homework 0, version 2

Submission by: ***Henry Luengas*** (HBot106@mit.edu)

```
student =   (name = "Henry Luengas",  kerberos_id = "HBot106")
```

```
# edit the code below to set your name and kerberos ID (i.e. email without @mit.edu)

student = (name = "Henry Luengas", kerberos_id = "HBot106")

# press the ▶ button in the bottom right of this cell to run your edits
# or use Shift+Enter

# you might need to wait until all other cells in this notebook have completed running.
# scroll down the page to see what's up
```

Homework 0: Getting up and running

First of all, ***welcome to the course!*** We are excited to teach you about real world applications of scientific computing, using the same tools that we work with ourselves.

Before we start next week, we'd like everyone to **submit this zeroth homework assignment**. It will not affect your grade, but it will help us get everything running smoothly when the course starts. If you're stuck or don't have much time, just fill in your name and ID and submit 🙂

Exercise 1 - *Square root by Newton's method*

Computing the square of a number is easy – you just multiply it with itself.

But how does one compute the square root of a number?

Algorithm:

Given: x

Output: $\sqrt{x}$

1. Take a guess a
2. Divide x by a
3. Set a = the average of x/a and a. (The square root must be between these two numbers. Why?)
4. Repeat until x/a is roughly equal to a. Return a as the square root.

In general, you will never get to the point where x/a is *exactly* equal to a. So if our algorithm keeps going until x/a == a, then it will get stuck.

So instead, the algorithm takes a parameter `error_margin`, which is used to decide when x/a and a are close enough to halt.

Exercise 1.1

Step 3 in the algorithm sets the new guess to be the average of x/a and the old guess a.

This is because the square root must be between the numbers x/a and a. Why?

`ex_1_1 =`

If a is greater than the true sqrt, then x/a must be less than the true sqrt. Conversly if a is less than the true sqrt, then x/a must be greater than the true sqrt. Therefore the true sqrt is always between a and x/a or equal to both of them.

```
ex_1_1 = md"""
If a is greater than the true sqrt, then x/a must be less than the true sqrt. Conversly if a is less than the true sqrt, then x/a must be greater than the true sqrt. Therefore the true sqrt is always between a and x/a or equal to both of them.
"""


# you might need to wait until all other cells in this notebook have completed running.
# scroll down the page to see what's up
```

Exercise 1.2

Write a function newton_sqrt(x) which implements the above algorithm.

```
newton_sqrt (generic function with 3 methods)
```

```
function newton_sqrt(x, error_margin=0.01, a=x / 2)
    input = x
    guess = a
    result = x/a
    average = (guess + result) / 2

    if (abs(guess - result) < error_margin)
        return(result)
    else
```

```
        return(newton_sqrt(input, error_margin, average))
    end
end
```

```
1.411764705882353
```

```
newton_sqrt(2)
```

Correct

Well done!

Hint

[redacted]

Hint

[redacted]

Exercise 2 - *Sierpinksi's triangle*

Sierpinski's triangle is defined *recursively*:

- Sierpinski's triangle of complexity N is a figure in the form of a triangle which is made of 3 triangular figures which are themselves Sierpinski's triangles of complexity N-1.
- A Sierpinski's triangle of complexity 0 is a simple solid equilateral triangle

To draw Sierpinski's triangle, we are going to use an external package, ***Compose.jl***. Let's set up a package environment and add the package.

A package contains a coherent set of functionality that you can often use a black box according to its specification. There are **lots of Julia packages**.

```
# setting up an empty package environment
begin
    import Pkg
    Pkg.activate(mktempdir())
    Pkg.Registry.update()
```

```
end
```

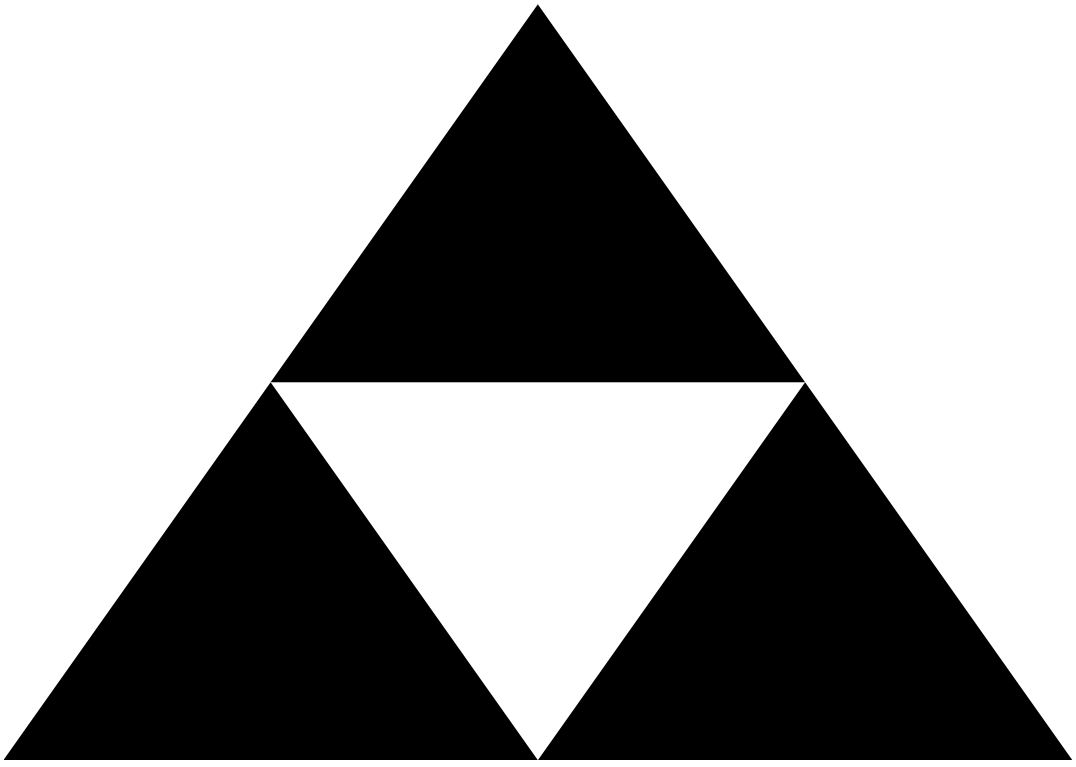
```
# add (ie install) a package to our environment
begin
    Pkg.add("Compose")
    # call `using` so that we can use it in our code
    using Compose
end
```

```
begin
    Pkg.add("PlutoUI")
    using PlutoUI
end
```

Just like the definition above, our `sierpinksi` function is *recursive*: it calls itself.

```
sierpinski (generic function with 1 method)
```

```
function sierpinski(n)
    if n == 0
        triangle()
    else
        t = sierpinski(n - 1) # recursively construct a smaller sierpinski's triangle
        place_in_3_corners(t) # place it in the 3 corners of a triangle
    end
end
```



```
sierpinski(complexity)
```

```
complexity = 1
```

```
. complexity = 1
```

Great! As you can see, all the cells in this notebook are linked together by the variables they define and use. Just like a spreadsheet!

Exercise 2.1

As you can see, the total area covered by triangles is lower when the complexity is higher.

Context[]

```
. sierpinski.(0:6)
```

Can you write a function that computes the *area of* `sierpinski(n)`, as a fraction of the area of `sierpinski(0)`?

So:

```
area_sierpinski(0) = 1.0
area_sierpinski(1) = 0.??
...
```

```
area_sierpinski (generic function with 1 method)
```

```
. function area_sierpinski(n)
.     if n == 0
.         return 1.0
.     else
.         return area_sierpinski(n-1) * 3 / 4
.     end
. end
```

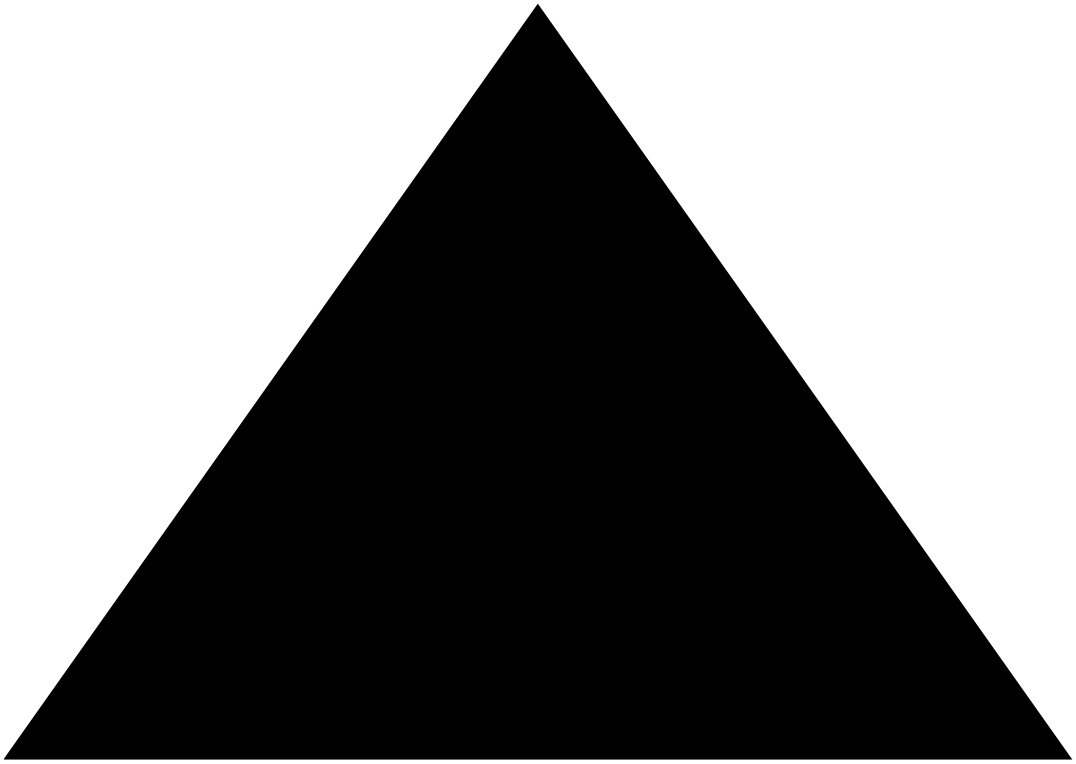
Correct

Well done!

Let's try it out below:

Complexity =  0

Sierpinski's triangle of complexity 0



has area **1.0**

Hint

That's it for now, see you next week!

```
triangle (generic function with 1 method)
```

```
place_in_3_corners (generic function with 1 method)
```