CHAPTER

2

Digging Deeper Into Python

OUTLINE

The secret, I don't know... I guess you've just gotta find something you love to do and then... do it for the rest of your life. For me, it's going to Rushmore.

–*"Max Fischer" in the movie* Rushmore

CHAPTER POINTS

- `For` loops make performing a fixed number of iterations simple.
- Python has data structures that can contain various groups of items and manipulate them efficiently.
- Floating point numbers are not exact, and this can lead to unexpected behavior.

2.1 A FIRST NUMERICAL PROGRAM

Thus far we have talked about the basics of Python and some of the rudimentary building blocks of a program. We have used these to make toy codes that did not do much useful. Now we will make things a bit more concrete and show how we can use Python to perform numerical calculations. We will start out very basic and expand our repertoire as we go.

DOI: 10.1016/B978-0-12-812253-2.00003-0

To begin, we will consider a common calculation in many applications (movie and computer graphics for computing the reflection angles for lighting and shading, for example):

$$y = \frac{1}{\sqrt{x}}.$$

Assuming that we cannot just use the built-in square root function, we could find the value of y such that

$$y\sqrt{x} - 1 = 0.$$

One way we could solve this equation is to start with a guess that we know is either high (or low), and then decrement (or increment) the guess until it is accurate enough. This is not a particularly good way of solving this problem, but it is easy to understand. To tell when we are close enough to an answer, we will evaluate the residual. In simple terms the residual for a guess y_i is given by

$$\text{residual} = y_i\sqrt{x} - 1.$$

When the guess is equal to the solution y, the residual will be zero. Similarly, when the residual is small, we are close to the answer.

For our problem we know that

$$x > \frac{1}{\sqrt{x}}$$

as long as x is greater than 1. So we could start at x and decrease the guess until we get the answer we desire. This is an example of exhaustive enumeration. It is called enumeration because we list (i.e., enumerate) possible solutions to the problem until we get one close enough. It is exhaustive because we list "all" the possible solutions up to some precision. As we will see, it is also exhaustive because it is a lot of work to solve a simple problem.

Here is Python code to solve this problem. The user inputs an x and the code computes $1/\sqrt{x}$.

```
In [1]: #this code computes 1/sqrt(x), for x > 1
        import math
        x = float(input("Enter a number greater than 1: "))
        if (x<=1):
            print("I said a number greater than 1")
        else:
            converged = 0
            answer = x #initial guess is x
            eps = 1.0e-6 #the residual tolerance
            converged = math.fabs(answer * math.sqrt(x) - 1.0) < eps
            iteration = 0
            while not(converged):
                answer = answer - 0.5*eps
                converged = math.fabs(answer * math.sqrt(x) - 1.0) < eps
                iteration += 1
            print("1/sqrt(",x,") =",answer)
            print("It took",iteration,"guesses to get that answer.")
```

```
Enter a number greater than 1: 3
1/sqrt( 3.0 ) = 0.5773504997552141
It took 4845299 guesses to get that answer.
```

Before discussing how the algorithm performs, it is worthwhile to discuss how it works. Notice that the code is using an if-else statement to check that the input provided by the user is greater than one. Users can enter any number of wacky things, so it is always a good idea to check user-specified input.

After assuring that the value of x is greater than or equal to one, the code starts with a guess at the answer of x. It then checks if the absolute value of the residual is smaller than a specified tolerance. If the absolute value of the residual is larger than the tolerance, then it enters the while loop and decreases the value of the answer by one-half times the tolerance until the residual is small enough in magnitude.

This is a really slow algorithm: it can easily take millions of guesses. Though it is easy to deride this simplistic algorithm, starting with a very basic algorithm that is slow, but that we know will work, is a good idea. In other words, having a slow, working algorithm is better than nothing. To paraphrase, an algorithm in code is worth two in your head.

If we can bracket the answer, that is say it is between two numbers, we can improve the algorithm by bisecting (dividing in two) the interval and zooming in on the answer. To start, we can bracket the solution to our equation because we know that the answer is between $1/x$ and x for $x > 1$. The following code starts out by defining this interval and checking on which half of the interval the solution is in.

```
In [2]: #this code computes 1/sqrt(x), for x > 1
        import math
        x = float(input("Enter a number greater than 1: "))
        if (x<=1):
            print("I said a number greater than 1")
        else:
            converged = 0
            upper_bound = x
            lower_bound = 1.0/x
            answer = (upper_bound + lower_bound)*0.5
            eps = 1.0e-6
            converged = math.fabs(answer * math.sqrt(x) - 1.0) < eps
            iteration = 0
            while not(converged):
                mid = answer
                if (mid < 1.0/math.sqrt(x)):
                    lower_bound = mid
                else:
                    upper_bound = mid
                answer = (upper_bound + lower_bound)*0.5
                #print(upper_bound,lower_bound)
                converged = (math.fabs(answer * math.sqrt(x) - 1.0)
                             < eps)
                iteration += 1
            print("1/sqrt(",x,") =",answer)
            print("It took",iteration,"guesses to get that answer.")
```

```
Enter a number greater than 1: 3
1/sqrt( 3.0 ) = 0.5773506164550781
It took 20 guesses to get that answer.
```

This method is called the bisection method, and we will take a closer look at it in the future. The way it works is it takes a range that brackets the root of an equation, and then looks at the midpoint of that range. Based on the value of the function at the midpoint, you then know if the root is in the lower half or upper half of the range. We will explain this in more detail in a future chapter when we talk about solving nonlinear equations.

Notice that it took a lot fewer guesses using bisection compared to exhaustive enumeration. Exhaustive enumeration is not very good for problems where we are trying to find a continuous variable.

2.2 FOR LOOPS

While loops are great, and they can do everything we need for iteration. Nevertheless, there are instances when we need to iterate a fixed number of times, it is useful to have a shorthand for this type of iteration structure. For instance, if we wanted to execute a block of code a set number of times, we have to define a counting variable and increment it by hand:

```
In [3]: #Some code that counts to ten
        count = 1
        while (count <= 10):
            print(count)
            count += 1

1
2
3
4
5
6
7
8
9
10
```

The `for` loop is built for such a situation. The way we typically use it is with the `range` function. This function takes 3 input parameters: `range(start, stop, [step])`. The `stop` parameter to the `range` function tells `range` to go to the number *before* `stop`. The parameter `step` is in brackets because it is optional. If you do not define it, Python assumes you want to count by 1 (i.e., step by 1). What `range` returns is a sequence that starts at `start` and counts up to `stop - 1` by `step`. The next example demonstrates this:

```
In [4]: print(list(range(1,10)))
        #the list command tells Python to write out the range

[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Also, if you just give range one parameter, it treats that as stop and assumes you want to start at 0:

```
In [5]: #These should be the same
        print(list(range(0,10)))
        print(list(range(10)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [6]: #Here's something using the step parameter
        print(list(range(0,10,2)))
```

```
[0, 2, 4, 6, 8]
```

With the range command we can have a for loop assign a variable a value in the range, in order, each time the code block of the for loop executes:

```
In [7]: for i in range(10):
            print(i+1)
```

```
1
2
3
4
5
6
7
8
9
10
```

BOX 2.1 PYTHON PRINCIPLE

The `range` function

```
range(start, stop, [step])
```

creates a list of integers that begins at `start`, increments by `step`, and stops before `stop`. The `step` parameter is assumed to be one if it not included. The `range` function can be called with one parameter:

```
range(stop)
```

This is equivalent to the three-parameter version with `start` equal to 0, and `step` equal to 1, that is, `range(0,stop,1)`.

Suppose we want to add a number to itself seven times. To do this we could use a `for` loop:

```
In [8]:  number = 10
         sum = 0
         for i in range(7):
             sum += number
         print(sum)
```

```
70
```

We could also do this using a `while` loop, but it takes two extra lines: one to initialize a variable, and another to increment it.

```
In [9]: #while loop version
        number = 10
        sum = 0
        i = 0
        while (i<7):
            sum += number
            i += 1
        print(sum)
```

```
70
```

BOX 2.2 PYTHON PRINCIPLE

The `for` loop is written in Python as

```
for i in X:
        [some code]
```

The code in the block `[some code]` will execute once for each item in the object `X` and each time through the code block `i` will take on the value of an item in `X`, in order.

Here is, perhaps, a more practical use of a for loop: to compute π. We do this using random numbers picked between -1 and 1 using the `random` library that comes with Python. In that library there is a function called uniform that gives a uniformly distributed random number between two endpoints.

```
In [10]: "'Compute pi by picking random points between x = -1 and 1,
         y = -1 and 1. The fraction of points
         such that x^2 + y^2 < 1, compared with the total number
         of points is an approximation to pi/4"'
         import random
         number_of_points = 10**5
         number_inside_circle = 0
         random.seed() #this seeds the random number generator
         for point in range(number_of_points):
             x = random.uniform(-1,1) #pick random number between -1 and 1
             y = random.uniform(-1,1) #pick random number between -1 and 1
             if x**2 + y**2 < 1: #is the point in the circle
                 number_inside_circle += 1
         pi_approx = 4.0*number_inside_circle/number_of_points
         print("With",number_of_points,
               "points our approximation to pi is",pi_approx)
```

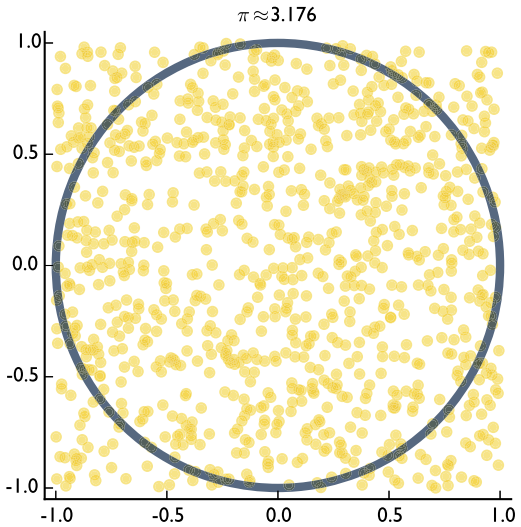
```
With 100000 points our approximation to pi is 3.1406
```

This works because the ratio of the number of points inside the circle to the total number of points will converge to the ratio of the area of the circle (π) to the total area of the square (4). In particular, `number_inside_circle/number_of_points` will converge to $\pi/4$ as the number of points chosen goes to infinity. The code above is our first example of a Monte Carlo method where we use random numbers to compute fixed quantities, and we will return to these methods in the last part of this text.

The random numbers generated by Python are actually items in a really long list of numbers that seem random (such random numbers are called pseudorandom numbers). In the code above we set where we start in the list using `random.seed()`, which then uses the system time to pick a starting point, so that each time the code is run, it starts somewhere different.

Using NumPy, which we have not covered yet, we can do this in an even fancier way. If we use Matplotlib, another Python library, we can get nice graphs as well. The code below picks 1000 random points to estimate π and plots the points on a graph in a particular color, and draws the circle in another color (the print version of the book will have the points in gray and the circle in black).

```
In [11]: import numpy as np
         import matplotlib.pyplot as plt
         #pick our points
         number_of_points = 10**3
         x = np.random.uniform(-1,1,number_of_points)
         y = np.random.uniform(-1,1,number_of_points)
         #compute pi
         pi_approx = 4.0*np.sum(x**2 + y**2 <= 1)/number_of_points
         #now make a scatter plot
         maize = "#ffcb05"
         blue = "#00274c"
         fig = plt.figure(figsize=(8,6), dpi=600)
         #scatter plot with hex color
         plt.scatter(x, y, alpha=0.5, color=maize)
         #draw a circle of radius 1 with center (0,0)
         circle = plt.Circle((0,0),1,color=blue, alpha=0.7,
                             fill=False, linewidth=4)
         #add the circle to the plot
         plt.gca().add_patch(circle)
         #make sure that the axes are square so that our circle is circular
         plt.axis('equal')
         #set axes bounds: axis([min x, max x, min y, max y])
         plt.axis([-1,1,-1,1])
         #make the title have the approximation to pi
         plt.title("$\\pi \\approx $" + str(pi_approx))
         #show the plot
         plt.show()
```



You can also simply accomplish other tasks `for` loops. For instance, you can have the control variable take on non-numeric things. If we have a list of strings, numbers, or whatever we can loop over all the elements of that sequence. In the case below we have the `for` loop make the control variable contain each string in a sequence.

```
In [13]: #silly hat code
         hats = ["fedora","trilby","porkpie","tam o'shanter",
                 "Phrygian cap","Beefeaters' hat","sombrero"]
         days = ["Monday","Tuesday","Wednesday","Thursday",
                 "Friday","Saturday","Sunday"]
         count = 0
         for today in hats:
             print("It is",days[count],"and I will wear a",today)
             count += 1

It is Monday and I will wear a fedora
It is Tuesday and I will wear a trilby
It is Wednesday and I will wear a porkpie
It is Thursday and I will wear a tam o'shanter
It is Friday and I will wear a Phrygian cap
It is Saturday and I will wear a Beefeaters' hat
It is Sunday and I will wear a sombrero
```

Notice what this code did: we defined a list called `days` that contained strings for the names of the days of the week. Inside our for loop we had a numeric variable that kept track of what number the day of the week was (0 for Monday in this case). Then when we access `days[count]` it returns the string in position count.

We can go one step beyond and plan our haberdashery decisions a month in advance using random numbers.

```
In [14]: #silly hat code
         import random
```

```
        hats = ["fedora","trilby","porkpie","tam o'shanter",
                "Phrygian cap","Beefeaters' hat","sombrero"]
        days = ["Monday","Tuesday","Wednesday","Thursday",
                "Friday","Saturday","Sunday"]
        for count in range(30):
            hat_choice = round(random.uniform(0,6))
            print("It is",days[count % 7],"and I will wear a",
                  hats[hat_choice])
```

```
It is Monday and I will wear a fedora
It is Tuesday and I will wear a Phrygian cap
It is Wednesday and I will wear a tam o'shanter
It is Thursday and I will wear a Beefeaters' hat
It is Friday and I will wear a Phrygian cap
It is Saturday and I will wear a Beefeaters' hat
It is Sunday and I will wear a Beefeaters' hat
It is Monday and I will wear a Phrygian cap
It is Tuesday and I will wear a Beefeaters' hat
It is Wednesday and I will wear a Phrygian cap
It is Thursday and I will wear a sombrero
It is Friday and I will wear a sombrero
It is Saturday and I will wear a trilby
It is Sunday and I will wear a Beefeaters' hat
It is Monday and I will wear a porkpie
It is Tuesday and I will wear a trilby
It is Wednesday and I will wear a Phrygian cap
It is Thursday and I will wear a sombrero
It is Friday and I will wear a tam o'shanter
It is Saturday and I will wear a porkpie
It is Sunday and I will wear a Phrygian cap
It is Monday and I will wear a fedora
It is Tuesday and I will wear a Beefeaters' hat
It is Wednesday and I will wear a Phrygian cap
It is Thursday and I will wear a Beefeaters' hat
It is Friday and I will wear a Beefeaters' hat
It is Saturday and I will wear a porkpie
It is Sunday and I will wear a Beefeaters' hat
It is Monday and I will wear a porkpie
It is Tuesday and I will wear a tam o'shanter
```

We will use the ability to iterate over a list in future codes to do things like investigate radioactive decay.

2.3 LISTS AND TUPLES

2.3.1 Lists

In the previous section we defined a list of strings as

```
In [15]: hats = ["fedora","trilby","porkpie","tam o'shanter","Phrygian cap",
                 "Beefeaters' hat","sombrero"]
         print(hats)
```

```
['fedora', 'trilby', 'porkpie', "tam o'shanter", 'Phrygian cap',
 "Beefeaters' hat", 'sombrero']
```

A list is a Python object that contains several other objects. A list is defined by enclosing the list members in square brackets. In the case above the objects were strings, but they could have been numbers or other objects. We can access items in a list using square brackets:

```
In [16]: hats[2]

Out[16]: 'porkpie'
```

Slicing using the colon is allowed on lists just as it is in strings, like we did in the previous chapter:

```
In [17]: print(hats[3:7])

["tam o'shanter", 'Phrygian cap', "Beefeaters' hat", 'sombrero']
```

You can also add items to a list using the `append` function. To do this you use the name of the list followed by `.append`:

```
In [18]: hats.append("toque")
         print(hats)

['fedora', 'trilby', 'porkpie', "tam o'shanter", 'Phrygian cap',
 "Beefeaters' hat", 'sombrero', 'toque']
```

You can delete an item from a list using the `remove` function in the same way we used the `append` function:

```
In [19]: hats.remove('trilby')
         print(hats)

['fedora', 'porkpie', "tam o'shanter", 'Phrygian cap',
 "Beefeaters' hat", 'sombrero', 'toque']
```

A feature of lists is that they can contain different types of items. For example you can mix strings and numbers:

```
In [20]: my_list = ["Item 0", "Item 1", 2]
         print(my_list)

['Item 0', 'Item 1', 2]
```

In many cases we would like to know how many elements are in a list. To find the length of a list use the `len` function:

```
In [21]: len(my_list)

Out[21]: 3
```

To see if an value is contained in a list use the `in` operator. This operator will return true if its argument is in the list, and false otherwise:

```
In [22]: "Item 2" in my_list

Out[22]: False

In [23]: 2 in my_list

Out[23]: True
```

The plus operator is overloaded so that we can use it to concatenate two lists, much like we did with strings:

```
In [24]: print(my_list + hats)

['Item 0', 'Item 1', 2, 'fedora', 'porkpie', "tam o'shanter",
 'Phrygian cap', "Beefeaters' hat", 'sombrero', 'toque']
```

BOX 2.3 PYTHON PRINCIPLE

A list in Python is a collection of data that can be changed after creation. The syntax to define a list is

```
list_var = [item1, item2, ..., itemN]
```

where the `...` indicates a number of other items separated by commas. Items in a list can be indexed using square brackets, and slice indexing using `:` is also supported. To add or remove items from the list use the `append` and `remove` commands with syntax of the form

```
list_var.append(item_add)
list_var.remove(item_remove)
```

where `item_add` and `item_remove` are items to add or remove from a list, respectively. Appending to a list adds the element to the end.

The `len` function will return the size of the list,

```
len(list_var)
#return the length of list_var
```

The `in` operator can be used to test if an item is in the list:

```
test_item in list_var
```

will evaluate to true if `test_item` is in the list `list_var`.

The plus operator, +, is overloaded so that

```
list_var1 + list_var2
```

will concatenate `list_var2` to the end of `list_var1`.

2.3.2 Tuples

A tuple is very similar to a list with two exceptions. The first exception is minor in that you define a tuple using regular parentheses:

```
In [25]: hats_tuple = ("fedora","trilby","porkpie",
                       "tam o'shanter","Phrygian cap",
                       "Beefeaters' hat","sombrero")
         print(hats_tuple)

('fedora', 'trilby', 'porkpie', "tam o'shanter", 'Phrygian cap',
 "Beefeaters' hat", 'sombrero')
```

You can access an element in a tuple with square brackets the same way you do a list

```
In [26]: hats_tuple[1]

Out[26]: 'trilby'
```

The major difference between lists and tuples is that tuples cannot be modified once created. The technical term for this is that tuples are immutable, whereas lists are mutable. If we try to change a tuple, we get an error:

```
In [27]: hats_tuple[1] = 'deer stalker'


---------------------------------------------------------------------
 TypeError            Traceback (most recent call last)

        <ipython-input-27-60a4d89f7f83> in <module>()
    ----> 1 hats_tuple[1] = 'deer stalker'


        TypeError: 'tuple' object does not support item assignment
```

The main point of tuples is to give the programmer the ability to group items together in a lightweight manner without all the behind-the-scenes infrastructure that a mutable type (like a list) has. It also can be useful if you want to define an object that you want to assure will not change. We will not have many uses for tuples in our work, but it is useful to know that they exist.

BOX 2.4 PYTHON PRINCIPLE

A tuple in Python is a collection of data that cannot be changed after creation. Parentheses are used to define a tuple:

```
tuple_var = (item1, item2, ..., itemN)
```

where the `...` indicates a number of other items separated by commas. Items in a tuple can be indexed using square brackets, and slice indexing using `:` is also supported.

The `len` function will return the size of the tuple,

```
len(tuple_var)
#return the length of tuple_var
```

2.4 FLOATS AND NUMERICAL PRECISION

Previously, it was mentioned that floating point numbers are not exact because a computer has a finite number of bits to represent the numbers (i.e., we cannot have an infinite number of decimal places in the number). Typically, this is not a major issue, but when it does matter, it can cause problems. In our work, one particular time that this matters is when we want to have a stopping criteria on a floating point number, for instance this is a bad idea:

```
In [28]: import random
         import math
         iteration = 0
         guess = 0
         closest = 0
         target = 0.3
         while (guess != target) and (iteration < 10**6):
             guess = random.random()#same as random.uniform(0,1)
             if (math.fabs(guess - target) <
                 math.fabs(closest - target)):
                 closest = guess
             iteration += 1
         print("In", iteration,
               "the closest random number we got to",
               target,"is", closest)

In 1000000 the closest random number we got to 0.3
is 0.3000001943632269
```

To most people, and for most computations, 0.3000001943632269 is close enough to 0.3, but the computer just knows that these two numbers are not equal.

What we probably wanted is something like

```
In [29]: import random
         import math
         iteration = 0
         guess = 0
         tolerance = 1.0e-6
         while (math.fabs(guess - 0.3) > tolerance and
                (iteration < 10**6)):
             guess = random.random()
             iteration += 1
         print("It took", iteration,
               "guesses to get within",tolerance,"of 0.3")
         print("The number we ended with is",guess)

It took 128626 guesses to get within 1e-06 of 0.3
The number we ended with is 0.29999952310331457
```

The bottom line, is that one should not use equality tests with floating point numbers. That is why in our exhaustive enumeration example in Section 2.1 we only tried to get within a tolerance rather than match the number exactly.

Even very simple equality tests with floating point numbers can fail.

```
In [30]: (0.1+0.1+0.1) == 0.3

Out[30]: False
```

Wait, what? The number 0.1 is not exactly represented in the computer because a computer stores numbers using a base 2 representation and not a base 10 representation like we are used to. To demonstrate this we can print out 0.1 to 20 digits.

```
In [31]: print("%.20f" % 0.1)

0.10000000000000000555
```

There is a tiny error that is amplified when we compute 0.1 + 0.1 + 0.1.

```
In [32]: 0.1 + 0.1 + 0.1

Out[19]: 0.30000000000000004
```

Later, especially in solving linear systems, we will see that the numerical precision can have a large effect on our answers, if we formulate algorithms that are sensitive to these errors. For now, let us agree not to use equality with floating point numbers.

FURTHER READING

A excellent example of the power and limitations of finite precision arithmetic can be found in the SIAM 100-Digit Challenge [3]. For further reading on Monte Carlo methods, Kalos and Whitlock is the standard reference [4].

PROBLEMS

Short Exercises

2.1. Write a Python code that asks the user to input a string. Then print back to the user their string backwards.

2.2. If we list all the natural numbers below 10 that are multiples of 3 or 7, we get 3, 6, 7, and 9. The sum of these multiples is 25. Find the sum of all the multiples of 3 or 7 below 10000.

2.3. Prompt the user for a number. Report back the cube root of the number. Test the code with some numbers that are perfect cubes, e.g., `125`. How accurate is the answer?

2.4. The public domain folk song "Dem Bones" is a song that describes in a pseudo-scientific manner the layout of the bones of the human body. An example couplet of this song reads as

> *The foot bone connected to the leg bone*
> *The leg bone connected to the knee bone*

which can be generalized to

The $\mathtt{b}_i$ *bone connected to the* $\mathtt{b}_{i+1}$ *bone*
The $\mathtt{b}_{i+1}$ *bone connected to the* $\mathtt{b}_{i+2}$ *bone*

Write a Python code that uses a `for` loop and a `list` that prints out an entire verse where the bones, in order, are

```
b = ["foot", "leg", "knee", "thigh", "back", "neck", "head"]
```

Programming Projects

1. Nuclear Reaction Q Values

Write a Python code that asks the user to input the masses (in amu) of two reactants in a nuclear reaction and the two products of the reaction. The code will output the Q value of the reaction in MeV. Here is an example reaction:

$$ {}^{A_X}_{Z_X}X + {}^{A_x}_{Z_x}x \rightarrow {}^{A_Y}_{Z_Y}Y + {}^{A_y}_{Z_y}y. $$

The user should be able to enter zero for the mass in case the reaction is a decay (i.e., has only one reactant) or is a reaction that has only one product. Use 1 amu = 931.494061 MeV/c^2.

2. Calculating e, the Base of the Natural Logarithm

The series

$$ \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} \cdots = e, $$

is a means of approximating e from the Taylor expansion of e^x. We can write a partial sum as

$$ e_{\text{approximate}} = \sum_{i=0}^{N} \frac{1}{i!}. $$

2.1. Using a `for` loop, compute an approximation using

$$ N = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100, 1000, 10000. $$

2.2. Describe how the solution converges to the exact answer as a function of N. That is, how does the error in the estimate change as a function of N?

2.3. How many digits of

$$ e = 2.71828182845904523536028747135266249775724709369995 $$

can you compute correctly?