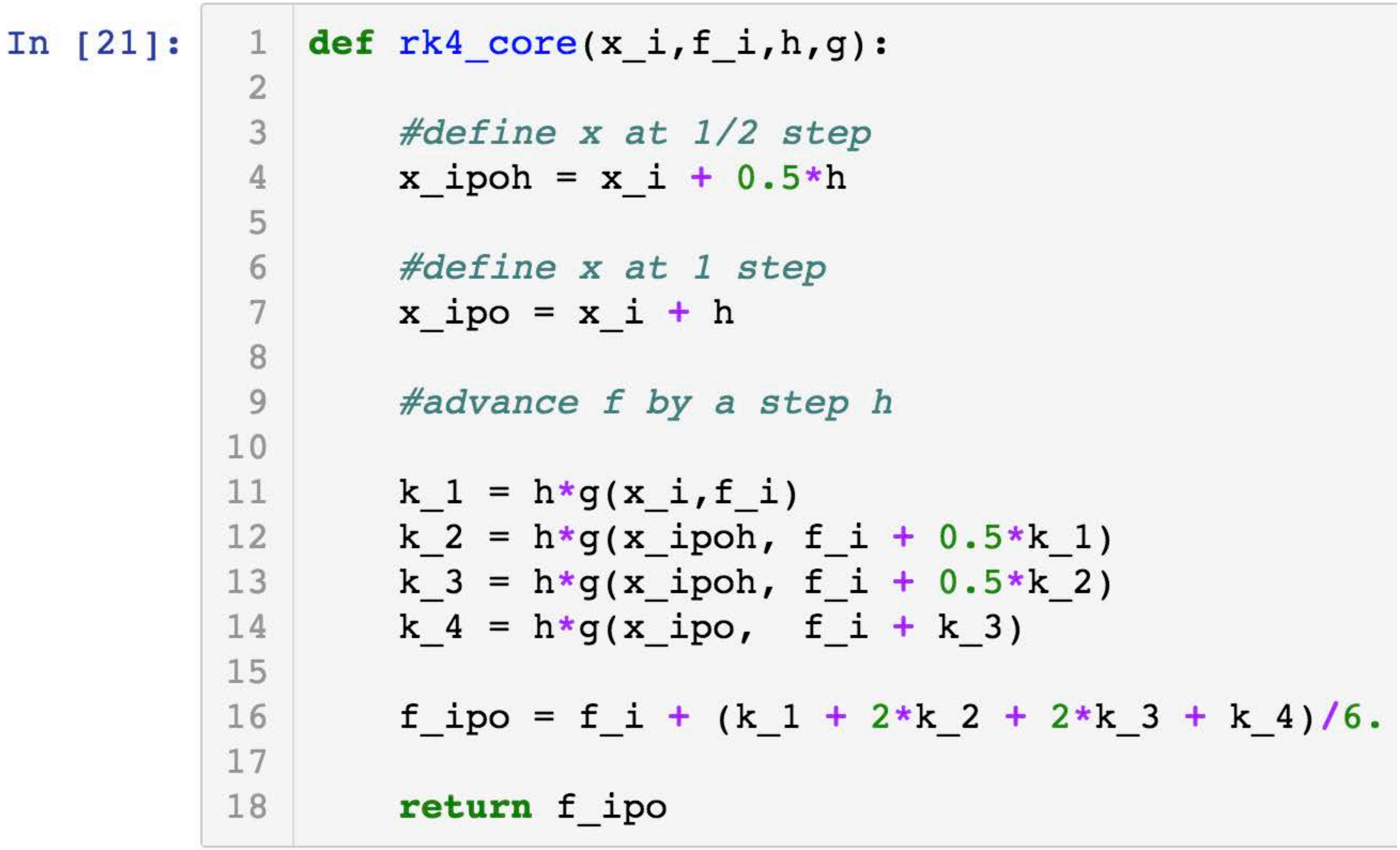
Define a wrapper routine for RK2



```python
def rk2(dfdx,a,b,f_a,N):

    #dfdx is the derivative wrt x
    #a is the lower bound
    #b is the upper bound
    #f_a is the boundary condition at a
    #N is the number of steps

    #define our steps
    x = np.linspace(a,b,N)

    #a single step size
    h = x[1]-x[0]

    #an array to hold f
    f = np.zeros(N,dtype=float)

    f[0] = f_a #value of f at a

    #evolve f along x
    for i in range(1,N):
        f[i] = rk2_core(x[i-1],f[i-1],h,dfdx)

    return x,f
```

Runge-Kutta Implementation

Define the 4th order RK method

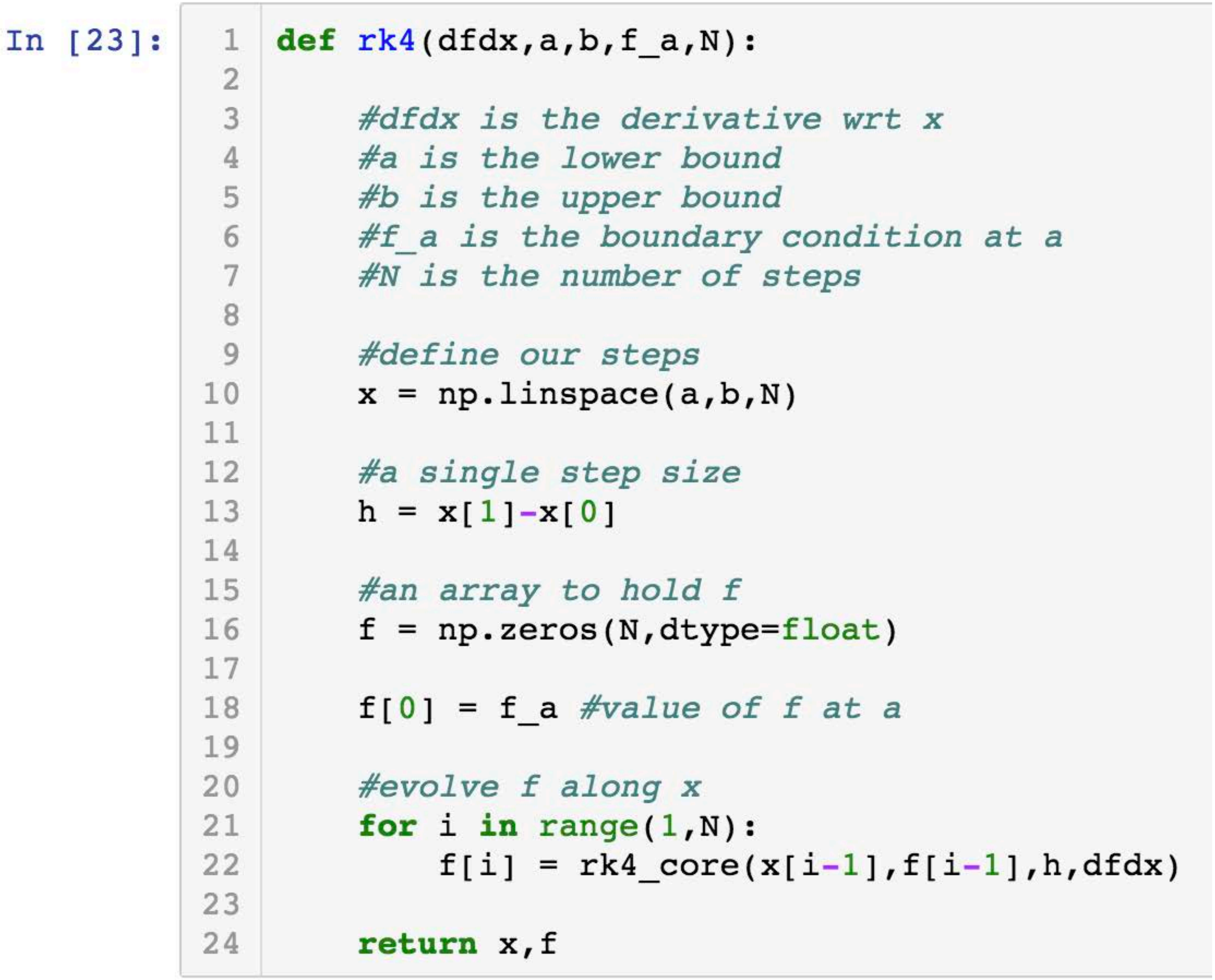


In [21]:

```
def rk4_core(x_i,f_i,h,g):

    #define x at 1/2 step
    x_ipoh = x_i + 0.5*h

    #define x at 1 step
    x_ipo = x_i + h

    #advance f by a step h

    k_1 = h*g(x_i,f_i)
    k_2 = h*g(x_ipoh, f_i + 0.5*k_1)
    k_3 = h*g(x_ipoh, f_i + 0.5*k_2)
    k_4 = h*g(x_ipo,  f_i + k_3)

    f_ipo = f_i + (k_1 + 2*k_2 + 2*k_3 + k_4)/6.

    return f_ipo
```

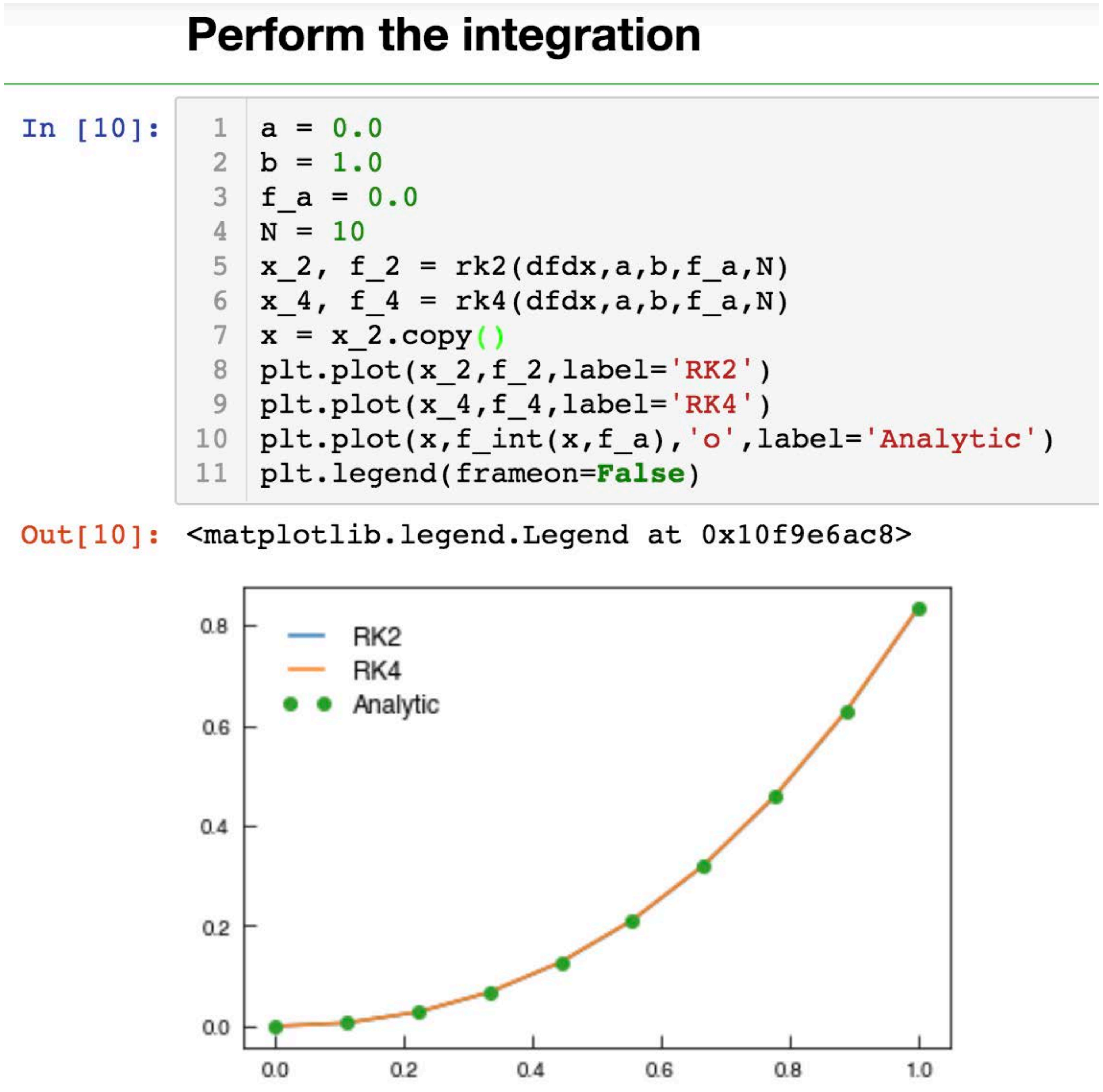
Runge-Kutta Implementation

Define a wrapper for RK4



```python
def rk4(dfdx,a,b,f_a,N):

    #dfdx is the derivative wrt x
    #a is the lower bound
    #b is the upper bound
    #f_a is the boundary condition at a
    #N is the number of steps

    #define our steps
    x = np.linspace(a,b,N)

    #a single step size
    h = x[1]-x[0]

    #an array to hold f
    f = np.zeros(N,dtype=float)

    f[0] = f_a #value of f at a

    #evolve f along x
    for i in range(1,N):
        f[i] = rk4_core(x[i-1],f[i-1],h,dfdx)

    return x,f
```

Runge-Kutta Implementation



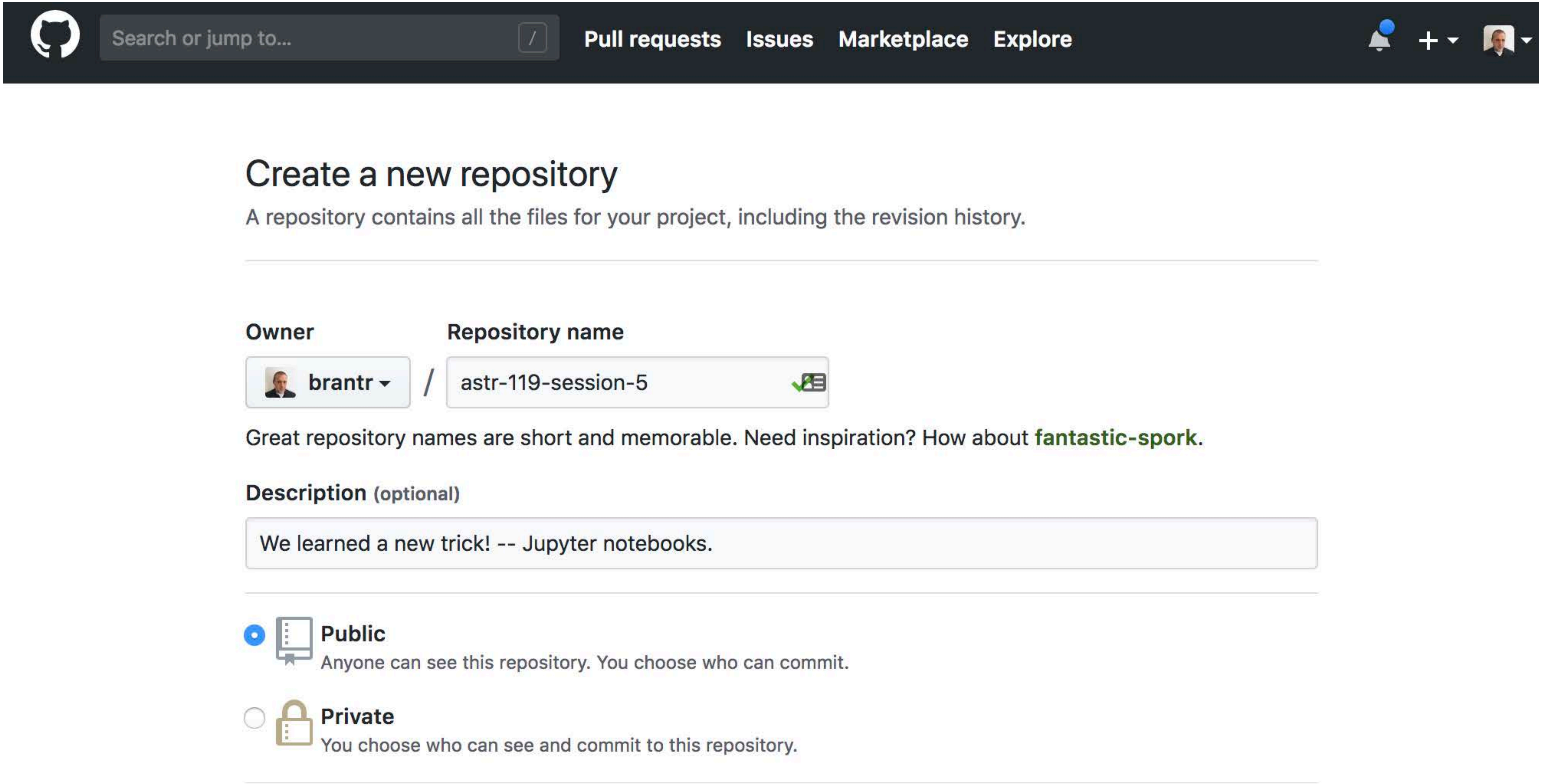
Perform the integration

```
In [10]: a = 0.0
         b = 1.0
         f_a = 0.0
         N = 10
         x_2, f_2 = rk2(dfdx,a,b,f_a,N)
         x_4, f_4 = rk4(dfdx,a,b,f_a,N)
         x = x_2.copy()
         plt.plot(x_2,f_2,label='RK2')
         plt.plot(x_4,f_4,label='RK4')
         plt.plot(x,f_int(x,f_a),'o',label='Analytic')
         plt.legend(frameon=False)
```

```
Out[10]: <matplotlib.legend.Legend at 0x10f9e6ac8>
```

Save Your Work

Make a GitHub project “astr-119-session-10”, and commit the programs my_first_jupyter_notebook.ipynb and test_matplotlib.ipynb you made today.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: brantr / Repository name: astr-119-session-5

Great repository names are short and memorable. Need inspiration? How about **fantastic-spork**.

Description (optional)

We learned a new trick! -- Jupyter notebooks.

● **Public**
Anyone can see this repository. You choose who can commit.

○ **Private**
You choose who can see and commit to this repository.