CSCI 141

Lecture 16
Finishing up Functions:
Returning Early
Tuples
Function Composition and Managing Complexity

Happenings

Tuesday, 5/14 – Peer Lecture Series: Unity Workshop

– 4 pm in CF 165

Tuesday, 5/14 – AIA Presents: AI in Education

– 6 pm in PH 228

Wednesday, 5/15 – Cybersecurity Lecture Series: Weaponizing Unicode with Aaron Brown

-- 5 pm in CF 105

Wednesday, 5/15 – Scholars Week, University Distinguished Lectures: Dr. Brian Hutchinson and The Broad Applicability of Deep Learning

– 5 pm in Carver 104

Announcements

- Midterm grades are not out yet. Working on it, and they'll be out ASAP.
- Get started on A4 soon: Demo in class today.
  - Due 1 week from today.

Goals

- Understand the basic usage of tuples:
  - using tuples to return multiple values from a function
  - packing and unpacking via the assignment operator
- Begin to understand how to use function composition to express complicated computations as clearly and simply as possible.

Function Calls: A Model for Execution

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

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

1. Evaluate all arguments
2. Assign argument values to parameter variables
3. Execute the function body
4. When done, replace the function call with its return value.

Variable Scope: Reminder

```
def print_rectangle_area(width, height):
    """Print area of width-by-height rectangle"""
    area = width * height
    print(area)
w = 4
h = 3
print_rectangle_area(w, h)
```

Facts:

- `width` and `height` are parameters
- `area` is a local variable
- `w`, and `h` are global variables

- All parameters are **also** local variables
- The scope of local variables is limited to the function they're defined in.

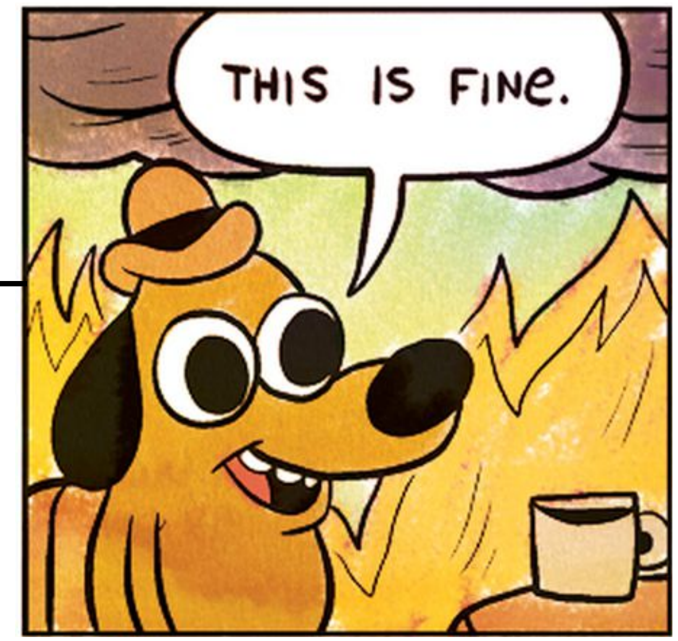
A note on the order of function definitions

```
def a():
    return 10

def b():
    return a() * 10

b()
```

Not much interesting going on here.

A note on the order of function definitions

```
def b():
    return a() * 10

def a():
    return 10

b()
```



(it actually is!)

- b calls a in its definition
- The call isn't *executed* until b is called
- As long as a has been defined by the time b is *called*, this is fine.

Today's Quiz

- 3 minutes

Today's Quiz

- 3 minutes
- Working with a neighbor: do your answers agree? (2 minutes)

Returning values

New statement: the `return` statement

Syntax: **`return`** *`expression`* (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

Returning Early: Demo

```
def sign(x):
    """ Return -1 if x < 0,
                1 if x > 0,
             or 0 if x == 0 """
    # code here
```

Returning Early: Demo

```
def sign(x):
    """ Return -1 if x < 0,
                1 if x > 0,
             or 0 if x == 0 """
    # code here
```

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)

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.

(of any types!)

```
(1, 4, “Mufasa”)
```

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

```
v = (1, 4, "Mufasa") # packing
(a, b, c) = v # "unpacking"
```

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
```

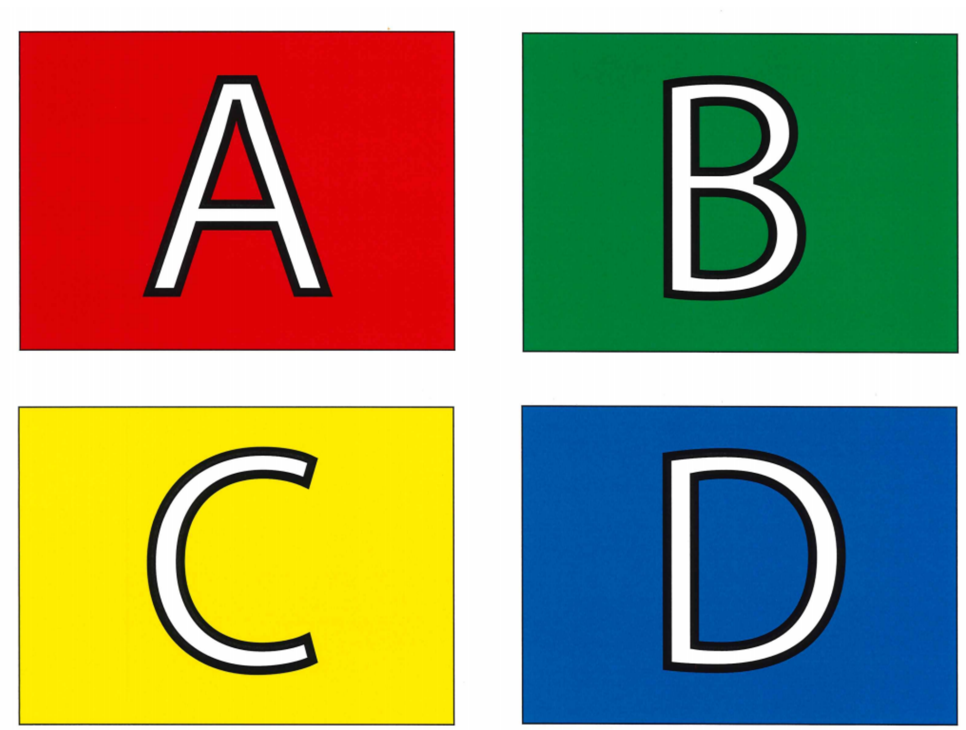
Tuples: Demo

Tuples: Demo

- assignment, packing, unpacking
- with and without parens (printing)
- swapping
- equality
- mismatched # values to unpack

ABCD: Tuples

See code example (tuples_abcd.py)



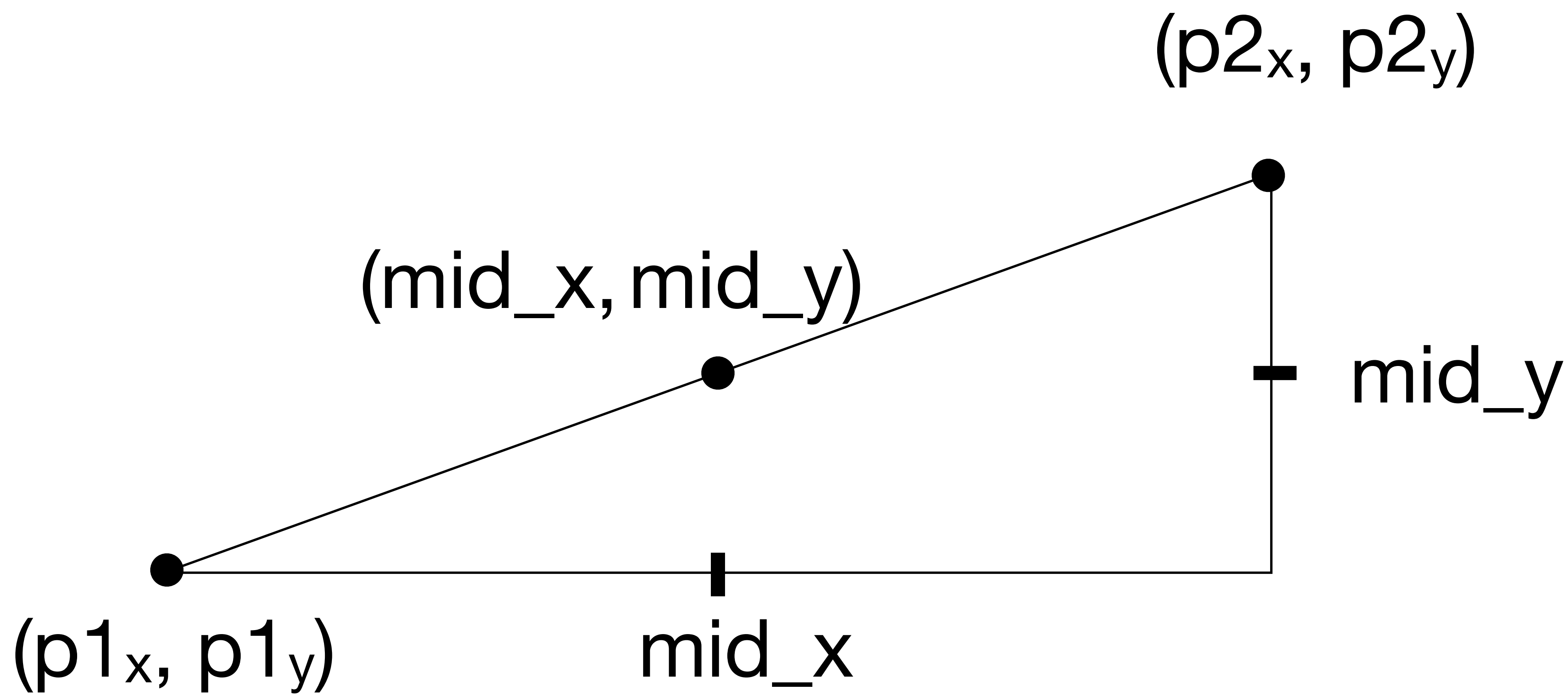
Midpoint Function

```
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
```

Midpoint Function

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

Okay, but how do you actually calculate this?



(on the board)

mid_x = $(p1_x + p2_x) / 2$

mid_y = $(p1_y + p2_y) / 2$

Demo: writing the midpoint function

- With tuple as return value
- Switch to tuples as parameters for points

Why write functions?

- The convenience of repetition:
  - you can define a function once then call it as many times as you want
  - Example: using **turtle_square** to create a snowflake
- The power of *customized* repetition:
  - you can define a function that takes arguments to customize the task it performs: this is powerful!
  - Example: using **draw_polygon** to draw an any-sided polygon
- The clarity of abstraction via function composition.
  - We can hide complexity behind simple function calls to make complicated calculations easier to think about and write.

Function Composition

Here's a made-up equation:

```
final_result = (a + b)**2 - d // 12 + (a**2 - 0.5*a*c)  + alpha * (dx**2 + dy**2)
```

It's pretty incomprehensible, even if you do know what a, b, d, c, alpha, dx, and dy mean.

Here's a nicer way to write it:

```
x = (a + b)**2 - d // 12
y = (a**2 - 0.5*a*c)
z = alpha * (dx**2 + dy**2)

final_result = x + y + z
```

Function Composition

Here's a made-up equation:

```
final_result = (a + b)**2 - d // 12 + (a**2 - 0.5*a*c)  + alpha * (dx**2 + dy**2)
```

What if x, y, and z weren't expressions, but more complicated computation requiring (for example) `for` loops to compute?

```
def calc_x(a, b, d):
    # calculation of x

def calc_y(a, c):
    # calculation of y

def calc_z(alpha, dx, dy):
    # calculation of z

x = calc_x(a, b, d)
y = calc_y(a, c)
z = calc_z(alpha, dx, dy)
final_result = x + y + z
```

Function Composition

Here's a made-up equation:

```
final_result = (a + b)**2 - d // 12 + (a**2 - 0.5*a*c)  + alpha * (dx**2 + dy**2)
```

What if x, y, and z weren't expressions, but more complicated computation requiring (for example) `for` loops to compute?

What if this is just an intermediate result that goes into an even **larger** calculation?

```
def calc_x(a, b, d):
    # calculation of x

def calc_y(a, c):
    # calculation of y

def calc_z(alpha, dx, dy):
    # calculation of z

x = calc_x(a, b, d)
y = calc_y(a, c)
z = calc_z(alpha, dx, dy)
intermediate_result = x + y + z
```

Function Composition

Here's a made-up equation:

```
final_result = (a + b)**2 - d // 12 + (a**2 - 0.5*a*c)  + alpha * (dx**2 + dy**2)
```

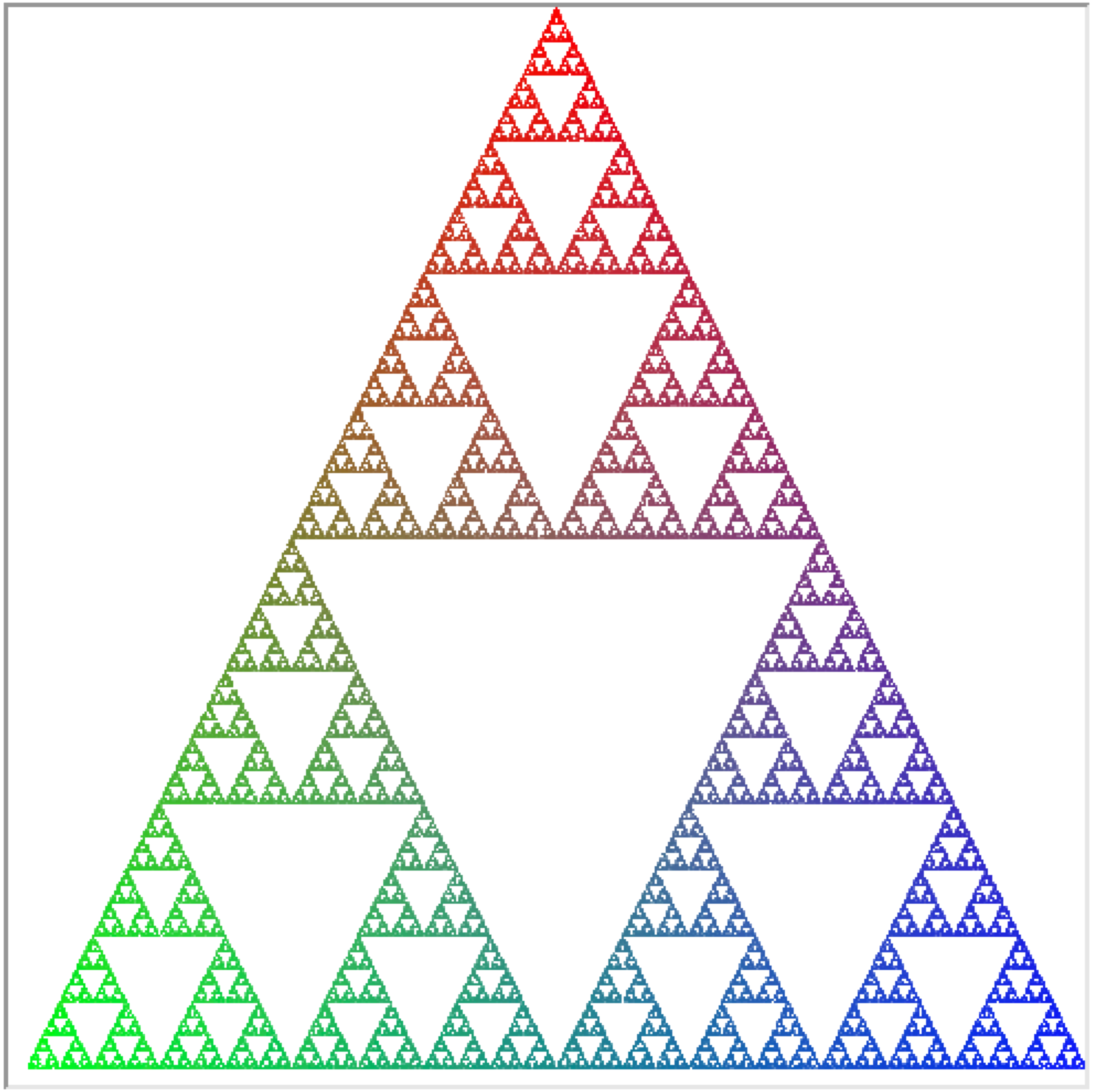
What if x, y, and z weren't expressions, but more complicated computation requiring (for example) `for` loops to compute?

What if this is just an intermediate result that goes into an even **larger** calculation?

```
def calc_x(a, b, d):
    # calculation of x

def calc_y(a, c):
    # calculation of y

def calc_z(alpha, dx, dy):
    # calculation of z

def calc_gamma(a,b,c,d,alpha,dx,dy):
    x = calc_x(a, b, d)
    y = calc_y(a, c)
    z = calc_z(alpha, dx, dy)
    return x + y + z
```

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
```

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:

- solution in action
- making up function names