CSCI 141

Lecture 15
More Scope; Return Values; A4; Tuples

Announcements

Announcements

- A4 out today! Due Wednesday 11/13

Announcements

- A4 out today! Due Wednesday 11/13
  - It's big.

Announcements

- A4 out today! Due Wednesday 11/13
  - It's big.
  - It's bad.

Announcements

- A4 out today! Due Wednesday 11/13
  - It's big.
  - It's bad.
  - 11/11 is a holiday.

Announcements

- A4 out today! Due Wednesday 11/13
  - It's big.
  - It's bad.
  - 11/11 is a holiday.
  - Start early and get help if you're stuck.

Goals

- Be able to execute functions the way Python does, and understand the implications for local variables and scope.
- Know how to return a value from a function, and understand the behavior of the `return` statement.
- Understand the task assigned in A4 and how to approach it.
- Understand the basic usage of tuples:
  - using tuples to return multiple values from a function
  - packing and unpacking via assignment

QOTD

In which of the lines marked with comments is the variable **`v2`** in scope?

In which of the lines marked with comments is the variable **`v3`** in scope?

```python
# M1
def a(v1, v2):
    # M2
    v3 = v1 + v2
    # M3
    print(v3)

# M4
a(4, 6)
# M5
```

How to Execute Function Calls

```python
def axpy(a, x, y):
    """ Print a*x + y """
    product = a * x
    result = product + y
    print(result)

a1 = 2
x1 = 3
print(axpy(a1, x1, 4))
print(a1)
```

1. Evaluate all arguments
2. Draw a local "box" inside the current "box"
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box

How to Execute Function Calls

```
def axpy(a, x, y):
    """ Print a*x + y """
    product = a * x
    result = product + y
    print(result)

a1 = 2
x1 = 3
print(axpy(a1, x1, 4))
print(a1)
```

1. Evaluate all arguments
2. Draw a local "box" inside the current "box"
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box

If multiple variables exist with the same name, use the **innermost** one available.

QOTD

- What does this program print?

```
def f(x):
    g(3 * x)

def g(x):
    print(x + 2)

f(4)
```

QOTD

To execute a function call:

1. Evaluate all arguments
2. Draw a local "box" inside the current "box"
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box

What does this program print?

```
def f(x):
    g(3 * x)

def g(x):
    print(x + 2)

f(4)
```

Variable Scope

```python
def print_rectangle_area(width, height):
    """ Print the area of a width-by-height
        rectangle """

    area = width * height
    print(area)

w = 4
h = 3
a = w * h
print_rectangle_area(w, h)

```

What if I want to do **further computation** with the result of the rectangle area?

Variable Scope

```python
def print_rectangle_area(width, height):
    """ Print the area of a width-by-height
        rectangle """

    area = width * height
    print(area)

w = 4
h = 3
a = w * h
print_rectangle_area(w, h)

```

What if I want to do **further computation** with the result of the rectangle area?

It got printed, then it was gone…

Writing Functions: Syntax

```
def name(parameters):
    statements
```

Two important questions:

1. How does the function use the arguments (inputs) passed to it?
2. **How does the function return a value?**

Returning values

New statement: the `return` statement

Syntax: `return` *`expression`*

Behavior:

1. *`expression`* is evaluated
2. the function stops executing further statements
3. the value of expression is returned
   i.e., the function call **evaluates** to the returned value

Returning values

New statement: the `return` statement

Syntax: `return` *`expression`*

this can **only** appear inside a function definition!

Behavior:

1. *`expression`* is evaluated
2. the function stops executing further statements
3. the value of expression is returned
   i.e., the function call **evaluates** to the returned value

Demo: add2.py

- Make add2 return instead of print
- Assign result to a variable
- function composition: call add2 on the results of add2 calls

Function Syntax: Summary

def keyword

function name

```
def name(parameters):
    """ docstring """
    statements
```

Specification

inputs

comma-separated list of parameters: variable names that will get assigned to the arguments

An indented code block that does any computation, executes any effects, and (optionally) returns a value

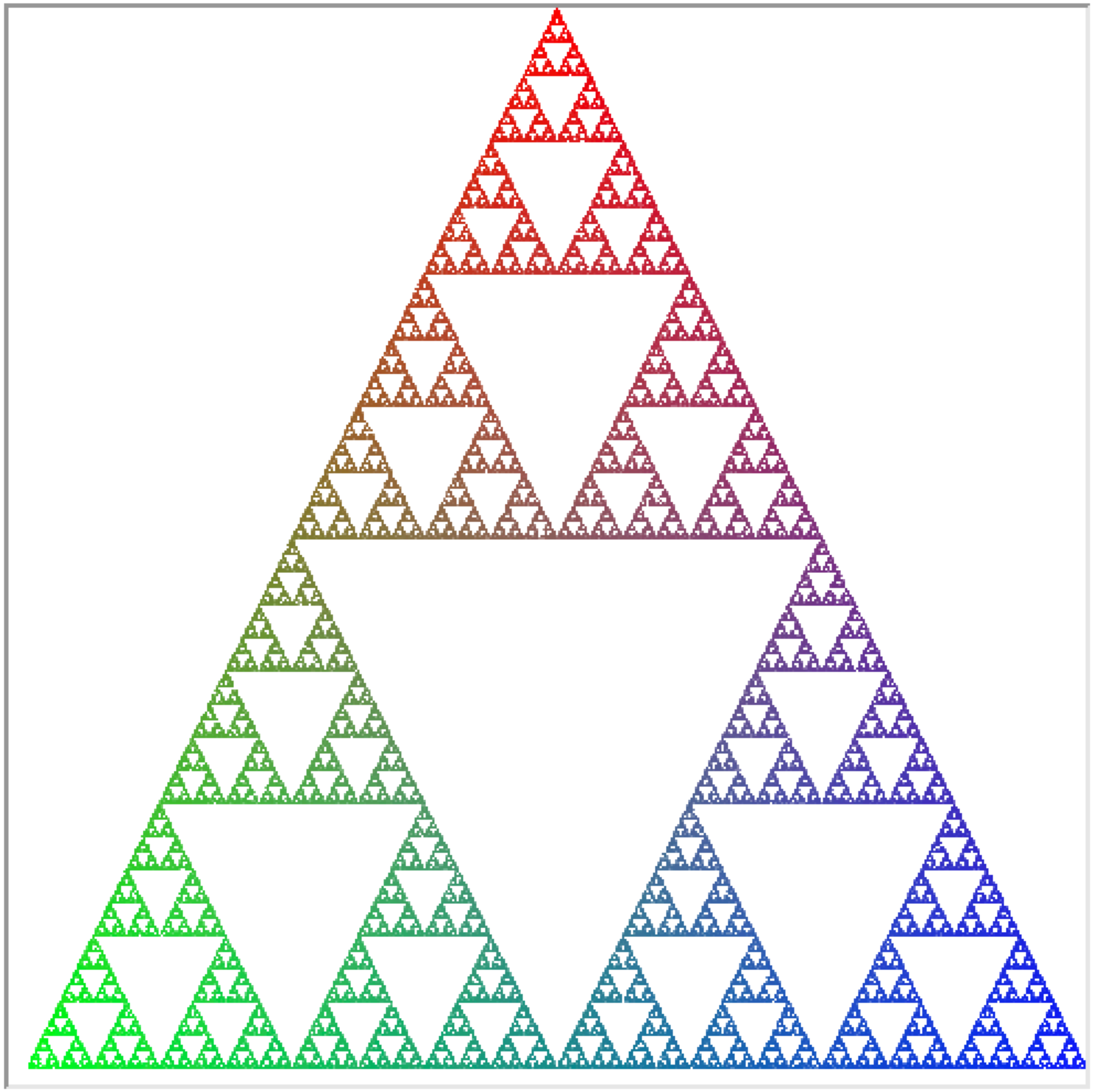
effects; return value

Why are functions great?

- **Concise** - wrap something complicated in an easy-to-use package:
  - define a function once then easily call it anywhere
- **Customizable** - make the easy-to-use package do different things:
  - customize the task your function performs based on its arguments
- **Composable** - use the result of one computation as input to (or as one step in) another.

A4

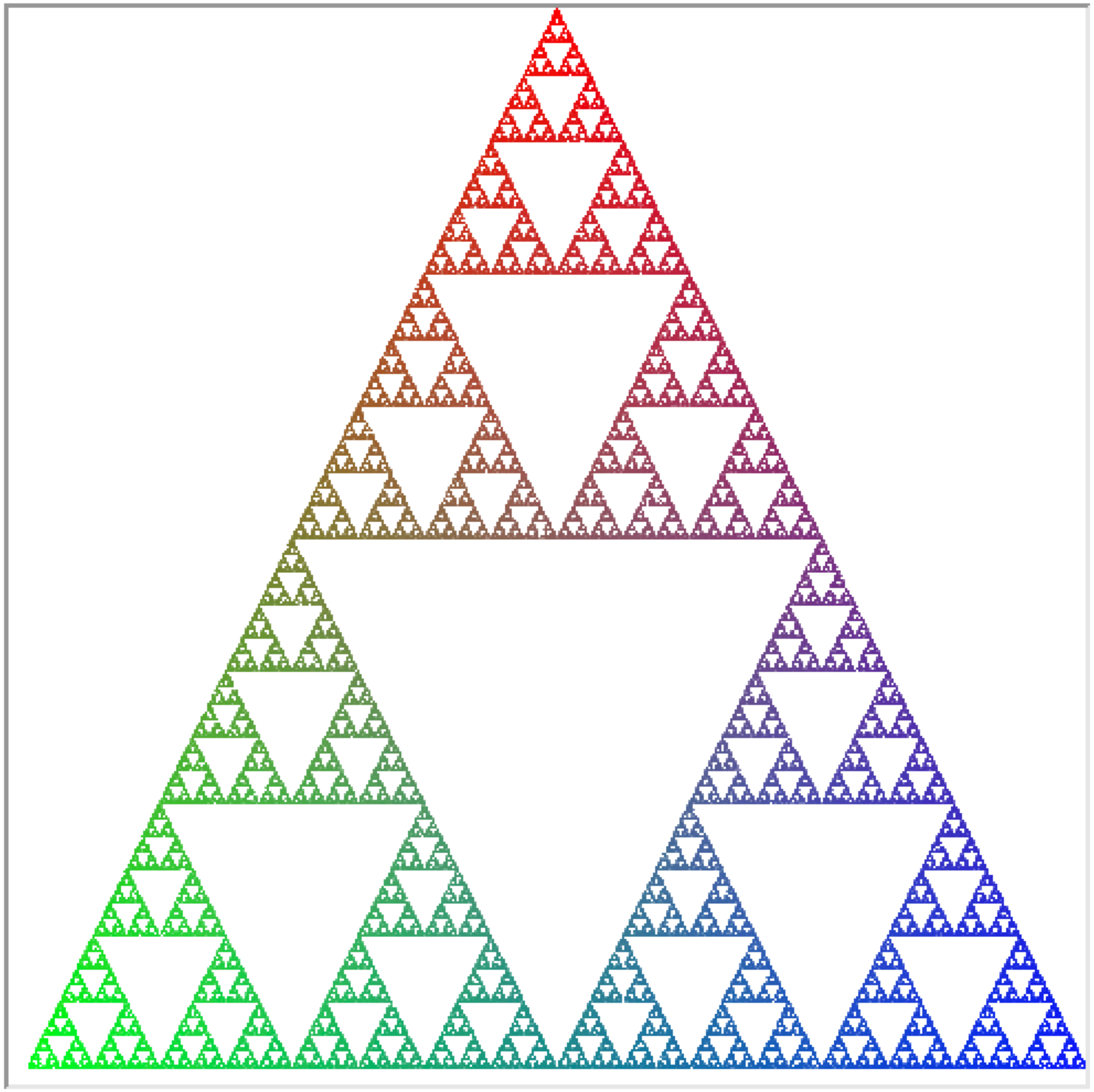
Your task:
Draw this.



A4

Your task:
Draw this.

Sounds simple, right?

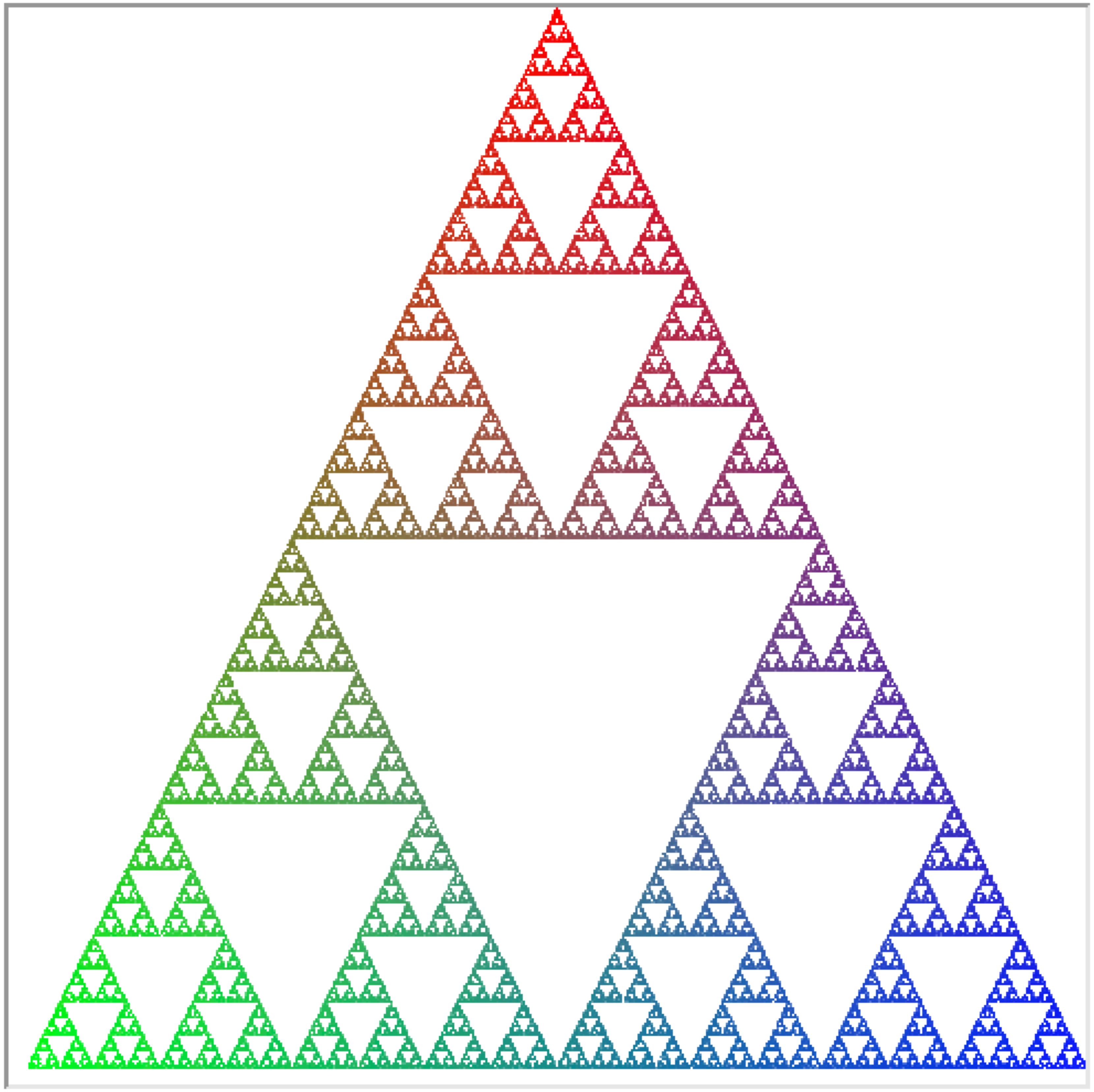


A4

Your task:
Draw this.

Sounds simple, right?

No.



A4: Pseudocode

```
# Let p be a random point in the window
# loop 10000 times:
#     c = a random corner of the triangle
#     m = the midpoint between p and c
#     choose a color for m
#     color the pixel at m
#     p=m
```

This pseudocode draws that crazy triangle thing.

A4: Pseudocode

```
# Let p be a random point in the window
# loop 10000 times:
#     c = a random corner of the triangle
#     m = the midpoint between p and c
#     choose a color for m
#     color the pixel at m
#     p=m
```

This pseudocode draws that crazy triangle thing.

Do you believe me?

A4: Pseudocode

```
# Let p be a random point in the window
# loop 10000 times:
#     c = a random corner of the triangle
#     m = the midpoint between p and c
#     choose a color for m
#     color the pixel at m
#     p=m
```

This pseudocode draws that crazy triangle thing.

Do you believe me?

(demo)

A4: Demo

```
# Let p be a random point in the window
# loop 10000 times:
#      c = a random corner of the triangle
#      m = the midpoint between p and c
#      choose a color for m
#      color the pixel at m
#      p=m
```

A4: Demo

```
# Let p be a random point in the window
# loop 10000 times:
#     c = a random corner of the triangle
#     m = the midpoint between p and c
#     choose a color for m
#     color the pixel at m
#     p=m
```

Demo:

- making up function names

Midpoint Function

```
def midpoint(p1x, p1y, p2x, p2y):
    """ Return the midpoint between
        (p1x, p1y) and (p2x, p2y)
    """
    # code here
```

(mid_x, mid_y)

This is **two** things!?
Can we return two things?

Midpoint Function

```
def midpoint(p1x, p1y, p2x, p2y):
    """ Return the midpoint between
        (p1x, p1y) and (p2x, p2y)
    """
    # code here
```

(mid_x, mid_y)

This is **two** things!? Can we return two things?

Midpoint Function

```
def midpoint(p1x, p1y, p2x, p2y):
    """ Return the midpoint between
        (p1x, p1y) and (p2x, p2y)
    """
    # code here
    # mid_x = . . .
    # mid_y = . . .
```

(mid_x, mid_y)

Midpoint Function

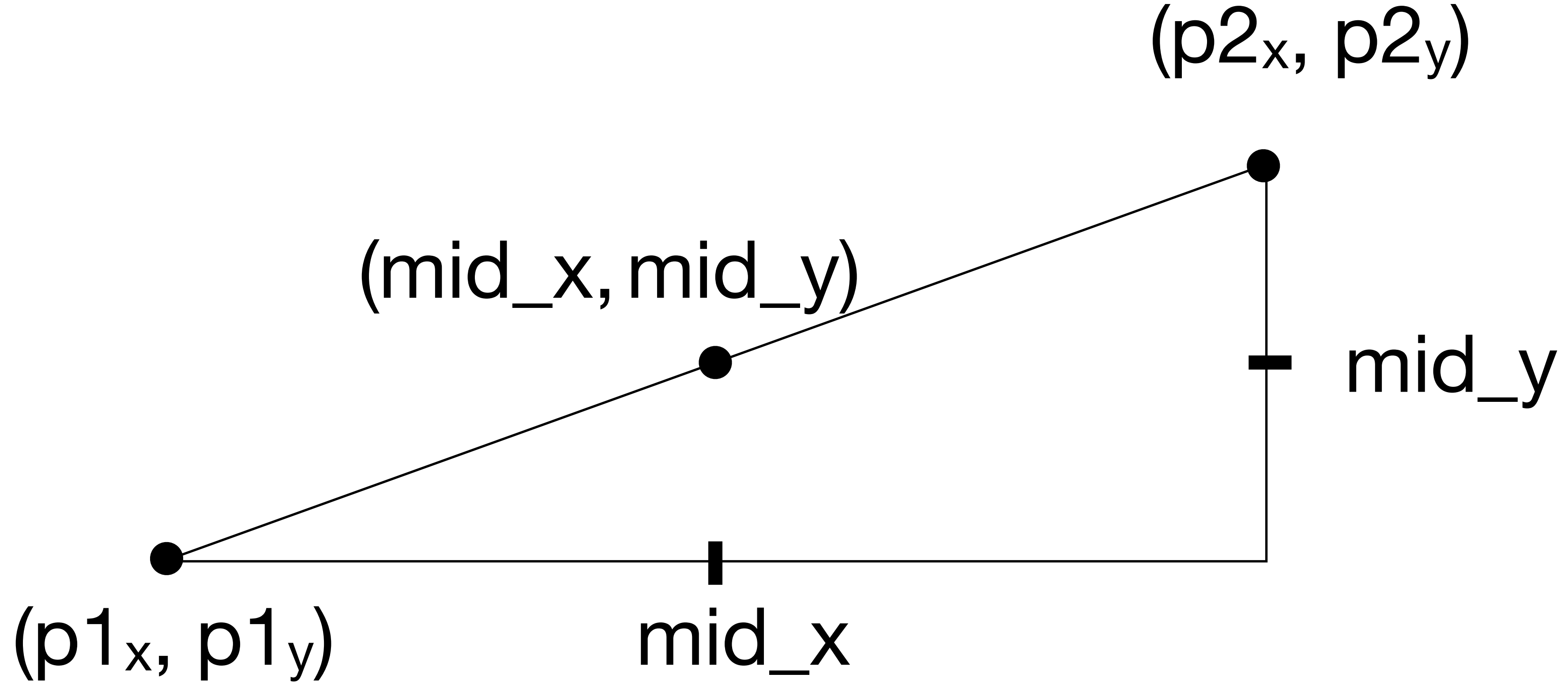
```
def midpoint(p1x, p1y, p2x, p2y):
    """ Return the midpoint between
        (p1x, p1y) and (p2x, p2y)
    """
    # code here
    # mid_x = . . .
    # mid_y = . . .

    return mid_x, mid_y
```

(mid_x, mid_y)

Midpoint Function

```
# mid_x = . . .
# mid_y = . . .
```

Okay, but how do you actually calculate this?

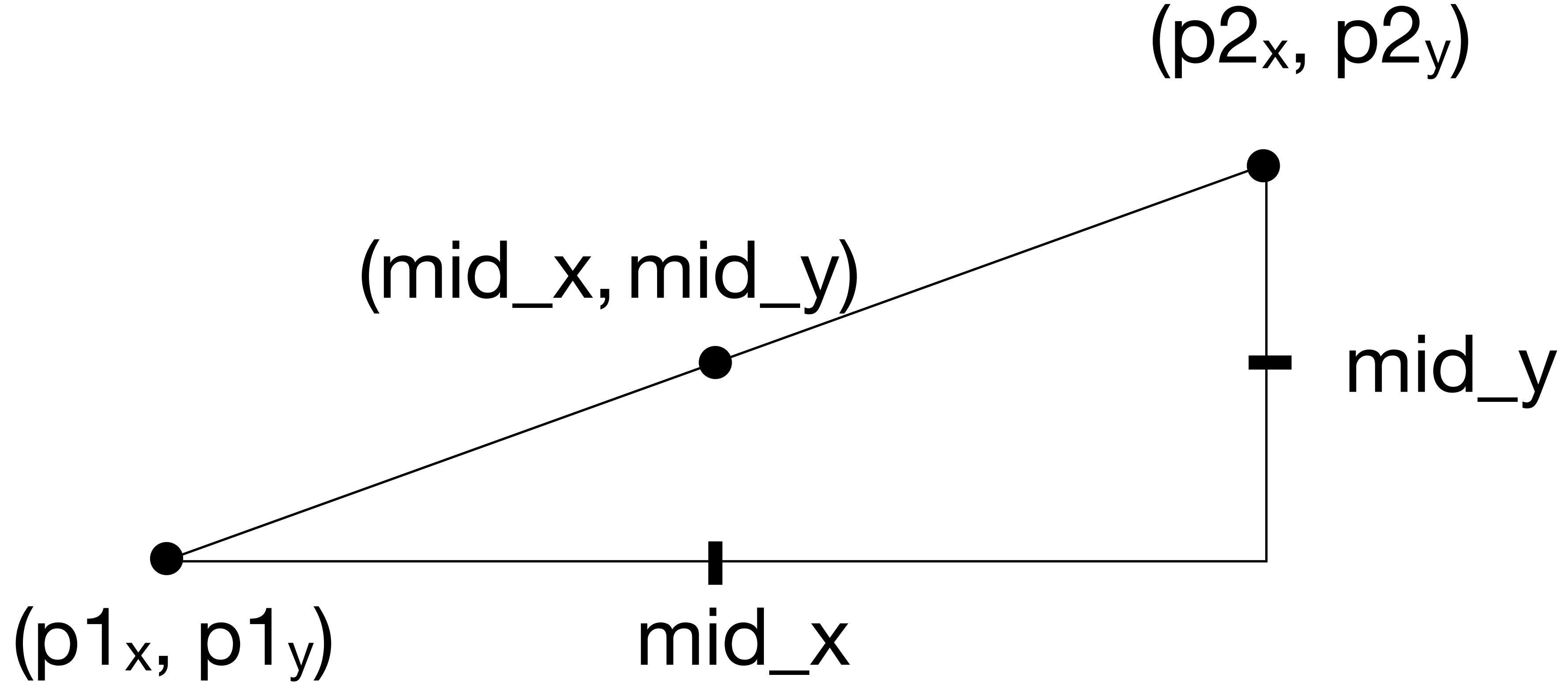


(whiteboard)

Midpoint Function

```
# mid_x = . . .
# mid_y = . . .
```

Okay, but how do you actually calculate this?



(whiteboard)

mid_x = $(p1_x + p2_x) / 2$

mid_y = $(p1_y + p2_y) / 2$

Returning Multiple Values

- You can return multiple values from a function by grouping them into a comma-separated sequence:

```
    return mid_x, mid_y
```

- You can assign each to a variable when calling the function:

```
mx, my = midpoint(p1x, p1y, p2x, p2y)
```

These are actually tuples

- A tuple is a sequence of values, optionally enclosed in parens.

```
(1, 4, “Mufasa”)
```

- You can “pack” and “unpack” them using assignment statements:

```
v = (1, 4, “Mufasa”)

(a, b, c) = v
```

These are actually tuples

- Tuples can also be passed *into* functions as arguments:

```
def midpoint(p1, p2):
    """Compute the midpoint between p1 and p2"""
    p1x, p1y = p1
    p2x, p2y = p2

    # . . .
    # return mx, my
```

Tuples: Demo