CSCI A201/A597 - Introduction to Programming I

Fall 2017

Programming Assignment 14

[100 points total]

Due Date

- at 4:00PM, on Monday December 04, 2017

Work Policy

Programming Assignment 14 is to be done individually - no group solutions or cooperative work.
You may discuss the programming assignment with other A201/A597 students, but you have to write the solutions and implement programming answers by yourself. If you discuss the programming assignment with other A201/A597 students, you must acknowledge them by including their names in a comment at the top of the Python files you turn in for this programming assignment. Do not share any source code or programming assignment answers with any students.

Notes

Keep a written log of your work for Programming Assignment 14, in a separate text document named **`pa14-log.txt`**, in which you annotate your work on these tasks.

At the top of the **`pa14-log.txt`** file, write:

- "A201/A597 Programming Assignment 14"
- your first and last name
- your IU username (that's the same as your IU email, *not* your numerical ID)

- in your `pa14-log.txt` file, write down notes about how you solved Programming Assignment 14 programming tasks.
  In particular, point out any difficulties, parts that may have been harder-than-usual to solve (as well as easier-than-usual), &c. Also (optionally) please include anything that you may have a question about, as well as any particularly good code you included in your solution, or anything else you may consider worth mentioning about it.

Solving A201/A597 programming assignments requires time - if you wait until the day when this assignment is due to start working on it, you may run out of time... we recommend that you start working on this assignment right away!

Date Date

- At 4:00PM, before your next A201/A597 lecture section begins (either Monday or Tuesday, depending on your enrollment section).

Work Policy

Programming Assignment 14 is to be done individually - no group solutions or cooperative work. You may discuss the programming assignment with other A201/A597 students, but you have to write the solutions and implement programming answers by yourself. If you discuss the programming assignment with other A201/A597 students, you must acknowledge them by including their names in a comment at the top of the Python files you turn in for this programming assignment. Do not share any source code or programming assignment answers with any students.

Getting Help

- attend **labs** and **office hours**
- ask the AIs, UIs, and Instructor:
  the recommended way to contact A201/A597 instructors is with a **Canvas message**: this way, you'll reach all A201/A597 AIs, UIs and Instructor at once.
  Log in to IU Canvas, select "Inbox" in the left-side window toolbar, then select the "Compose a new message" icon at the top. Select "Course:CSCI A201...Fall 2017" → "To: Teachers" → "All in Teachers", and include A201 (or A597) in the Subject.
- consult the textbook
- consult lecture notes
- consult the references provided in reading assignments

Work Material for Programming Assignment 14 Tasks:

To begin with, here's some Python code (including many descriptive comments) which you need to read and understand first. You may also download the same code here.

```
def f1(x):
    return f1(x)

# This causes an infinite loop, which crashes when it gets 1000 levels deep
# When we call f1(5), it just keeps calling f1(5) again to try and calculate
# the result, until the stack gets too big.

def f2(x):
    return f2(x-1)

# Again, we get an infinite loop, which crashes at 1000 recursions
# This time, the first time through, it tries to calculate f(4), and then f(3),
# and then f(2), etc., but it never ends (until it crashes)

def f3(x):
    if x == 0:
        return 55
    else:
        return f3(x-1)

# In this case, as long as we give a non-negative integer as input, it will
# eventually return 55, and then pass that number back up the stack and return
# 55 for the whole function.
# In PYTHON, if we passed a number bigger than about 1000, it will crash not
# because it's inherently wrong, but because Python can't handle real recursion.
# If you pass a negative number, the values will never stop and it will crash

def f4(x):
    if x==0:
        return 0
    else:
        return f4(x-1)+5

# If you chase down all the recursions here, basically it adds 5 for each
```

```
# recursive step, and since it always calls with one smaller, that means
# if the argument is x, then it adds 5 x times. In other words, this function
# is a recursive implementation of times-five.

# Let's actually do something a little more practical:
# QUIZ: Without using recursion, define a function called gauss(), which takes
# a non-negative integer n and returns 1+2+3+4+...+n.

def gauss1(n):
    total=0
    while n!=0:
        total=total+n
        n=n-1
    return total

def gauss2(n):
    numbers=range(1,n+1)
    total = 0
    for number in numbers:
        total=total+number
    return total

# There are other methods.

# Now let's do this with recursion.

# Remember from high school algebra that a natural number (nat) is a
# non-negative integer. (i.e., 0, 1, 2, 3, ...)
def gauss(n):
    """returns the sum of all integers from 1 up to n

    natural-number -> natural-number"""
    if n == 0:
        return 0                  # base case. should not be recursive.
    else:
        return gauss(n-1) + n     # recursive case. we call the function again
                                  # with a value that is closer to the base value

# Every recursive function needs two cases:
# A base case, where no recursion is used.
# And a recursive case, where the function is called again with a value closer
# to the base value, and usually some other calculation is done with that value

# Think about the recursive case like this:
    # If I have the answer for n-1, then how do I get the answer for n
    # gauss(n-1) represents the answer for n-1. We add n to that to get the
    # answer we want for the number n. Think of gauss(n-1) not as an instruction
    # but as the result of the calculation.

# Binary Number Trees are a kind of data, and here's how we'll represent them
# in python:

# A Binary Number Tree (BNT) is either:
#  - a number, or:
#  - a tuple (BNT, BNT)           (a tuple of two Binary Number Trees)

# So for example, a BNT with four numbers may look like this:

tree1 = ((2,(3,2)),5)

# This is a recursive data definition, which uses Binary Number Trees as part
# of the definition of a Binary Number Tree.

def count_leaves(bnt):
    """returns the number of leaves in bnt
```

```
    BNT -> int"""

    if isinstance(bnt,tuple):

        # this is the recursive case
        # In this case, I have two smaller trees, the left branch and the
        # right branch
        # the left branch is: bnt[0] and the right branch is bnt[1]
        # These values are closer to the base case, because they are smaller
        # trees. So we use them as part of recursive calls:
        # count_leaves(bnt[0]) (the number of leaves in the left branch)
        # and count_leaves(bnt[1]) (the number of leaves in the right branch)

        return count_leaves(bnt[0]) + count_leaves(bnt[1])

    else:
        # Then it must be a number, so this is the base case:

        return 1
```

Technically you already have all the tools you need to write all sorts of recursive functions, but this reading will walk through a few examples of more complex recursive functions working with trees. This should give you a sense of the variety that is possible with this technique and to point out how it's really the same strategy every time.

This is how we represented Binary Number Trees in the above example Python code (please read the comments in the above example Python code, for more details about).

```
# A Binary Number Tree (BNT) is one of:
  # - a number
  # - a tuple: (Binary Number Tree, Binary Number Tree)
```

In other words, we represent trees as either just a number (if there's only one leaf in the entire tree) or a two-item tuple where the first item is another tree (representing the left branch) and the second item is also another tree (representing the right branch).

Every recursive function that works on Binary Number Trees has basically the exact same structure:

```
def func(bnt):
    """do something with the tree bnt

    BNT -> ???"""
    if isinstance(bnt,tuple):
        # do something with func(bnt[0]) and func(bnt[1])
        # and return the result
    else:
        # do something with bnt (which is just a number here)
        # and return the result
```

Comparing Leaves

Suppose we wanted to write a function that takes a Binary Number Tree and returns the largest value that appears on any leaf. Before getting into the details, let's just look at the overall structure of the function:

```
def largest_leaf(bnt):
    """return the largest leaf in bnt

    BNT -> number"""
```

```
    if isinstance(bnt,tuple):
        # recursive case
        # ... largest_leaf(bnt[0]) ... largest_leaf(bnt[1]) ...
    else:
        # base case
        # ... bnt ...
```

First, let's look at the base case. If `bnt` is just a number representing a single leaf, then it's really obvious what the value of the largest leaf is: it is just `bnt`! So we return that.

```
def largest_leaf(bnt):
    """return the largest leaf in bnt

    BNT -> number"""
    if isinstance(bnt,tuple):
        # recursive case
        # ... largest_leaf(bnt[0]) ... largest_leaf(bnt[1]) ...
    else:
        return bnt
```

Now we look at the recursive case, where `bnt` is a tuple whose two members are the left and right branches. The left branch is `bnt[0]` and the right branch is `bnt[1]`. Next, we should think about `largest_leaf(bnt[0])`, which will return the largest leaf *in the left branch*. Try not to think of this as an *instruction* for doing a calculation, but as the *result* of the calculation. Also, try not to think about *how* that calculation was made; that's not our job right now. Just imagine that the calculation has already been made: `largest_leaf(bnt[0])` has returned the value of largest leaf in the left branch, and similarly `largest_leaf(bnt[1])` has returned the value of the largest leaf in the right branch. In particular, that means that `largest_leaf(bnt[0])` and `largest_leaf(bnt[1])` are *numbers*. (Because `largest_leaf()` always outputs a number: checkc out its signature.)

Once we have that idea in mind, that we already *have* the largest leaf in the left branch and the largest leaf in the right branch, our job is to figure out how to take those two values and figure out what the largest leaf in the entire tree is. It's actually quite a simple calculation. Look at the biggest value from the left (call it *L*), and if it's bigger than the biggest value from the right (call it *R*), then use *L*! And if it's not bigger, then use *R*!

```
        if L>R:
            return L
        else:
            return R
```

Of course, the biggest value on the left isn't called *L*; it's called `largest_leaf(bnt[0])`. And instead of *R*, we actually have `largest_leaf(bnt[1])`. So putting that all together, we get:

```
def largest_leaf(bnt):
    """return the largest leaf in bnt

    BNT -> number"""

    if isinstance(bnt,tuple):
        if largest_leaf(bnt[0]) > largest_leaf(bnt[1]):
            return largest_leaf(bnt[0])
        else:
            return largest_leaf(bnt[1])
    else:
        return bnt
```

There are other ways to figure out which of `largest_leaf(bnt[0])` and `largest_leaf(bnt[1])` is bigger. Some are more efficient. Some are shorter. But all of them use the values `largest_leaf(bnt[0])` and `largest_leaf(bnt[1])` as the biggest value from the left and the biggest from the right.

Recursive Functions That Return Booleans

Now let's tackle a seemingly completely different problem. Suppose we wanted to write a function that determines whether or not there are any zeroes in the tree. This is still a recursive function that takes a Binary Number Tree as an argument, so the overall structure is exactly the same as last time:

```
def has_zero(bnt):
    """returns true if bnt has a zero leaf and false otherwise

    BNT -> boolean"""

    if isinstance(bnt,tuple):
        # ... has_zero(bnt[0]) ... has_zero(bnt[1]) ...
    else:
        # ... bnt ...
```

As usual, let's examine the base case first, where `bnt` is just a number. In this situation, we have to think a little (but only a little). If that one number is zero, we want to return `True`. If that one number is not zero, we want to return `False`. So we might do something like this for the base case:

```
def has_zero(bnt):
    """returns true if bnt has a zero leaf and false otherwise

    BNT -> boolean"""

    if isinstance(bnt,tuple):
        # ... has_zero(bnt[0]) ... has_zero(bnt[1]) ...

    else:
        if bnt == 0:
            return True
        else:
            return False
```

And that will work.

A side trip about unnecessary `if`/`else`...

But... It's actually kind of a silly thing to do. We're asking Python to calculate `bnt==0`. And if that calculation returns `True`, we are telling it to return `True`. And if that calculation returns `False`, we are telling it to return `False`. So why don't we just return the result directly?

```
def has_zero(bnt):
    """returns true if bnt has a zero leaf and false otherwise

    BNT -> boolean"""

    if isinstance(bnt,tuple):
        # ... has_zero(bnt[0]) ... has_zero(bnt[1]) ...

    else:
        return bnt==0
```

You may not be comfortable with returning the result of a "test" like`bnt==0`, but you should get used to it. `bnt==0` is just a calculation like any other Python command; it just happens to return a boolean instead of a number. And we *can* test booleans with `if`, we can also just return those results like any other result, if that's what we want to do, which is true here.

But back to recursion...

So where were we?

```
def has_zero(bnt):
    """returns true if bnt has a zero leaf and false otherwise

    BNT -> boolean"""

    if isinstance(bnt,tuple):
        # ... has_zero(bnt[0]) ... has_zero(bnt[1]) ...

    else:
        return bnt==0
```

Right. We've handled the base case, and now we're about to handle the recursive case. We need to work with `has_zero(bnt[0])` and `has_zero(bnt[1])`. As always, `bnt[0]` is the left branch of the tree and `bnt[1]` is the right branch. When Python calculates `has_zero(bnt[0])` for us, it is going to calculate whether or not the left branch has any zeroes or not. If the left branch does have a zero, then `has_zero(bnt[0])` will be `True`. If the left branch does *not* have a zero, then `has_zero(bnt[0])` will be `False`. How do we know? That's what our purpose statement and signature said!

Last time, `largest_leaf(bnt[0])` was a number, but that was only because `largest_leaf()` returned a number. `has_zero()` always returns a boolean, so that means that `has_zero(bnt[0])` must be a boolean.

Similarly, `has_zero(bnt[1])` will be `True` if the *right* branch has any zeroes, and it will be `False` if it doesn't.

That means that I've got two booleans to work with: `has_zero(bnt[0])` and `has_zero(bnt[1])`, describing whether or not the left and right branches have zeroes. And I need to use those two answers to determine whether there are any zeroes anywhere in the tree.

There are four possibilities, right? Either both are `has_zero(bnt[0])` and `has_zero(bnt[1])` are `True`, meaning that both branches have at least one zero. In that case, I should return `True` for the whole tree. Or it might be the case that `has_zero(bnt[0])` is `True` and `has_zero(bnt[1])` is `False`, meaning that there's a zero in the left branch, but none in the right. In that case, I should also return `True` for the whole tree. The case where `has_zero(bnt[0])` is `False` and `has_zero(bnt[1])` is `True` is the same. In fact, the only time I should return `False` is if both `has_zero(bnt[0])` and `has_zero(bnt[1])` are `False` (meaning that neither the left nor the right branch has a zero).

The above paragraph actually makes the whole thing seem more complicated than it is. Really, as long as either one of `has_zero(bnt[0])` ***OR*** `has_zero(bnt[1])` (or both!) is `True`, then I should return `True`, and otherwise, I should return `False`. That's what `or` is for! So I might put it all together like this:

```
def has_zero(bnt):
    """returns true if bnt has a zero leaf and false otherwise

    BNT -> boolean"""

    if isinstance(bnt,tuple):
        if has_zero(bnt[0]) or has_zero(bnt[1]):
            return True
        else:
            return False
    else:
        return bnt==0
```

And that would work.

But... remember what I said above about silly `if`/`then` commands? I can actually make this much simpler:

```
def has_zero(bnt):
    """returns true if bnt has a zero leaf and false otherwise

    BNT -> boolean"""
```

```
    if isinstance(bnt,tuple):
        return has_zero(bnt[0]) or has_zero(bnt[1])
    else:
        return bnt==0
```

Functions That Take Trees *and* Return Trees

One last example. Suppose I wanted to write a recursive function that takes a Binary Number Tree and returns a new tree where each leaf has had 1 added to it.

As always, the structure looks like this:

```
def increment_leaves(bnt):
    """returns a copy of bnt with all leaves increased by 1

    BNT -> BNT"""

    if isinstance(bnt,tuple):
        # ... increment_leaves(bnt[0]) ... increment_leaves(bnt[1]) ...
    else:
        # ... bnt ...
```

Note the signature. This function doesn't return a number like `largest_leaf()` and it doesn't return a boolean like `has_zero()`. It returns another tree. In other words, it returns either a number or a two-item tuple where the two items are both trees. This will be important in two places.

First, it guides us in what we want to return: we can either return a number or we can take two trees and put them together in a tuple (`(tree1,tree2)`) and return that. (As long as we pick the *right* two trees.)

Second, it lets us know what we'll get *out* of the recursive calls. We can always depend on the fact that `increment_leaves(bnt[0])` and `increment_leaves(bnt[1])` will both be *trees*.

But let's not get ahead of ourselves too much. First, let's handle the base case. If `bnt` is just a single-leaf tree (i.e., a number), then it's easy to get a new single-leaf tree (i.e., a number) that is one bigger. We just add 1 to it and return the result.

```
def increment_leaves(bnt):
    """returns a copy of bnt with all leaves increased by 1

    BNT -> BNT"""

    if isinstance(bnt,tuple):
        # ... increment_leaves(bnt[0]) ... increment_leaves(bnt[1]) ...
    else:
        return bnt + 1
```

Now it's time to think about the recursive case. We've got a tree with two branches: `bnt[0]` and `bnt[1]`. But we actually have more than that. We have `increment_leaves(bnt[0])` and `increment_leaves(bnt[1])`. What is `increment_leaves(bnt[0])`? We've already mentioned that it's a tree (look at the signature!). But if we look at the purpose statement, we see that it's not just any tree; it's a copy of `bnt[0]` with all leaves increased by 1. To say it again: `increment_leaves(bnt[0])` is a copy of the left branch with all the leaves increased by 1. And similarly, `increment_leaves(bnt[1])` is a copy of the *right* branch with all leaves increased by 1.

So if I have a copy of the left branch (call it *L*) with all the leaves properly incremented and a copy of the right branch (call it *R*) with all its leaves also increased by 1, then how do I put them together to get the entire tree with all its leaves increased by 1? Well, I just have to put the two smaller trees together as the branches of the new tree: `( L, R )`.

Of course, the modified left and right branches aren't called *L* and *R*. They are called `increment_leaves(bnt[0])` and `increment_leaves(bnt[1])`. But we do the same thing with them; we combine them as the two members of a two-item tuple `( increment_leaves(bnt[0]), increment_leaves(bnt[1]) )` and return that.

```
def increment_leaves(bnt):
    """returns a copy of bnt with all leaves increased by 1

    BNT -> BNT"""

    if isinstance(bnt,tuple):
        return ( increment_leaves(bnt[0]), increment_leaves(bnt[1]) )
    else:
        return bnt + 1
```

And there you have it: three seemingly completely different recursive functions that can all be coded with the same set of steps:

1. Write down the header and the docstring, including a purpose statement and signature. (*Don't skip this and don't do it last!* It helps with the recursive step!)
2. Determine whether you're in the base case or a recursive case. (Usually with an `if`/`then`.)
3. Fill out what to return in the base case. Remember that the argument probably doesn't have any structure here to work with.
4. Write down what you have to work with in the recursive case. This will always be in the form `func(`*`smaller_value`*`)`, where *smaller_value* is of the same type as the main argument, but it's closer to the base case.
    - When the argument is a natural number `n`, then *smaller_value* is usually just `n-1`. And so you would write down `func(n-1)`.
    - When the argument is a binary tree `bnt`, then there are *two smaller_value*s: `bnt[0]` and `bnt[1]`, and so you would write down `func(bnt[0])` and `func(bnt[1])`.
5. Figure out what that `func(`*`smaller_value`*`)` represents. Start by identifying its type (look at the signature) and then think about what it actually is (look at the purpose statement).
    - In `gauss()`, `gauss(n-1)` was a number: specifically the sum of all the integers from 1 up to `n-1` (i.e., everything added up except the last number).
    - In `count_leaves()`, `count_leaves(bnt[0])` is a number, and it represents the number of leaves in the left branch. Similarly, `count_leaves(bnt[1])` is the number of leaves in the right branch.
    - In `largest_leaf()`, `largest_leaf(bnt[0])` is a number, and it represents the largest leaf in the left branch. Similarly, `largest_leaf(bnt[1])` is the largest leaf in the right branch.
    - In `has_zero()`, `has_zero(bnt[0])` is a boolean, and it represents whether or not there is a zero in the left branch. Similarly, `has_zero(bnt[1])` represents whether or not there is a zero in the right branch.
    - In `increment_leaves()`, `increment_leaves(bnt[0])` is another Binary Number Tree, and it is a copy of the left branch with all the leaves incremented. Similarly, `increment_leaves(bnt[1])` is a copy of the right branch with all the leaves incremented.
6. Figure out how to take the answer(s) for the smaller value(s) and use that to calculate the answer for the current argument.
    - In `gauss()`, we add `gauss(n-1)` (the sum of everything up to `n-1`) to `n` to get the sum of all the integers from 1 to `n`.
    - In `count_leaves()`, we add `count_leaves(bnt[0])` (the number of leaves in the left branch) to `count_leaves(bnt[1])` (the number of leaves in the right branch) to get the total number of leaves in the whole tree.
    - In `largest_leaf()`, we figured out which of `largest_leaf(bnt[0])` (the largest leaf in the left branch) and `largest_leaf(bnt[1])` (the largest leaf in the right branch) was bigger and returned that value, which is the biggest leaf in the whole tree.
    - In `has_zero()`, we determined if either `has_zero(bnt[0])` (is there a zero in the left branch?) or `has_zero(bnt[1])` (is there a zero in the right branch?) was `True` and that determined if there were any zeroes in the entire tree.

- In `increment_leaves()`, we took `increment_leaves(bnt[0])` (a copy of the left branch with all the leaves incremented) and `increment_leaves(bnt[1])` (a copy of the right branch with all the leaves incremented) and put them together as `( increment_leaves(bnt[0]), increment_leaves(bnt[1]) )`, a new tree with all the leaves of the original tree (only increased by 1).

Tasks

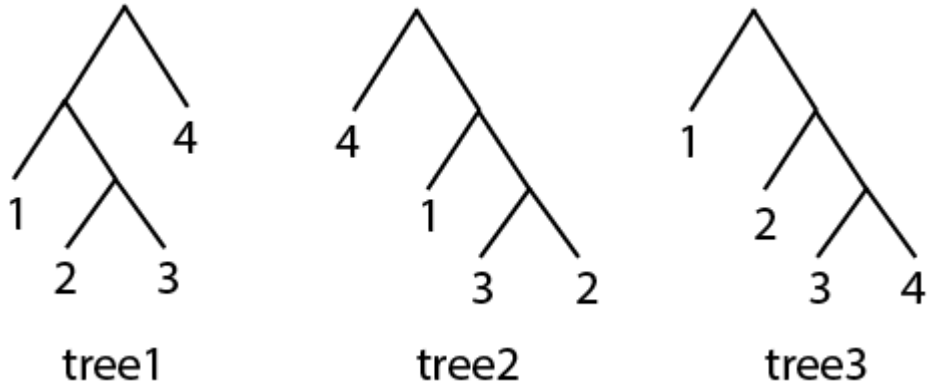
As for previous programming assignments, for Programming Assignment 14 it is *required* to include properly formatted docstrings with purpose statements and signatures for ***all*** methods and functions you are asked to define. If you don't include properly formatted docstrings with purpose statements and signatures for all methods and functions you define for Programming Assignment 14, your assignment will be *rejected*.

Note: save your IDLE interactive shell transcript into a file named `pa14-transcript.py` for your interactions with the Python scripts you write for all Tasks.

A. Save your script for Task A in a file named `pa14-BNTs.py`.

Remember the data definition we saw for Binary Number Trees in the above notes:

```
# A Binary Number Tree (BNT) is one of:
# - Number
# - (Binary-Number-Tree, Binary-Number-Tree)
```



Following the above three examples of Binary Number Trees, write a Python script that does the following:

1. Create three examples of Binary-Number-Trees to represent each of the examples in the above picture.
2. Store these trees in the variables `tree1`, `tree2`, and `tree3`.
3. Don't forget to include the data definition as a comment in the script.

B. In the source code above, we wrote a function that counted up the total number of leaves in a BNT. It was a recursive function, and it looked something like this:

```
def count_leaves(tree):
    """returns the number of leaves in tree

    BNT -> int"""

    if isinstance(tree,tuple):
        return count_leaves(tree[0]) + count_leaves(tree[1])
```

```
    else:
        return 1
```

Answer the following questions about this function definition.
Put your answers in a file named `pa14-answers.txt`.

1. Explain in your own words why in the base case, the function always returns the number `1`.
2. Explain what `tree[0]` and `tree[1]` represent in the recursive case.
3. Why can't you write `tree[0]` in the base case?
4. What do `count_leaves(tree[0])` and `count_leaves(tree[1])` represent in the recursive case?
5. Why are we adding `count_leaves(tree[0]) + count_leaves(tree[1])`?

C. Save your script for Task C in the same file as for Task A, i.e. a Python script file named `pa14-BNTs.py`.

Write a recursive function called `add_leaves()` that takes a Binary-Number-Tree and returns the sum of all the values of all of its leaves. All three of these example trees would result in `10`. Remember that this is a recursive function, so no loops and no lists!

D. *and now, for something almost (but not completely) entirely different:*

For this task, you'll be using the `turtle` module, which allows you to draw pictures by telling a "turtle" to move forward or backward or to turn to the left or right.

We've seen this module in previous assignments and homework...

You can read all about turtle graphics here, but you only need to know a few functions* in order to use it for this problem.

*To be specific, you need to understand `forward()`, `left()`, and `right()`. You may also find `clear()`, `reset()`, `speed()`, `backward()`, `setpos()` and `setheading()` useful too.

Play around with turtle before you move on to solving this task. Put `from turtle import *` at the top of a program, and then try out the commands `forward()`, `left()`, and `right()`. If things are going too slow for your taste, you can enter `speed(0)`.

Consider the following program:

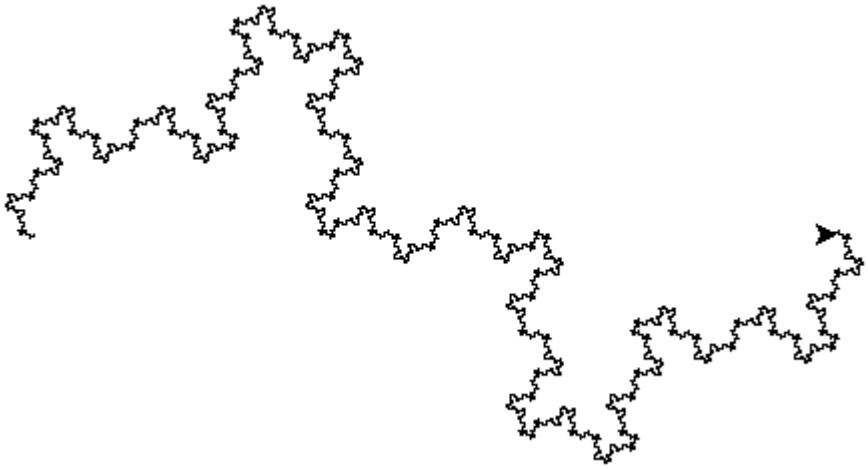
```
from turtle import *

def koch(length):
    if length < 2:
        forward(length)
    else:
        koch(length/3)
        left(60)
        koch(length/3)
        right(120)
        koch(length/3)
        left(60)
        koch(length/3)
```

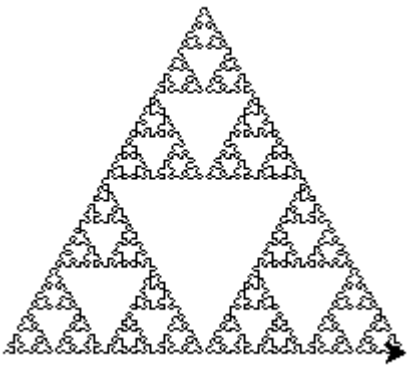
See if you can figure out what this function does and why it works. You can read more about this curve here. When you think you understand what it does, your job is to come up with your own fractal and draw it using turtle.

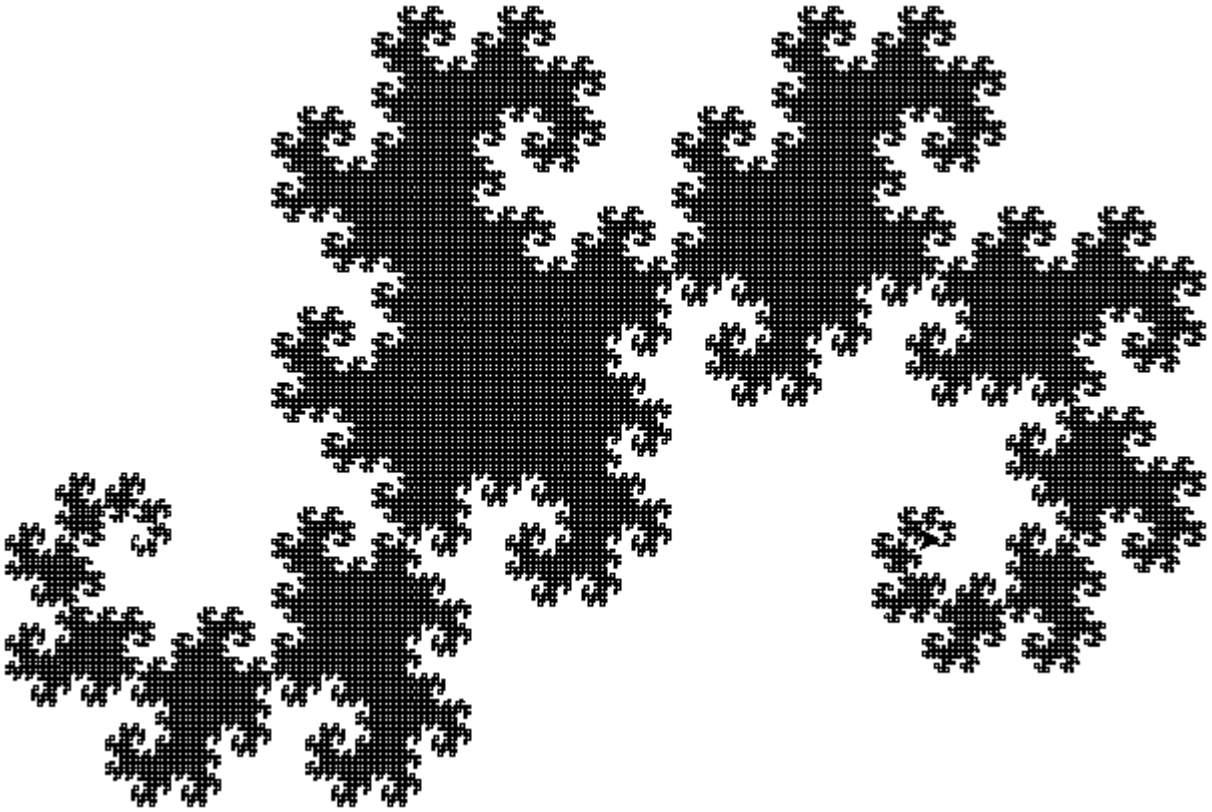
Save your script for Task D in a file named `pa14-fractals.py`.

Also turn in a screenshot of your screen running your fractal, showing what the fractal looks like. Save your screenshot into a file named `pa14-fractals.png` .

Here are some pictures of a few fractals that can be created that way:







Some of these are pretty famous, but I found them all accidentally by playing around with altering the recursive case of the `koch()` function above.

One vital key to getting a fractal (instead of some ugly mass of scribbles) is that in the recursive case, you should never move forward (or backward) a *fixed* amount. For example, never include `forward(5)`. This would ruin your fractal because it would move 5 pixels no matter what scale you're at. In fact, I wouldn't use `forward()` or `backward()` at all. Every time you want to move forward, I would instead use a recursive call with some fraction of `length`. If you want to move half the length forward, do something like `my_fractal(length/2)`. If you want to move a very short distance forward, do something like `my_fractal(length/10)`. Really, the only absolute values in the recursive case should be angles for turning.

Another key to getting a good-looking fractal is that the starting and ending position for the recursive case needs to be the same as for the base case. In `koch()`, the base case (when `length < 2`) just moves forward by the given `length`. In the recursive case, it moves forward by a third of `length`, then turning left at a 60 degree angle and moving a third of `length`, then turning right by 60 degrees and moving for a third, and finally turning left again by 60 degrees (meaning going back in the original direction) and moving one more third. The end result is that I'm exactly one `length` forward from the starting position, even though I took a bunch of detours. If you pick a different route to get to the same place (different angles, different proportions of `length`, different numbers of turns), you'll almost always end up with a good looking fractal. If your recursive case leaves you in a different position than the base case, then things will be a bit weirder and less fractal-like.

Last updated: 2017-12-01
mitja@indiana.edu