CSCI 141

Lecture 16

How to approach A4, or:
Managing Complexity with Functions

Tuples

Announcements

Announcements

- Midterm grades should be out by the end of the weekend.

Announcements

- Midterm grades should be out by the end of the weekend.
- I'm working my way through the mid-quarter eval feedback. I'll discuss findings on Monday.

Goals

- Understand the task assigned in A4 and how to approach it.
- Understand how to use function composition to express complicated computations as clearly and simply as possible.
- Understand the basic usage of tuples:
    - using tuples to return multiple values from a function
    - packing and unpacking via the assignment operator

First: An Apology

Last lecture, I told you a lie.

First: An Apology

How to Execute Function Calls

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

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

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

1. Evaluate all arguments
2. Draw a local "box" inside the global one*
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

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

*Global variables can be *read* but not *modified* unless you mark them as such using `global var_name` at the top of the function definition.

In this course, we will never modify global variables from inside a function and will only rarely read them.

1. Evaluate all arguments
2. Draw a local "box" inside the global one*
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

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

*Global variables can be *read* but not *modified* unless you mark them as such using `global var_name` at the top of the function definition.

In this course, we will never modify global variables from inside a function and will only rarely read them.

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

*Unless the function is defined inside another function or class. This won't happen in this course.

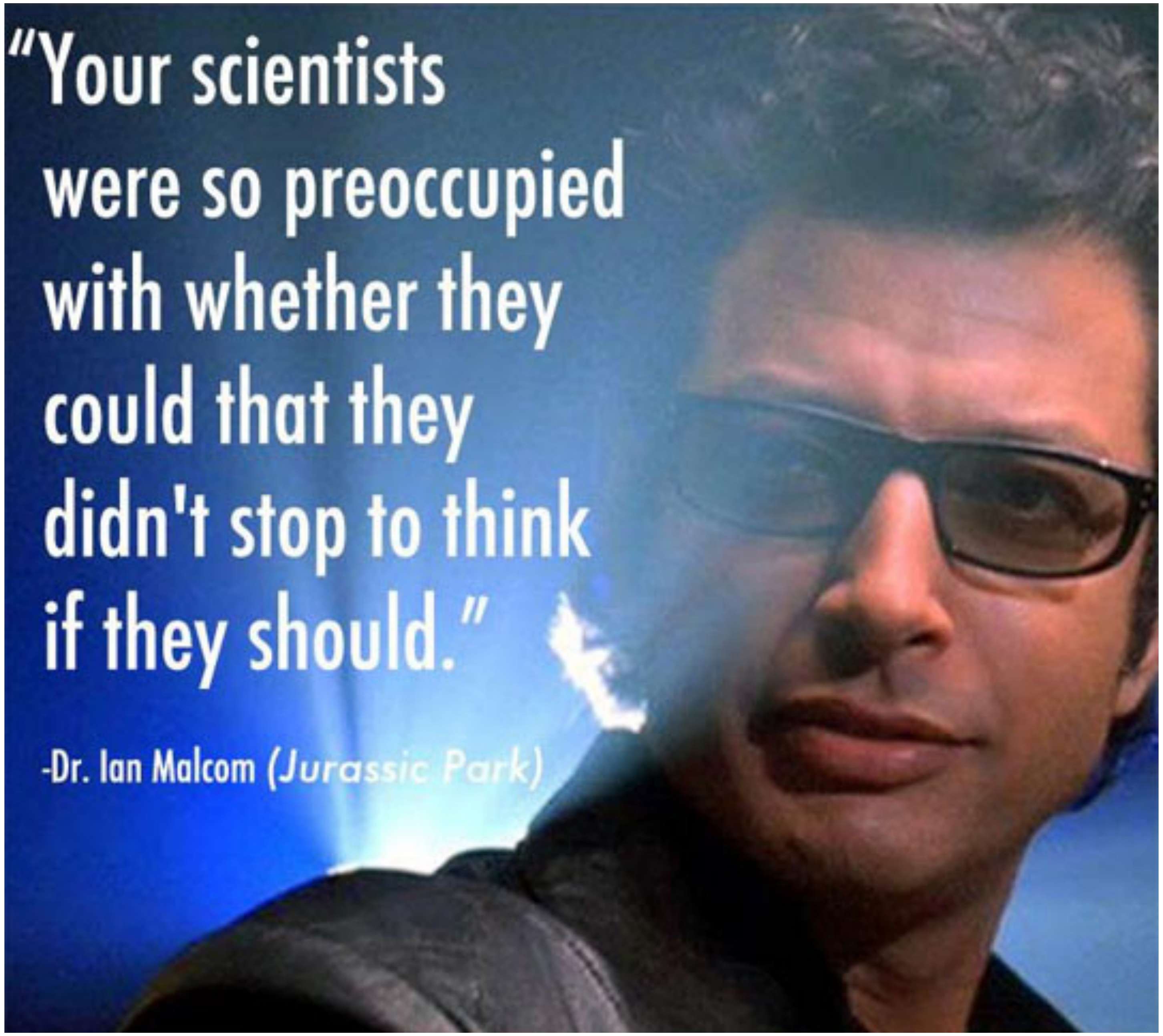
How did Scott get this wrong?

How did Scott get this wrong?

No excuses! Shame!

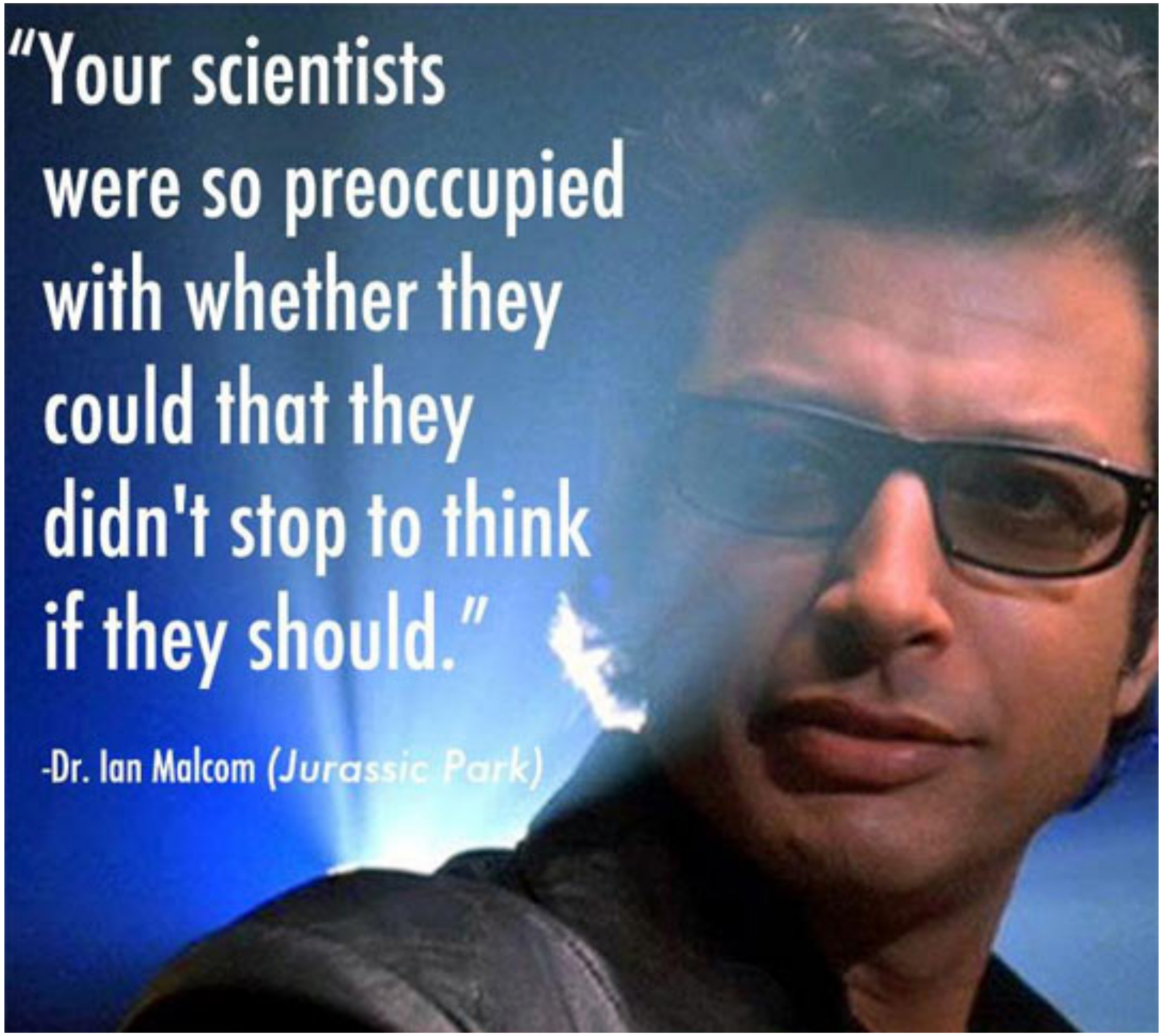
How did Scott get this wrong?

How did Scott get this wrong?



How did Scott get this wrong?

Your professor was so accustomed to doing what you **should** that he lost track of the details of what you **could**.



How did Scott get this wrong?

Your professor was so accustomed to doing what you **should** that he lost track of the details of what you **could**.

How did Scott get this wrong?

The specification of the Python language says:

- You **can** access variables that are not local to the function.

Conventional software engineering wisdom says:

- You **should not** access variables that are not local to the function.

Your professor was so accustomed to doing what you **should** that he lost track of the details of what you **could**.

How did Scott get this wrong?

The specification of the Python language says:

- You **can** access variables that are not local to the function.

Conventional software engineering wisdom says:

- You **should not** access variables that are not local to the function.

Your professor was so accustomed to doing what you **should** that he lost track of the details of what you **could**.

Bottom line: If your function needs a piece of data, that data should be passed in as an argument.

Why is accessing globals bad?

- The function's behavior becomes unpredictable, because it depends on global state.
- "Pure" functions are ideal: the output is fully determined by the inputs.

Bottom line: If your function needs a piece of data, that data should be passed in as an argument.

QOTD

Which of the following belongs in a function's docstring? Select all that apply.

- Preconditions
- Postconditions
- The steps that the function takes to accomplish its task
- Information about any side-effects the function has
- Information about what arguments the function takes

One more modification

To execute a function call:

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

We now know how to return a value - what does Python do with it?

One more modification

To execute a function call:

1. Evaluate all arguments
2. Draw a local "box" inside the global one
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box
6. **Replace the function call with its return value**

We now know how to return a value - what does Python do with it?

QOTD

To execute a function call:

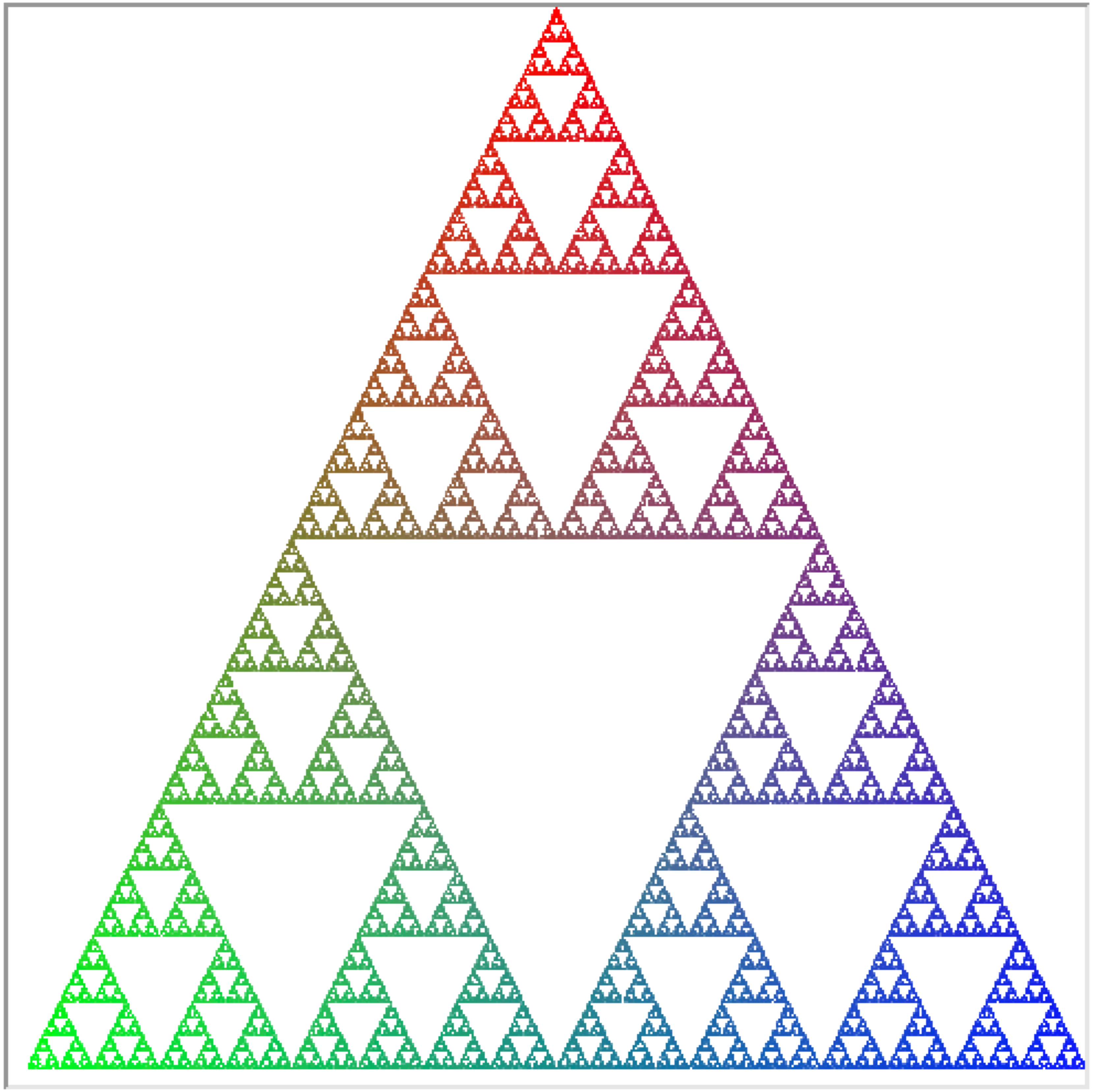
1. Evaluate all arguments
2. Draw a local "box" inside the global one
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box
6. **Replace the function call with its return value**

What does this program print?

```
x = 4

def f(x):
    return 3 * x

def g(x):
    return x + 2

print(f(g(x)))
print(g(f(x)))
```

A4

Your task:
Draw this.



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: 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
```

Demo: break this down into manageable pieces by inventing functions that solve pieces of the problem!

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
```

Midpoint Function

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

Midpoint Function

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

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 = . . .

    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

Tuples - 1

```
a = 1
b = 2
c = 3

v = (a, a, c)

print(v, sep=" ")

# What does this print?
# A: 1 2 3
# B: 1 1 3
# C: (1, 2, 3)
# D: (1, 1, 3)
```

Tuples - 2

```
a = 1
b = 2
c = 3

a, b, c = (a, a, c)

print(a, b, c, sep=" ")

# What does this print?
# A: 1 2 3
# B: 1 1 3
# C: (1, 2, 3)
# D: (1, 1, 3)
```

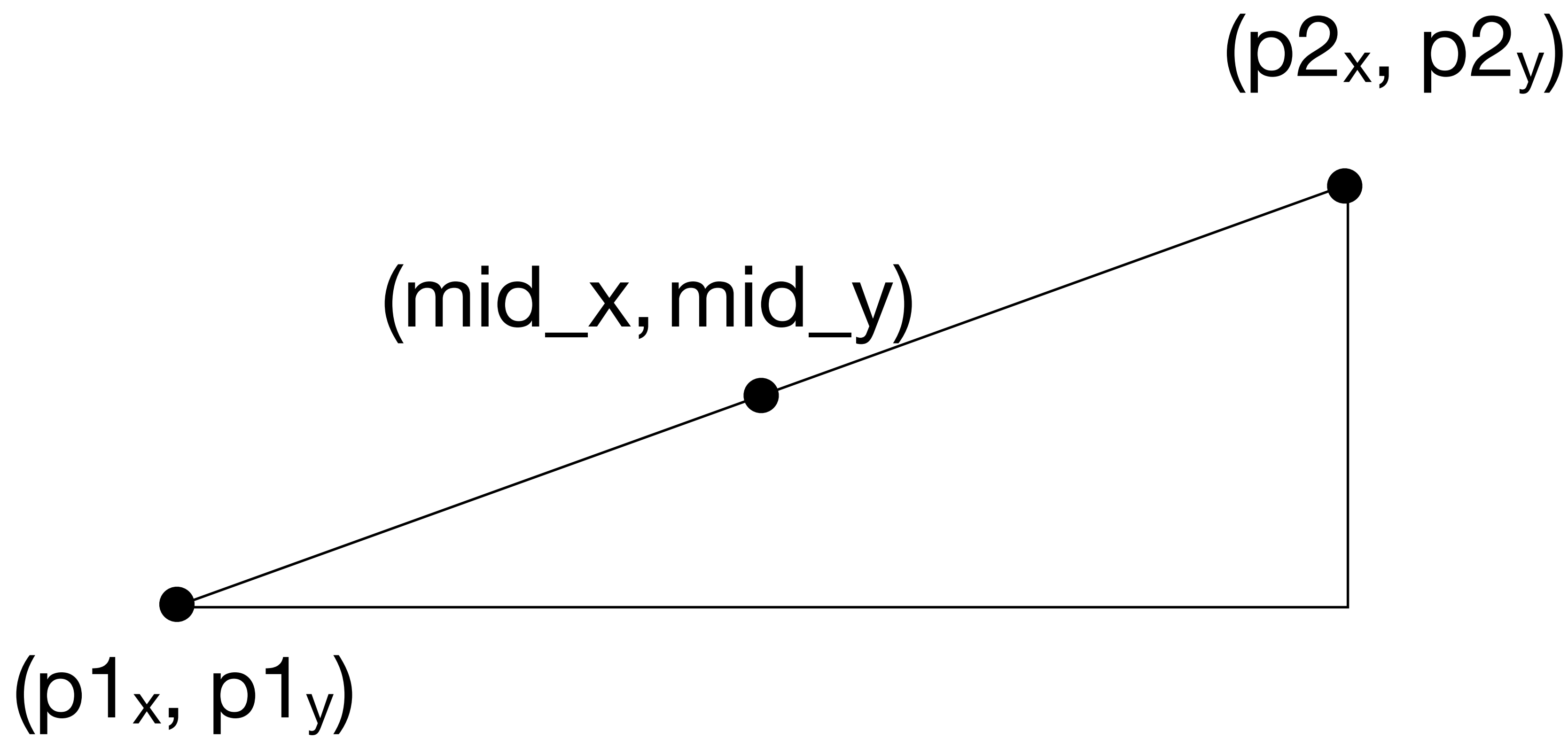
Midpoint Function

```
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 = . . .
```

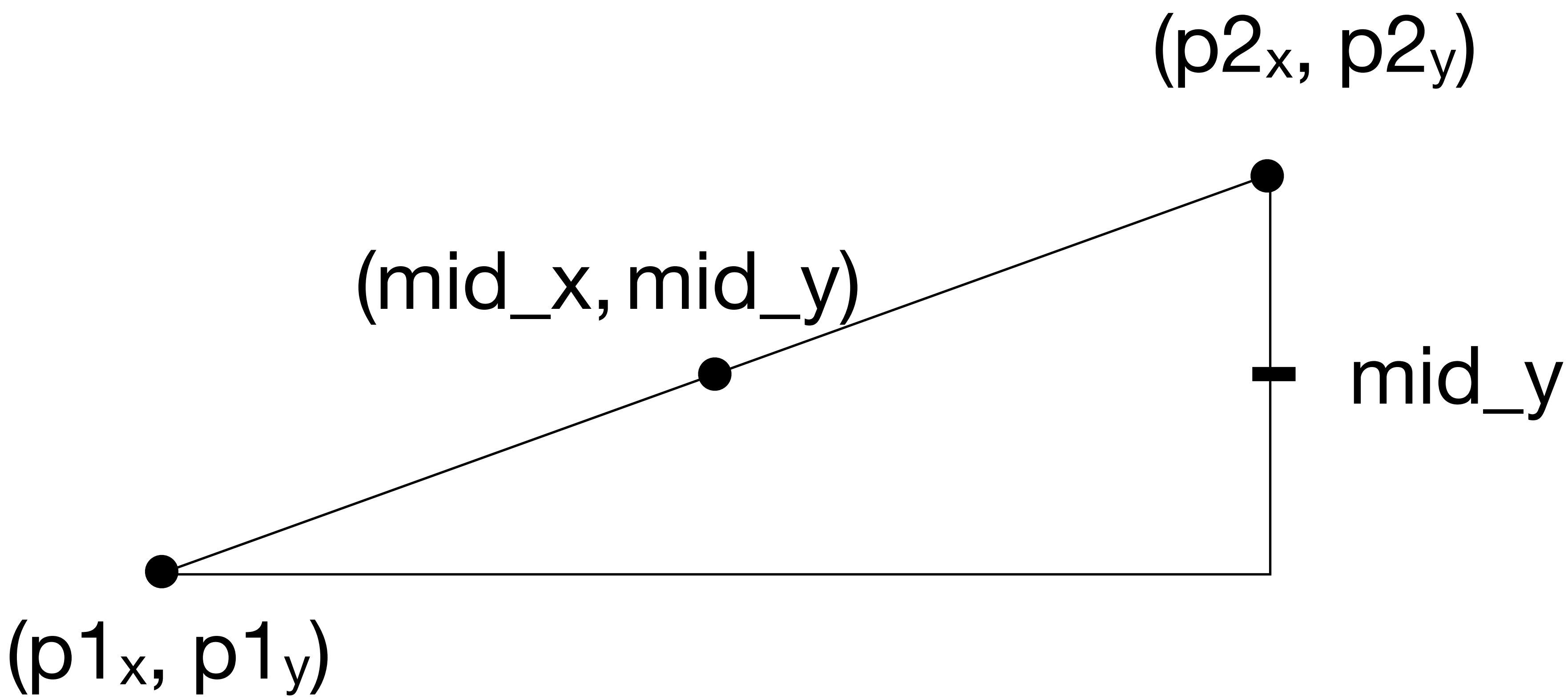
Okay, but how do you actually calculate this?



Midpoint Function

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

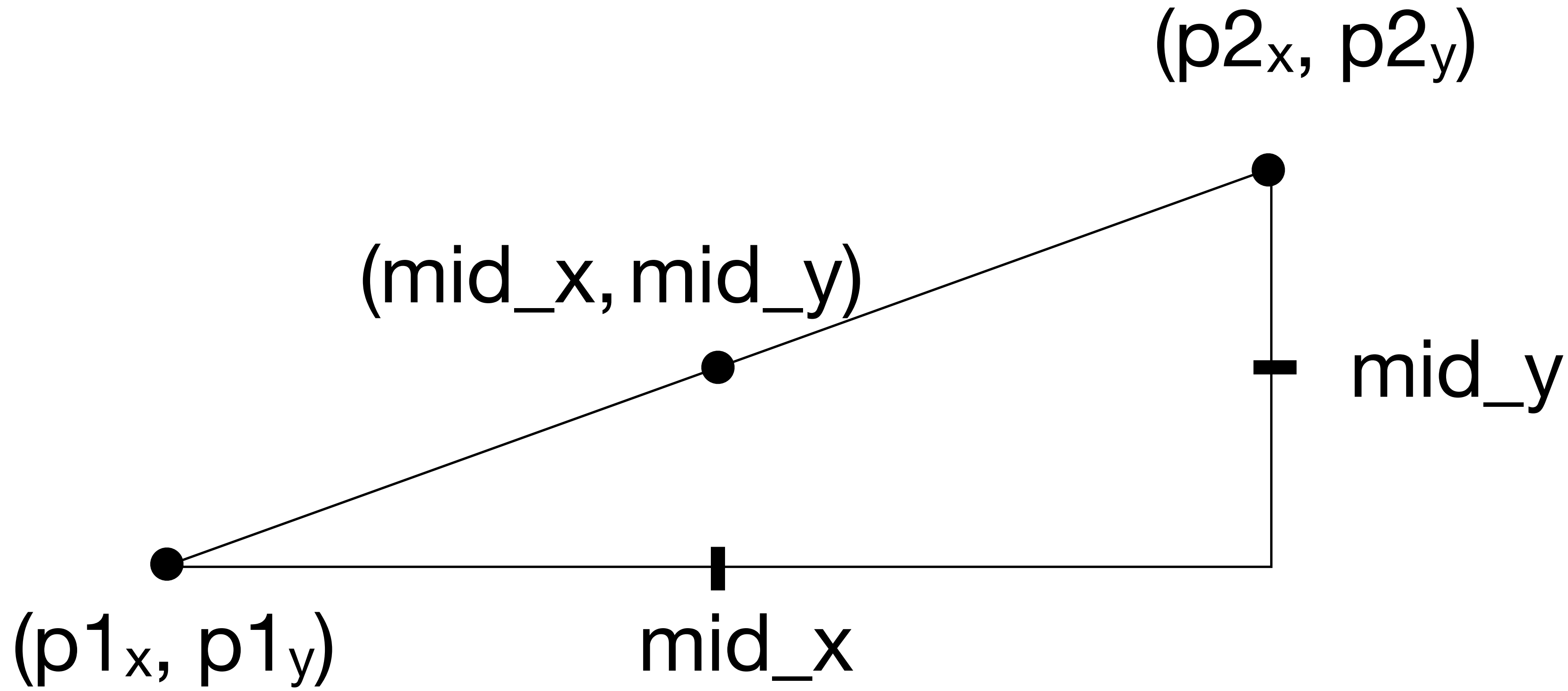
Okay, but how do you actually calculate this?



Midpoint Function

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

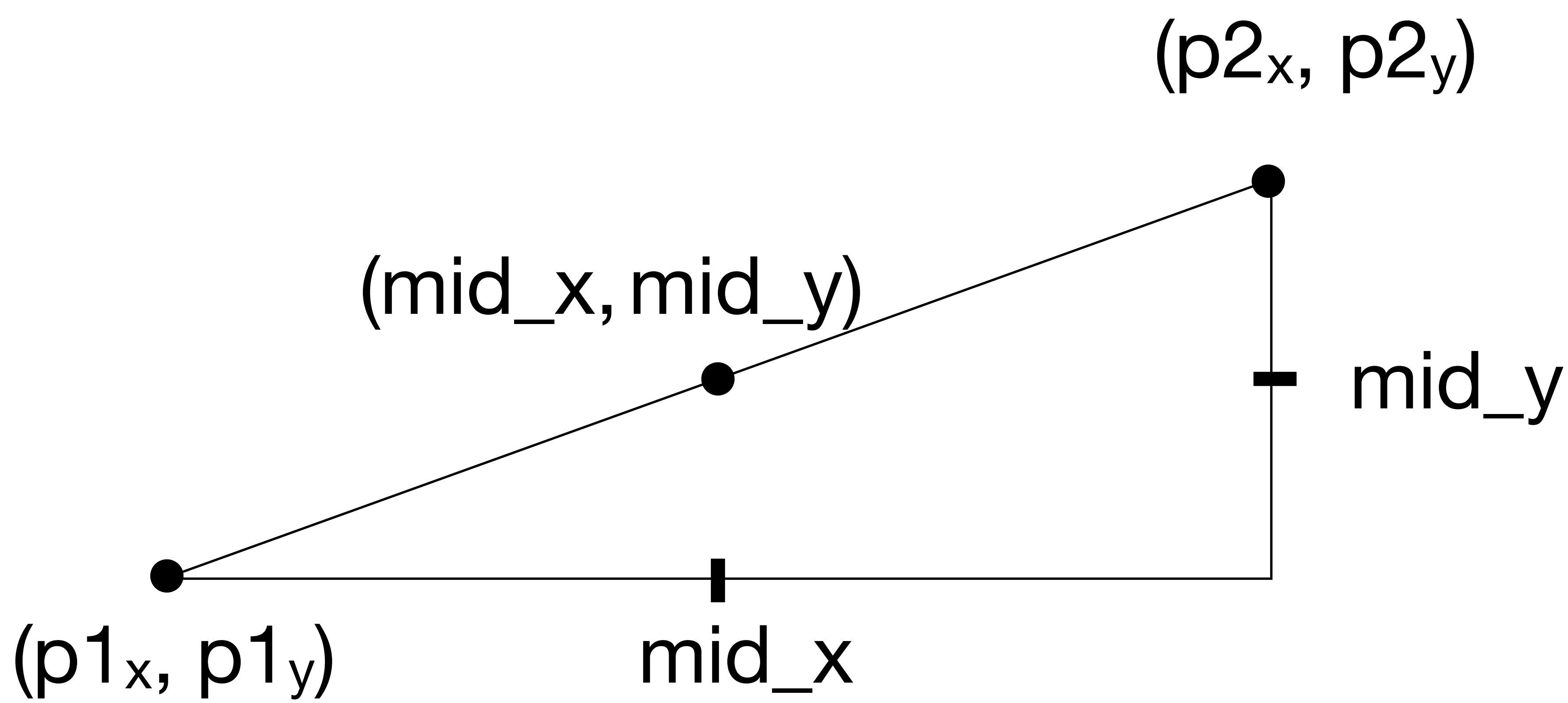
Okay, but how do you actually calculate this?



Midpoint Function

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

Okay, but how do you actually calculate this?

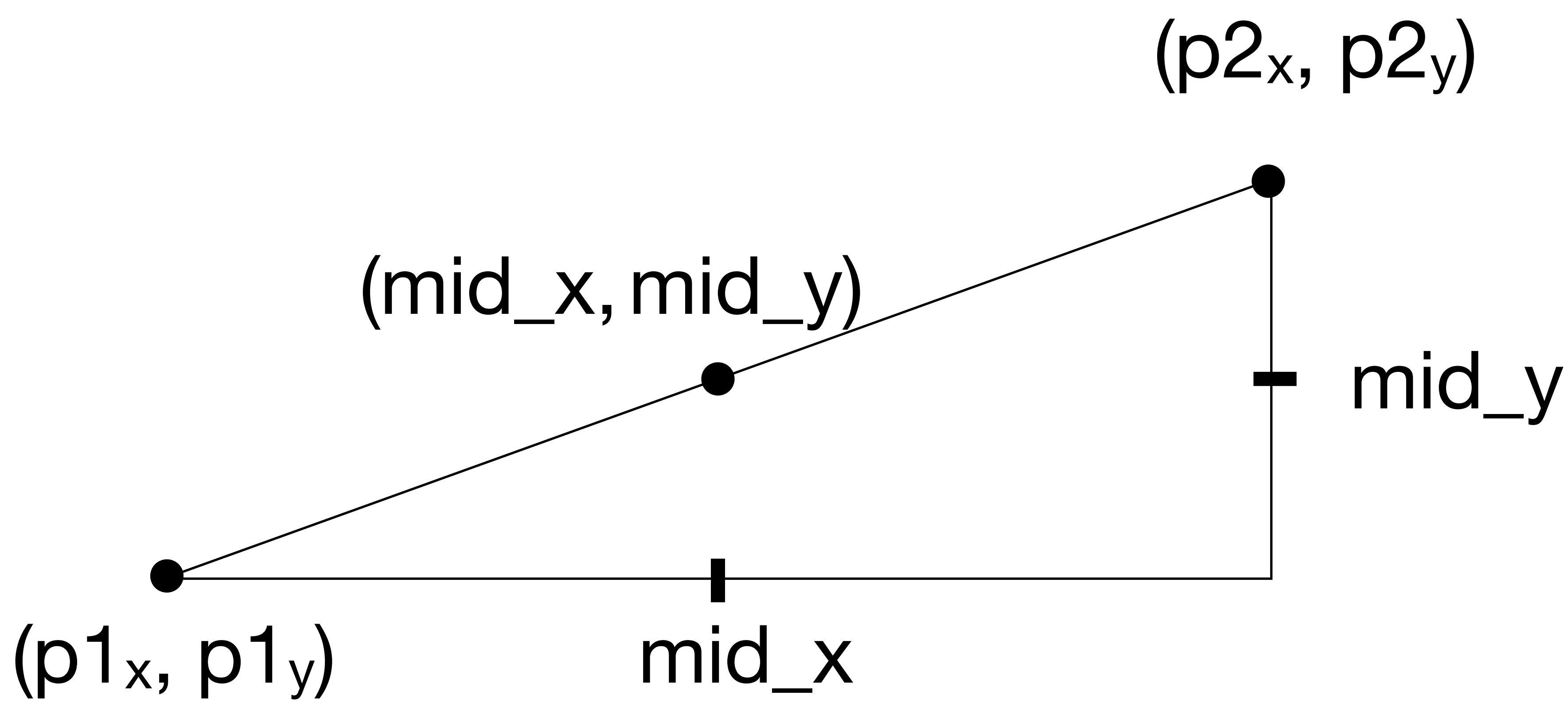


(on the board)

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

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
```

Color is chosen based on distance from each corner.
(details on the handout)

Subproblem: compute the distance between two points.

Exercise: Implement This

on paper!

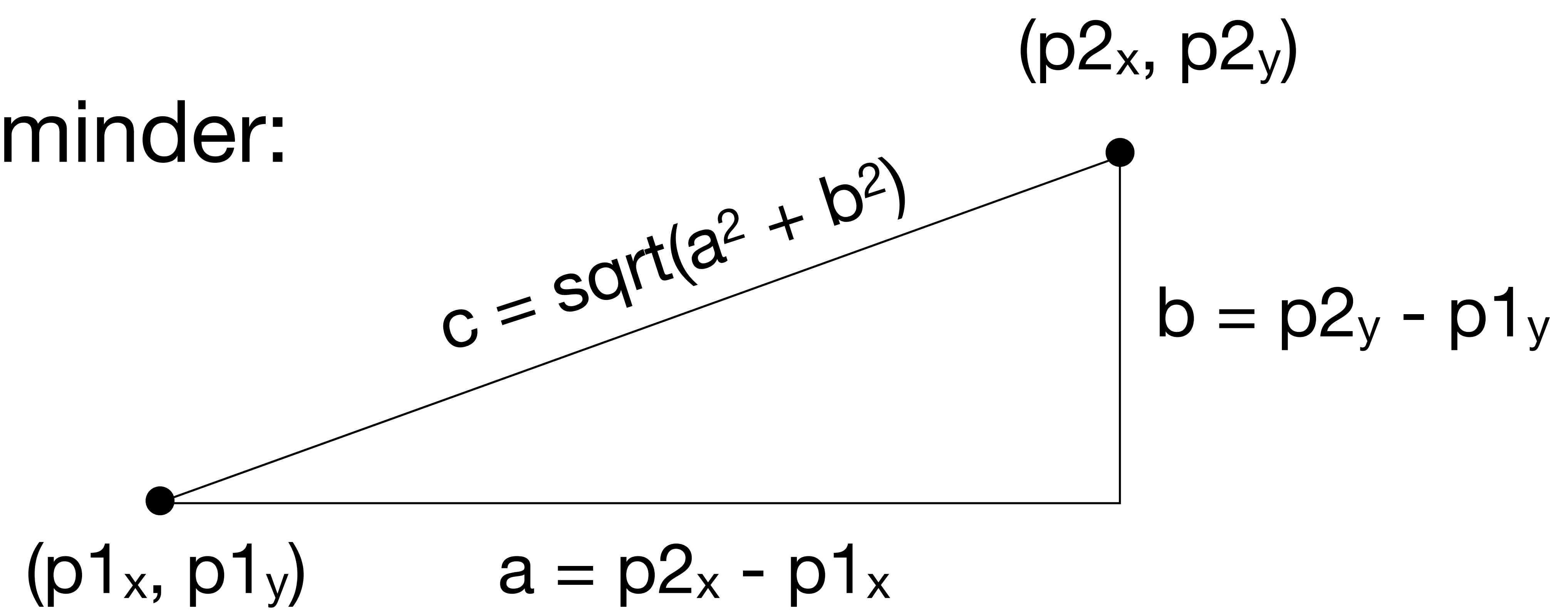
```
def distance(p1x, p1y, p2x, p2y):
    """ Return the distance between p1 and p2,
        which are points with coordinates
        (p1x, p1y) and (p2x, p2y)"""
```

Exercise: Implement This

on paper!

```
def distance(p1x, p1y, p2x, p2y):
    """ Return the distance between p1 and p2,
        which are points with coordinates
        (p1x, p1y) and (p2x, p2y)"""
```

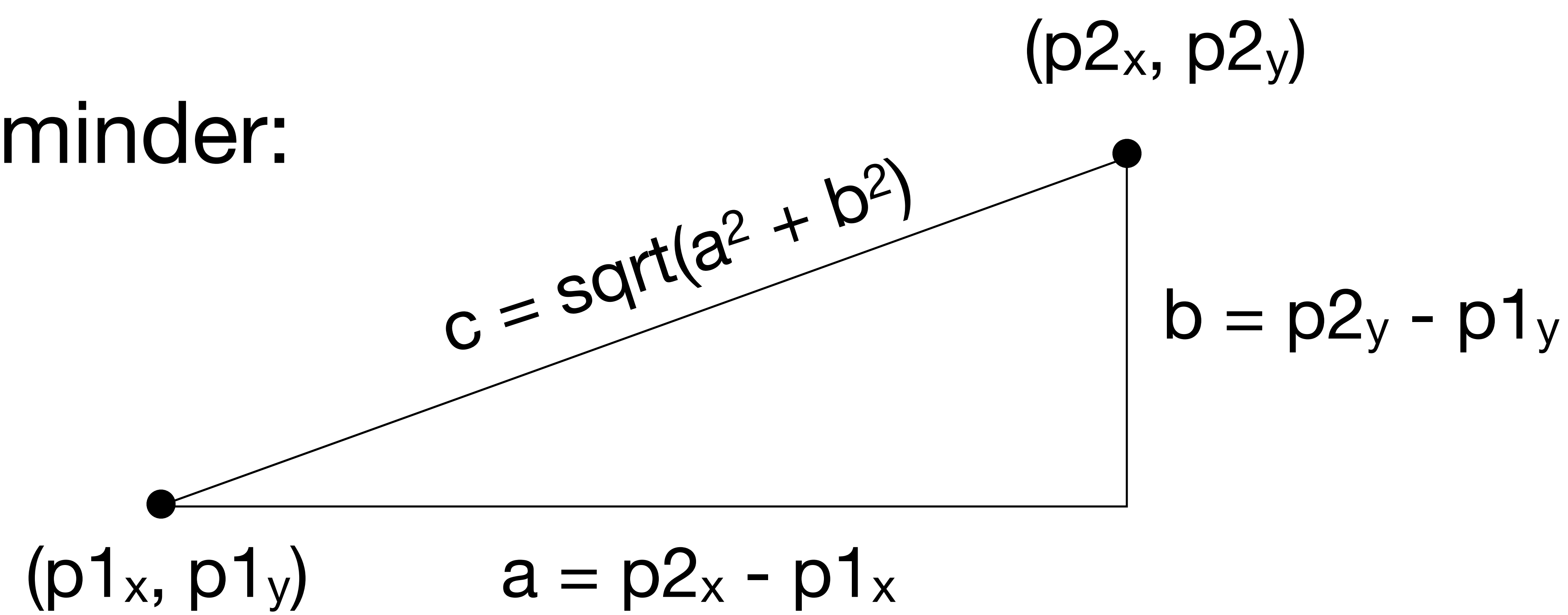
Math reminder:



Demo: Distance Function

```
def distance(p1x, p1y, p2x, p2y):
    """ Return the distance between p1 and p2,
        which are points with coordinates
        (p1x, p1y) and (p2x, p2y)"""
```

Math reminder:



Reminder: Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.

    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

Reminder: Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.

    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

Reminder: Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.

    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

```
>>> split_bill(34.78, 18.0, 0)
```

Reminder: Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.

    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

```
>>> split_bill(34.78, 18.0, 0)
ZeroDivisionError: float division by zero
```

Reminder: Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.

    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

```
>>> split_bill(34.78, 18.0, 0)
ZeroDivisionError: float division by zero
```

Bad news:

Reminder: Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.

    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

```
>>> split_bill(34.78, 18.0, 0)
ZeroDivisionError: float division by zero
```

Bad news: This is your fault.

Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.
        Precondition: num_diners > 0
    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.
        Precondition: num_diners > 0
    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.
        Precondition: num_diners > 0
    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

```
>>> split_bill(34.78, 18.0, 0)
```

Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.
        Precondition: num_diners > 0
    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

```
>>> split_bill(34.78, 18.0, 0)
ZeroDivisionError: float division by zero
```

Docstrings, Preconditions and Postconditions

Example. Suppose you wrote this function:

```
def split_bill(bill_amt, tip_pct, num_diners):
    """ Return the total owed by each diner for a
        restaurant bill of bill_amt, assuming a tip
        percent of tip_pct and splitting the bill
        evenly among num_diners people.
        Precondition: num_diners > 0
    """
    total = bill_amt + (bill_amt * tip_pct/100)
    return total / num_diners
```

```
>>> split_bill(34.78, 18.0, 0)
ZeroDivisionError: float division by zero
```

This is my fault.

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)
```

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.

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:

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?

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?

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?

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
```