CHAPTER

5 Dictionaries and Functions as Arguments

OUTLINE

DICTIONARY, n. A malevolent literary device for cramping the growth of a language and making it hard and inelastic. This dictionary, however, is a most useful work.

–*Ambrose Bierce,* The Devil's Dictionary

CHAPTER POINTS

- Dictionaries are lists where the elements are accessed via special names, called keys.
- In Python it is possible to pass the name of a function as an argument to another function. This allows functions, such as numerical integration, to operate on an arbitrary function.
- Lambda functions allow the programmer to define a function in one line and use all the variables in the current scope.

5.1 DICTIONARIES

Previously, we learned about lists as a sequence of items that we can access via position using square brackets. There may be cases where we do not want to access items based on a numerical index, rather we want to access them based on a name. A typical example of this might be the children in a family. You could have a list of children that you access via the

DOI: 10.1016/B978-0-12-812253-2.00006-6

order that they were born; however, this would be fairly impersonal and not a useful ordering for anyone other than the parents in that family. The solution to this problem in Python can be found in the dictionary. A dictionary is like a list in many ways, but you access it with the name of the item.

In technical terms, a dictionary is a set of `key:value` pairs. The key is the analog to the index of a list, and is, in effect, the name of the item. You define a dictionary using curly braces.

```
In [1]: #simple dictionary
        days_of_week = {"M":"Monday", "T":"Tuesday",
                        "W":"Wednesday", "R":"Thursday",
                        "F":"Friday", "S":"Saturday",
                        "U":"Sunday"}
        print("Key M gives", days_of_week["M"])
        print("Key R gives", days_of_week["R"])
        #is G a key in days_of_week?
        print("G" in days_of_week.keys())

Key M gives Monday
Key R gives Thursday
False
```

Instead of accessing the dictionary using a position, like we have done with strings, lists, and NumPy arrays, we use the key. This is useful because we then do not have to remember the order we have listed the values in. For example, in the days of the week above we don't have to remember how we ordered the days (e.g., Monday first or Sunday first).

Also, the above example used the `in` operator to indicate if a particular key is in a dictionary. In particular, it tells us that `"G"` is not a key in the dictionary.

BOX 5.1 PYTHON PRINCIPLE

A dictionary is a sequence of items that is accessed via a name called a key. The elements of the dictionary that the key refers to is called the value. To define a dictionary we use curly brackets as in the following example

```
my_dictionary ={key_1:value_1,
                key_2:value_2, ...}
```

A list of the keys in a dictionary can be obtained via the function `d.keys()`, where `d` is the name of a dictionary.

For a further example we will read in a comma-separated-values text file (often called a csv), using the module `csv`. The text file will be used to give the `key:value` pairs in a dictionary. The format of this file is `chemical symbol, element name`. The first few lines of the file are

```
Ac,Actinium
Ag,Silver
Al,Aluminum
Am,Americium
```

The following code reads in the file and uses the chemical symbol as the key and the chemical name as the value. It also asks the user to input a chemical symbol, and will return the name.

```
In [2]: import csv
        #create a blank dictionary
        element_dict = {}
        #this block will only execute if the file opens
        with open('ChemicalSymbols.csv') as csvfile:
            chemreader = csv.reader(csvfile)
            for row in chemreader: #have for loop that loops over each line
                element_dict[row[0]] = row[1] #add a key:value pair
        key = input("Enter a valid chemical symbol: ")
        if key in element_dict:
            print(key,"is",element_dict[key])
        else:
            print("Not a valid element")

Enter a valid chemical symbol: Pu
Pu is Plutonium
```

Dictionaries can be made even more powerful, if we make a dictionary of dictionaries. Yes, you read that correctly: the `value` in the `key:value` pair can be another dictionary. For many applications, this is where dictionaries become very useful. In the following example we use idea of a dictionary of dictionaries to store extra information about the days of the week.

```
In [3]: #simple dictionary of dictionaries
        days_of_week = {"M":{"name":"Monday","weekday":True,"weekend":False},
                        "T":{"name":"Tuesday","weekday":True,"weekend":False},
                        "W":{"name":"Wednesday","weekday":True,"weekend":False},
                        "R":{"name":"Thursday","weekday":True,"weekend":False},
                        "F":{"name":"Friday","weekday":True,"weekend":False},
                        "S":{"name":"Saturday","weekday":False,"weekend":True},
                        "U":{"name":"Sunday","weekday":False,"weekend":True}}
        print("The days that are weekdays:")
        for day in days_of_week: #for loop over dictionary, loops over keys
            if days_of_week[day]["weekday"] == True:
                print(days_of_week[day]["name"],"is a weekday.")

        for day in days_of_week: #for loop over dictionary, loops over keys
            if days_of_week[day]["weekend"] == True:
                print(days_of_week[day]["name"],"is a weekend, whoop.")

The days that are weekdays:
Thursday is a weekday.
Wednesday is a weekday.
Tuesday is a weekday.
Monday is a weekday.
Friday is a weekday.
Saturday is a weekend, whoop.
Sunday is a weekend, whoop.
```

Notice that when a dictionary is iterated over in a `for` loop, the loop variable will get each of the keys of the dictionary. Also, the order of the keys is not guaranteed to match the order in which they were input. In the above loop, the keys were not printed out in the order Monday through Sunday.

We can use the idea of a dictionary of dictionaries idea to make a code that can compute radioactive decay for us, automatically. To do this I will create a dictionary where the key is the atomic number (Z), and the value will be a dictionary with the element name and symbol. The file that is read below is of the format `Z, Symbol, Name`.

```
In [4]: import csv
        element_dict = {} #create a blank dictionary
        #this block will only execute if the file opens
        with open('ChemicalSymbolsZ.csv') as csvfile:
            chemreader = csv.reader(csvfile)
            #have for loop that loops over each row
            for row in chemreader:
                #add a key:value pair
                element_dict[row[0]]={"symbol":row[1],"name":row[2]}
        key = input("Enter a valid atomic number: ")
        if key in element_dict:
            print(key,"is",element_dict[key]["symbol"],
                  ":",element_dict[key]["name"])
        else:
            print("Not a valid element")

Enter a valid atomic number: 34
34 is Se : Selenium

In [5]: key = input("Enter a valid atomic number: ")
        if key in element_dict:
            print(key,"is",element_dict[key]["symbol"],
                  ":",element_dict[key]["name"])
        else:
            print("Not a valid element")

Enter a valid atomic number: 104
104 is Rf : Rutherfordium
```

Given that we have a dictionary where we can look up an element by its atomic number, we can write a function that computes the product of alpha decay of a particular nuclide. We will pass the function the atomic number, the mass number, and the dictionary of elements, and it will return the atomic number and the mass number of the product, along with printing some information to the screen.

```
In [6]: def alpha_decay(Z,A,elements):
             """Alpha decay a nuclide

             Args:
                 Z: atomic number of nuclide
                 A: mass number of nuclide
                 elements: dictionary of elements
```

```
            Returns:
                Z and A of daughter nuclide (both ints)
            Side effects:
                Prints a descriptive string of the decay
            """
            newZ = int(Z) - 2 #lose two protons in alpha decay
            newA = int(A) - 4 #lose four nucleons in alpha decay
            print(elements[str(Z)]["name"],"-",A,"(",
                  elements[str(Z)]["symbol"],"-",A,"), alpha decays to",
                  elements[str(newZ)]["name"],"-",newA,"(",
                  elements[str(newZ)]["symbol"],"-",newA,")")
            return newZ,newA
        z_value = input("Enter the Z of the nuclide: ")
        a_value = input("Enter the mass number of the nuclide: ")
        Z,A = alpha_decay(z_value, a_value, element_dict)

Enter the Z of the nuclide: 94
Enter the mass number of the nuclide: 239
Plutonium - 239 ( Pu - 239 ), alpha decays to Uranium - 235 ( U - 235 )
```

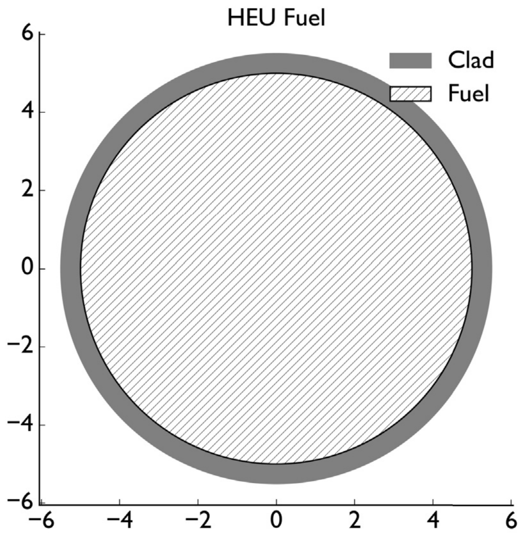
Given that the function returns the atomic and mass numbers of the products, we can run `alpha_decay` in a loop.

```
In [7]: #alpha decay something 10 times
        Z = 94
        A = 239
        for decays in range(10):
            Z,A = alpha_decay(Z, A, element_dict)

Plutonium - 239 ( Pu - 239 ), alpha decays to Uranium - 235 ( U - 235 )
Uranium - 235 ( U - 235 ), alpha decays to Thorium - 231 ( Th - 231 )
Thorium - 231 ( Th - 231 ), alpha decays to Radium - 227 ( Ra - 227 )
Radium - 227 ( Ra - 227 ), alpha decays to Radon - 223 ( Rn - 223 )
Radon - 223 ( Rn - 223 ), alpha decays to Polonium - 219 ( Po - 219 )
Polonium - 219 ( Po - 219 ), alpha decays to Lead - 215 ( Pb - 215 )
Lead - 215 ( Pb - 215 ), alpha decays to Mercury - 211 ( Hg - 211 )
Mercury - 211 ( Hg - 211 ), alpha decays to Platinum - 207 ( Pt - 207 )
Platinum - 207 ( Pt - 207 ), alpha decays to Osmium - 203 ( Os - 203 )
Osmium - 203 ( Os - 203 ), alpha decays to Tungsten - 199 ( W - 199 )
```

This example does not check if such an alpha decay is possible or likely, however, with an appropriate modification to the dictionary we could add information about the decay mode for a particular nuclide. This dictionary would be much more complicated, because it might require a three-level hierarchy consisting of a top level dictionary where the keys are the atomic number and the values are a dictionary of dictionaries where the key is the mass number and the values are the decay modes. Setting this up and filling it with data would be messy, but in principle doable.

Another use of dictionaries would be to store information about the different fuel elements in a reactor. We will consider a reactor that has two types of fuel, high-enriched uranium (HEU) and low-enriched uranium (LEU). We will use a dictionary to describe the properties of each type of fuel. Then using this information, we plot the HEU fuel geometric cross-section.

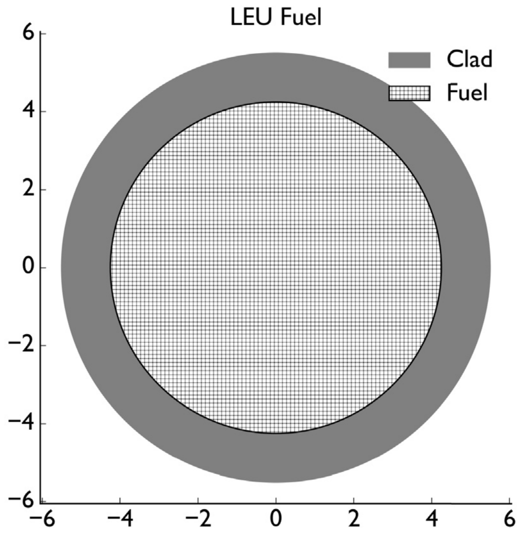
```
In [8]:  fuel_types = {}
         fuel_types["heu"] = {"fuel":{"nu sigma_f":12.0,
                                     "D":3.0, "thickness":5.0},
                             "clad":{"nu sigma_f":0.0, "
                                     D":300.0, "thickness":0.5}}
         fuel_types["leu"] = {"fuel":{"nu sigma_f":8.5,
                                     "D":1.25, "thickness":4.25},
                             "clad":{"nu sigma_f":0.0,
                                     "D":300.0, "thickness":1.25}}
         #plot heu
         #heu fuel
         fuel_radius = fuel_types["heu"]["fuel"]["thickness"]
         clad_radius = fuel_radius + fuel_types["heu"]["clad"]["thickness"]
         fuel = plt.Circle((0,0),fuel_radius,
                           facecolor="white",label="Fuel", hatch="//")
         clad = plt.Circle((0,0),clad_radius,color='gray',label="Clad")
         fig = plt.figure(figsize=(8,6), dpi=600)
         plt.gca().add_patch(clad)
         plt.gca().add_patch(fuel)
         plt.title("HEU Fuel")
         plt.axis('equal')
         plt.legend()
         plt.axis([-clad_radius,clad_radius,-clad_radius,clad_radius])
         plt.show();
```



In a similar manner, the LEU fuel can be visualized by using the dictionary.

```
In [9]:  fuel_radius = fuel_types["leu"]["fuel"]["thickness"]
         clad_radius = fuel_radius + fuel_types["leu"]["clad"]["thickness"]
         fig = plt.figure(figsize=(8,6), dpi=600)
         fuel = plt.Circle((0,0),fuel_radius,
                           facecolor="white",label="Fuel", hatch="+")
```

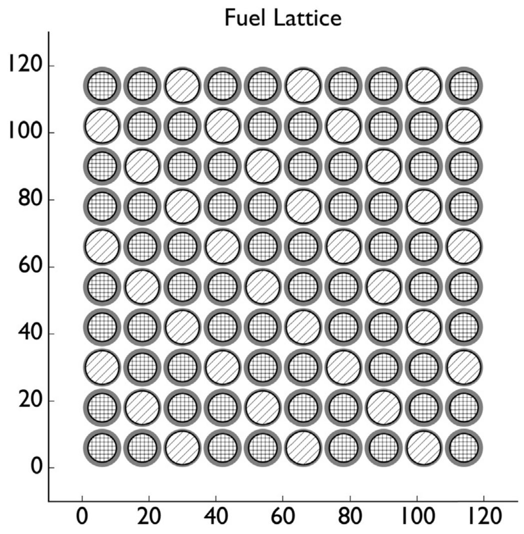
```
        clad = plt.Circle((0,0),clad_radius,color='gray',label="Clad")
        plt.gca().add_patch(clad)
        plt.gca().add_patch(fuel)
        plt.title("LEU Fuel")
        plt.axis('equal')
        plt.legend()
        plt.axis([-clad_radius,clad_radius,-clad_radius,clad_radius])
        plt.show();
```



Given that we have a dictionary describing each type of fuel, we can define a lattice of fuel elements. We will make a 10 by 10 lattice of fuel with a 0.5 cm spacing between elements. Also, we will make every third element HEU and the rest LEU. The code below creates this lattice and then plots it.

```
In [10]: fuel_placements = {}
         #10 x 10 lattice with 0.5 cm spacing
         #every third pin is heu
         x = np.arange(6.,120,12)
         y = np.arange(6.,120,12)
         fig = plt.figure(figsize=(8,6), dpi=600)
         count = 1 #set up counting variable
         for i in x:
             for j in y:
                 if not(count % 3): #if count mod 3 is 0, then heu
                     pin_type = "heu"
                     hatch = "/"
                 else: #else leu
                     pin_type = "leu"
                     hatch = "+"
                 fuel_radius = fuel_types[pin_type]["fuel"]["thickness"]
                 clad_radius = fuel_radius +
                               fuel_types[pin_type]["clad"]["thickness"]
```

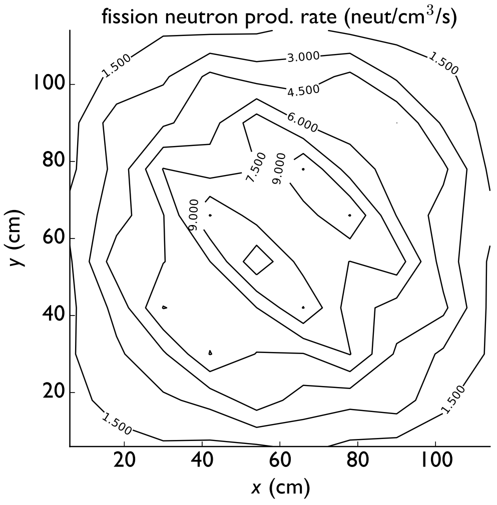
```
            fuel = plt.Circle((i,j),fuel_radius,facecolor="white",
                              edgecolor="black",hatch=hatch,
                              label="Fuel")
            clad = plt.Circle((i,j),clad_radius,color='gray',label="Clad")
            plt.gca().add_patch(clad)
            plt.gca().add_patch(fuel)
            count += 1 #increment count
        plt.title("Fuel Lattice")
        plt.axis('equal')
        plt.axis([0,120,0,120])
        plt.show();
```

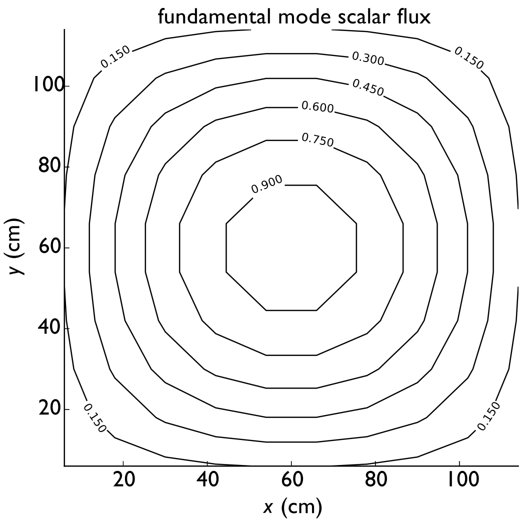


With this lattice, if we knew the shape of the fundamental mode of the scalar flux, we could compute the fission neutron production rate density at each point of the reactor. We will assume a simple scalar flux shape and multiply the flux by the value of $\nu\Sigma_f$, that is the product of the average number of fission neutrons produced, times the macroscopic fission cross-section, to get the fission neutron production rate density. The following code computes this quantity by evaluating $\nu\Sigma_f$ at the middle of each fuel element.

```
In [11]: fuel_placements = {}
         #10 x 10 lattice with 0.5 cm spacing
         #every third pin is heu
         x = np.arange(6.,120,12)
         y = np.arange(6.,120,12)
         fig = plt.figure(figsize=(8,6), dpi=600)
         X =  np.zeros((x.size, x.size))
         Y = X.copy()
         Z = X.copy()
         Zflux = Z.copy()
         row = 0
         col = 0
         count = 1
```

```
for i in x:
    for j in y:
        if not(count % 3): #if count mod 3 is 0, then heu
            pin_type = "heu"
        else: #else leu
            pin_type = "leu"
        nusigf = fuel_types[pin_type]["fuel"]["nu sigma_f"]
        X[row,col] = i
        Y[row,col] = j
        Z[row,col]=nusigf*np.sin(i*np.pi/120)*np.sin(j*np.pi/120)
        Zflux[row,col]=np.sin(i*np.pi/120)*np.sin(j*np.pi/120)
        row += 1 #increment row
        count += 1 #increment count
    col += 1 #increment column
    row = 0

CS = plt.contour(X,Y,Z, colors='k')
plt.clabel(CS, fontsize=9, inline=1)
plt.xlabel("x (cm)");
plt.ylabel("y (cm)");
plt.title("fission neutron production rate (neutrons/cm$^3$/s)");
plt.show();
CS = plt.contour(X,Y,Zflux, colors='k')
plt.clabel(CS, fontsize=9, inline=1)
plt.xlabel("x (cm)");
plt.ylabel("y (cm)");
plt.title("fundamental mode scalar flux");
plt.show();
```





When we solve the diffusion equation for the scalar flux of neutrons in a system, we will revisit these techniques for calculating the fission neutron production rate, and other quantities.

5.2 FUNCTIONS PASSED TO FUNCTIONS

In Python it is possible to pass the name of an existing function as a parameter to another function. This name can then be used to execute the function and manipulate the results. This ability to call a generic function results in very powerful codes as we will see.

BOX 5.2 PYTHON PRINCIPLE

In Python a function can take the name of an arbitrary function as a parameter. In the generic example

```
def new_function(f,x):
      return f(x)
```

The function `new_function` returns the value of function `f` called as `f(x)`. In this example, the function name that is passed to `new_function` must be a single parameter and `x` must be of the correct type.

A salient example of this a numerical integration function. If we wanted to write a function to apply an integration formula, we would not want to have to write a new integration routine for each integrand. Instead, we can make the function that forms the integrand a parameter, and then call that function every time we want to evaluate the integrand. The example below uses the midpoint rule to integrate a generic integrand, `f(x)` between points `a` and `b`.

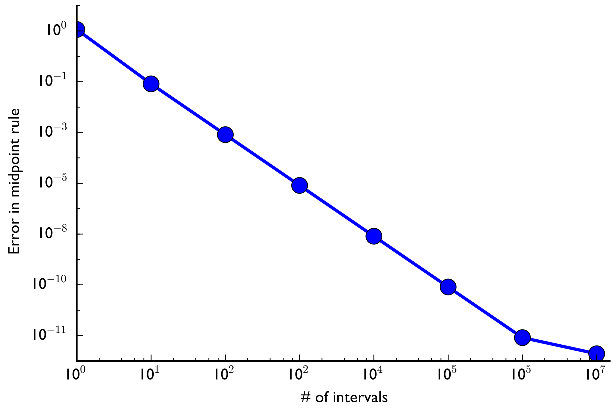
```
In [12]: def midpoint_rule(f,a,b,num_intervals):
             """integrate function f using the midpoint rule

             Args:
                 f: function to be integrated, it must take 1 argument
                 a: lower bound of integral range
                 b: upper bound of integral range
                 num_intervals: the number of intervals in [a,b]
             Returns:
                 estimate of the integral
             """
             L = (b-a) #how big is the range
             dx = L/num_intervals #how big is each interval
             #midpoints are a+dx/2, a+3dx/2, ..., b-dx/2
             midpoints = np.arange(num_intervals)*dx+0.5*dx+a
             integral = 0
             for point in midpoints:
                 integral = integral + f(point)
             return integral*dx
```

To integrate $\sin x$ from 0 to π we pass in the name of NumPy sine function, `np.sin`. The exact value of this integral is 2. With 10 intervals, we get a pretty good answer.

```
In [13]: print(midpoint_rule(np.sin,0,np.pi,10))

2.00824840791
```

We can see how the numerical integration technique converges to the exact answer as a function of the number of intervals by calling the midpoint rule function with several different values of `num_intervals`. In the next code snippet we do this using a `for` loop to compute the integral using 10^i intervals for $i = 0, 1, 2, \ldots, 8$. Then we plot the error as a function of the number of intervals on a log-log scale. Later, when we study numerical integration in more detail this type of plot will be important.

```
In [14]: num_intervals = 8 #number of interval sizes
         #run several different intervals
         intervals = 10**np.arange(num_intervals)
         ierror = np.zeros(num_intervals)
         fig = plt.figure(figsize=(8,6), dpi=600)
         count = 0
         a = 0
         b = np.pi
         for interval in intervals:
             error[count] = np.fabs(midpoint_rule(np.sin,a,b,interval)-2)
             count += 1
         plt.loglog(intervals,error,marker="o",
                    markersize = 10,linewidth=2);
         plt.xlabel("# of intervals")
         plt.ylabel("Error in midpoint rule")
         plt.show()
```



We are not limited to integrating only the sine function. We can define our own functions and pass them to the midpoint rule function we defined. In other words, the midpoint rule can approximate any 1-D definite integral. For example, we can use our midpoint rule function to compute an estimate of the exponential integral function,

$$E_n(x) = \int_1^\infty dt\, \frac{e^{-xt}}{t^n}.$$

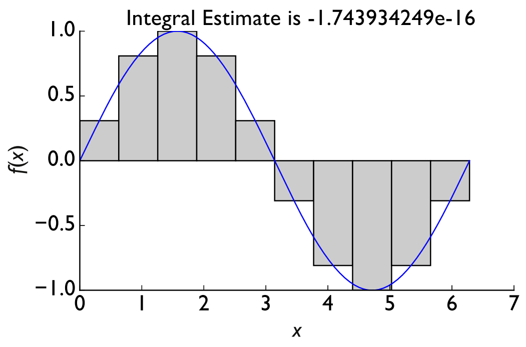
Because this is an improper integral, we have to introduce a finite upper bound, and this is another approximation.

```
In [15]: def exp_int_argument(t,n=1,x=1):
             return np.exp(-x*t)/t**n
         num_points = 10**6
         upper_bound = 1000
         print("Exact answer is 0.2193839343")
         print("Our approximation with upper bound",upper_bound,
               "and",num_points,
               "points is",
               midpoint_rule(exp_int_argument,1,upper_bound,num_points))

Exact answer is 0.2193839343
Our approximation with upper bound 1000 and 1000000 points is 0.2193839038
```

Using Matplotlib we can add the functionality to the integration function to graphically show how the midpoint rule estimates the integral. The function below draws the areas that comprise the integral estimate. It should give the same answer as the previous midpoint rule function, but with pretty graphics. Notice in the docstring for the function, the fact that the function produces a plot is listed as a side effect. A side effect is something that the function does other than return a value. In this case it makes a plot, but a side effect could be printing something to the screen, writing to file, or modifying a NumPy array that was passed to the function.

```
In [16]: def midpoint_rule_graphical(f,a,b,num_intervals):
             """integrate function f using the midpoint rule

             Args:
                 f: function to be integrated, it must take one argument
                 a: lower bound of integral range
                 b: upper bound of integral range
                 num_intervals: the number of intervals to break [a,b] into
             Returns:
                 estimate of the integral
             Side Effect:
                 Plots intervals and areas of midpoint rule
             """
             fig = plt.figure()
             ax = plt.subplot(111)
             L = (b-a) #how big is the range
             dx = L/num_intervals #how big is each interval
             midpoints = np.arange(num_intervals)*dx+0.5*dx+a
             x = midpoints
             y = np.zeros(num_intervals)
             integral = 0
             count = 0
             for point in midpoints:
                 y[count] = f(point)
                 integral = integral + f(point)
                 verts = [(point-dx/2,0)] + [(point-dx/2,f(point))]
                 verts += [(point+dx/2,f(point))] + [(point+dx/2,0)]
                 poly = plt.Polygon(verts, facecolor='0.8', edgecolor='k')
                 ax.add_patch(poly)
                 count += 1
```

```
        y = f(x)
        smooth_x = np.linspace(a,b,10000)
        smooth_y = f(smooth_x)
        plt.plot(smooth_x, smooth_y, linewidth=1)
        plt.xlabel("x")
        plt.ylabel("f(x)")
        plt.title("Integral Estimate is " + str(integral*dx))
        plt.show()
        return integral*dx
    midpoint_rule_graphical(np.sin,0,2*np.pi,10)
```



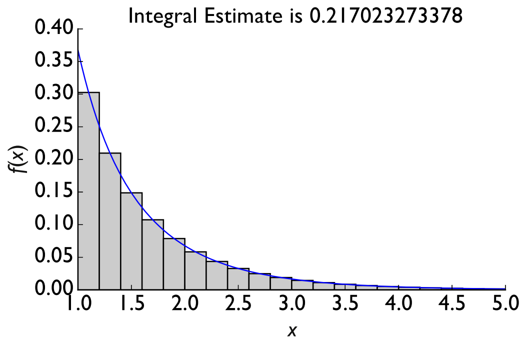
BOX 5.3 PYTHON PRINCIPLE

A side effect of a function is something that the function does that affects or interacts with the function caller in some way. This can be printing to the screen, making a graph, or changing something in a mutable variable (e.g., a NumPy vector) passed to the function. It is important to document side effects so that the code calling the function can handle them.

We can use the same function to compute the exponential integral and visualize how well the approximation of the midpoint rule is doing.

```
In [17]: num_points = 20
         upper_bound = 5
         print("Answer is 0.2193839343")
         print("Our approximation with upper bound",upper_bound,
              "and",num_points,"points is",
               midpoint_rule_graphical(exp_int_argument,1,upper_bound,num_points))
```

```
Answer is 0.2193839343
```



```
Our approximation with upper bound 5 and 20 points is 0.217023273378
```

It appears that the area is better approximated near $x = 5$ than near $x = 1$. Later, we will see other approaches to estimating integrals that give better approximations than rectangles.

5.3 LAMBDA FUNCTIONS

Python also allows you to define simple, one line functions called lambda functions. One of the benefits of a lambda function is that they are very easy to define. Because of this, they are especially good if we want to combine previously defined functions in a simple manner.

Lambda functions also have different scope rules than standard functions. Lambda functions give you access to all the variables available in the scope they are defined. This means that lambda functions do not have their own variable scope.

BOX 5.4 PYTHON PRINCIPLE

Lambda functions are short functions that you define in a single line. The syntax to define a lambda function named `lambda_func` is

```
lambda_func = lambda [parameter list]:
              expression
```

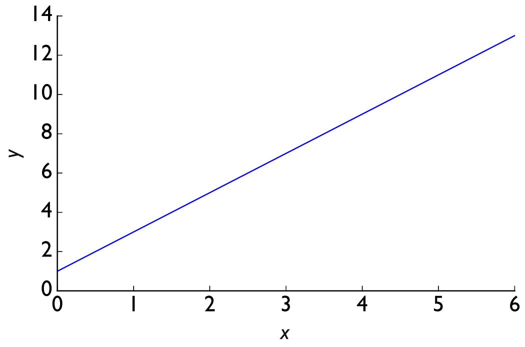
where `[parameter list]` is a comma-separated list of the input parameters to the function and the result of evaluating `expression` is what the function returns.

Lambda functions have the same scope as the scope in which they are defined. For example, if you define a lambda function inside of a function, it has the same scope as that function.

The following example uses a lambda function to define a line.

```
In [17]: simple_line = lambda x: 2.0*x + 1.0
         print("The line at x = 0 is", simple_line(0))
         print("The line at x = 1 is", simple_line(1))
         print("The line at x = 2 is", simple_line(2))
```

```
        x = np.linspace(0,6,50)
        y = simple_line(x)
        plt.plot(x,y)
        plt.ylabel("y")
        plt.xlabel("x")
        plt.show()

The line at x = 0 is 1.0
The line at x = 1 is 3.0
The line at x = 2 is 5.0
```

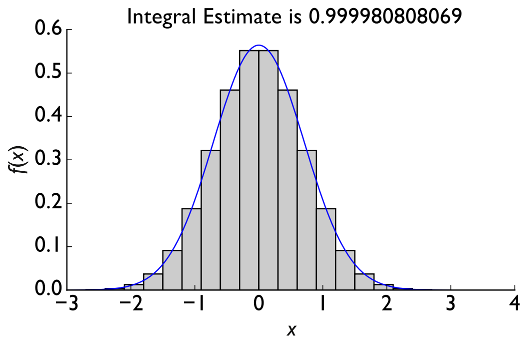


We can use lambda functions in our midpoint integration routine as well. Here we define the probability density function of a Gaussian as the integrand. A unit variance and zero mean Gaussian is given by

$$f(x) = \frac{e^{-x^2}}{\sqrt{\pi}}.$$

The integral over $x \in [-\infty, \infty]$ should be 1. Lambda functions are very useful in this context because the exponential is already defined; we just want to integrate it with a particular form of argument and multiply it by a constant.

```
In [18]: #function to compute gaussian
        gaussian = lambda x: np.exp(-x**2)/np.sqrt(np.pi)
        midpoint_rule_graphical(gaussian,-3,3,20)
```



We can use the fact that lambda functions have the same scope as where they are defined to make our midpoint rule integrate two-dimensional functions by defining two lambda functions. In effect, what this code does is treat a 2-D integral as a 1-D integral:

$$\int_{a_y}^{b_y} dy \int_{a_x}^{b_x} dx\, f(x,y) = \int_{a_y}^{b_y} dy\, g(y),$$

where

$$g(y) = \int_{a_x}^{b_x} dx\, f(x,y).$$

This is possible to define because we can define the $g(y)$ using lambda functions. For a test we will estimate the integral

$$\int_0^{\pi} dy \int_0^{\pi} dx\, \sin(x)\sin(y) = 4.$$

```
In [19]: def midpoint_2D(f,ax,bx,ay,by,num_intervals_x,num_intervals_y):
             """integrate function f(x,y) using the midpoint rule
                 Args:
                     f: function to be integrated, it must take 2 arguments
                     ax: lower bound of integral range in x
                     bx: upper bound of integral range in x
                     ay: lower bound of integral range in y
                     by: upper bound of integral range in y
                     num_intervals_x: the number of intervals in x
                     num_intervals_y: the number of intervals in y
                 Returns:
                     estimate of the integral
                 """
                 g = lambda y: midpoint_rule(lambda x: f(x,y),ax,bx,num_intervals_x)
                 return midpoint_rule(g,ay,by,num_intervals_y)
             sin2 = lambda x,y:np.sin(x)*np.sin(y)
             print("Estimate of the integral of sin(x)sin(y), over [0,pi] x [0,pi] is",
                   midpoint_2D(sin2,0,np.pi,0,np.pi,1000,1000))
```

```
Estimate of the integral of sin(x)sin(y), over [0,pi] x [0,pi] is 4.00000328987
```

The lambda functions in this example tell Python to treat only a single variable as the function parameter, and evaluate everything else based on the current scope. This means that the code `lambda x: f(x,y)` evaluates `f(x,y)` using whatever the current value of `y` is and the parameter `x` that the function was passed. The value of `y` is supplied by a parameter to the lambda function `g`.

The intricacies of defining lambda functions inside other lambda functions can get a bit abstract, but remembering that they treat all variables in an expression that is not specified in the parameter list as “known”, is the key to understanding how they function.

PROBLEMS

Short Exercises

5.1. Write a Python dictionary that contains the key:value pairs that have for a key the name of the common subatomic particles (i.e., proton, neutron, and electron) and the value the mass of the particle in kilograms.

5.2. Using the `midpoint_rule` function defined above, compute the integral of $\sin^2 x$, over the range $[0, 2\pi]$ with 10, 100, and 1000 intervals.

5.3. Estimate π to five digits of accuracy by computing the integral of $f(x) = 4\sqrt{1-x^2}$ for $x \in [0, 1]$.

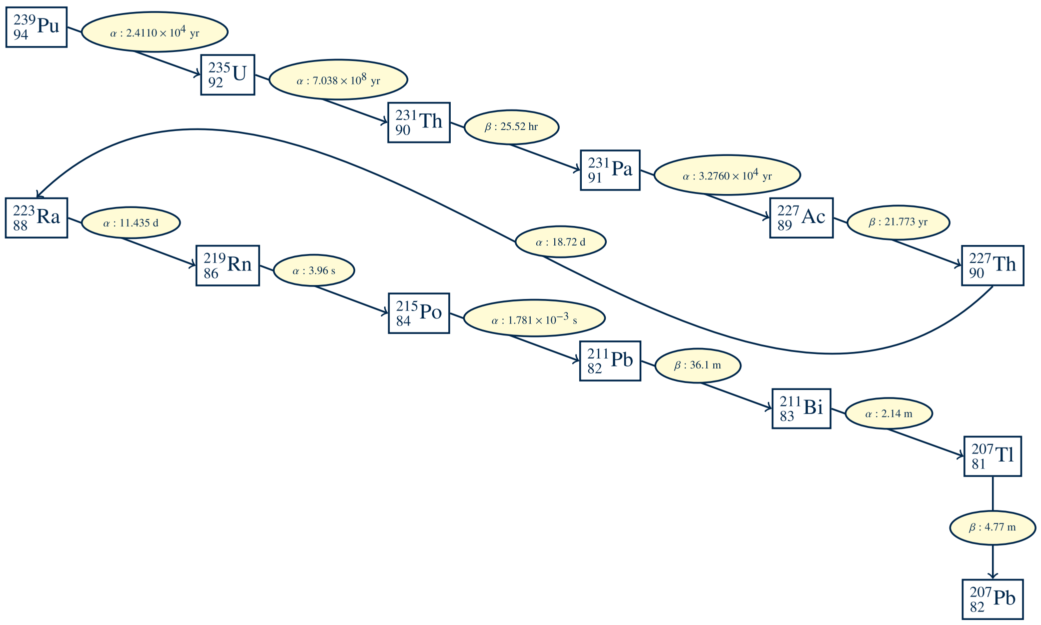
5.4. Integrate the function $f(x, y, z) = \exp(-z^2)\sin(x)\sin(y)$ over the (x, y, z) range $[0, \pi] \times [0, \pi] \times [-4, 4]$ using 10, 100, and 1000 intervals.

Programming Projects

1. Plutonium Decay Chain

Consider the plutonium decay chain from ^{239}Pu to stable ^{207}Pb, as shown below. Construct a dictionary with the keys are `A-X` where `A` is the mass number of the nuclide and `X` is the atomic symbol for the nuclide. For example one key is `239-Pu`. The value for the aforementioned keys should be a dictionary with `key:value` pairs given by:

- key: `half-life`, value: the half-life of the decay in seconds,
- key: `decay_mode`, value: the decay mode (i.e., alpha, beta, or stable in this case), and
- key: `mass`, value: the mass of the nuclide.



Your code should use the dictionary to print out

- All the nuclides that decay by alpha decay
- The activity of 1 gram of each nuclide that is a beta emitter.

2. Simple Cryptographic Cipher

To transmit a message you desire to encrypt it. In the terminology of cryptography the original message is the plain text and the encrypted message is called the cipher text. The means of encrypting the message is called a cipher. A simple method is the ROT-13 cipher, which is an example of the Caesar cipher. In this cipher the letter is replaced by a letter 13 places away in the alphabet. This can be encoded easily in a dictionary:

```
cipherDict = {a:"n", b:"o",...,m:"z",n:"a",...}
```

Write a function called `cipher` that takes in a string and returns an encrypted cipher text using the ROT-13 cipher, your function must also take in cipher text and return plain text. Your code must have the following behavior:

- Handle lower case and capital letters,
- Not do anything to characters that are not alphabet characters, e.g., numbers, punctuation, and other characters, and
- Only take a single parameter as an argument to the function. This argument will be a string containing the plain text or cipher text.

Test your code on the following cipher text “Gur bayl rzcrebe vf gur rzcrebe bs vpr-pernz.” Show that it can give the correct plain text, and that it can recover the cipher text by applying the function to the plain text.