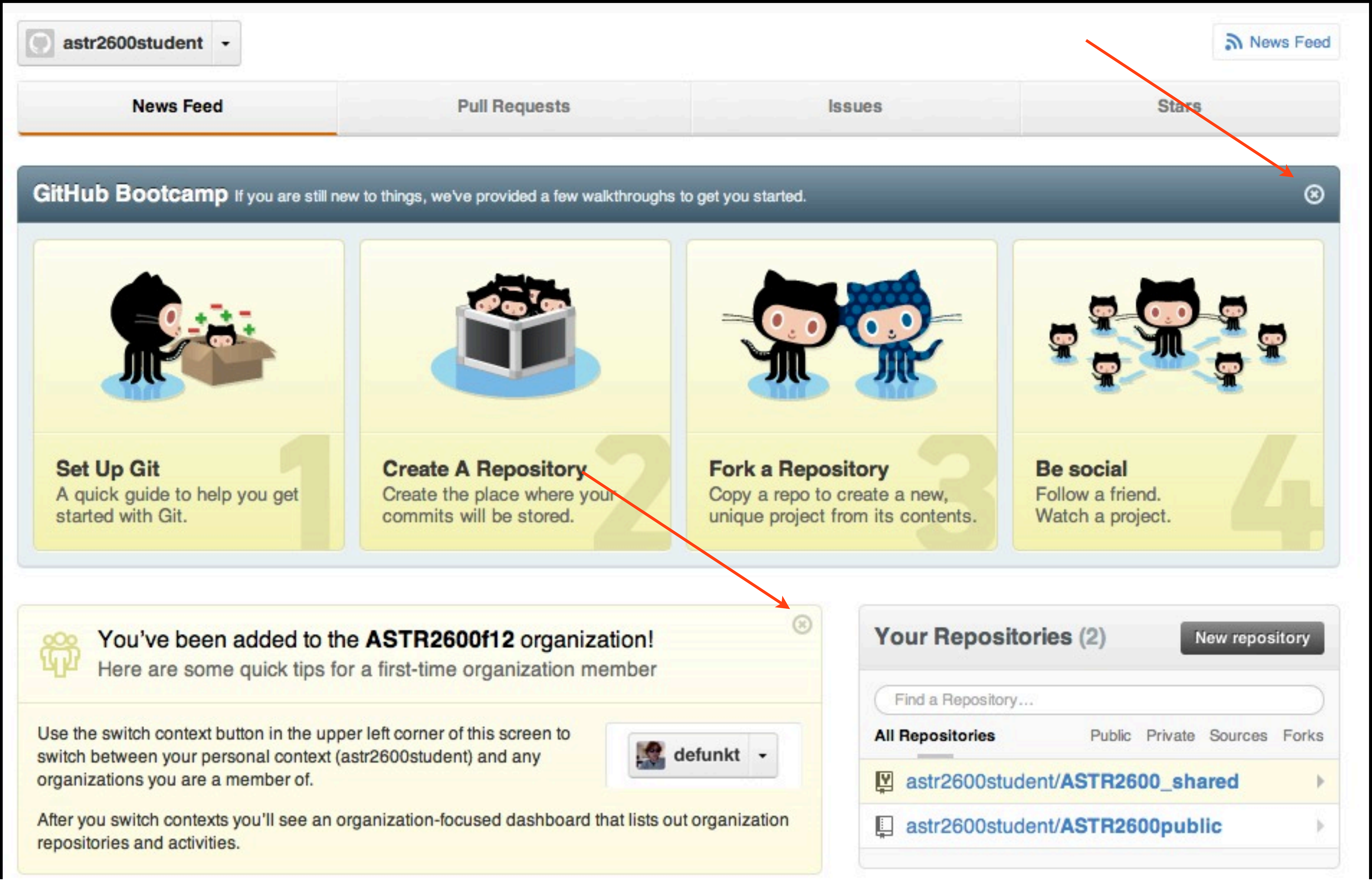
# Does your homepage look like this?



**GitHub Bootcamp** If you are still new to things, we've provided a few walkthroughs to get you started.

**Set Up Git** 1
A quick guide to help you get started with Git.

**Create A Repository** 2
Create the place where your commits will be stored.

**Fork a Repository** 3
Copy a repo to create a new, unique project from its contents.

**Be social** 4
Follow a friend. Watch a project.

You've been added to the **ASTR2600f12** organization!
Here are some quick tips for a first-time organization member

Use the switch context button in the upper left corner of this screen to switch between your personal context (astr2600student) and any organizations you are a member of.

After you switch contexts you'll see an organization-focused dashboard that lists out organization repositories and activities.

**Your Repositories** (2)

astr2600student/**ASTR2600_shared**

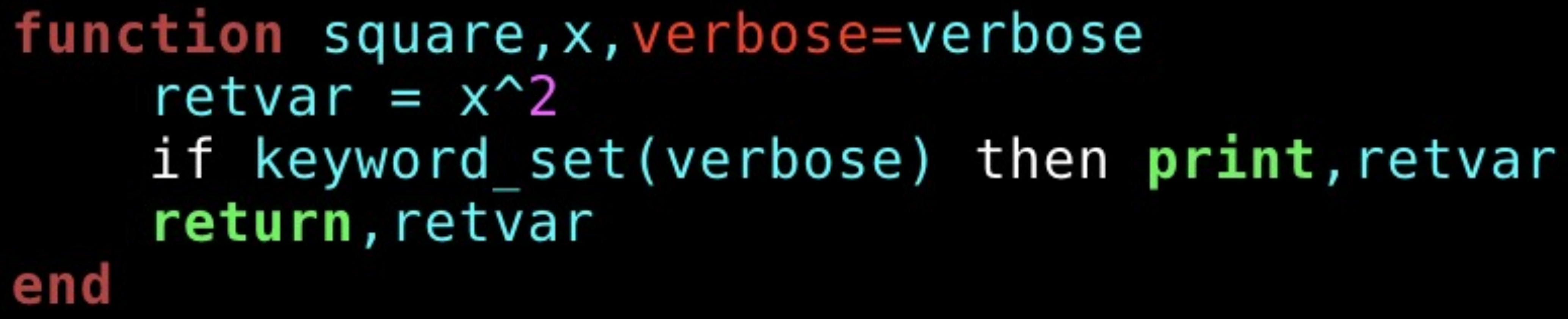
astr2600student/**ASTR2600public**

# Programs: Keywords

# Keyword Variables

- Optional “named” arguments
- Optional means the user doesn’t have to set them, so your code needs to know how to deal with empty keywords
- Often used as boolean flags

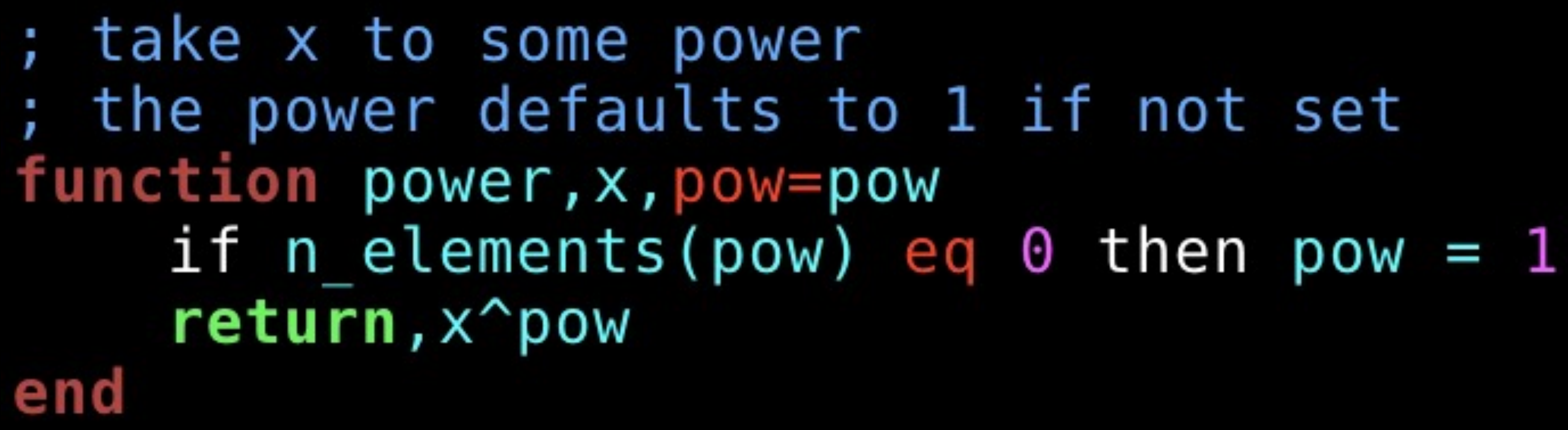
# Boolean Keyword



```
function square,x,verbose=verbose
    retvar = x^2
    if keyword_set(verbose) then print,retvar
    return,retvar
end
```

- print `retvar` only if `verbose` is set to non-zero
- keyword_set checks whether the keyword has been set
  - it recognizes *any non-zero* as True

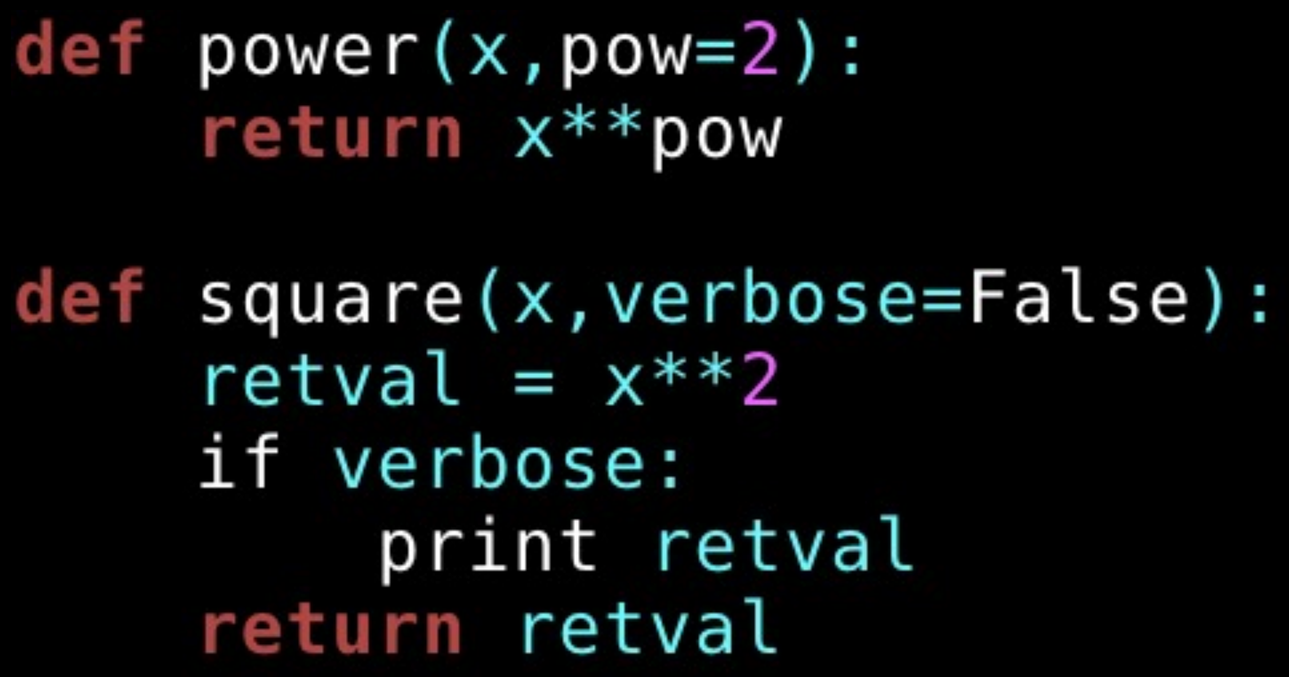
# Keyword



```
; take x to some power
; the power defaults to 1 if not set
function power,x,pow=pow
    if n_elements(pow) eq 0 then pow = 1
    return,x^pow
end
```

- `pow` is some number
- can’t use `keyword_set` because `0` is a valid power
- `if n_elements(keyword) eq 0 ...` sets a *default value* for the keyword

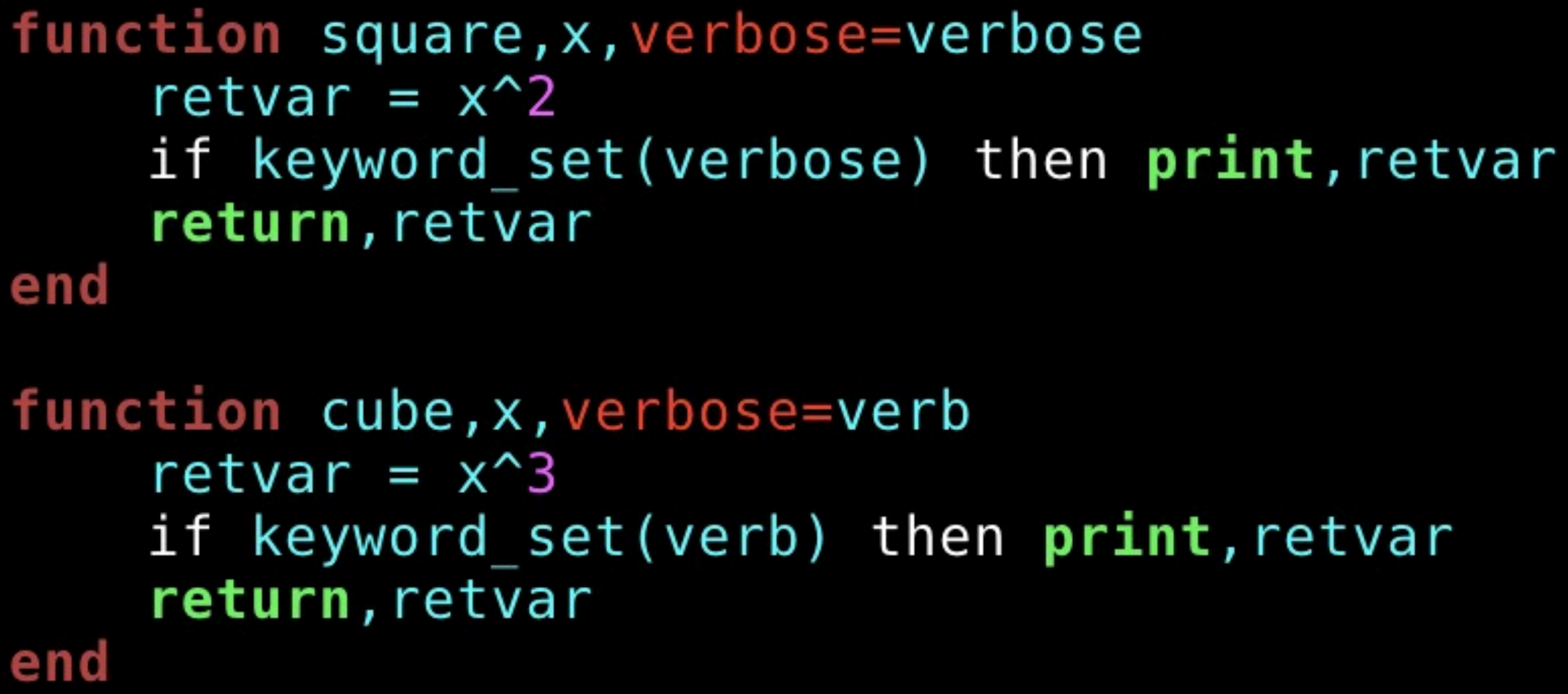
# Keywords in Python



```
def power(x,pow=2):
    return x**pow

def square(x,verbose=False):
    retval = x**2
    if verbose:
        print retval
    return retval
```

- Declare defaults along with the keyword name

# Keyword Oddities

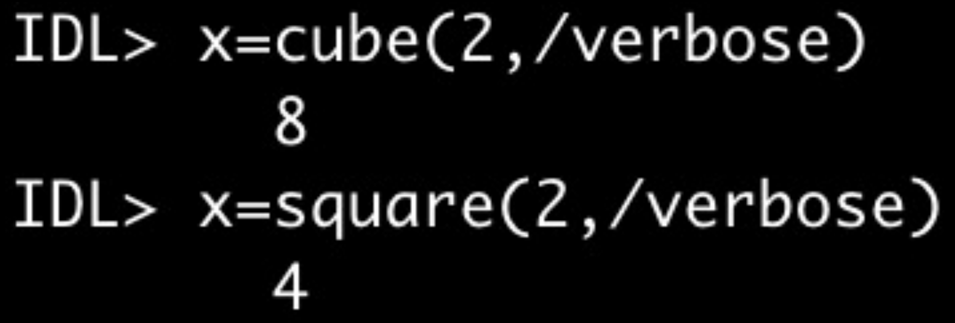


```
function square,x,verbose=verbose
    retvar = x^2
    if keyword_set(verbose) then print,retvar
    return,retvar
end

function cube,x,verbose=verb
    retvar = x^3
    if keyword_set(verb) then print,retvar
    return,retvar
end
```

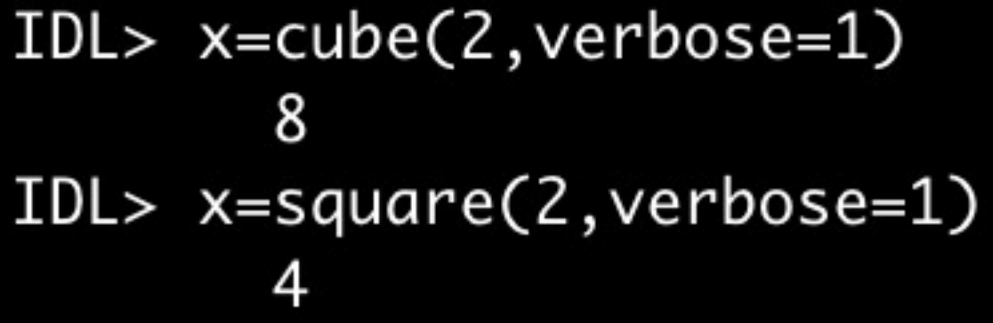
- The variable name is on the *right side* of the keyword declaration

# Keyword Oddities

- The variable name is on the *right side* of the keyword declaration
- but the keyword you call with is on the *left side*



```
IDL> x=cube(2,/verbose)
       8
IDL> x=square(2,/verbose)
       4
```



```
IDL> x=cube(2,verbose=1)
       8
IDL> x=square(2,verbose=1)
       4
```

# Debugging

# Bugs and Errors

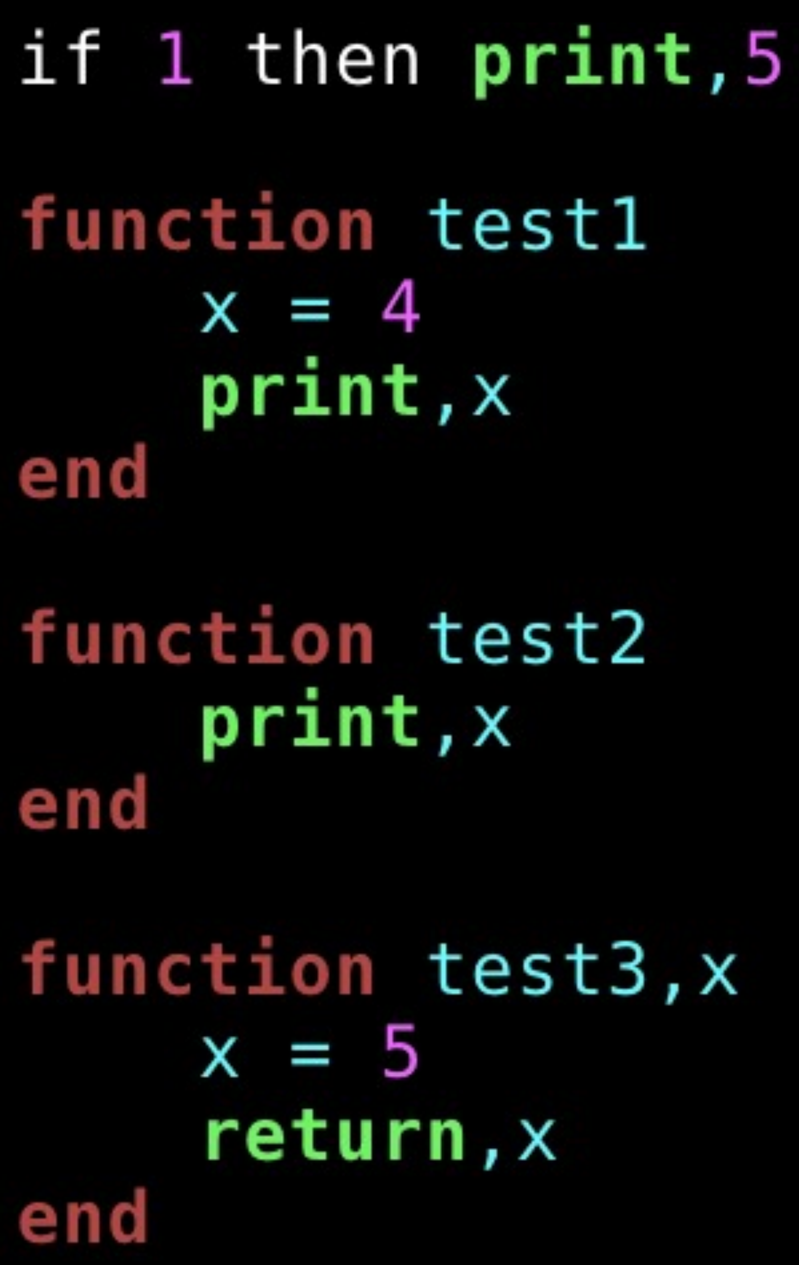
- Errors are things that IDL knows are wrong
    - syntax errors - caught at compile time
    - “Variable is undefined” errors - caught at run time

# Bugs and Errors

- “Bugs” are things that cause undesired behavior, but aren’t wrong code
  - i.e., if the function `square(2)` returned 5, but the correct answer is 4, that is a bug
  - it “works” - it gives you output and doesn’t crash - but it works wrong.

# Errors

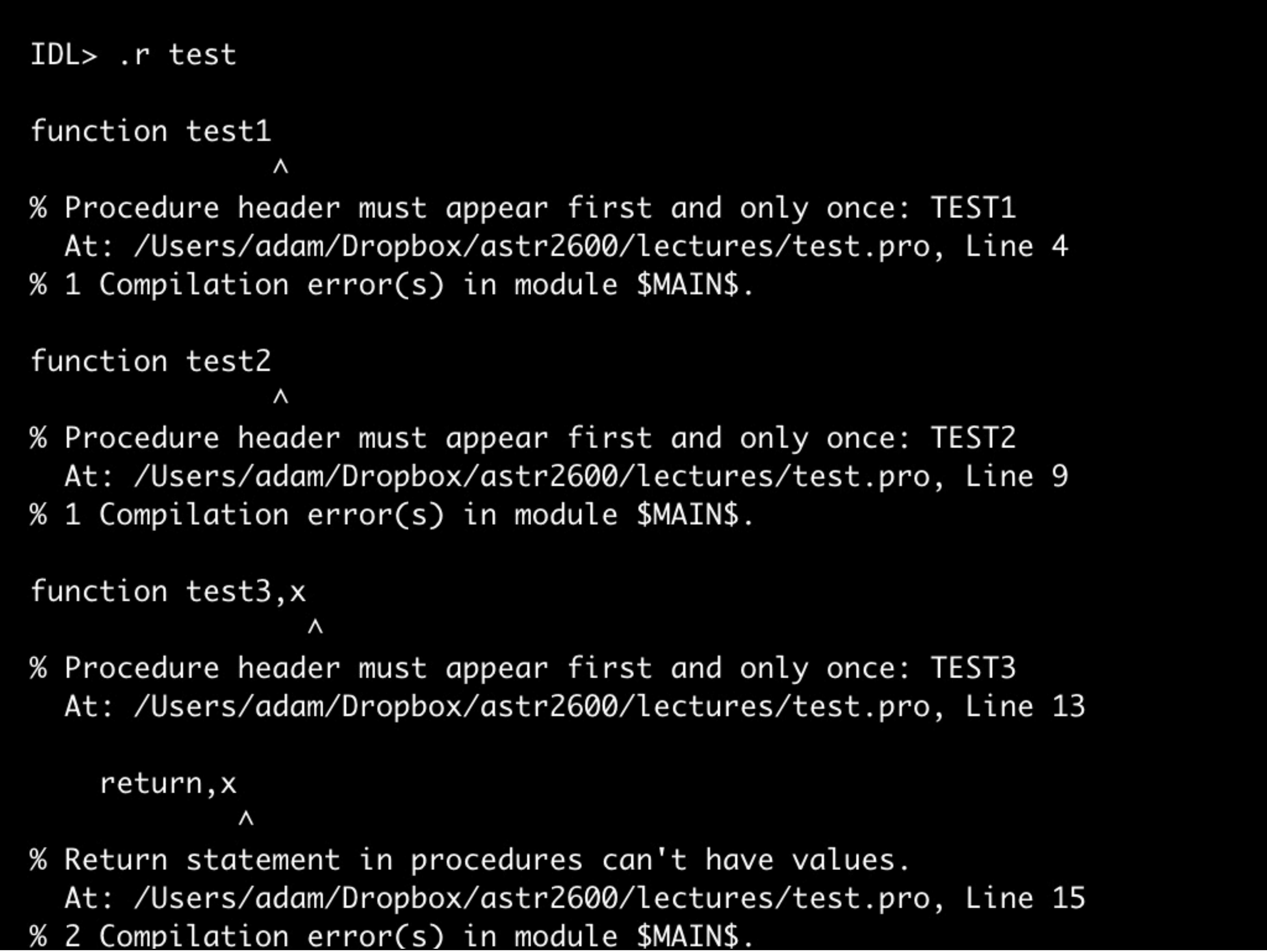
- Syntax errors are caught when you "compile" (`.compile` or `.run`) your code
- IDL will try to compile the rest of the code even after a syntax error!
  - This can result in a LONG list of errors!
  - *ALWAYS address the first error first!*



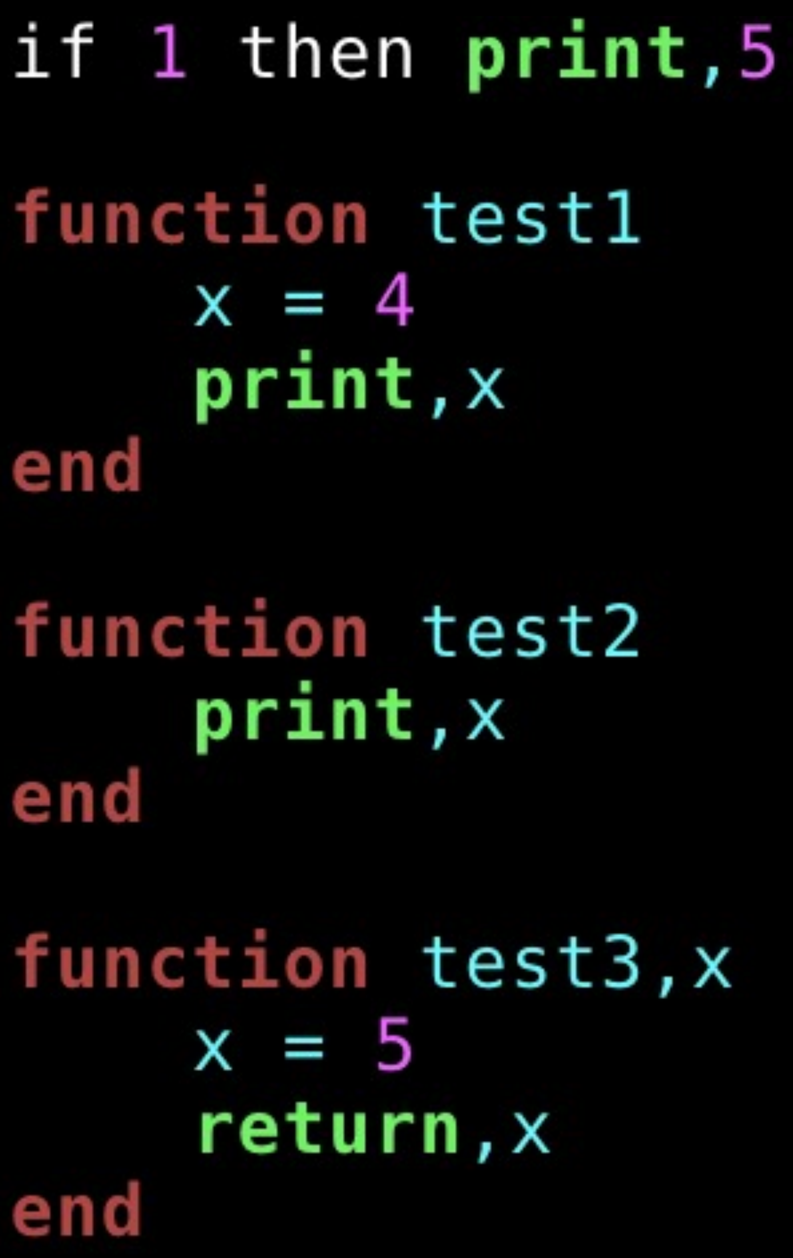
```
if 1 then print,5

function test1
    x = 4
    print,x
end

function test2
    print,x
end

function test3,x
    x = 5
    return,x
end
```

Here's a `.pro` file. I'll try to compile it and show you the errors.

All 3 of these functions are syntactically correct.



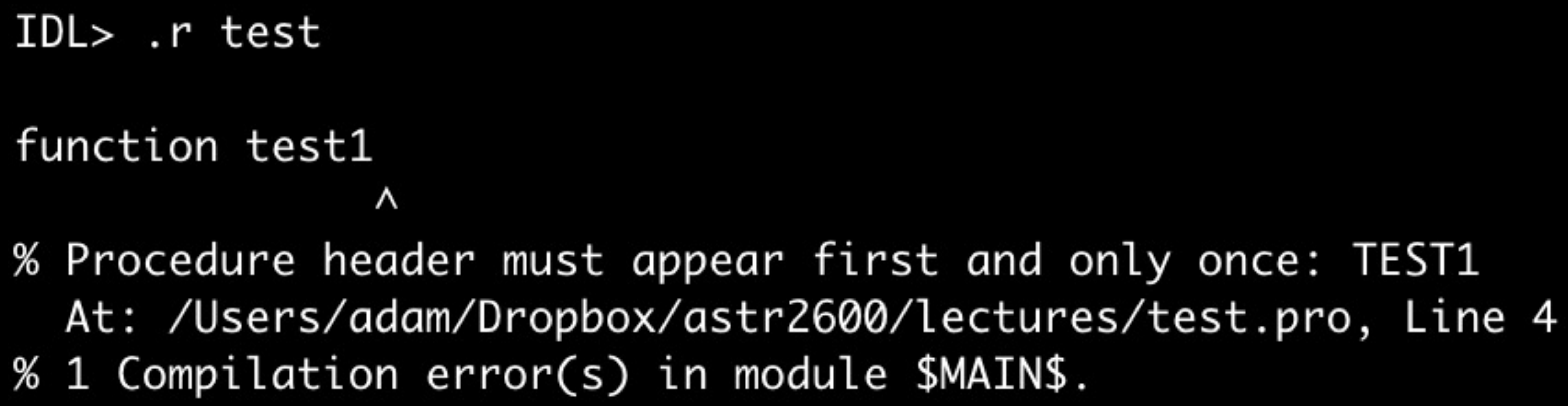
```
IDL> .r test

function test1
              ^
% Procedure header must appear first and only once: TEST1
  At: /Users/adam/Dropbox/astr2600/lectures/test.pro, Line 4
% 1 Compilation error(s) in module $MAIN$.

function test2
              ^
% Procedure header must appear first and only once: TEST2
  At: /Users/adam/Dropbox/astr2600/lectures/test.pro, Line 9
% 1 Compilation error(s) in module $MAIN$.

function test3,x
                ^
% Procedure header must appear first and only once: TEST3
  At: /Users/adam/Dropbox/astr2600/lectures/test.pro, Line 13

    return,x
            ^
% Return statement in procedures can't have values.
  At: /Users/adam/Dropbox/astr2600/lectures/test.pro, Line 15
% 2 Compilation error(s) in module $MAIN$.
```



```
if 1 then print,5

function test1
    x = 4
    print,x
end

function test2
    print,x
end

function test3,x
    x = 5
    return,x
end
```

This is the only real error (you’re not allowed to have anything but functions and procedures in a function/ procedure file)

...unless the “something else” is at the end, followed by `end`

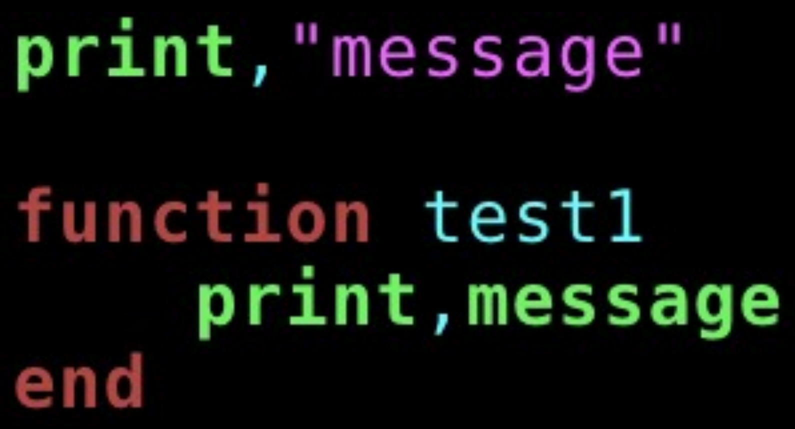


```
IDL> .r test

function test1
              ^
% Procedure header must appear first and only once: TEST1
  At: /Users/adam/Dropbox/astr2600/lectures/test.pro, Line 4
% 1 Compilation error(s) in module $MAIN$.
```

- The “compiler” tells you the closest location of an error, but does not make clear how to fix the error

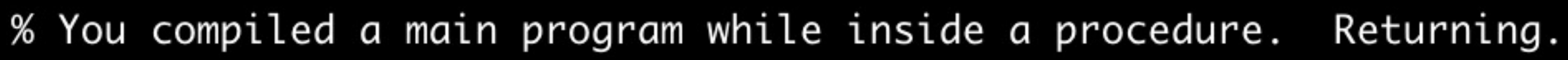
- This is one of the challenges of programming - figuring out what obscure error messages mean

  - also a great reason to make YOUR error messages explicit!

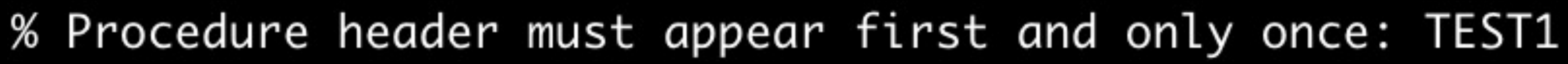


```
print,"message"

function test1
    print,message
end
```

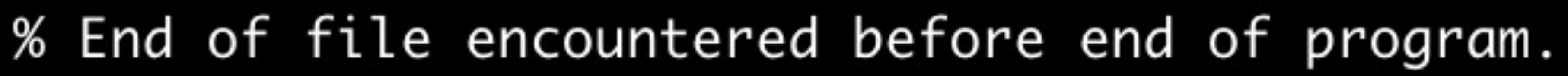
What error message will you get if you compile this file?

A) 

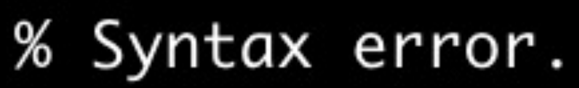
```
% You compiled a main program while inside a procedure.  Returning.
```

B) 

```
% Procedure header must appear first and only once: TEST1
```

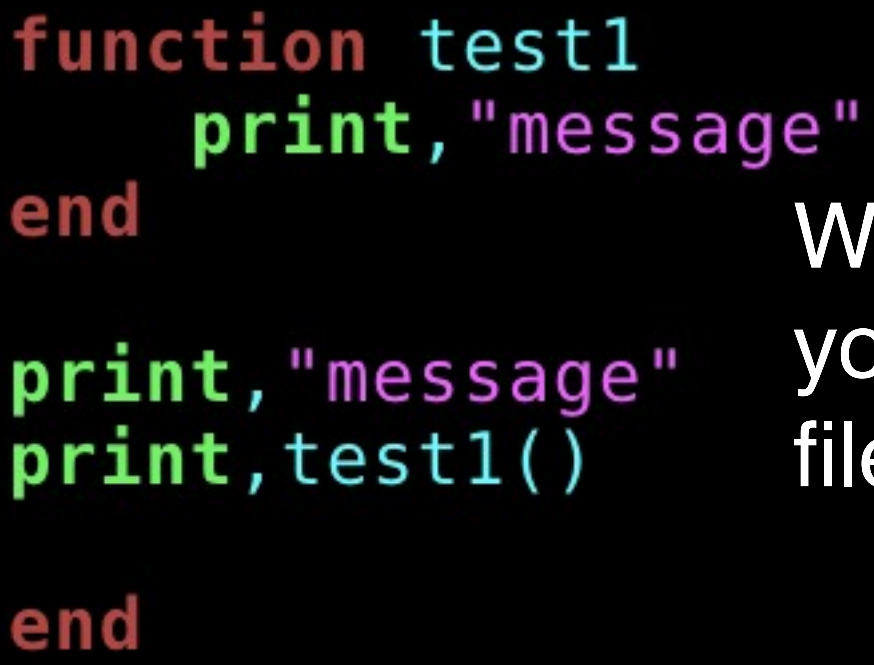
C) 

```
% End of file encountered before end of program.
```

D) 

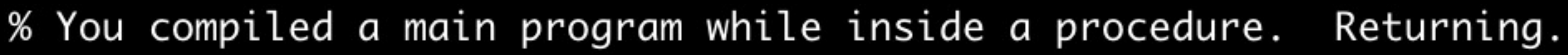
```
% Syntax error.
```

E) None of the above

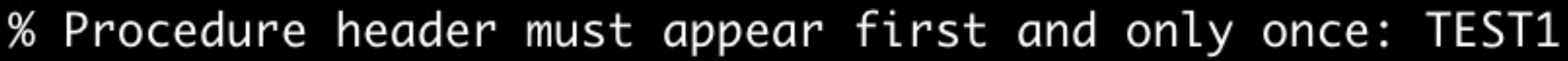


```
function test1
    print,"message"
end

print,"message"
print,test1()

end
```

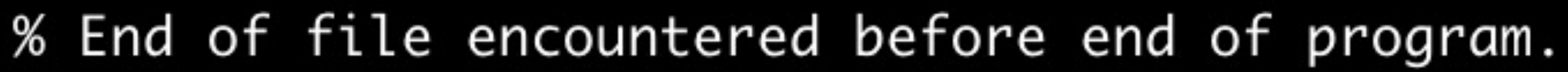
What error message will you get if you compile this file?

A) 

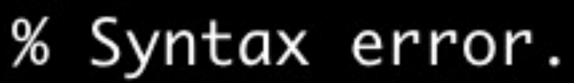
```
% You compiled a main program while inside a procedure.  Returning.
```

B) 

```
% Procedure header must appear first and only once: TEST1
```

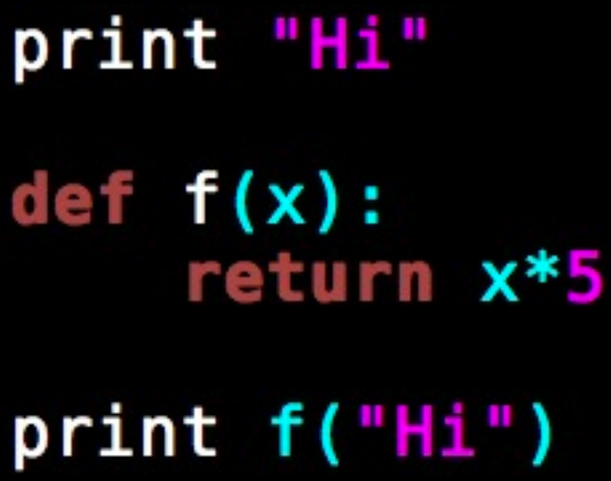
C) 

```
% End of file encountered before end of program.
```

D) 

```
% Syntax error.
```

E) None of the above



```
print "Hi"

def f(x):
    return x*5

print f("Hi")
```

Will this code work? (hint: `f(“Hi”)` does)

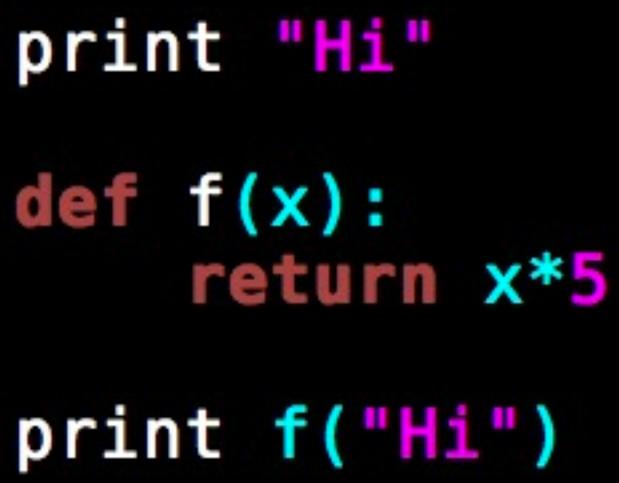
A) Yes

B) No, can’t have `prints` before and after the function definition

C) No, I don’t believe your hint, you can’t multiply a string by a number

D) Just no.

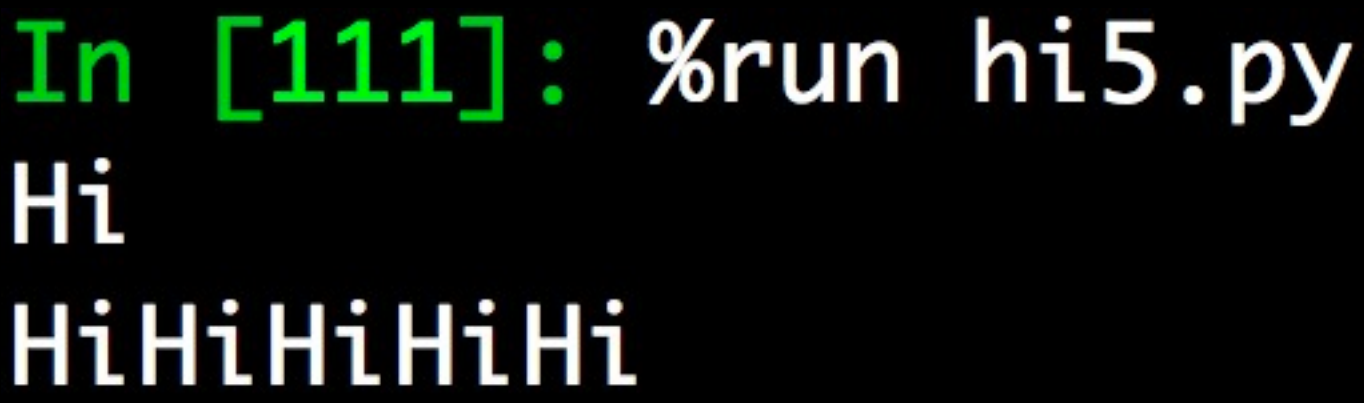
E) None of the above



```
print "Hi"

def f(x):
    return x*5

print f("Hi")
```

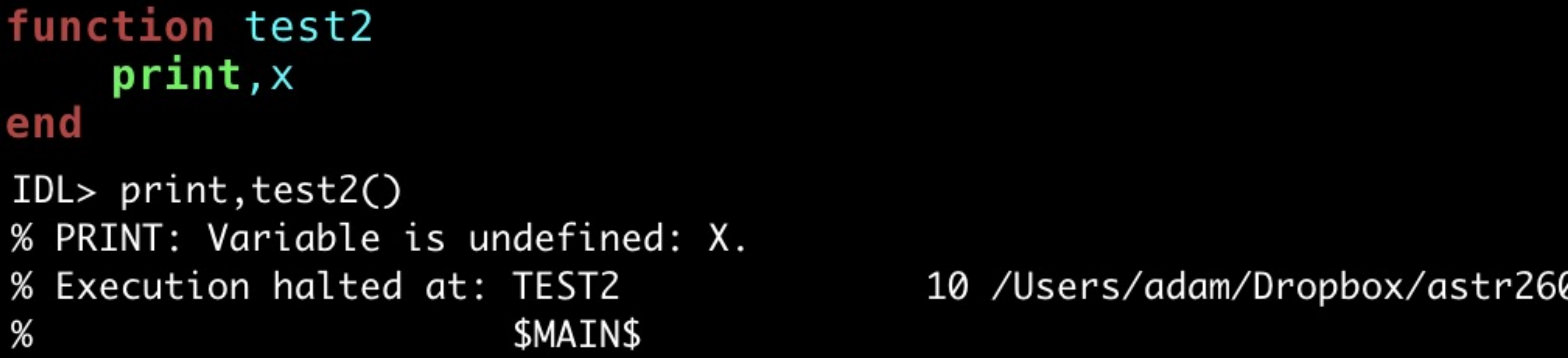
Will this code work?  (hint: `f(“Hi”)` does)

A) Yes



```
In [111]: %run hi5.py
Hi
HiHiHiHiHi
```

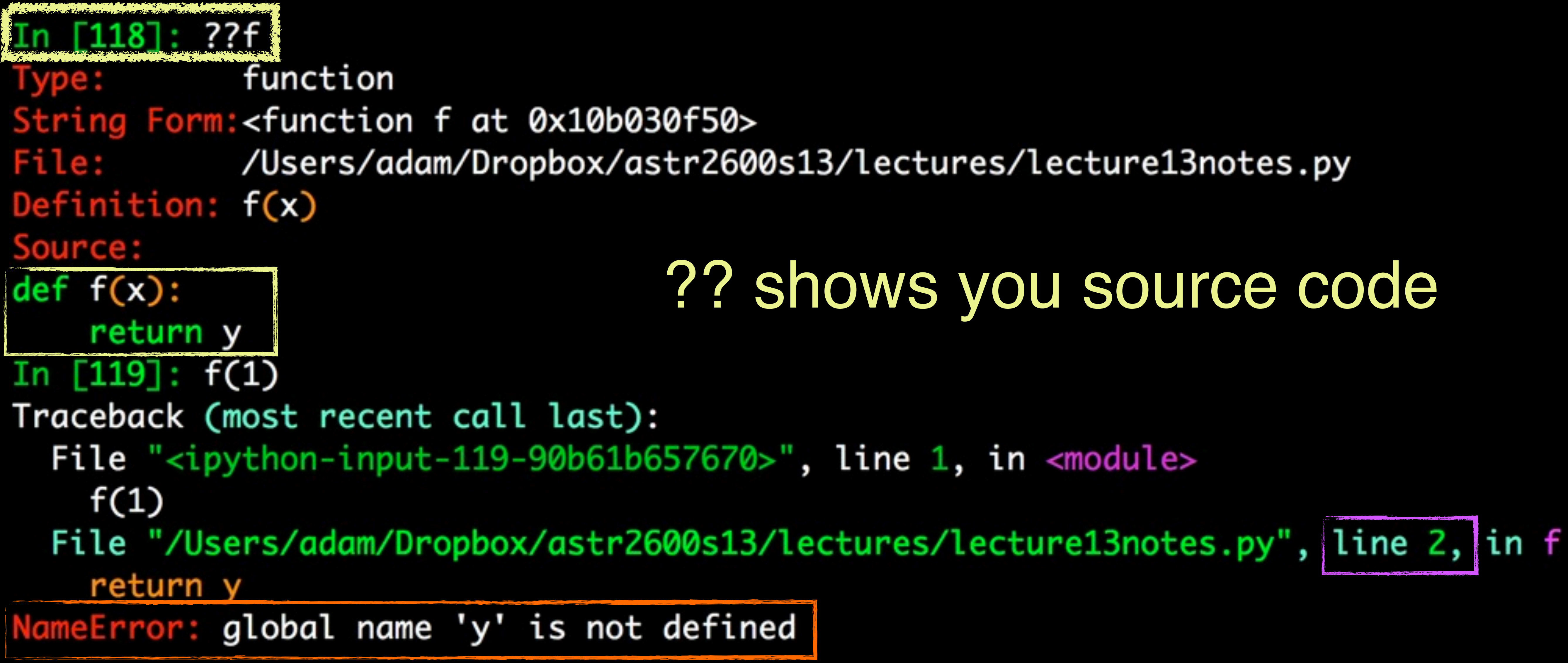
# Runtime Errors



```
function test2
    print,x
end
```

```
IDL> print,test2()
% PRINT: Variable is undefined: X.
% Execution halted at: TEST2                   10 /Users/adam/Dropbox/astr26
%                      $MAIN$
```

- `test2` is syntactically correct, but `x` is undefined
- Error message tells you:
  - `x` is undefined when you try to print it
  - the error is on line 10 of [filename] in function `test2`



```
In [118]: ??f
Type:        function
String Form:<function f at 0x10b030f50>
File:        /Users/adam/Dropbox/astr2600s13/lectures/lecture13notes.py
Definition: f(x)
Source:
def f(x):
    return y
In [119]: f(1)
Traceback (most recent call last):
  File "<ipython-input-119-90b61b657670>", line 1, in <module>
    f(1)
  File "/Users/adam/Dropbox/astr2600s13/lectures/lecture13notes.py", line 2, in f
    return y
NameError: global name 'y' is not defined
```

?? shows you source code

Can’t return an undefined variable:
‘y’ is not defined!

Problem is on line 2
of the function “f”

# If your code doesn’t work...

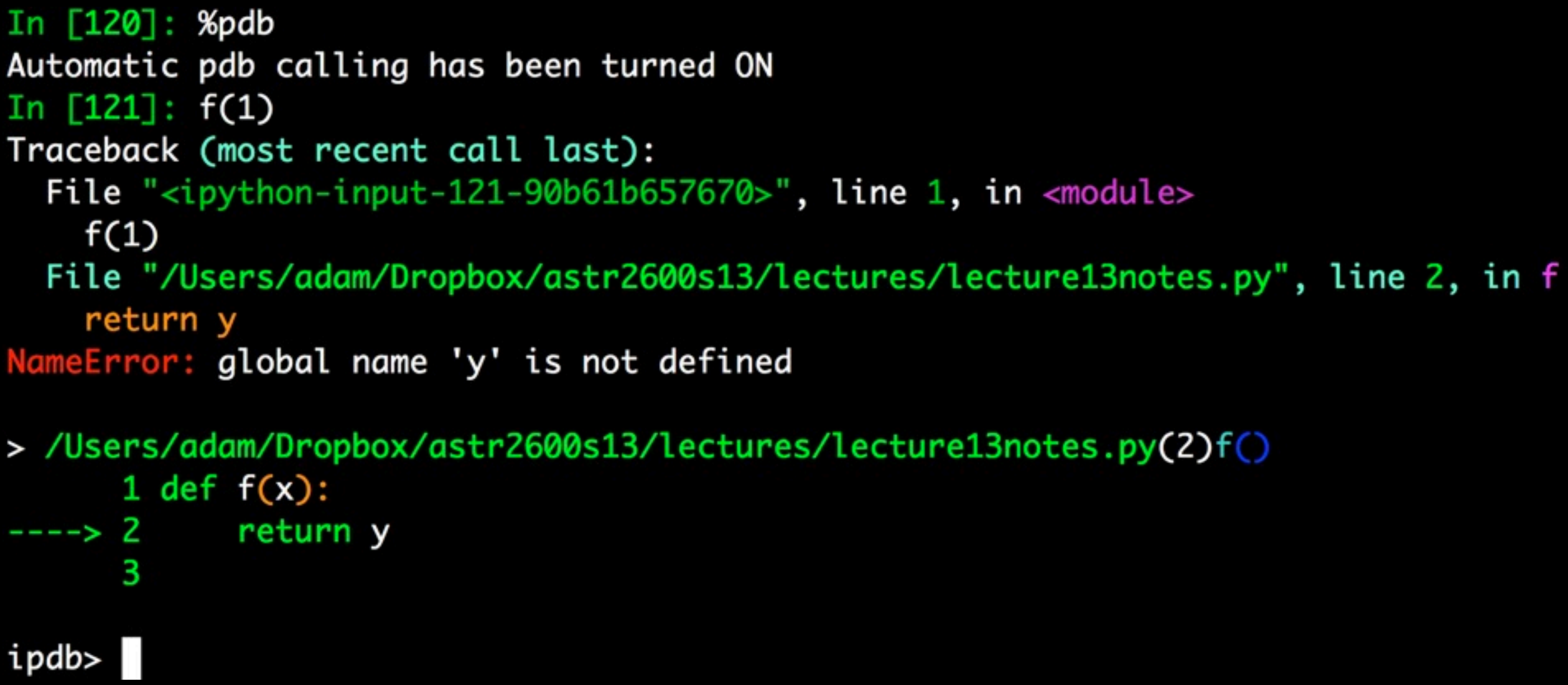
- You should ask first, “What error messages is it giving me?”
- “Does that tell me why it doesn’t work?”
- Ask yourself these questions before you ask me or Cameron.

# Runtime Errors

- In IDL, act just like `stop` statements
- They halt the code wherever the error was, allowing you to inspect *local* variables
- To get out of the halted code (the *debugger*), use `retall` (“return all”)

# Runtime Errors

- Python doesn’t kick you out into the debugger automatically, you need to enable it first:



```
In [120]: %pdb
Automatic pdb calling has been turned ON
In [121]: f(1)
Traceback (most recent call last):
  File "<ipython-input-121-90b61b657670>", line 1, in <module>
    f(1)
  File "/Users/adam/Dropbox/astr2600s13/lectures/lecture13notes.py", line 2, in f
    return y
NameError: global name 'y' is not defined

> /Users/adam/Dropbox/astr2600s13/lectures/lecture13notes.py(2)f()
      1 def f(x):
----> 2     return y
      3

ipdb> █
```

# Reading

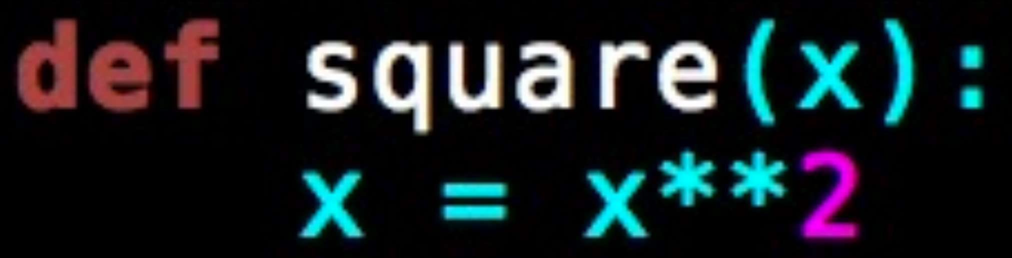
- Read section 13.6.  It is about debugging.  Debugging is important, but there are too many words for me to repeat in class.

# Debugging

1. Determine that there is a bug
2. Identify the bug
3. Treat the bug like a “whuduzitdo” - understand what the code is *actually* doing
4. Correct the bug

# Debugging

1. Determine that there is a bug
2. Identify the bug
   A. Create a test that “fails” on the bug
3. Treat the bug like a “whuduzitdo” - understand what the code is *actually* doing
4. Correct the bug
   A. Check that the test now does NOT fail

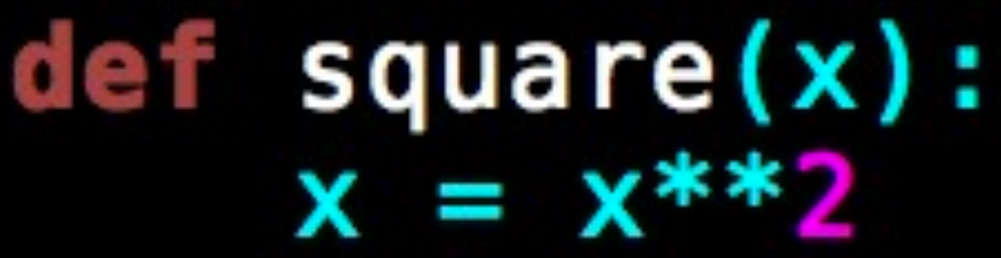


```python
def square(x):
    x = x**2
```

Will this function run?

A) Yes

B) No

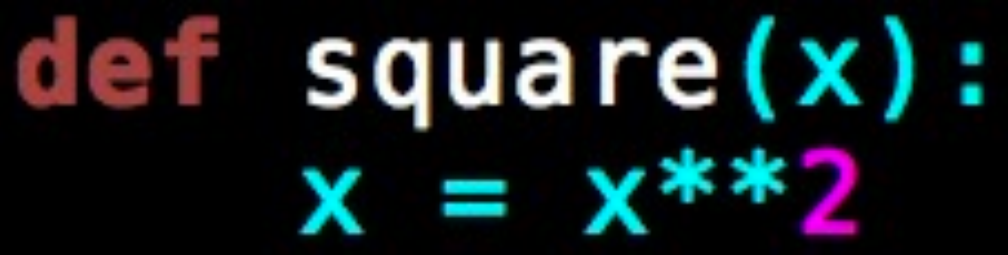


```
def square(x):
    x = x**2
```

Will it set x to be its square?

A) Yes

B) No

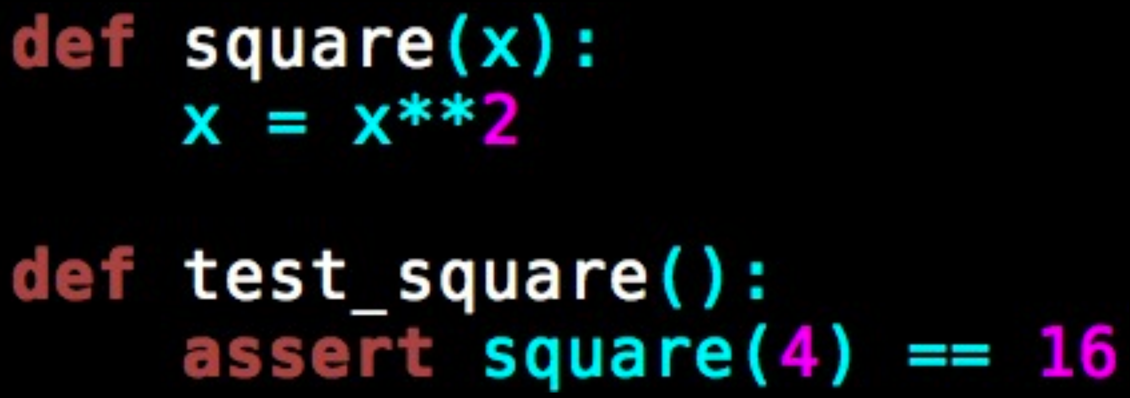


```
def square(x):
    x = x**2
```

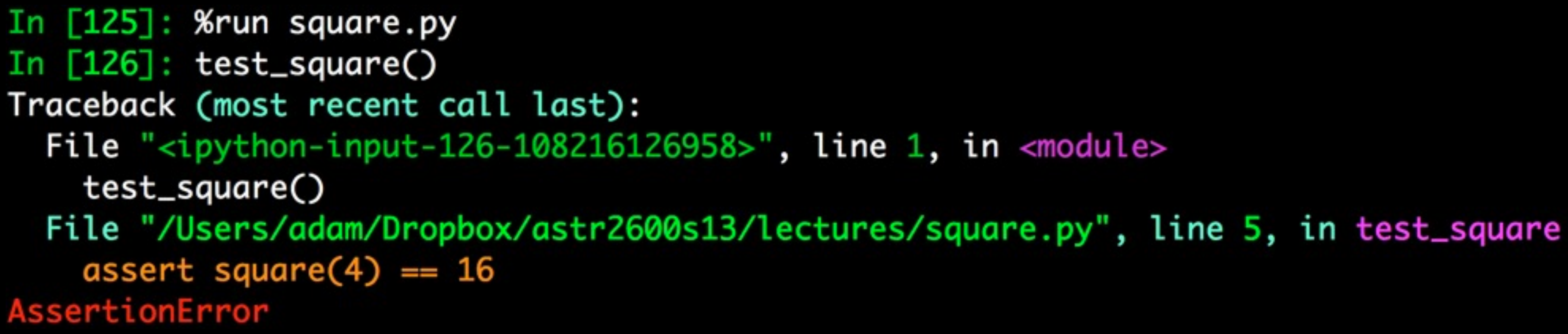
Will it return $x^2$?

A) Yes

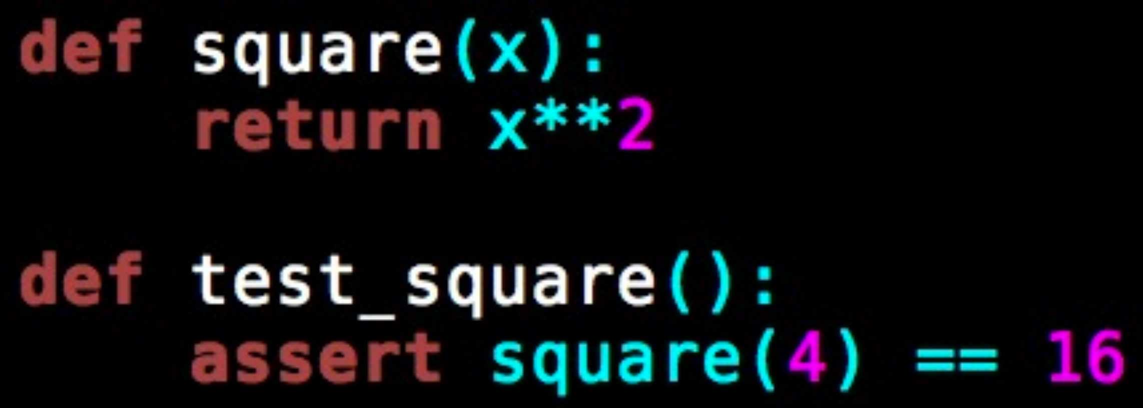
B) No



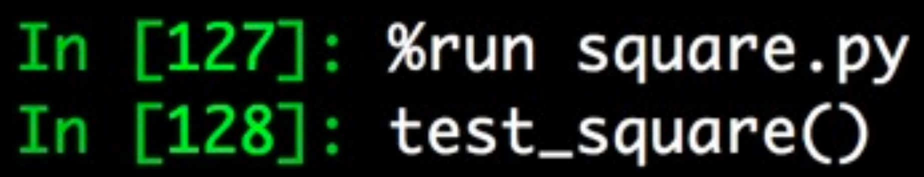
```
def square(x):
    x = x**2

def test_square():
    assert square(4) == 16
```



```
In [125]: %run square.py
In [126]: test_square()
Traceback (most recent call last):
  File "<ipython-input-126-108216126958>", line 1, in <module>
    test_square()
  File "/Users/adam/Dropbox/astr2600s13/lectures/square.py", line 5, in test_square
    assert square(4) == 16
AssertionError
```



```python
def square(x):
    return x**2

def test_square():
    assert square(4) == 16
```

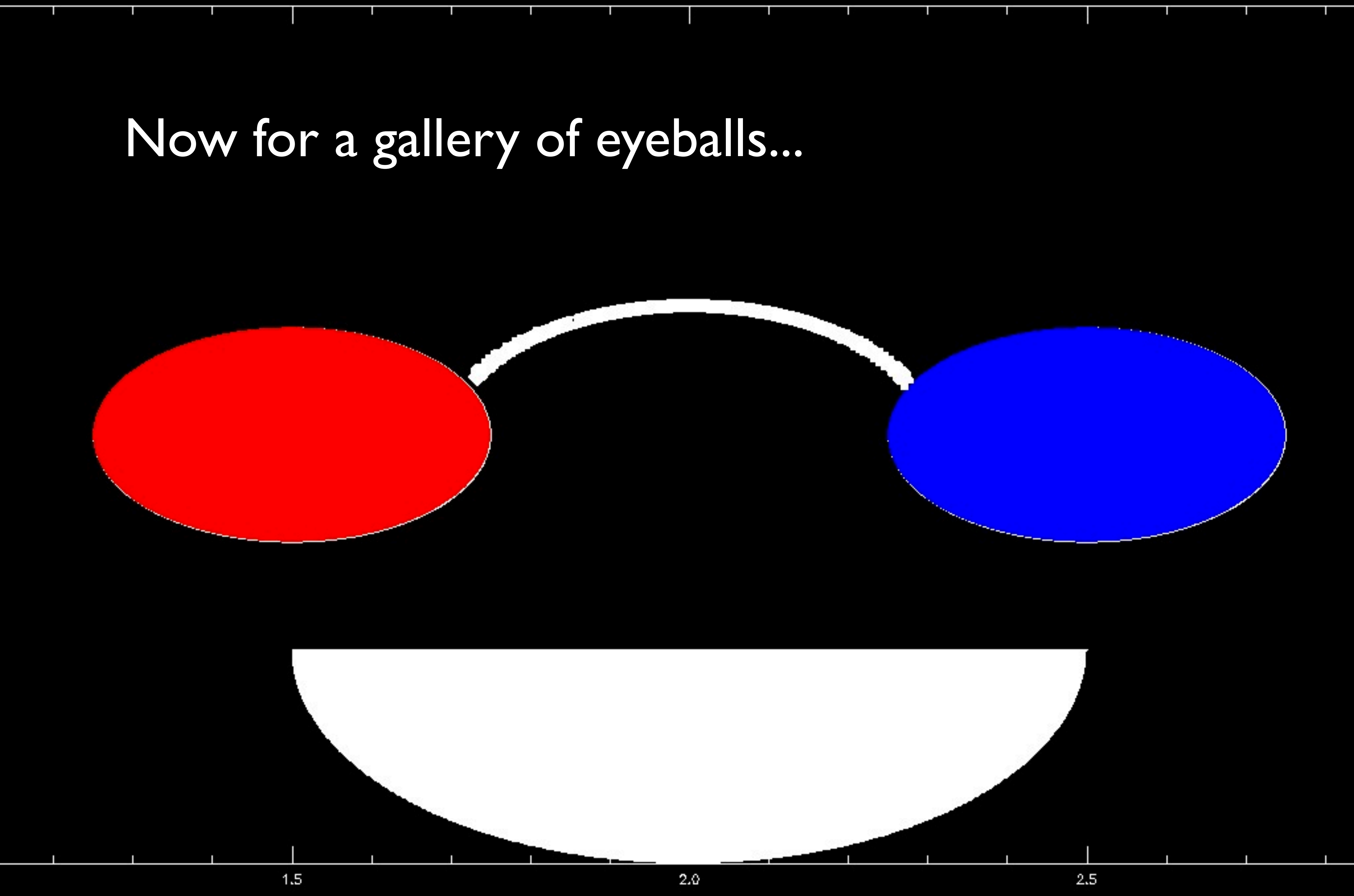


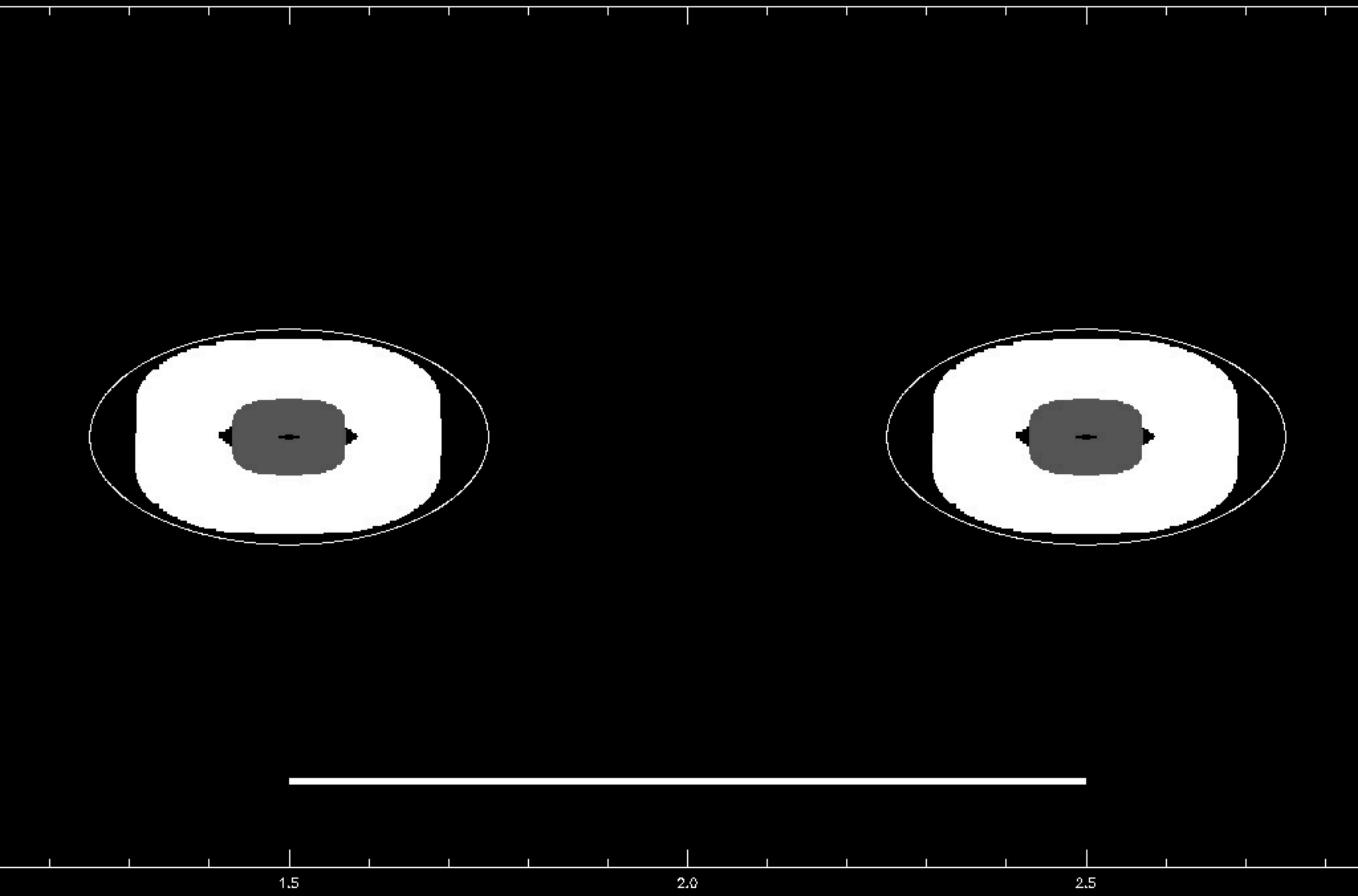
```
In [127]: %run square.py
In [128]: test_square()
```

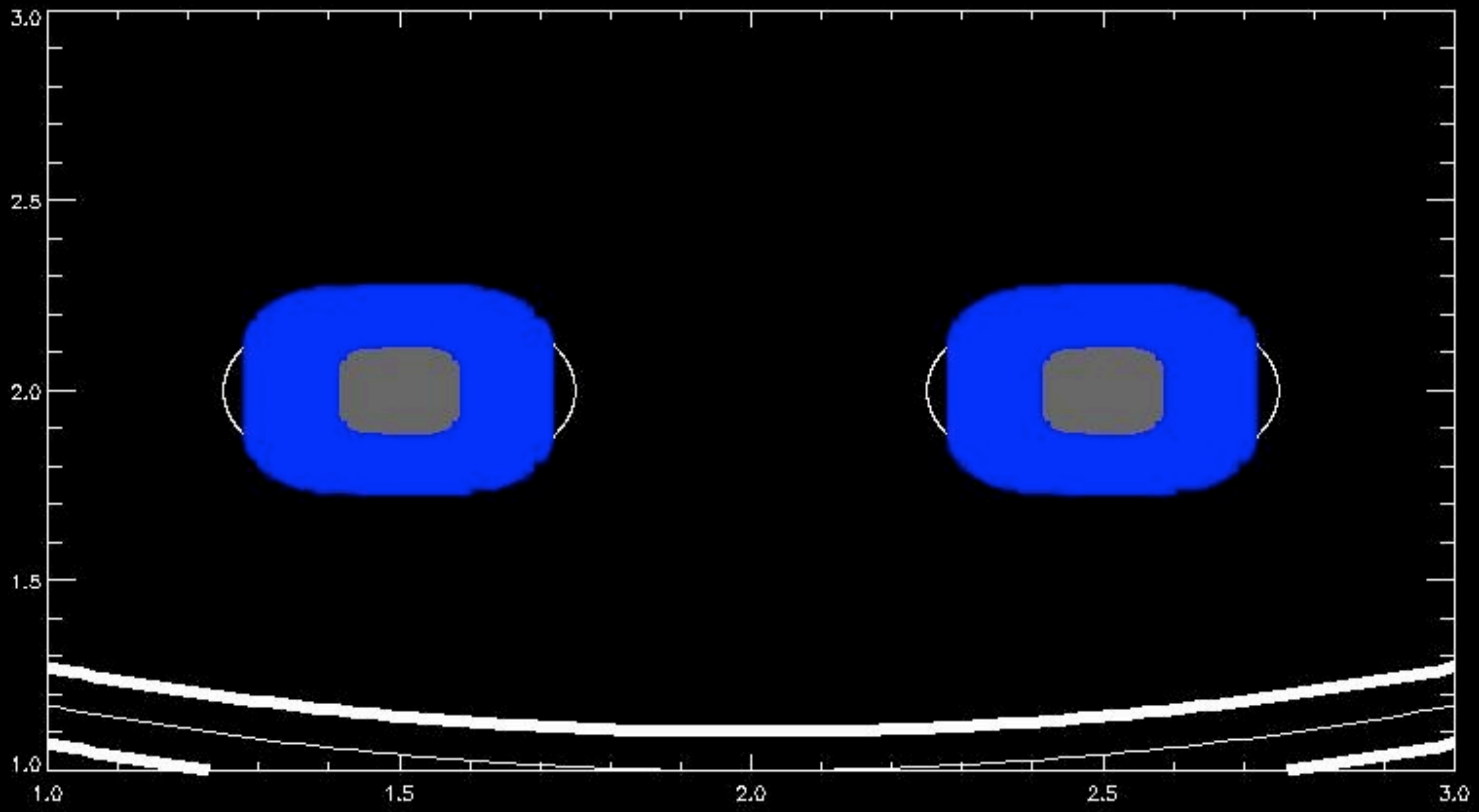
# Debugging

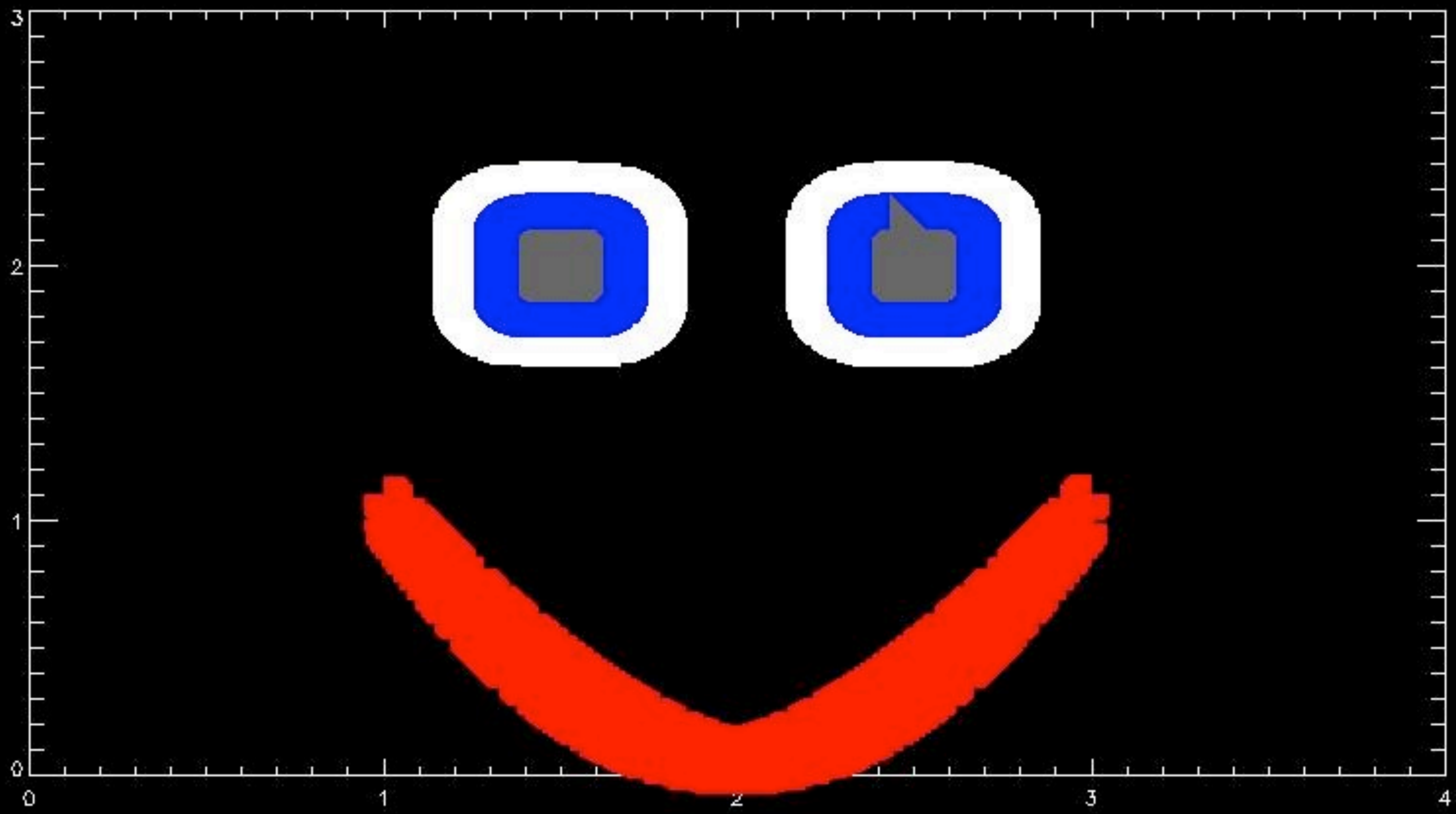
- Debugging is a skill developed over time
- whuduzitdo’s are designed to give you that experience
- debugging = you are the compiler

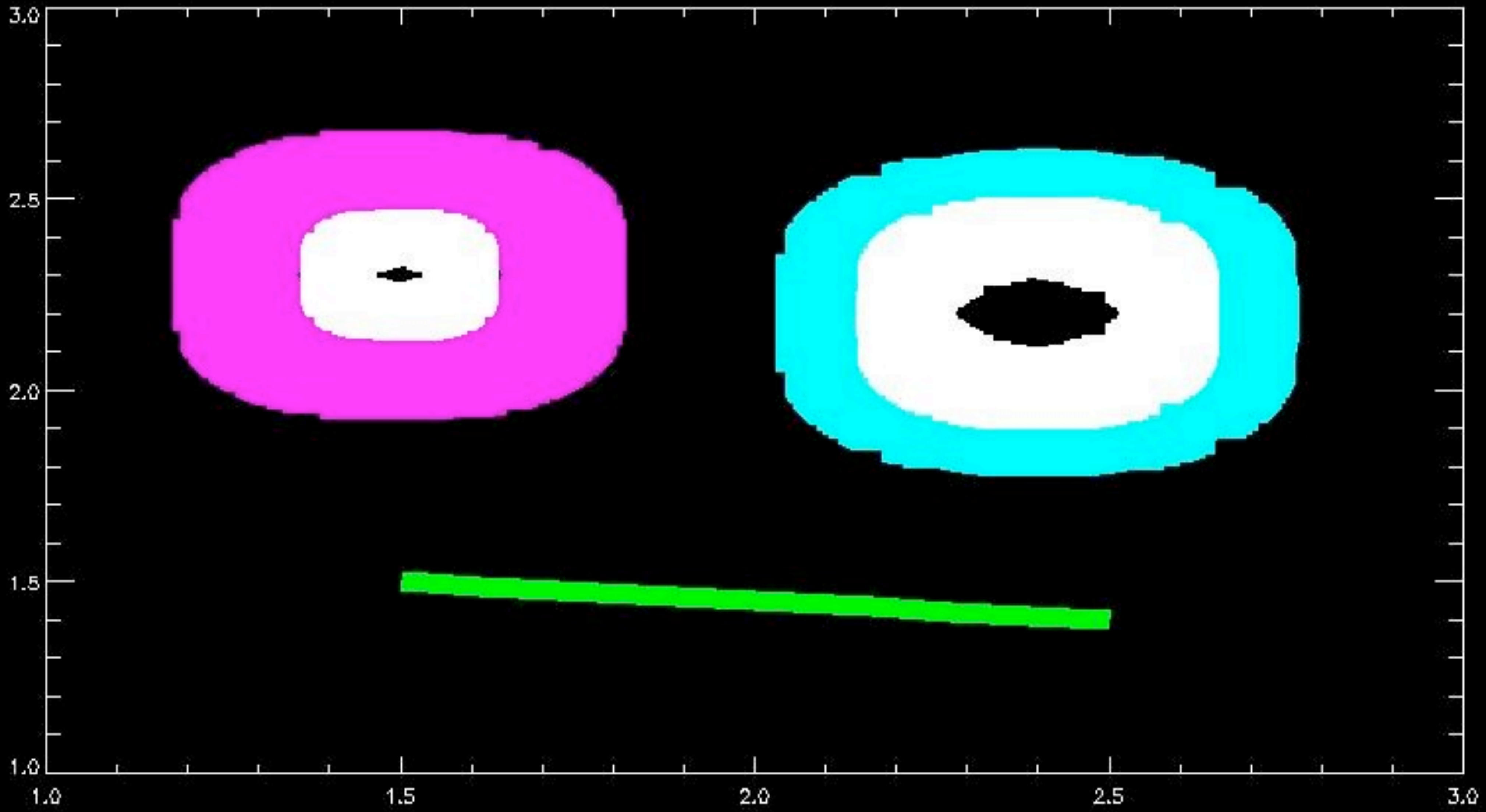
# Now for a gallery of eyeballs...

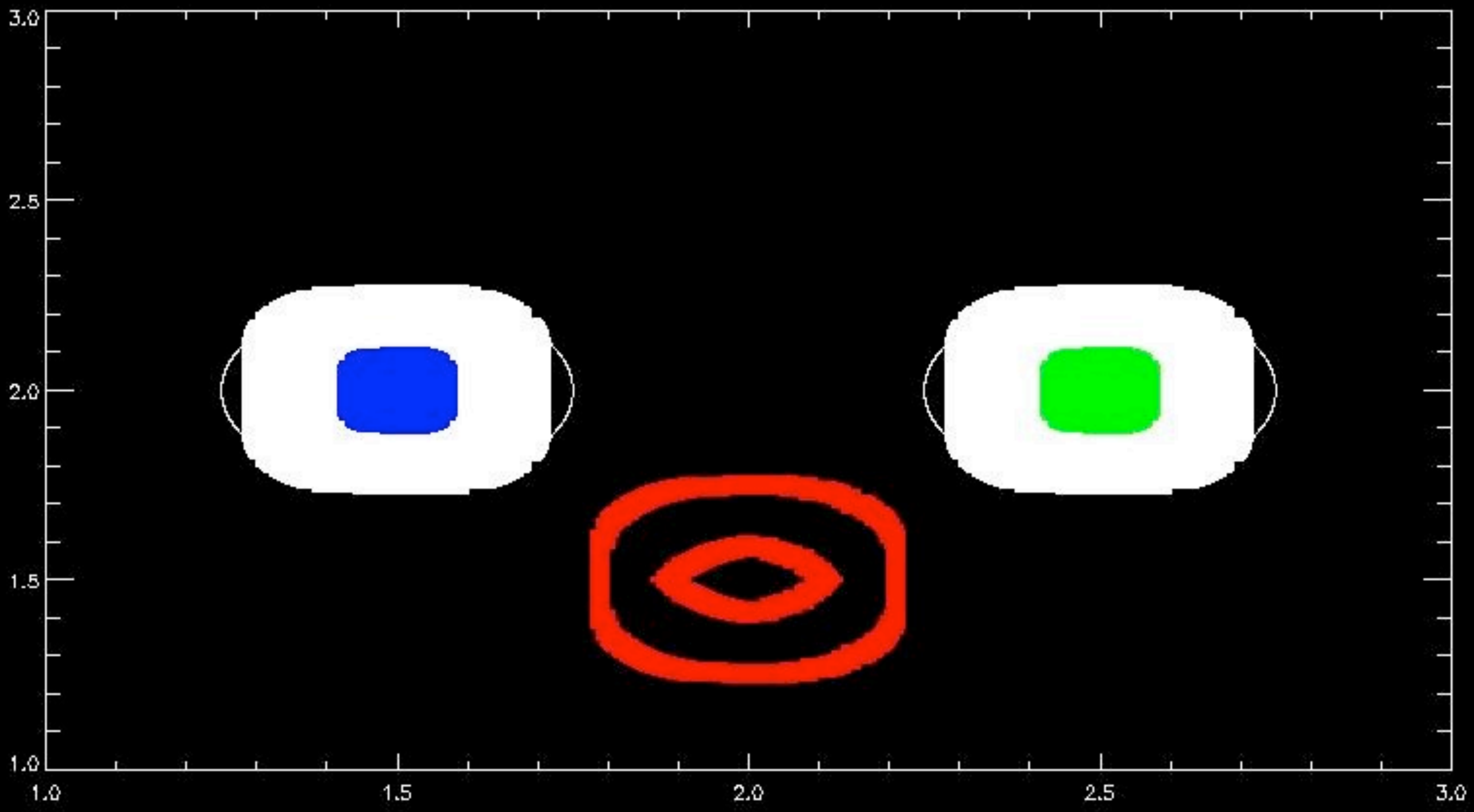


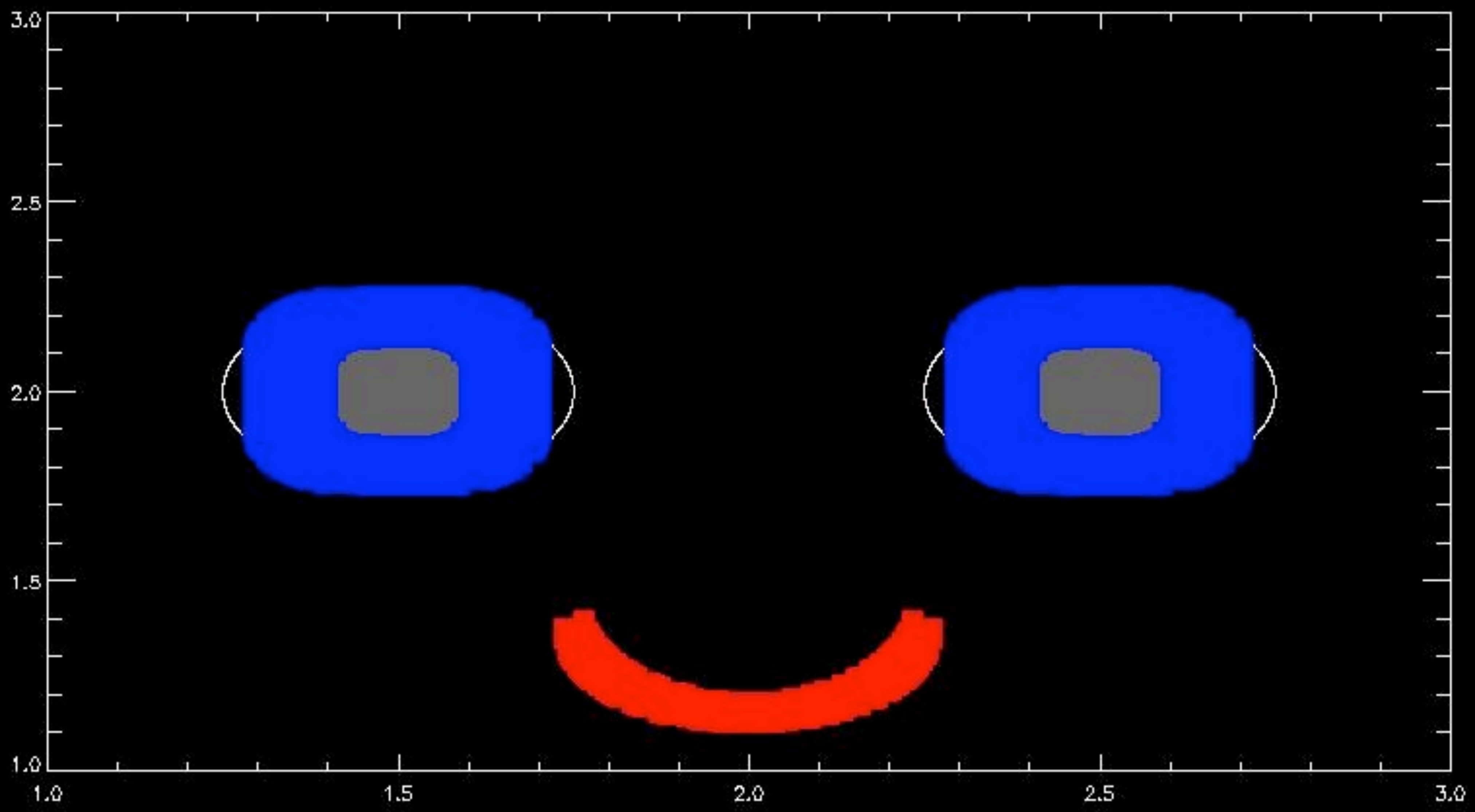

1.5
2.0
2.5

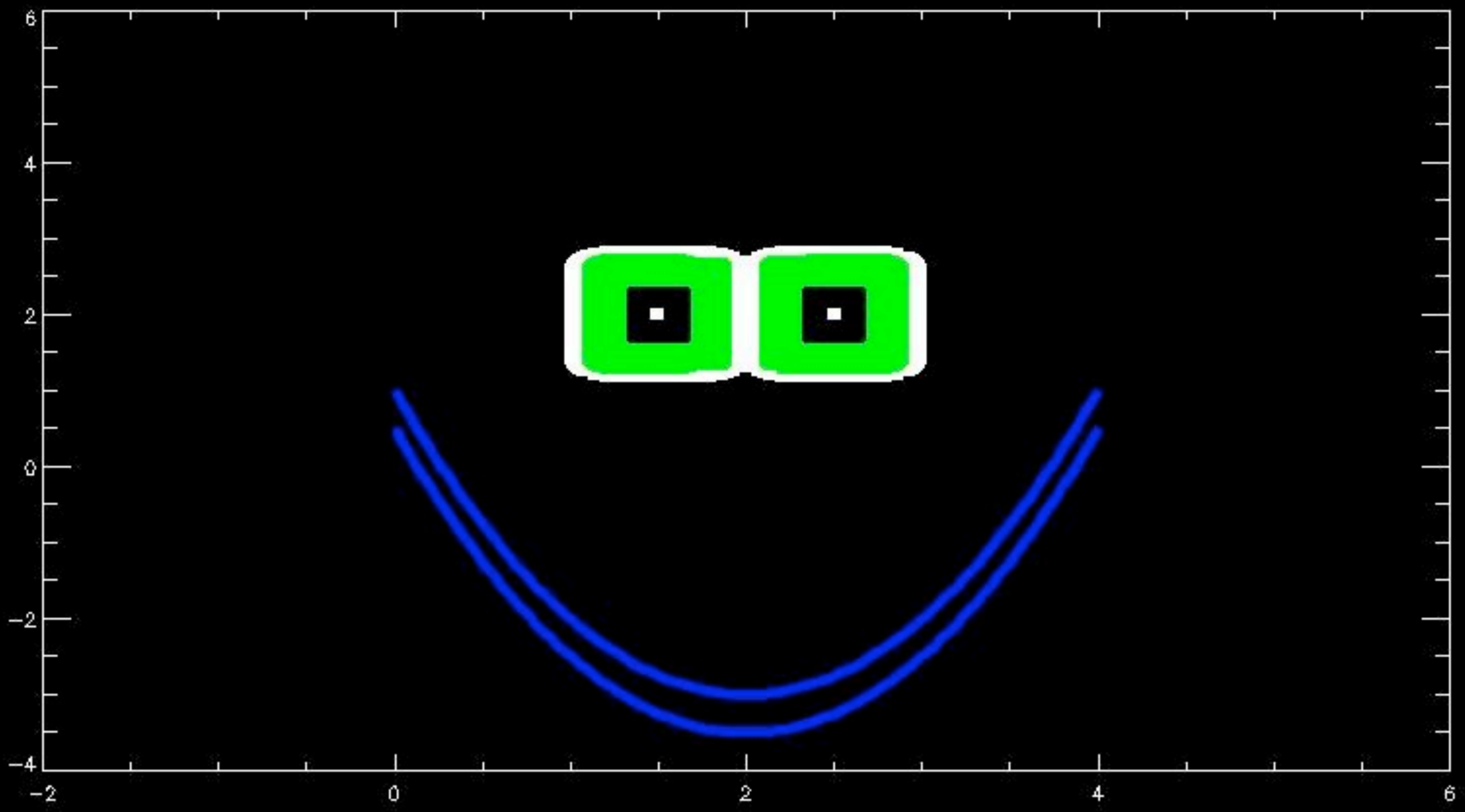

3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0

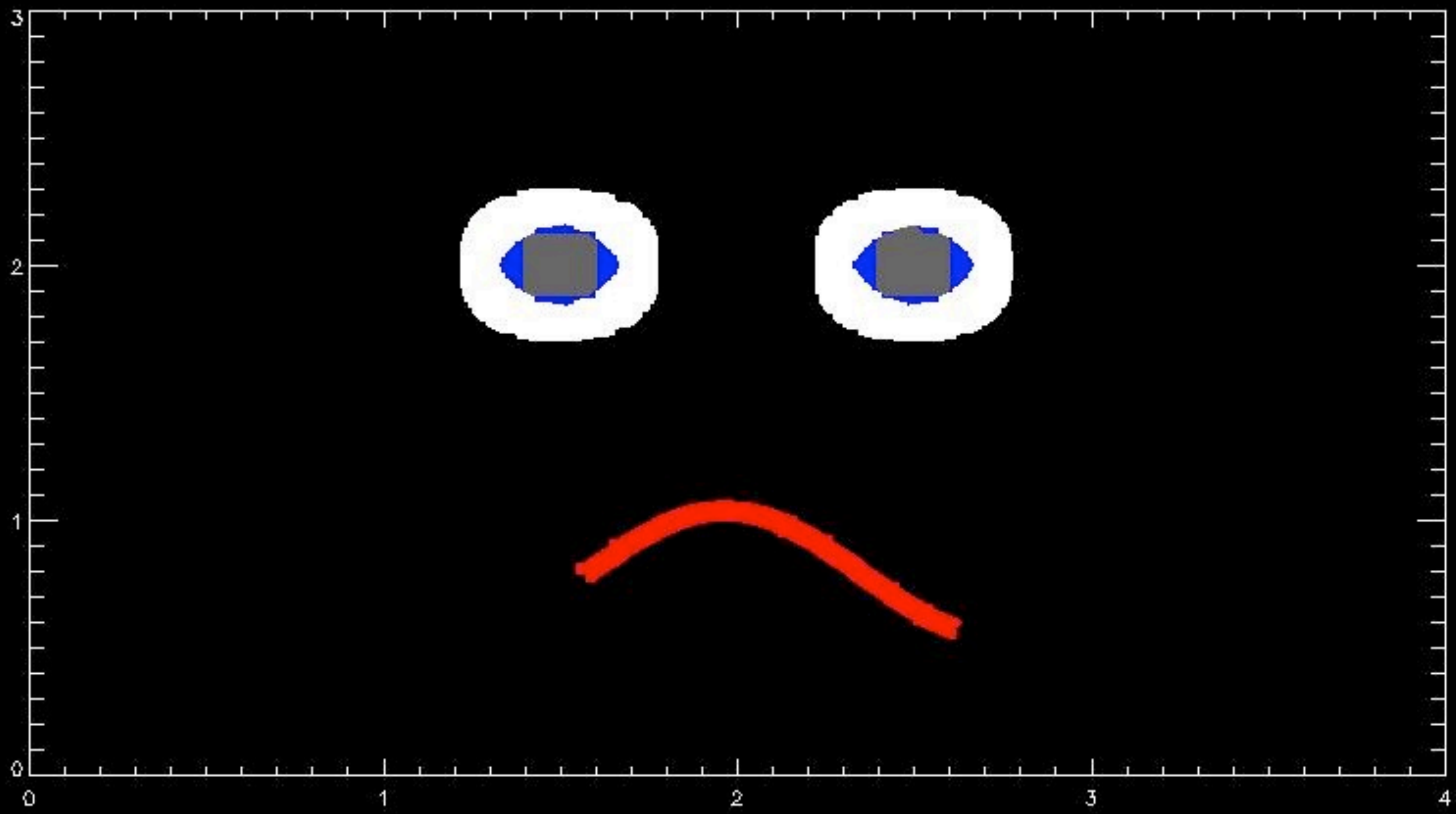

3
2
1
0
0
1
2
3
4

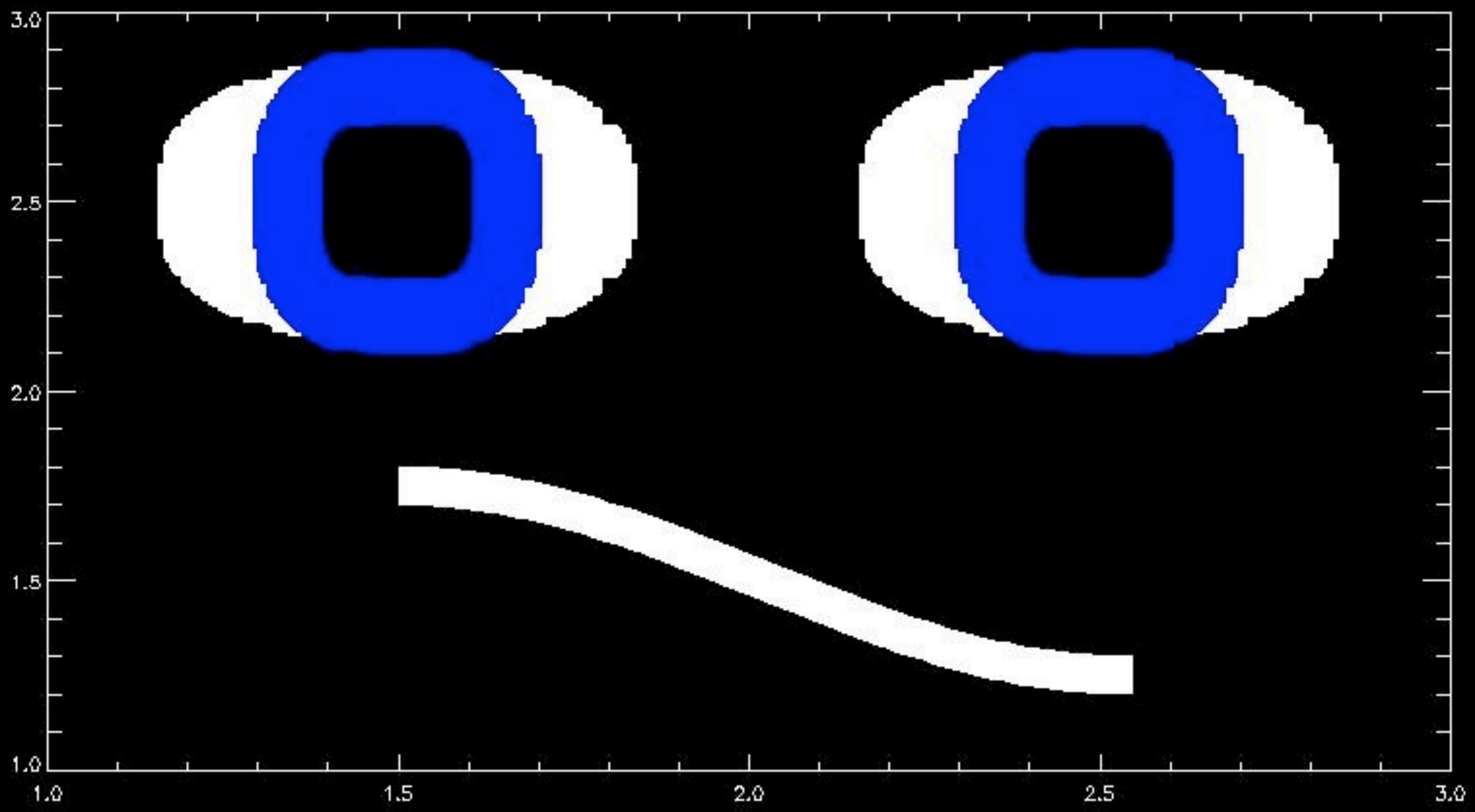

3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0

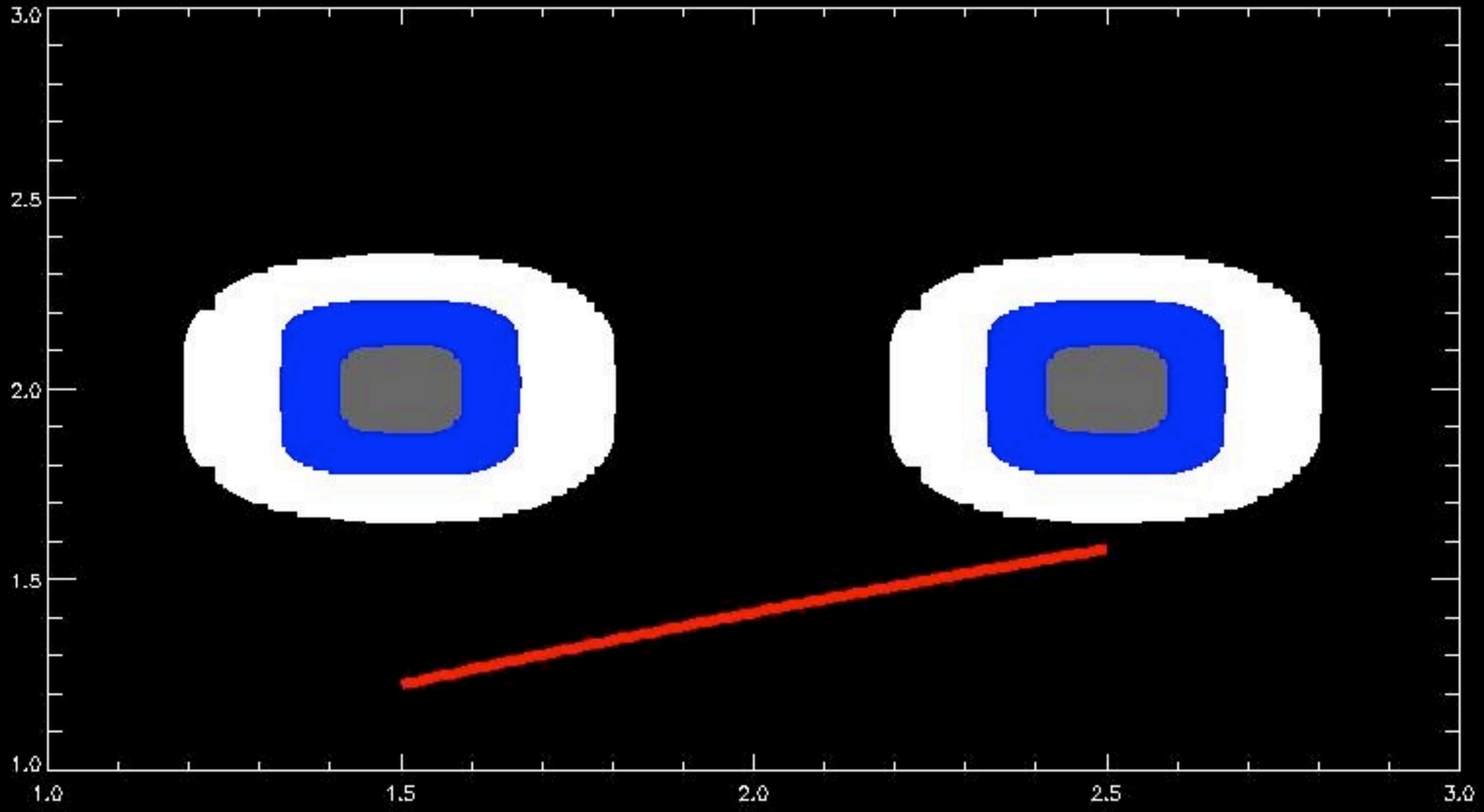

3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0


3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0


6
4
2
0
−2
−4
−2
0
2
4
6


3
2
1
0
0
1
2
3
4


3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0


3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0

# Chapter 14: Program Design and Development

# General Rules

- Start small: Make testable, individual pieces of code
    - debugging is much slower than code writing

# Top Down Design

- Look at the big picture first (start with a vague outline, with the big bullets written first)

# Example

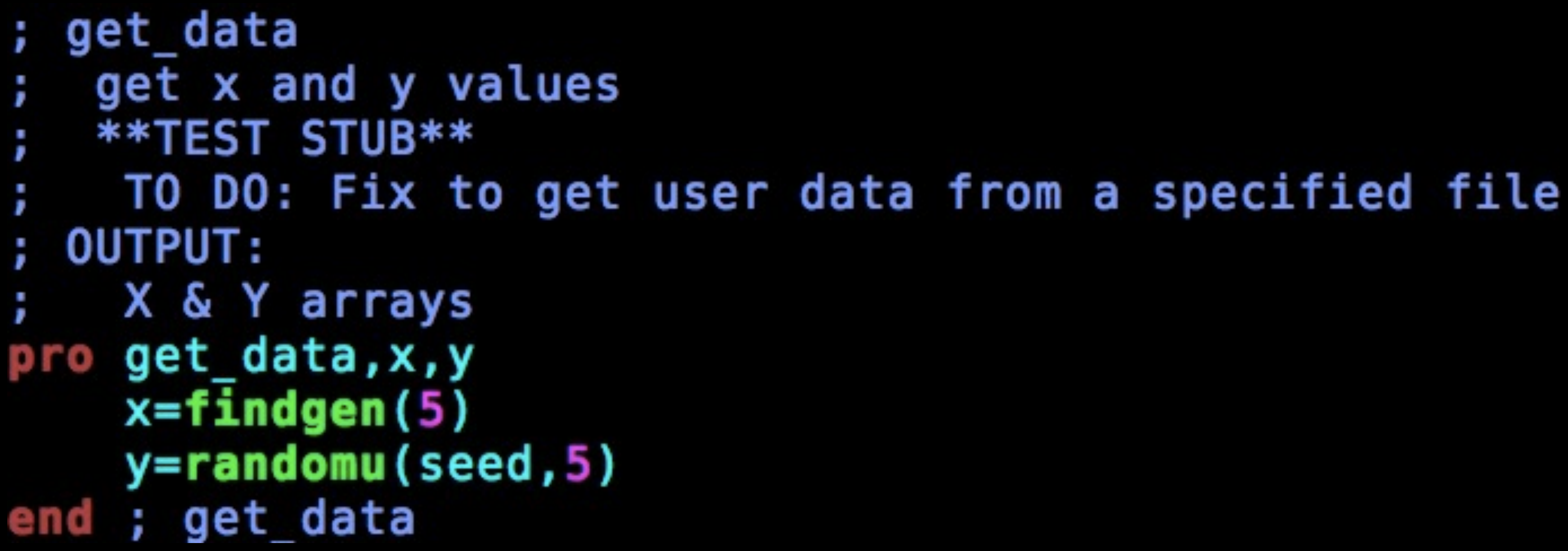
1. Get input data
2. Do computation
3. Output results

# Example

| | |
|---|---|
| 1. Get input data | `get_data,x,y` |
| 2. Do computation | `z = compute_z(x,y)` |
| 3. Output results | `output_results,z` |

# Example

- Fill in individual functions with test stubs:



```
; get_data
;  get x and y values
;  **TEST STUB**
;   TO DO: Fix to get user data from a specified file
; OUTPUT:
;   X & Y arrays
pro get_data,x,y
    x=findgen(5)
    y=randomu(seed,5)
end ; get_data
```

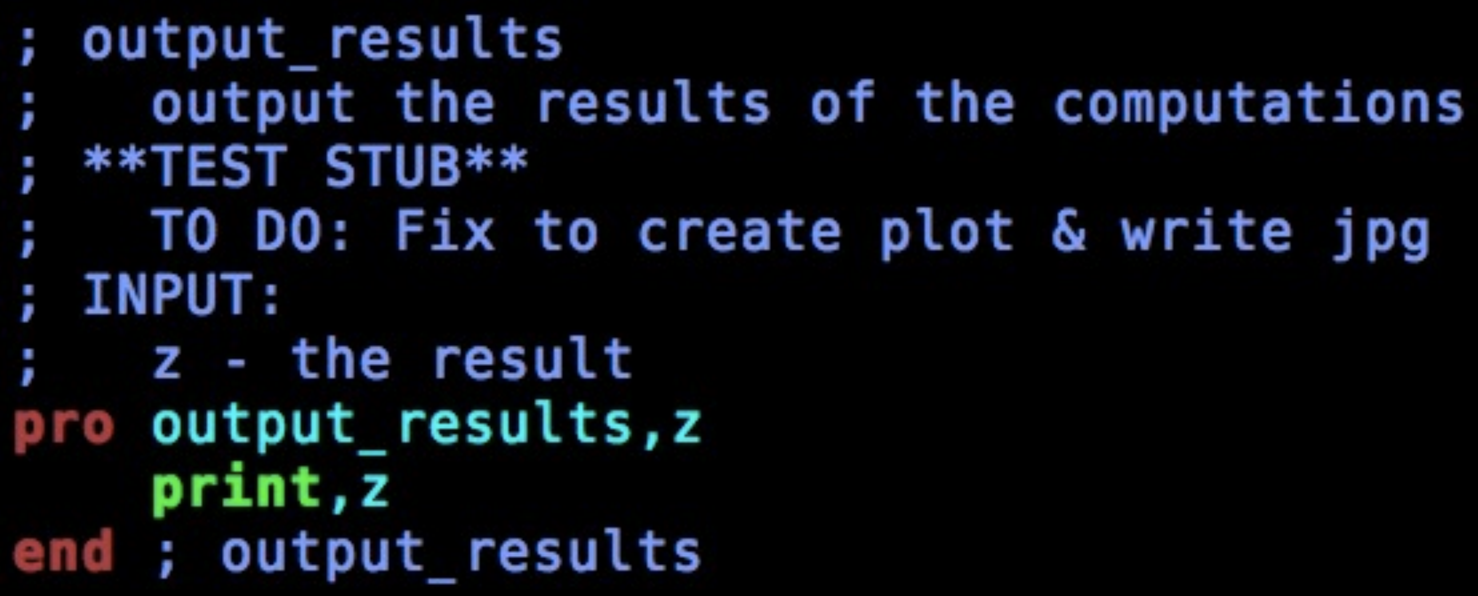
- Doesn’t actually *read* anything, but outputs data in the right format
  - i.e., does enough for the next step

# Example

- `get_data` does just enough for `compute_z` to work

# Example

- Overly simplified output



```
; output_results
;   output the results of the computations
; **TEST STUB**
;   TO DO: Fix to create plot & write jpg
; INPUT:
;   z - the result
pro output_results,z
    print,z
end ; output_results
```

- Just enough to see if the prior steps worked (not “pretty” yet)

# Example - Outline revisited

1. Get input data `get_data,x,y`
    - Reads reasonable test data
2. Do computation `z = compute_z(x,y)`
    - still not implemented
3. Output results `output_results,z`
    - May be ugly, but does enough to see if compute_z works

# Example

- Create `compute_z` now
  - you already have a “testing framework” built around it
- Then, iterate: maybe you need more complicated data for `compute_z`? change `get_data`

# Another approach: Top-Down

- Advantages:
  - Big Picture is set up early
  - Can break down into small chunks
    - Good for team development
  - Visualize as “flow charts”

# Top-Down

- Frequently utilizes “pseudo-code”
    - pseudo-code is “almost code” that reads more like English
    - it’s code, but you don’t care about syntax - you aim for readability
    - pseudocode is usually easy to translate into real code, but easier to read (as a human) than code

# Pseudo-Code example

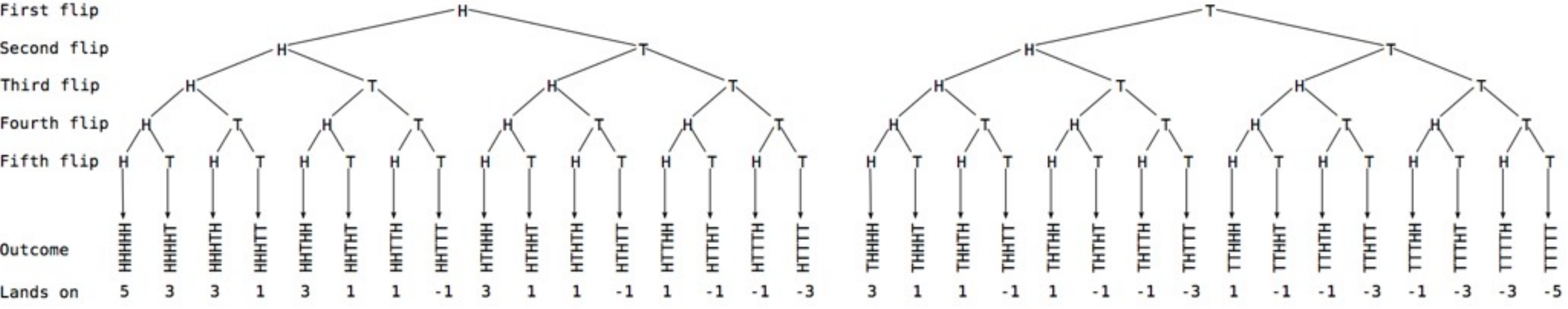
- Open my flux and wavelength files
- Plot flux versus wavelength
- Highlight the spectral line at 1020A
- Fit a gaussian to that line

# Bottom-Up approach

- Start with small details
- Test each small step along the way
  - (for small projects, this will end up looking the same as “Top Down”, except you won’t write an outline at the start)

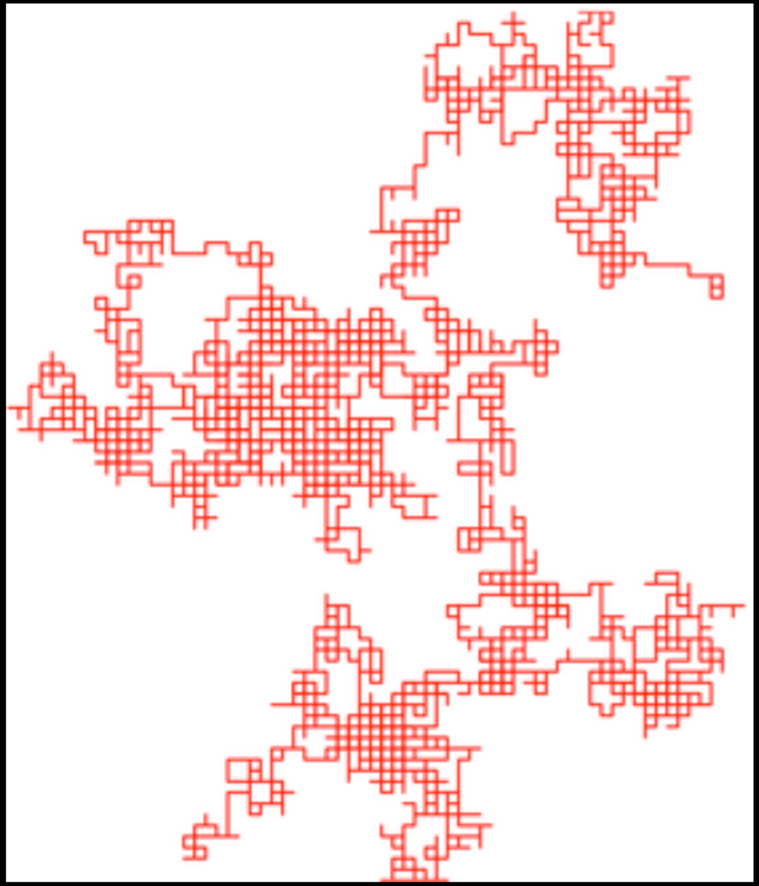
# Case Study: Random Walk

- A “random walk” is taking sequential coin flips to decide which direction to move



# Random Walk

- Take N steps of some size, where each step is random
- This results in brownian motion (motion of, e.g., air molecules)



# Bottom-up

- Start with the simplest component:
  - Take one *random* step
- If motion is restricted to be in one dimension (easiest case), that means there must be equal likelihood to go *forward* and *backward*

# What does this do?

```
step = randomu(seed,1) gt 0.5 ? 1 : -1
```

A) Makes `step` a random variable greater than 0.5

B) Makes `step` +1 50% of the time and -1 50% of the time

C) Makes `step` an array of length `seed` with values +1 and -1 randomly distributed

D) Crashes

E) None of the above

# 2D steps

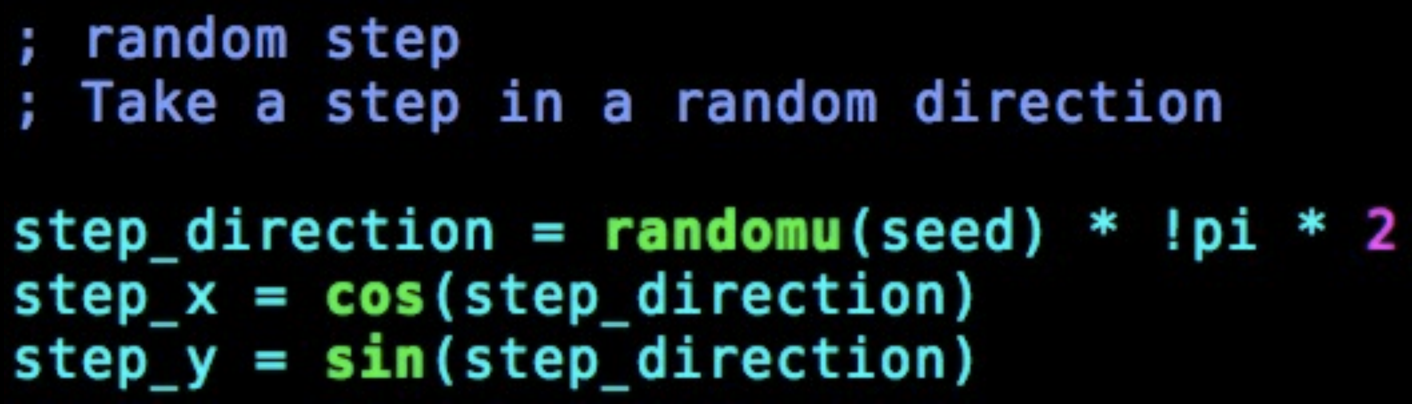
- On a 2 dimensional grid, can take a step in any direction
  - step size is always the same
  - all directions are equally likely

Which is best for determining a random direction?

A) `randomn(seed) * 360`

B) `randomu(seed) * 360`

C) Neither



```
; random step
; Take a step in a random direction

step_direction = randomu(seed) * !pi * 2
step_x = cos(step_direction)
step_y = sin(step_direction)
```

Should random step be...

A) A procedure

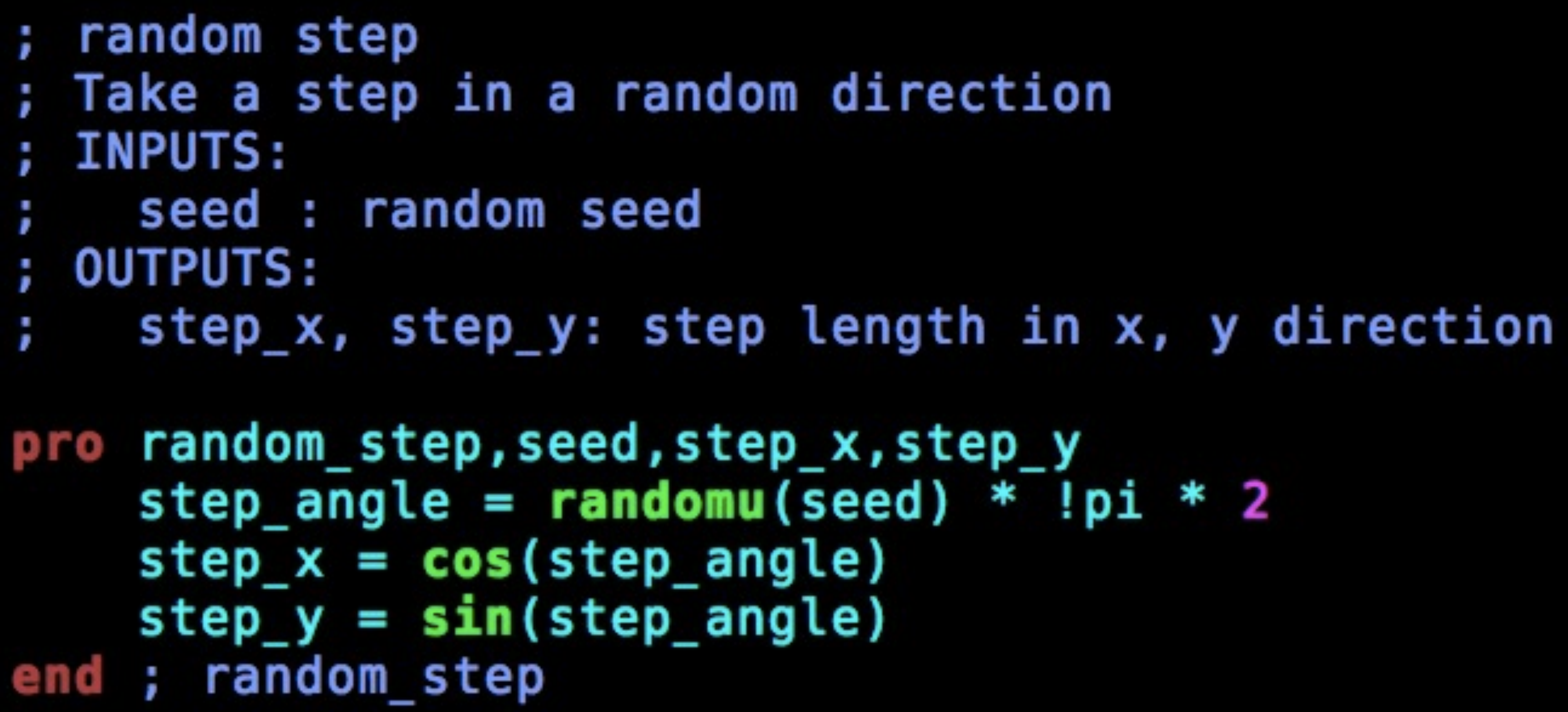
B) A function

C) A program

D) A script

E) None of the above

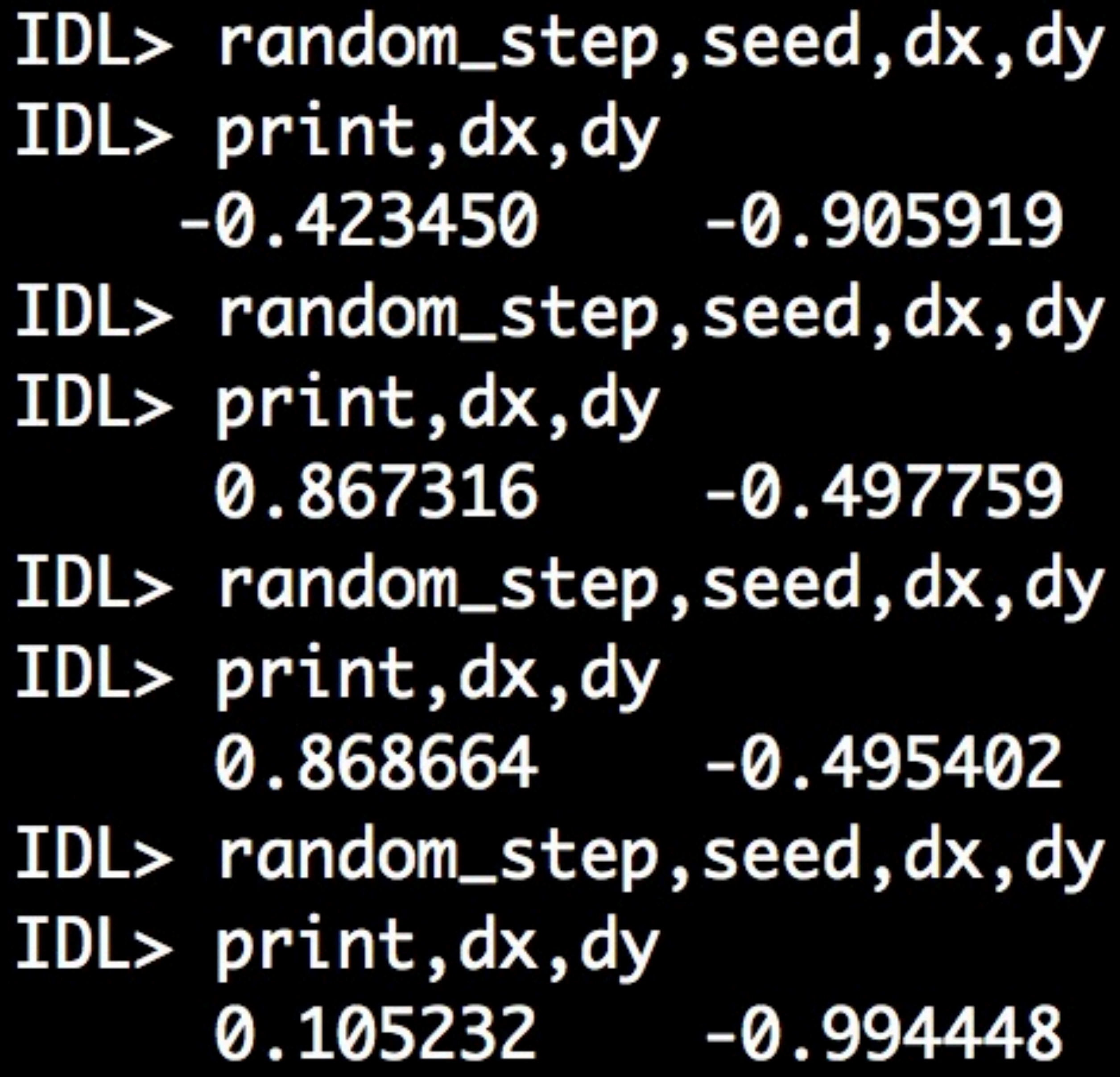
# Random Step pro



```
; random step
; Take a step in a random direction
; INPUTS:
;   seed : random seed
; OUTPUTS:
;   step_x, step_y: step length in x, y direction

pro random_step,seed,step_x,step_y
    step_angle = randomu(seed) * !pi * 2
    step_x = cos(step_angle)
    step_y = sin(step_angle)
end ; random_step
```

# Random step...

- It works, I think....
- How do I test it?
- What aspects could we test?

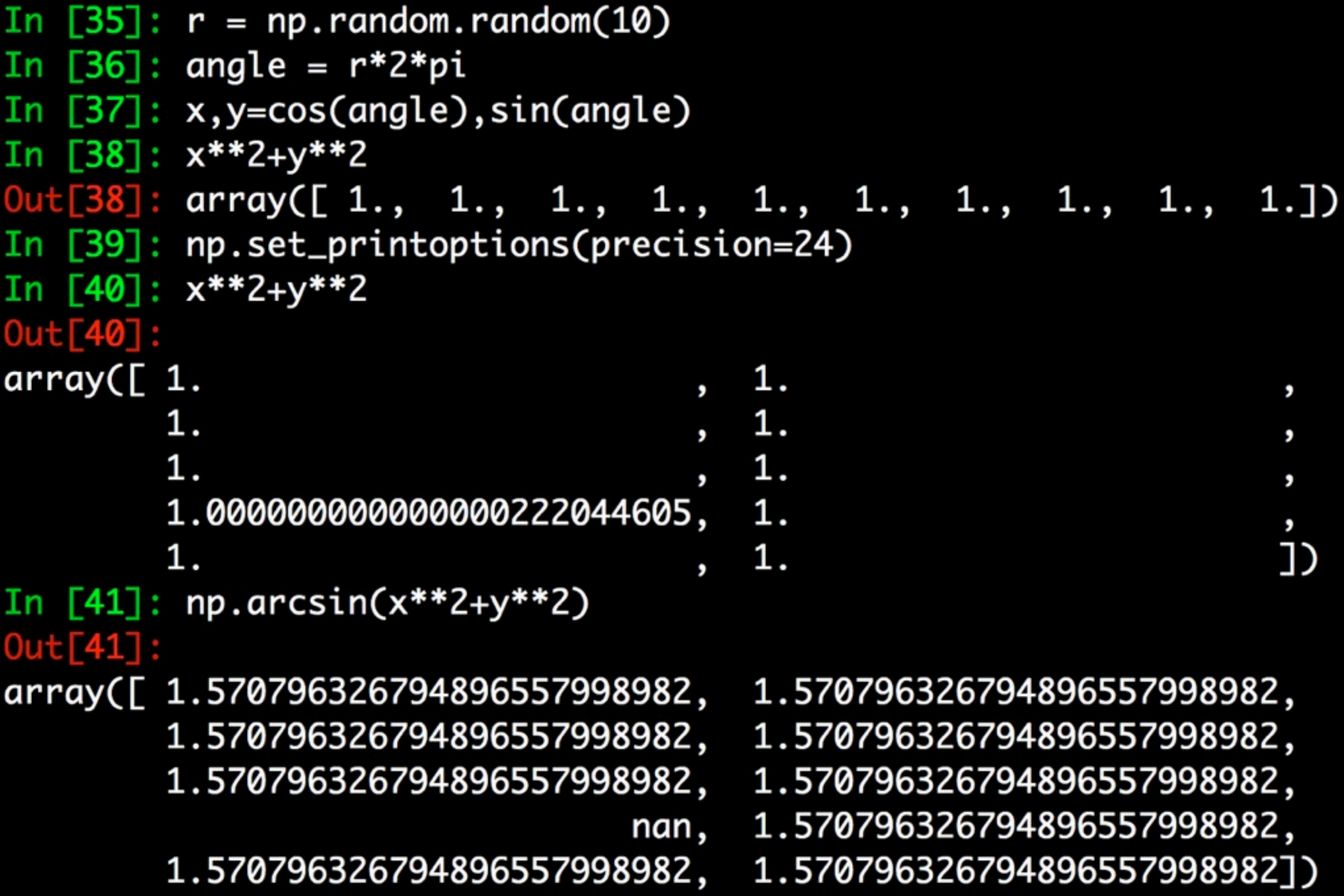


```
IDL> random_step,seed,dx,dy
IDL> print,dx,dy
    -0.423450    -0.905919
IDL> random_step,seed,dx,dy
IDL> print,dx,dy
     0.867316    -0.497759
IDL> random_step,seed,dx,dy
IDL> print,dx,dy
     0.868664    -0.495402
IDL> random_step,seed,dx,dy
IDL> print,dx,dy
     0.105232    -0.994448
```

Which is a useful test of `random_step`?
(you just called `random_step,seed,dx,dy`
so `dx` and `dy` should be random variables with
some other properties....)

```
A)print,abs((dx^2+dy^2)^0.5 - 1.0) lt 1e-7

B)print,(dx^2+dy^2)^0.5 eq 1.0

C)print,abs(dx + dy) - 1.0 lt 1e-7

D)print,dx^2 lt dy^2
```

E) None of these



```
In [35]: r = np.random.random(10)
In [36]: angle = r*2*pi
In [37]: x,y=cos(angle),sin(angle)
In [38]: x**2+y**2
Out[38]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
In [39]: np.set_printoptions(precision=24)
In [40]: x**2+y**2
Out[40]:
array([ 1.                        ,  1.                        ,
        1.                        ,  1.                        ,
        1.                        ,  1.                        ,
        1.000000000000000222044605,  1.                        ,
        1.                        ,  1.                        ])
In [41]: np.arcsin(x**2+y**2)
Out[41]:
array([ 1.570796326794896557998982,  1.570796326794896557998982,
        1.570796326794896557998982,  1.570796326794896557998982,
        1.570796326794896557998982,  1.570796326794896557998982,
                               nan,  1.570796326794896557998982,
        1.570796326794896557998982,  1.570796326794896557998982])
```

```
RuntimeWarning: invalid value encountered in arcsin
```

# Next Step ...insteption...

- We made a random step, but we want to random walk
- How many steps should we take?
    - Leave that decision to the user
- How should we take those steps?

# Random Walk

You have a `random_step` procedure. What should you use to make a `random_walk` procedure now?
(how many steps are we taking/when do we stop walking?)

A) `while` loop

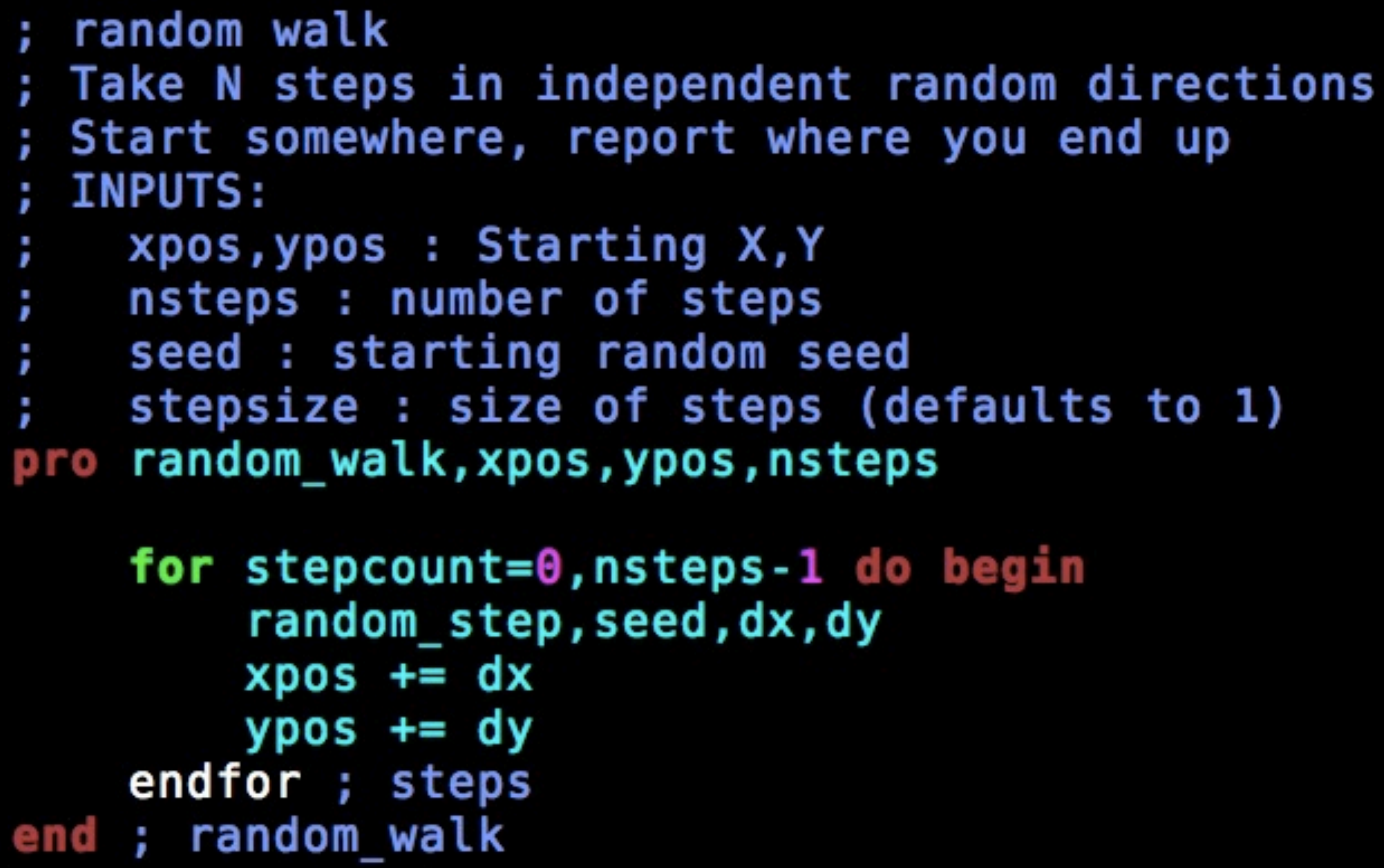
B) `repeat ... until` loop

C) `foreach` loop

D) `for` loop

E)

# Random Walk



```
; random walk
; Take N steps in independent random directions
; Start somewhere, report where you end up
; INPUTS:
;   xpos,ypos : Starting X,Y
;   nsteps : number of steps
;   seed : starting random seed
;   stepsize : size of steps (defaults to 1)
pro random_walk,xpos,ypos,nsteps

    for stepcount=0,nsteps-1 do begin
        random_step,seed,dx,dy
        xpos += dx
        ypos += dy
    endfor ; steps
end ; random_walk
```

# Other improvements?

- What are we really interested in?
    - Each step?  The total distance?  The path taken?  Maybe all of these!
    - Need more features!
        - referred to as a “feature request” on the intertubes

# Expanding Random Walk

- Make `xpos,ypos` arrays instead of scalars
    - can keep track of each step
- Plot the random walk!
    - Obviously.  It will look cool.
- Determine the total distance traveled (at each step)

# Building Up Code

- Tutorial today will be about adding features and code development
- You should have “Diagnostic Code” (i.e. `print` statements) dispersed throughout
- If you don’t want it to print, “comment it out” (start the line with a `;`)

# Chapter 14

- Use Chapter 14 as a reference
- Dewey goes through the whole development process - many of the same ideas we went over in lecture, but with concrete examples throughout
- There are LOTS OF MISTAKES in programming, and Ch 14 shows some

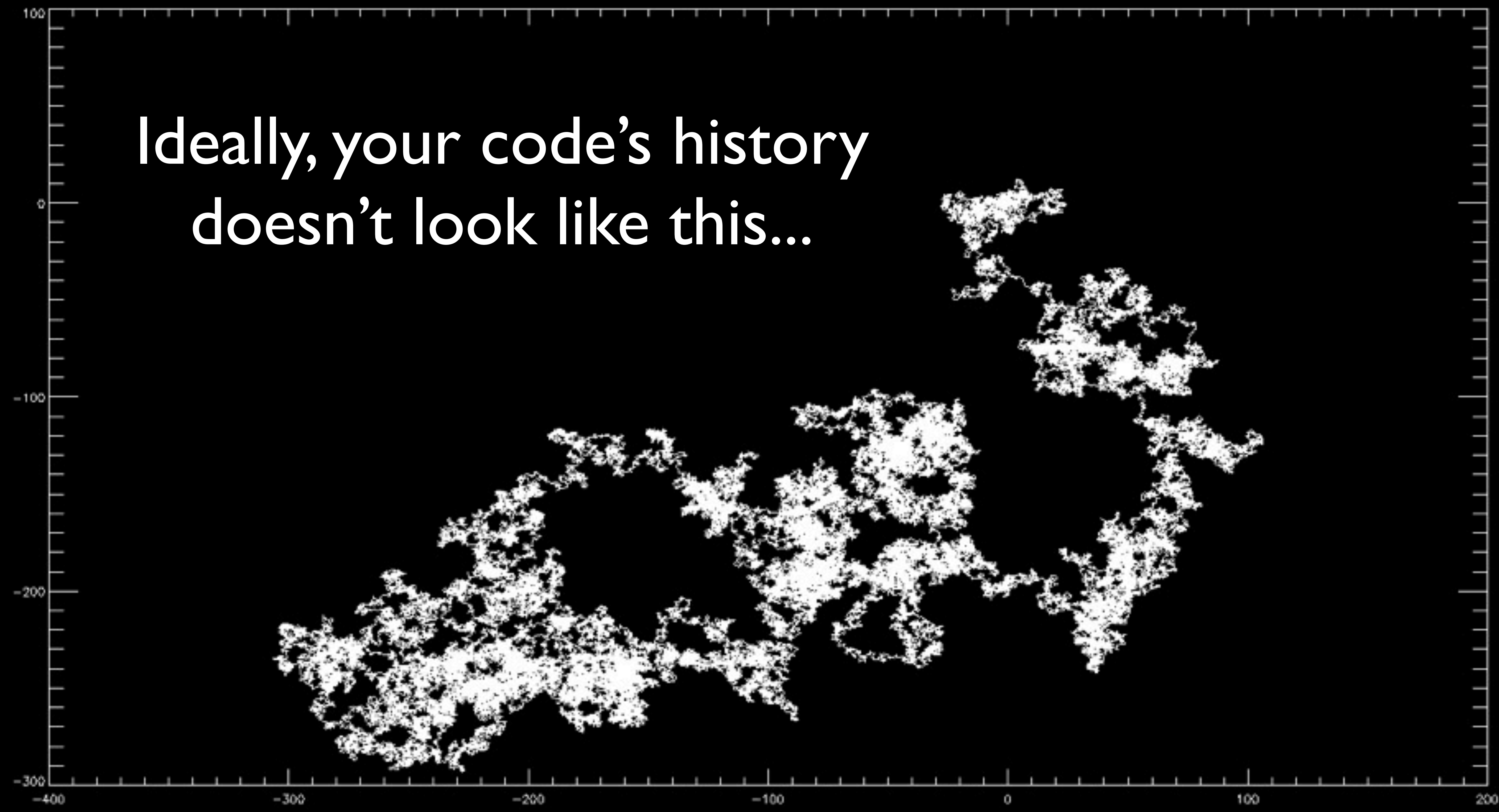
# Ch 14 vs Lecture

- Chapter 14 implements the same general idea, but with different approaches
  - I coded things in a different order
  - I chose different variable manipulations
- You can choose either way!

# “Refactoring”

- If you want to change the code you’re handed to look more like Dewey’s, go ahead
- When you change code so that the code looks different but it does the same thing, that’s called “refactoring”
  - usually, you do it to “clean up” messy code or remove duplicate code

# Brownian Motion Again



Ideally, your code's history doesn't look like this...

# Tutorial 15

- For tutorial 15, we will do Chapter 14
  - With a twist: You will do a “`git commit`” every 10 minutes
  - In this case, it doesn’t matter if your code works at any given step, just keep committing (which means, SAVING too!)
    - Good practice: Save often
    - Will show your “work flow”

# Random Walk Goal

- How does the distance from the starting point change *on average*?
    - The book shows you how to do this: Average the distances of many random walks

# Code Golf

- Bad technique, but useful thought exercise:
- Officially, “Code Golf” is trying to accomplish a task using the fewest characters possible.
  - It’s actually kind of dumb.
- BUT, it can be useful to try to accomplish a task in the fewest *commands* possible

# “Good” code golf

- Terse code is often easier to parse than verbose code
- “Brevity is the soul of wit.”
- But really, we’re interested in a different brevity - *faster* code is better most of the time
  - But, it is NEVER worth sacrificing “correctness” for speed

# Code Optimization

- As a rule, you don't optimize code unless you have to (i.e., unless your code is slow)
- But, I brought up optimization and code golf because the functions in Chapter 14 can be accomplished in probably fewer than 10 lines of code.  Kudos* if you can figure out how!

*I am not actually offering food, just props... for now