Problem 3: Numeric Integration

Favoris

Week 3: Functions and Recursion > Homework 3 > Problem 3: Numeric Integration

In this lab you will build a Python program that can compute the areas underneath arbitrary mathematical functions (mathematicians call these integrals).

Doing this, you'll:

- practice writing Python functions
- gain experience with list comprehensions
- write helper functions and compose large functions from smaller ones

When you're finished with this assignment, submit your code at the bottom of this page.

Start by making a copy of this trinket!

Trying Out List Comprehensions

To begin, try out *list comprehensions* in trinket. This should help build intuition about how list comprehensions work:

```
>>> lc_mult( 10 )  # multiplication example
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> lc_mult( 5 )  # a smaller example
[0, 2, 4, 6, 8]

>>> lc_idiv( 10 )  # integer division
[0, 0, 1, 1, 2, 2, 3, 3, 4, 4]

>>> lc_fdiv( 10 )  # floating-point division
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
```

Read over those three one-line functions; be sure you have internalized how they work!

Note that the calls `lc_idiv( 10 )` and `lc_fdiv( 10 )` return *different* lists; the first uses integer division (rounding down); the second uses floating-point division.

To do

Change the `lc_idiv` function so that it becomes

```
return [ float(x/2) for x in range(N) ]
```

Before running it, decide if you think `lc_idiv( 10 )` will now be the same or different than before.

Then, run `lc_idiv( 10 )`. Did it match your expectations? Nothing to write here, but be sure you're happy with this why this new output is what it is!

Introducing Integration

Integration is sometimes described as finding the area between a mathematical function and the horizontal (x) axis.

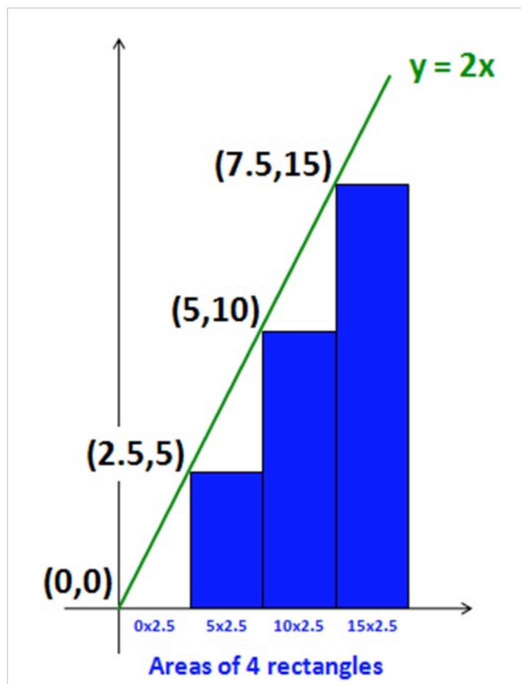
More generally, integration is simply the *sum* of a numeric function's output values, i.e., y-values across a chosen span.

It is important because it provides a one-number summary of what the function is "doing" over an interval. It's used to determine how fields of forces act on a particular point, surface, or object over time. It is also essential in defining a function's "average value" on an interval or region.

As an example, recall the `dbl` function:

```
def dbl(x):
  """ input: a number x (int or float)
    output: twice the input
  """
  return 2*x
```

Suppose you wanted to estimate the integral of `dbl`, on the interval between `0.0` and `10.0`. You could create the following (rough) approximation with rectangles:



1. Here, the interval is divided into 4 parts. The list `[0, 2.5, 5, 7.5]` represents the x value of the *left-hand endpoint* of each of the four subintervals.
2. We next find the output values from `dbl(x)` for each possible `x` in the list above. Here, name these output values `Y`: `Y = [0, 5, 10, 15]`.
3. Add up the areas of the rectangles whose upper-left-hand corner (height) is at these `Y` values. Each rectangle extends down to the x-axis. Each rectangle's individual width is `2.5`, because there are four equal-width rectangles spanning a total of `10` units of width.
4. We find the rectangles' areas in the usual way. Their heights are contained in the list `Y` and their widths are `2.5`, so the total area is `0*2.5 + 5*2.5 + 10*2.5 + 15*2.5` or `(0 + 5 + 10 + 15)*2.5`, which is `(30)*2.5`, or `75`.

Yes, this is a very rough approximation for the "area under the curve," i.e., the integral. If we make the width of the rectangles smaller, however, their sum will approximate that area as closely as we'd like.

More importantly, this example suggests a general way to approximate a function's integral over a known interval:

1. Divide the interval into `N` equal parts and create a list with the corresponding `x` (input) values.
2. Find the `y` values (outputs) of the function for each value of `x` calculated in step 1
3. Calculate the area of each rectangle under the curve. Their heights are the `y` values; their widths are the separation between the `x` values.
4. Sum the rectangles' areas and return the result.

That's the integral, or an approximation that can be made arbitrarily close!

The rest of this problem involves writing functions for each of these four steps; then you'll use those functions to answer questions about the results.

Calculating the x Values to be Evaluated

First, you will write a function named `unitfracs(N)` and then one named `scaledfracs(low,hi,N)`.

Writing `unitfracs`

Take a look at how `unitfracs(N)` works:

```
>>> unitfracs(2)
[0.0, 0.5]

>>> unitfracs(4)
[0.0, 0.25, 0.5, 0.75]

>>> unitfracs(5)
[0.0, 0.2, 0.4, 0.6, 0.8]

>>> unitfracs(3)
[0.0, 0.3333333333333333, 0.6666666666666666]

>>> unitfracs(10)
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

As its name suggests, it returns a list of evenly-spaced left-hand endpoints (fractions) in the unit interval [0,1).

Hint! Copy, paste, and alter the example function `lc_fdiv` in order to write `unitfracs`. *You will only need to change a* ***single*** *character from that code!*

Be sure to fix the docstring of `unitfracs` to reflect what it does, as well.

Writing `scaledfracs`

Next, take a look at how `scaledfracs(low,hi,N)` works:

```
>>> scaledfracs( 10, 30, 5 )
[10.0, 14.0, 18.0, 22.0, 26.0]

>>> scaledfracs( 41, 43, 8 )
[41.0, 41.25, 41.5, 41.75, 42.0, 42.25, 42.5,
42.75]

>>> scaledfracs( 0, 10, 4 )
[0.0, 2.5, 5.0, 7.5]
```

`scaledfracs(low,hi,N)` creates `N` left endpoints uniformly through the interval [low,hi).

Hint! This is tricky! Here is some additional explanation:

Use `unitfracs`. For example, use this line as a starting point:

```
return [             for x in unitfracs(N)
]
```

This way, you won't have to redo the work of `unitfracs`!

You might feel that this is closely related to the `interp` function you wrote in Lab 1—you're right!

You don't need to use that `interp` function, but you will want to use the ideas from it! Here it is, for reference:

```
def interp(low,hi,frac):
    """ returns a value frac of the way from low to hi
"""
    return low + (hi-low)*frac
```

Note that the role of `frac` above is as the "interpolating fraction," which is exactly what `x` is doing in the list comprehension!

Do include a docstring that reflects what `scaledfracs` does.

Calculating the y Values

Your `scaledfracs` function can produce arbitrary lists of evenly-spaced `x` values.

Next, you'll need to calculate the `y` values (outputs) of a function at each of these x positions. Again, you'll use list comprehensions to make this process simple and fast!

Although the goal is to handle arbitrary functions, we'll start with a concrete function and build up.

Writing `sqfracs`

Write a function `sqfracs(low,hi,N)` that works as follows:

```
>>> sqfracs(4,10,6)
[16.0, 25.0, 36.0, 49.0, 64.0,
81.0]

>>> sqfracs(0,10,5)
[0.0, 4.0, 16.0, 36.0, 64.0]
```

Here, `sqfracs` is very similar to `scaledfracs` *except that each value is squared.*

Hint! Use `scaledfracs` here. In the same way that `scaledfracs` used `unitfracs`, `sqfracs` can use `scaledfracs`! Consider the snippet:

```
for x in
scaledfracs(low,hi,N)
```

Writing `f_of_fracs`

Write a function `f_of_fracs(f,low,hi,N)` that works as follows:

```
>>> f_of_fracs(dbl, 10, 20, 5)
[20.0, 24.0, 28.0, 32.0, 36.0]

>>> f_of_fracs(sq, 4, 10, 6)
[16.0, 25.0, 36.0, 49.0, 64.0, 81.0]

>>> f_of_fracs(sin, 0, pi, 4)  # the sine
function
[0.0, 0.71, 1.0, 0.71]
# the above values are rounded versions of what
# will actually be displayed
```

Note that `f_of_fracs` takes a *function* as its first input—this is no problem in Python.

You might copy-and-paste `sqfracs` as a *model*: only a few characters need to be changed! (3 of them, to be precise!)

Calculate the Area and Put it all Together

You now have functions that calculate both the `x` and, more importantly, `y` values of a function at regularly-spaced intervals.

Next, you'll write `integrate(f,low,hi,N)` which will return the final, desired value: the integral of `f` from `low` to `hi` using `N` steps.

Take a look at a few examples and hints below.

As these examples highlight, `integrate` does not return a list; it returns a single, floating-point value:

```
>>> import math       # to use math.pi and math.sin

>>> integrate(dbl, 0, 10, 4)
75.0            # the example from the top of this lab

>>> integrate(dbl, 0, 10, 1000)
99.9            # rounded from 99.8999 (precise value: 100)

>>> integrate(sq,0,3,1000000) # a million steps will give Python
pause
8.9999865000044963     # close! (precise value: 9.0)

>>> integrate(math.sin, 0, math.pi, 1000)
1.9999983550656628     # pretty good! (precise value: 2.0)
```

Don't worry about small errors in the rightmost (least-significant) digits. They occur from small differences in Python versions.

To get started, here is the "signature" (the `def` line) and docstring for `integrate`. Feel free to use this:

```
def integrate(f,low,hi,N):
  """ integrate returns an estimate of the definite integral
     of the function f (the first input)
     with lower limit low (the second input)
     and upper limit hi (the third input)
     where N steps are taken (the fourth input)

    integrate simply returns the sum of the areas of
rectangles
    under f, drawn at the left endpoints of N uniform steps
    from low to hi
  """
```

Hints:

- **Do NOT use a list comprehension!** No list is being created here.
- Rather, use `f_of_fracs` to generate the list of heights you need.
- Remember that the output of `f_of_fracs` is a big list of `y`-values!
- You want to multiply those `y`-values (heights) by the rectangles' width. ***But all the widths are the same!***
- So, you can sum the heights ***then*** multiply by the width!
- Use Python's built-in `sum`. `sum(L)` returns the sum of the elements in the list `L`.
- **Got 60 instead of 75?** Then you wrote `((hi-low)/N)`, which divides two integers! (As a result it rounds down, as with all integer division). Perhaps the easiest way to remedy this is to multiply `low` in the expression above by `1.0`. This will make it—and everything it touches—into `float`s.

Some examples on how `sum` works:

```
>>> sum( [10, 4, 1] )
15

>>> sum( range(0,101)
)
5050
```

If your `integrate` function works, congratulations! You've built a general-purpose routine that can provide integrals for any computable function (including those for which there is no closed-form integral).

Next, you'll put `integrate` to use.

Questions to Answer

You should put your answers either within comments or within triple-quoted strings in your file. Strings are easier because you don't need to preface multiple lines with the comment character, `#`.

Do not answer this zeroth question, we have placed the answer here, just as an example of how to answer the other two.

Question 0 (example)

Explain why `integrate(dbl, 0, 10, N)` converges to 100 as `N` increases.

The answer might look like:

```
"""
Q0.

The value of integrate(dbl, 0, 10, N) as N increases is equal to
the area under the dbl function, the line
y=2x, between x=0.0 to x=10.0.

This value is the area of the triangle in the image at the top of the problem's
page.

That area is 100, because the triangle's height is 20 and its width is 10.
For a triangle, A = 0.5*h*w .
"""
```

Question 1.

As noted, the exact value of the integral of the `dbl` function, $y=2x$, from $x=0$ to $x=10$ is 100, which is the area of the triangle in the image at the top of this page. That area is 100 because the triangle's height is 20 and its width is 10.

The calls to `integrate(dbl,0,10,4)` and `integrate(dbl,0,10,1000)`, shown above, output a little **less** than 100.

In a sentence explain why `integrate` will always *underestimate* the correct value of this particular integral.

As a follow up, what is a function whose integral would always be *overestimated* on the same interval, from `0` to `10`? (If you're not sure about this, answer the next two questions first.)

Question 2.

The following function, `c`, traces a part of a circular arc with radius 2.

```
def c(x):
    """ c is a semicircular function of radius two
"""
    return (4-x**2)**0.5
```

Place this function into your trinket and confirm the values of

```
>>> integrate(c,0,2,2)
3.732            #
rounded

>>> integrate(c,0,2,20)
3.228            #
rounded
```

Next, find the values of `integrate(c,0,2,200)` and `integrate(c,0,2,2000)` and make a note of them in you answer.

As `N` goes to infinity, what does the value of this integral become? Why?

Submit Homework 3, Problem 3

30.0/30.0 points (graded)

To submit your Homework 3, Problem 3 code, you'll need to copy it from the trinket above and paste it into the box below. After you've pasted your code below, click the "Check" button.

IMPORTANT: Make sure that there aren't spaces at the beginning of your code, and that you copied all of the characters. If there are extra spaces or you are missing spaces, our server won't be able to run your code and we won't be able to give you any of the points you deserve for your hard work.

1

2

3

4

```
from math import *

def dbl(x):
    """ doubler!  input: x, a number
"""
```

```
    return 2*x

def sq(x):
    """ squarer!  input: x, a number """
    return x**2


def lc_mult(N):
    """ this example takes in an int N
        and returns a list of integers
```

25

```
        from 0 to N-1, **each multiplied by
2**
```

Press ESC then TAB or click outside of the code editor to exit
correct

correct

Test results
CORRECT See full outputSee full output

You have used 1 of 3 attempts Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.