Some reminders

- Commit & push your code often, even if it’s not done
  - Just make sure your commit message says “Done with assignment X” if it’s meant to be the last one
- Check the twitter feed when working on homework, it has useful tips

Do you know what vectors are?

A) Yes, I use them frequently in math & physics classes

B) Yes, but haven’t used them lately

C) I’ve heard of them but don’t remember anything about them

D) No, never heard of them

E) None of the above (please tell me what you mean)

Using Loops *for science!*

- For planetary / stellar dynamics, the only important force is gravity
- We can use the equations of motion to determine how orbits progress
- We’ll use a numerical approximation of the equations of motion

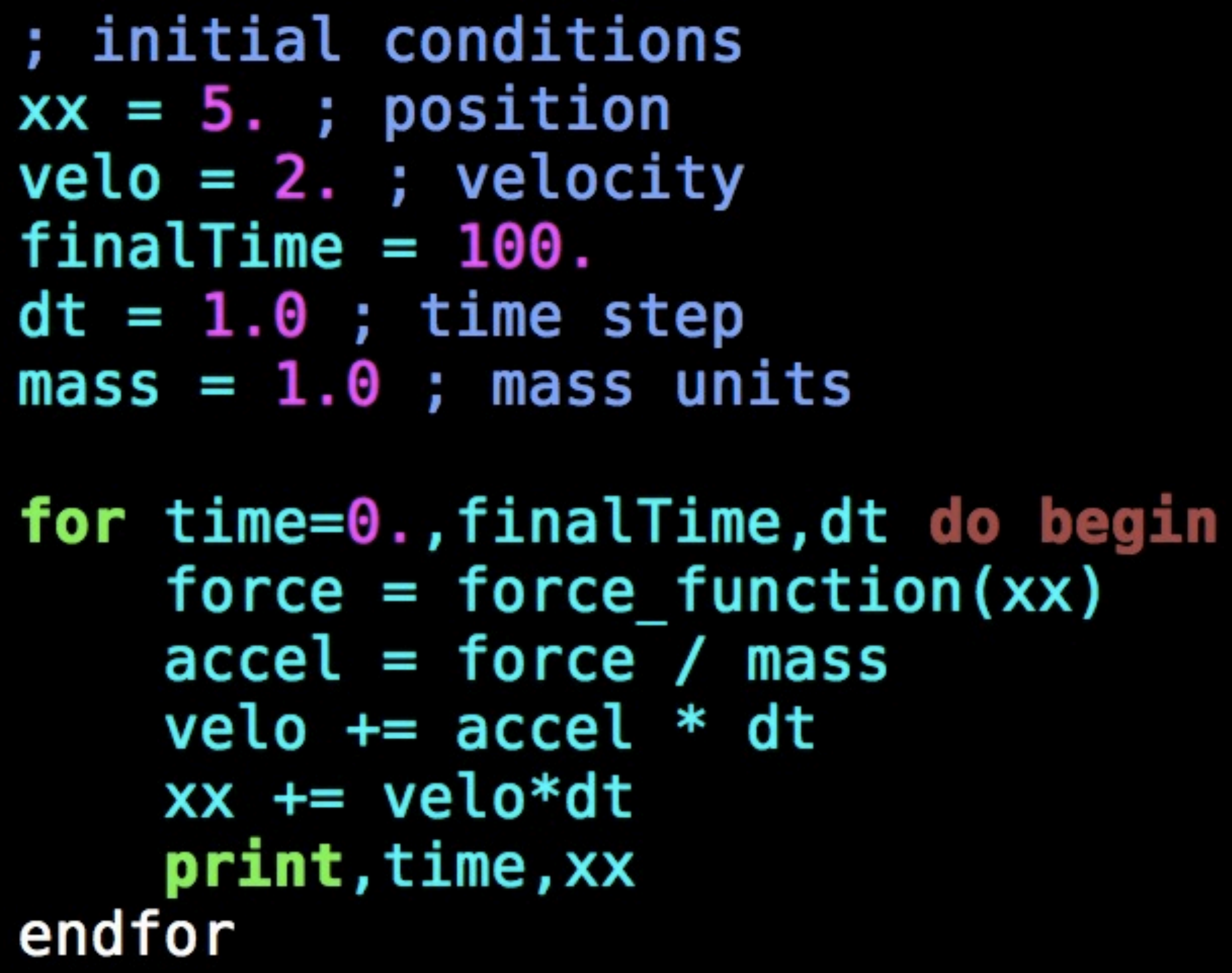
Using Loops *for science!*

- `F = m a` (Newton's 2nd law)
- `a = F / m`
- $a \approx \Delta v / \Delta t$
- $\Delta v \approx a \, \Delta t$
- $v \approx v_0 + \Delta v$
- $\Delta x = v \, \Delta t$
- $x = x_0 + \Delta x$

Looping the EoM

- Start with some initial position and velocity
- Calculate the force at that position
- Calculate the acceleration caused by that force
- Calculate the new velocity due to the acceleration
- Calculate the new position due to the new velocity

Looping the EoM



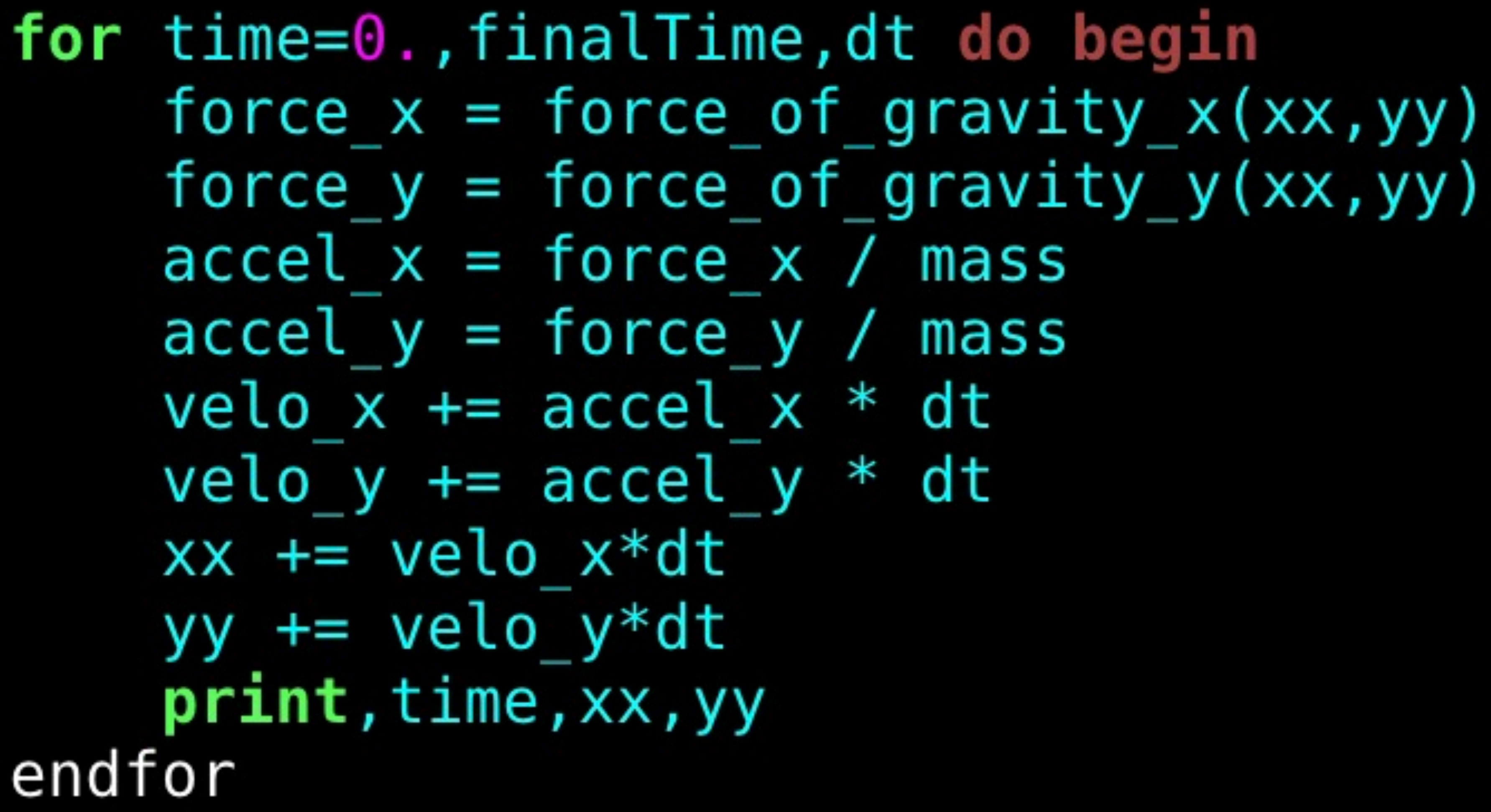
```
; initial conditions
xx = 5. ; position
velo = 2. ; velocity
finalTime = 100.
dt = 1.0 ; time step
mass = 1.0 ; mass units

for time=0.,finalTime,dt do begin
    force = force_function(xx)
    accel = force / mass
    velo += accel * dt
    xx += velo*dt
    print,time,xx
endfor
```

Numerical Methods

- This simple “step forward” method is called the “(forward) Euler Method”
- The error in this method is proportional to the step size (so a smaller step size is better)
- it is also “unstable” - it can diverge from the correct answer drastically

Generalizing to N dimensions

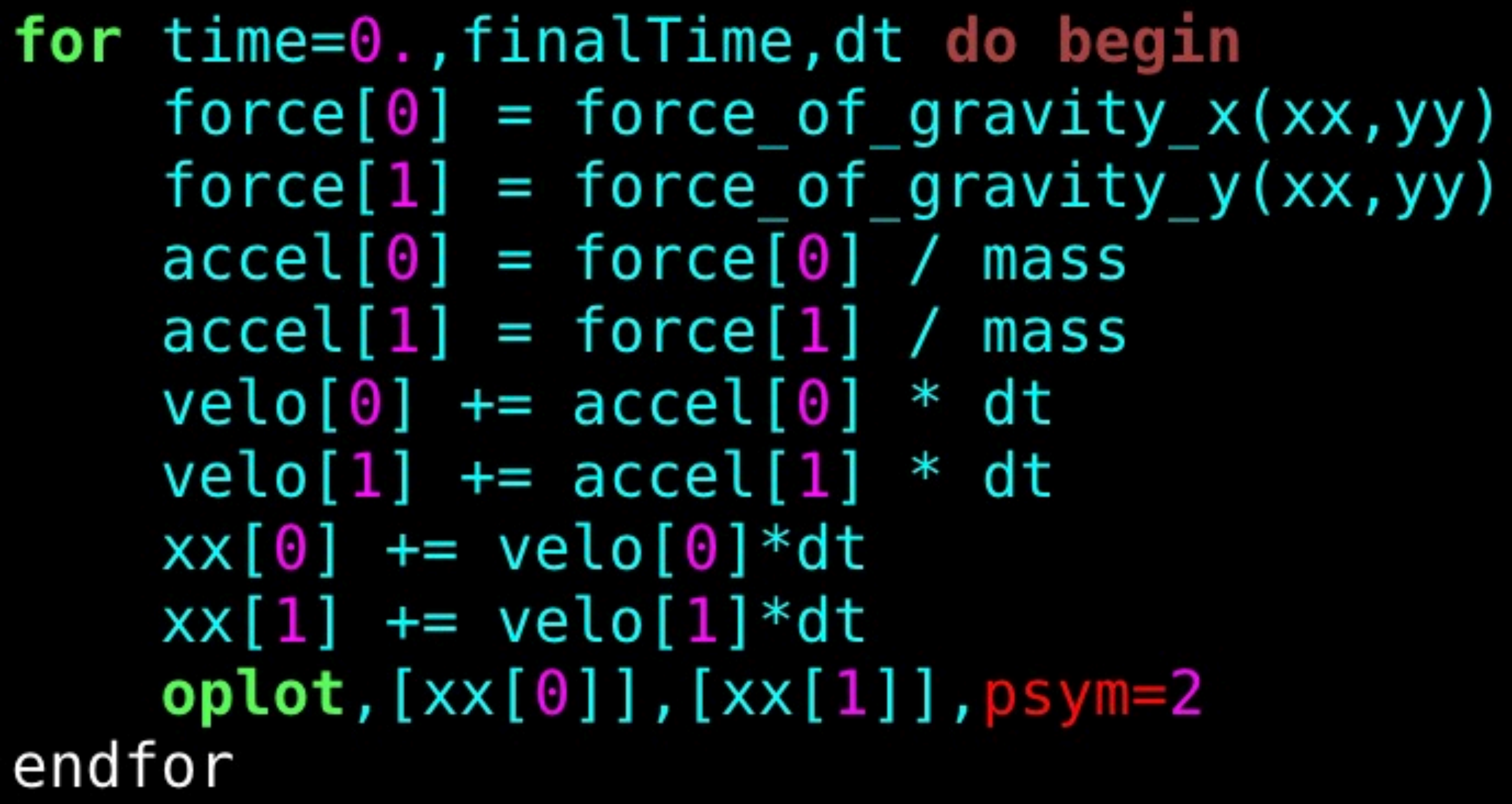
- First, let’s do 2 dimensions:



```
for time=0.,finalTime,dt do begin
    force_x = force_of_gravity_x(xx,yy)
    force_y = force_of_gravity_y(xx,yy)
    accel_x = force_x / mass
    accel_y = force_y / mass
    velo_x += accel_x * dt
    velo_y += accel_y * dt
    xx += velo_x*dt
    yy += velo_y*dt
    print,time,xx,yy
endfor
```

Generalizing to N dimensions

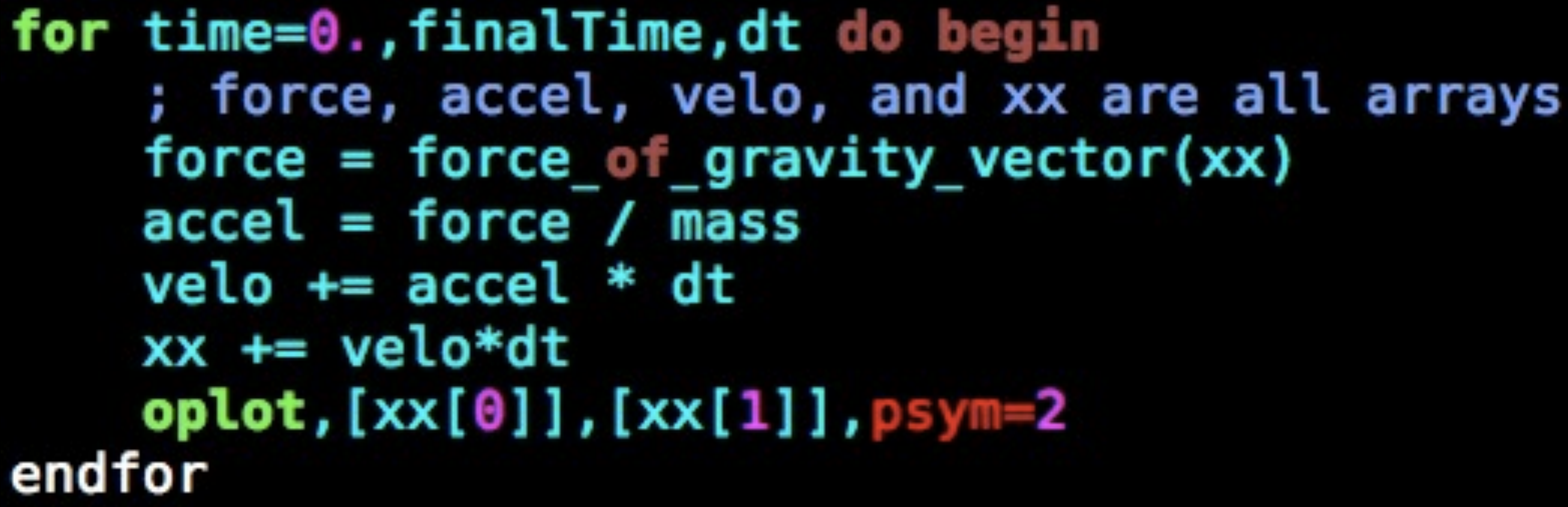
- But then, change to a vector representation:



```
for time=0.,finalTime,dt do begin
    force[0] = force_of_gravity_x(xx,yy)
    force[1] = force_of_gravity_y(xx,yy)
    accel[0] = force[0] / mass
    accel[1] = force[1] / mass
    velo[0] += accel[0] * dt
    velo[1] += accel[1] * dt
    xx[0] += velo[0]*dt
    xx[1] += velo[1]*dt
    oplot,[xx[0]],[xx[1]],psym=2
endfor
```

Generalizing to N dimensions

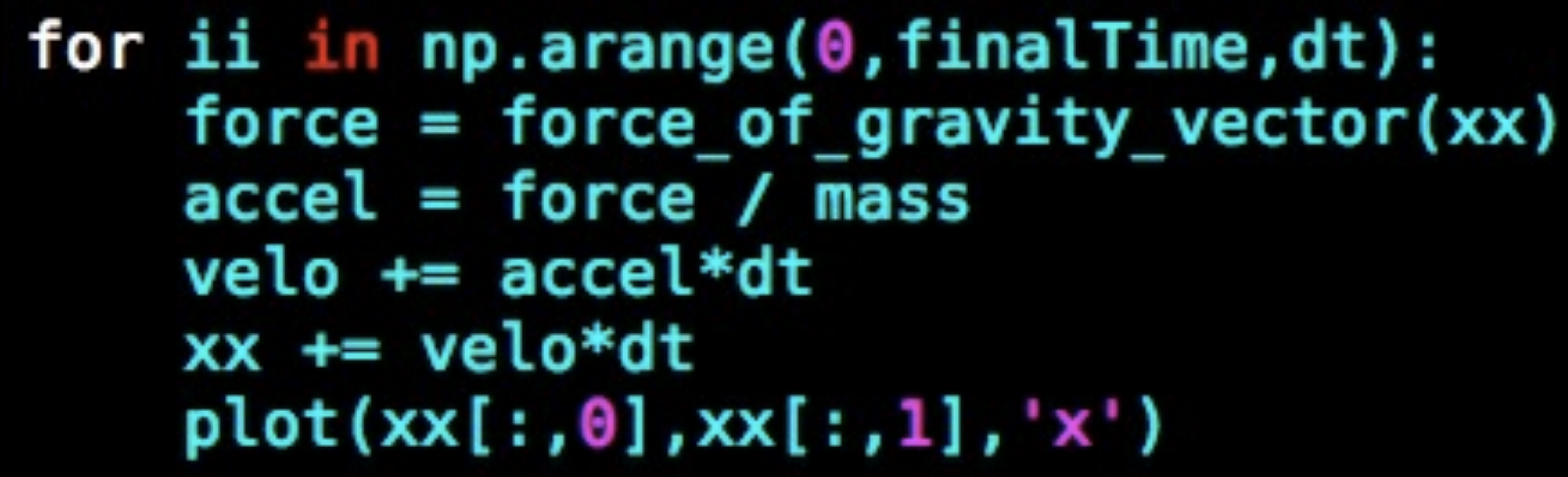
- But we should be taking advantage of array arithmetic



```
for time=0.,finalTime,dt do begin
    ; force, accel, velo, and xx are all arrays
    force = force_of_gravity_vector(xx)
    accel = force / mass
    velo += accel * dt
    xx += velo*dt
    oplot,[xx[0]],[xx[1]],psym=2
endfor
```

Generalizing to N dimensions

- But we should be taking advantage of array arithmetic



```
for ii in np.arange(0,finalTime,dt):
    force = force_of_gravity_vector(xx)
    accel = force / mass
    velo += accel*dt
    xx += velo*dt
    plot(xx[:,0],xx[:,1],'x')
```

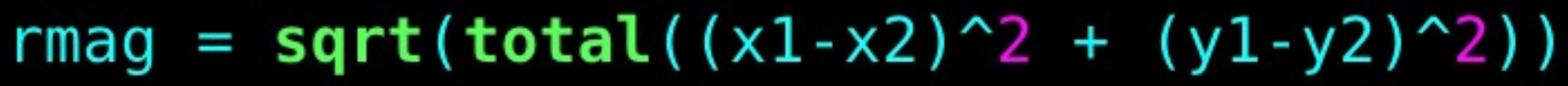
Generalizing to N dimensions

- Now that our code is “general”, i.e. we don’t specify how many dimensions there are at the start, we can do 1D, 2D, or 3D with the same code

Gravity

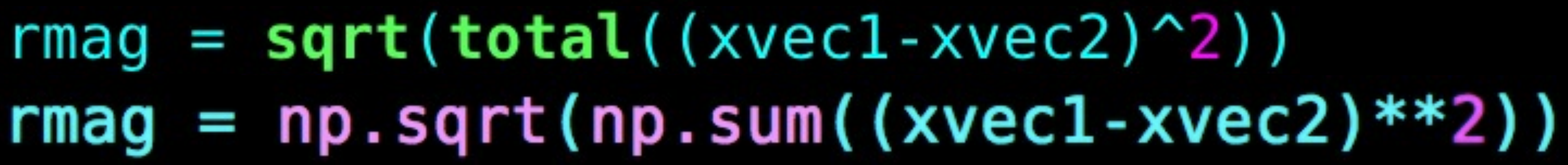
- What is the force vector?
- Newton’s Law of Gravitation:
- $F = \frac{GM_1M_2}{r^2}\hat{r}$
- $\hat{r}$ is a unit vector (i.e., it has a direction, but length=1).
- We know the masses and G, need to find r

Distance

- `|r|`, the "magnitude" of `r`, is the sum of the squares of the distances
- i.e., $|r| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 + ...}$
- 

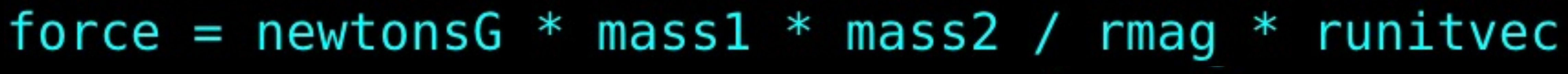
```
rmag = sqrt(total((x1-x2)^2 + (y1-y2)^2))
```

- This can be vectorified easily:



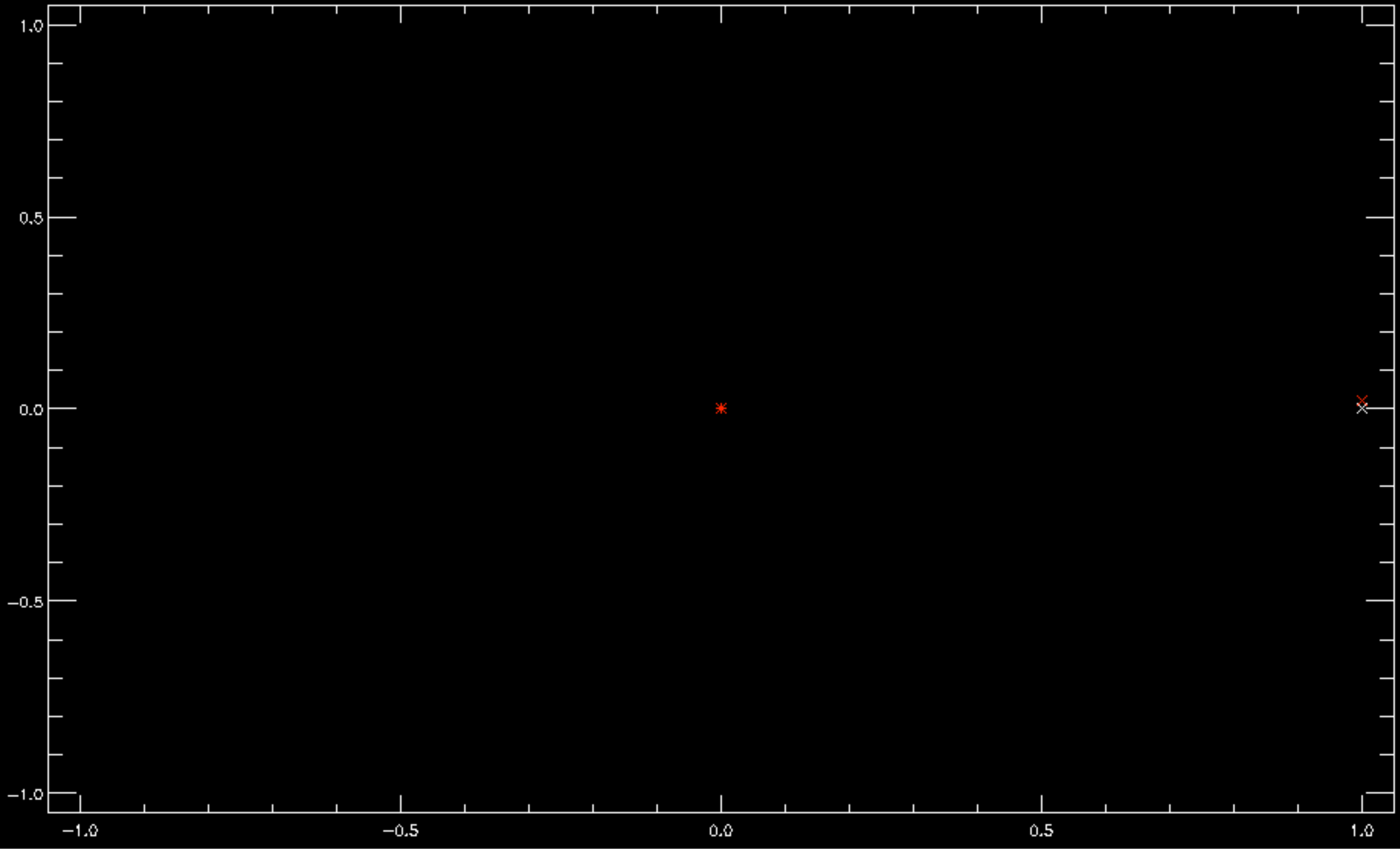
```
rmag = sqrt(total((xvec1-xvec2)^2))
rmag = np.sqrt(np.sum((xvec1-xvec2)**2))
```

Math -> Vector

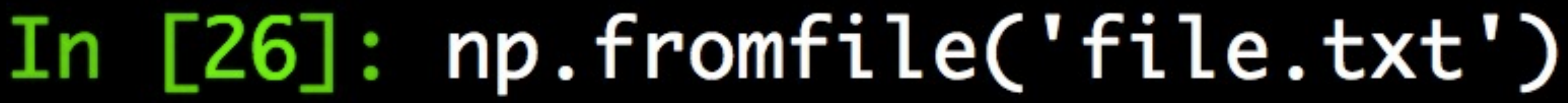
- This equation:
- $F = \frac{GM_1 M_2}{r^2}\hat{r}$
- Becomes:
- 

```
force = newtonsG * mass1 * mass2 / rmag_ * runitvec
```

Example Gravity Simulation



Assuming `file.txt` is a real file, what will this do?



```
In [26]: np.fromfile('file.txt')
```

A) Read the data from `file.txt` into an array, then store it in a named variable

B) Read the data from `file.txt` into an array, then discard that array

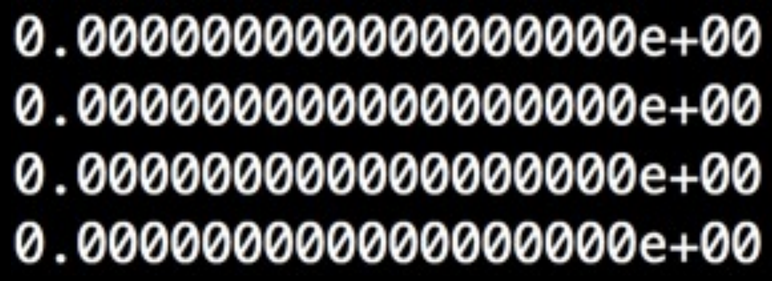
C) Nothing

D) Crash

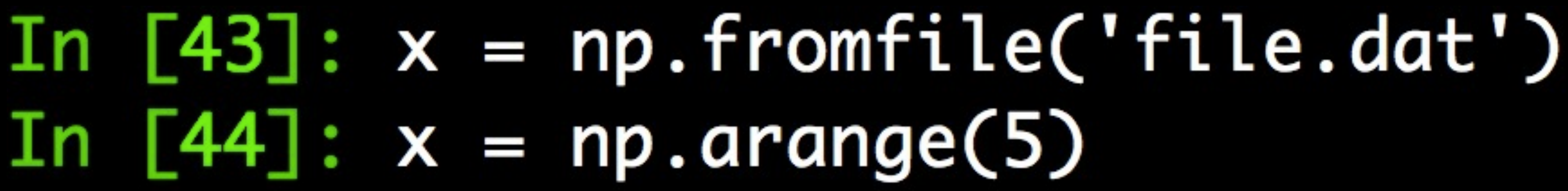
E) Automatically complete & turn in your next 3 assignments

What values will `x` have?

`file.dat` contents:



```
0.000000000000000000e+00
0.000000000000000000e+00
0.000000000000000000e+00
0.000000000000000000e+00
```



```
In [43]: x = np.fromfile('file.dat')
In [44]: x = np.arange(5)
```

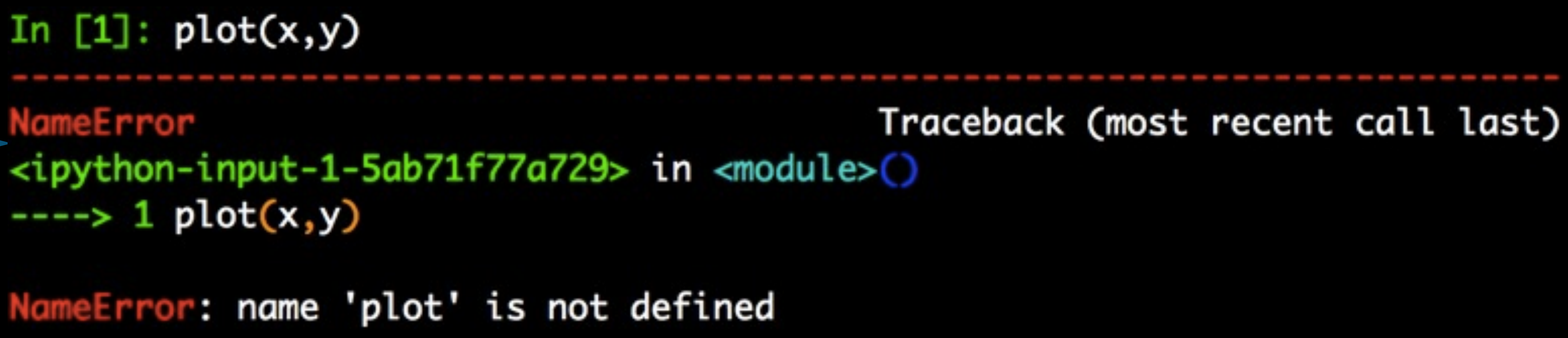
A) [0,0,0,0]

B) [0,1,2,3,4]

C) [0,1,2,3,4,5]

D) [1,2,3,4,5]

E) None of the above



```
In [1]: plot(x,y)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-5ab71f77a729> in <module>()
----> 1 plot(x,y)

NameError: name 'plot' is not defined
```

Which action will resolve (fix) the above error?

A) `from pylab import *`

B) Restart session with `ipython --pylab`

C) If the error is from a script, `%run -i` from a pylab session

D) Change `plot(x,y)` to `pl.plot(x,y)` if you've already done `import pylab as pl`

E) All of the above

What can you do if your plot doesn’t show up?

A) Restart your session as a pylab session

B) Use `show()` on the command line or in the script

C) `%run -i` your script

D) All of the above

E) None of the above

Programs and Functions

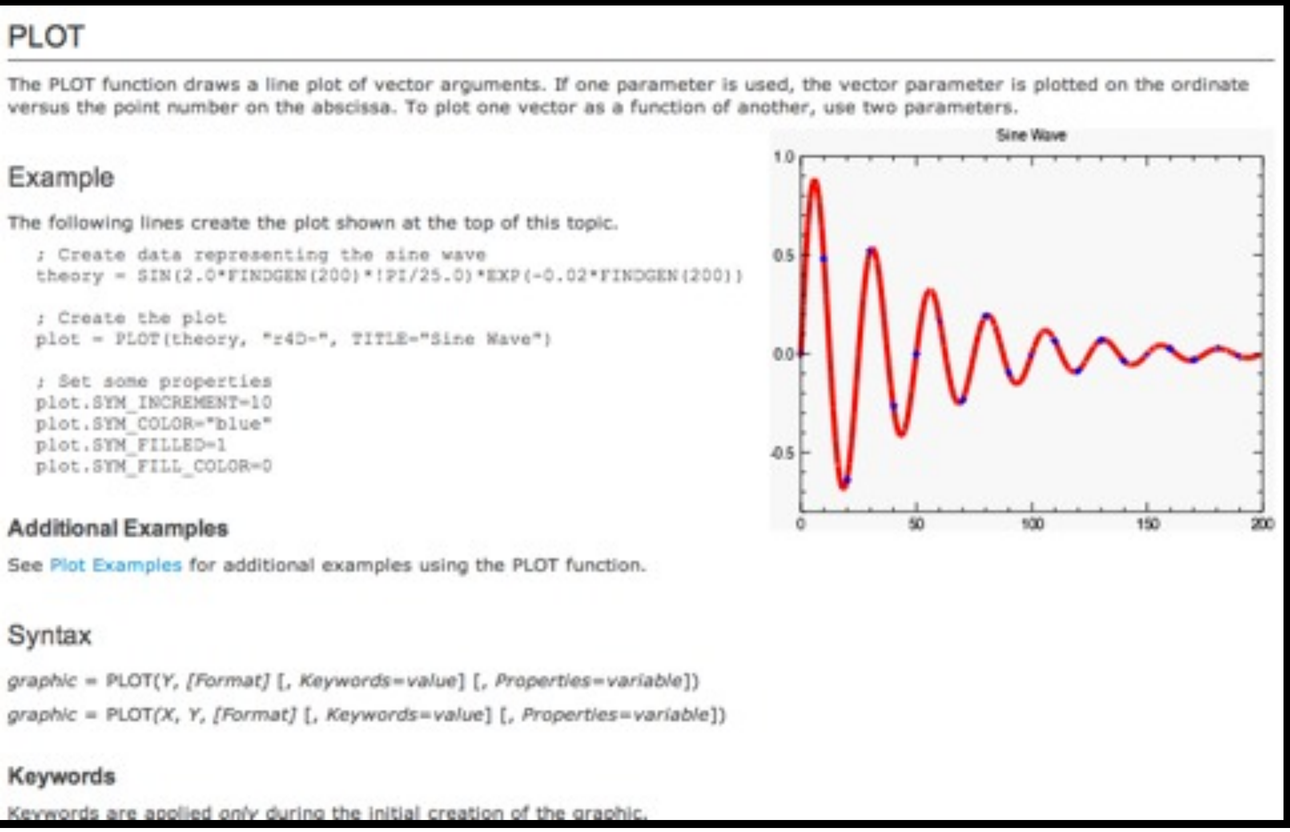
REMINDER: procedure vs function

- The plot function vs the plot procedure
- The plot function is new in IDL 8
  - it’s a lot like the Python plot function
- the plot function is a complicated object-oriented thing
- the plot procedure is what we’re used to

plot procedure vs function

- If you go to the IDL help, make sure you get the right one:

function



PLOT

The PLOT function draws a line plot of vector arguments. If one parameter is used, the vector parameter is plotted on the ordinate versus the point number on the abscissa. To plot one vector as a function of another, use two parameters.

Example

The following lines create the plot shown at the top of this topic.

```
; Create data representing the sine wave
theory = SIN(2.0*FINDGEN(200)*!PI/25.0)*EXP(-0.02*FINDGEN(200))

; Create the plot
plot = PLOT(theory, "r4D-", TITLE="Sine Wave")

; Set some properties
plot.SYM_INCREMENT=10
plot.SYM_COLOR="blue"
plot.SYM_FILLED=1
plot.SYM_FILL_COLOR=0
```

Additional Examples

See Plot Examples for additional examples using the PLOT function.

Syntax

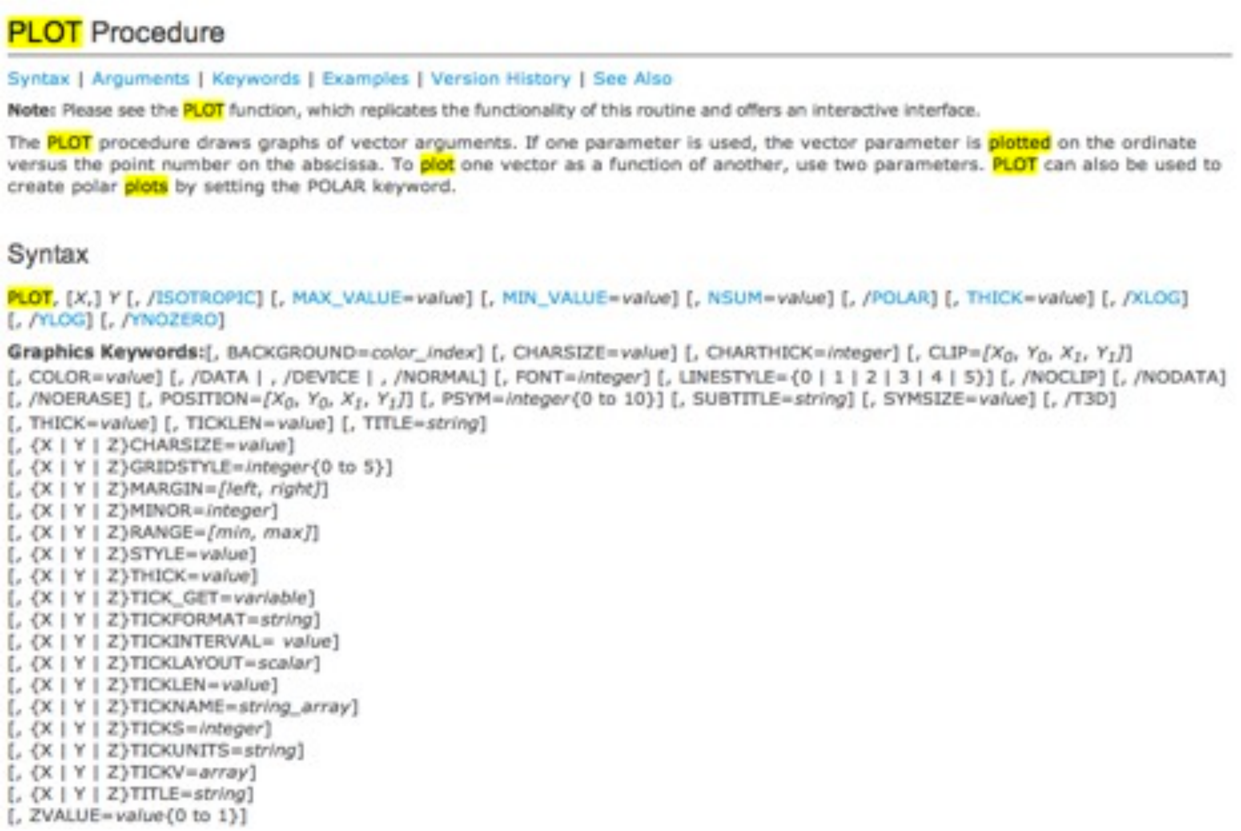
graphic = PLOT(*Y*, *[Format]* [, *Keywords*=*value*] [, *Properties*=*variable*])

graphic = PLOT(*X*, *Y*, *[Format]* [, *Keywords*=*value*] [, *Properties*=*variable*])

Keywords

Keywords are applied *only* during the initial creation of the graphic.

procedure



PLOT Procedure

Syntax | Arguments | Keywords | Examples | Version History | See Also

Note: Please see the PLOT function, which replicates the functionality of this routine and offers an interactive interface.

The PLOT procedure draws graphs of vector arguments. If one parameter is used, the vector parameter is plotted on the ordinate versus the point number on the abscissa. To plot one vector as a function of another, use two parameters. PLOT can also be used to create polar plots by setting the POLAR keyword.

Syntax

PLOT, [*X*,] *Y* [, /ISOTROPIC] [, MAX_VALUE=*value*] [, MIN_VALUE=*value*] [, NSUM=*value*] [, /POLAR] [, THICK=*value*] [, /XLOG] [, /YLOG] [, /YNOZERO]

Graphics Keywords:[, BACKGROUND=*color_index*] [, CHARSIZE=*value*] [, CHARTHICK=*integer*] [, CLIP=[X_0, Y_0, X_1, Y_1]] [, COLOR=*value*] [, /DATA | , /DEVICE | , /NORMAL] [, FONT=*integer*] [, LINESTYLE={0 | 1 | 2 | 3 | 4 | 5}] [, /NOCLIP] [, /NODATA] [, /NOERASE] [, POSITION=[X_0, Y_0, X_1, Y_1]] [, PSYM=*integer*{0 to 10}] [, SUBTITLE=*string*] [, SYMSIZE=*value*] [, /T3D]
[, THICK=*value*] [, TICKLEN=*value*] [, TITLE=*string*]
[, {X | Y | Z}CHARSIZE=*value*]
[, {X | Y | Z}GRIDSTYLE=*integer*{0 to 5}]
[, {X | Y | Z}MARGIN=[*left*, *right*]]
[, {X | Y | Z}MINOR=*integer*]
[, {X | Y | Z}RANGE=[*min*, *max*]]
[, {X | Y | Z}STYLE=*value*]
[, {X | Y | Z}THICK=*value*]
[, {X | Y | Z}TICK_GET=*variable*]
[, {X | Y | Z}TICKFORMAT=*string*]
[, {X | Y | Z}TICKINTERVAL= *value*]
[, {X | Y | Z}TICKLAYOUT=*scalar*]
[, {X | Y | Z}TICKLEN=*value*]
[, {X | Y | Z}TICKNAME=*string_array*]
[, {X | Y | Z}TICKS=*integer*]
[, {X | Y | Z}TICKUNITS=*string*]
[, {X | Y | Z}TICKV=*array*]
[, {X | Y | Z}TITLE=*string*]
[, ZVALUE=*value*{0 to 1}]

What are the differences between functions and procedures?

A) Procedures have `end` statements, functions do not

B) Procedures have `return` statements, functions do not

C) Functions have `return` statements but procedures have `end` statements instead

D) Functions have `return` statements, procedures don’t return anything

E) None of the above

Functions

-
  ```
  function name,arguments
        return,something
  end
  ```

- actually, the previous was a trick question: functions DON’T have to have a return statement!

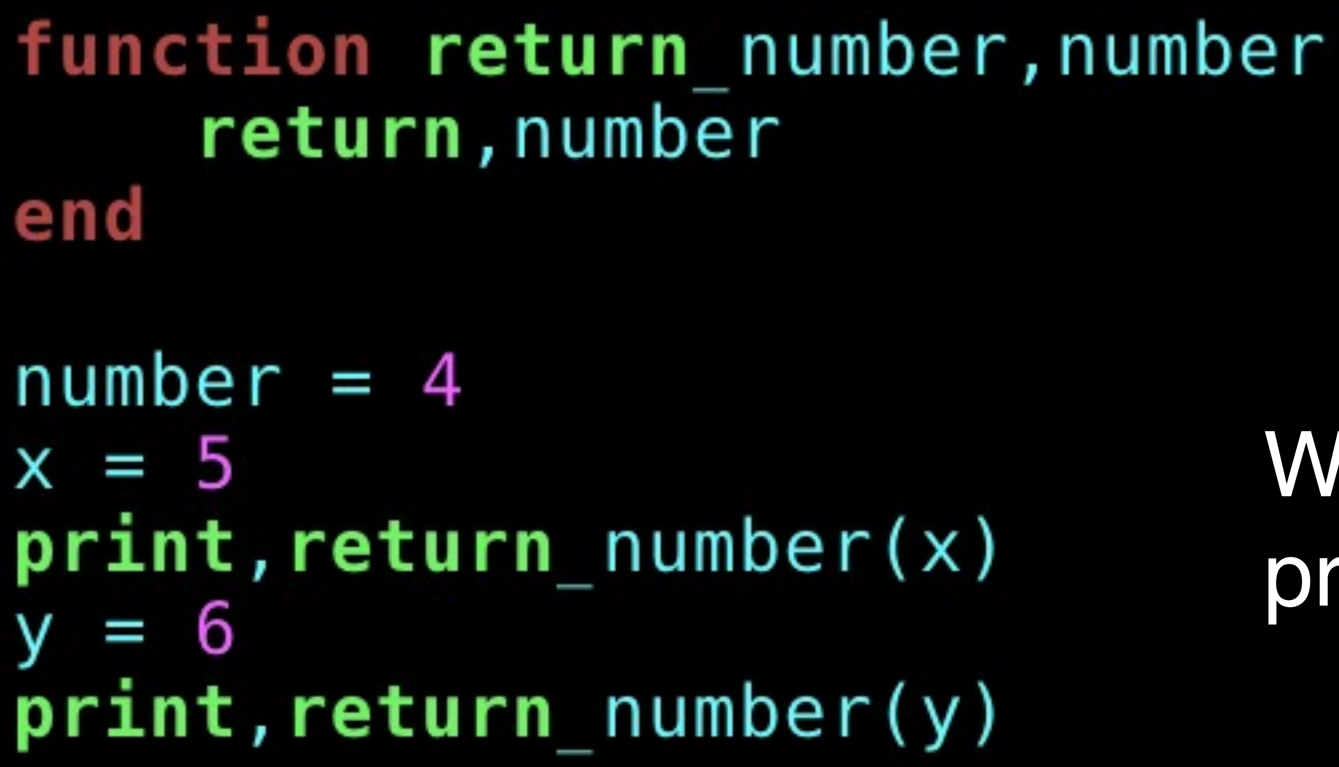
  - if a function has no return statement, it will `return,0`

Procedures

-
  ```
  pro name,arguments
          ; do something
  end
  ```

Arguments

- Arguments are the things passed into functions or procedures
- You can have both non-keyword and keyword arguments



```
function return_number,number
    return,number
end

number = 4
x = 5
print,return_number(x)
y = 6
print,return_number(y)
```

What will this print?

A) 4,4

B) 5,6

C) 5,5

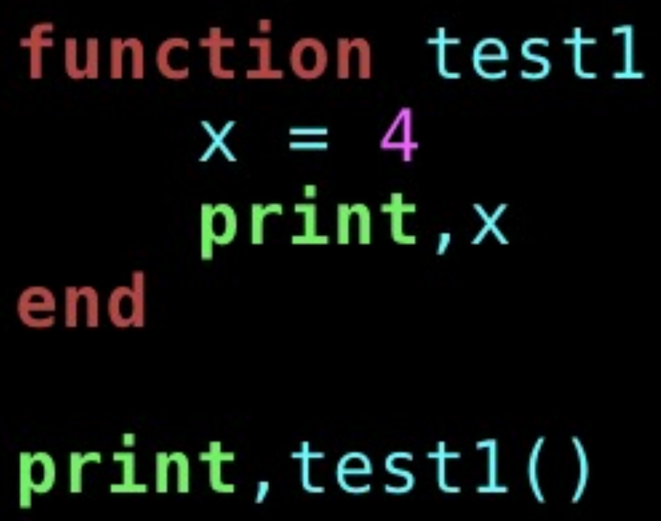
D) 4,5

E) None of the above

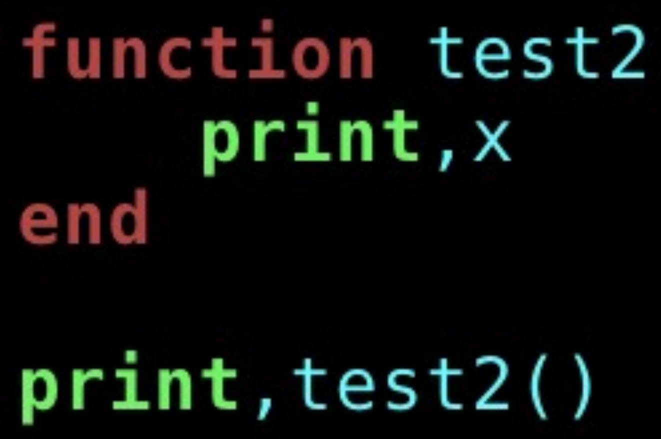
Namespaces

- “namespaces” are a description of where variable names are respected
  - sometimes referred to as “scope”
- Recall that we said *scripts* know what is defined on the interactive command line, but *programs* do not
- Functions, procedures, and programs know nothing about what’s defined in other functions, procedures, and programs

What will these print?



```
function test1
    x = 4
    print,x
end

print,test1()
```



```
function test2
    print,x
end

print,test2()
```

A) 4,0 and crash

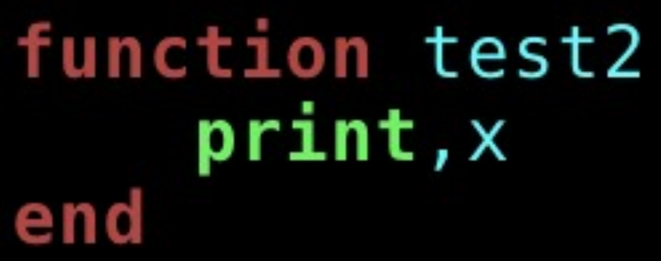
B) 4,0 and 4,0

C) 4 and 4

D) 4 and crash

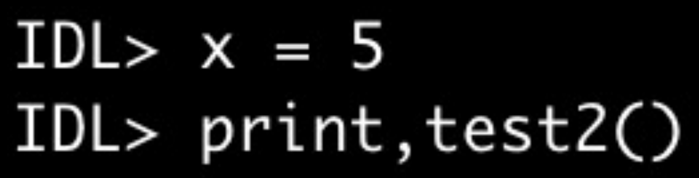
E) None of the above / I don’t know

Given that you’ve compiled this function:



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

Will this work?



```
IDL> x = 5
IDL> print,test2()
```

A) Yes

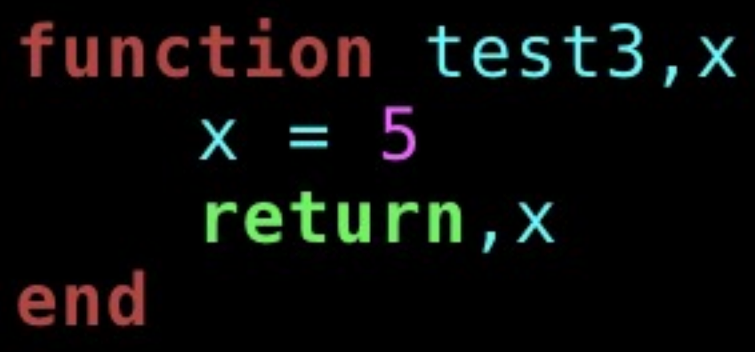
B) No

C) C, D, or E

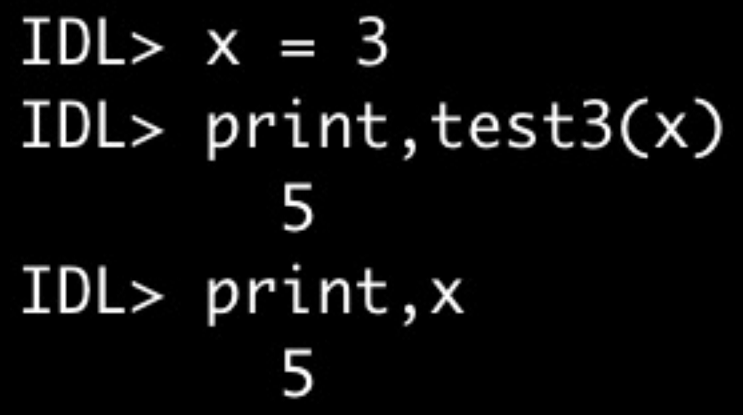
Pass by reference vs Pass by value

- If you pass a *named variable* to an IDL function, any changes to that variable will persist afterwards
    - This is called “pass by reference” because you’re passing what the variable points to

“Pass by Reference” example



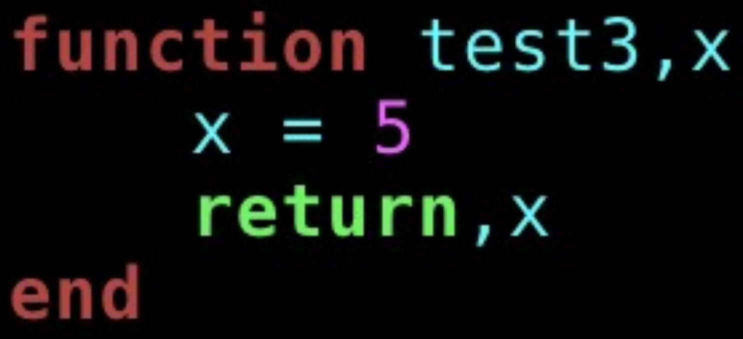
```
function test3,x
    x = 5
    return,x
end
```



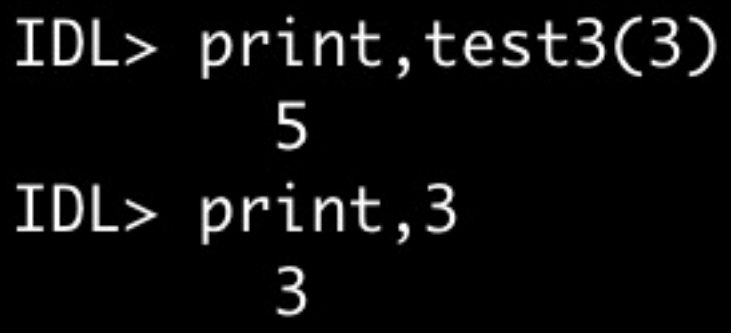
```
IDL> x = 3
IDL> print,test3(x)
       5
IDL> print,x
       5
```

- test3 *changes the value of x*

“Pass by Reference” example



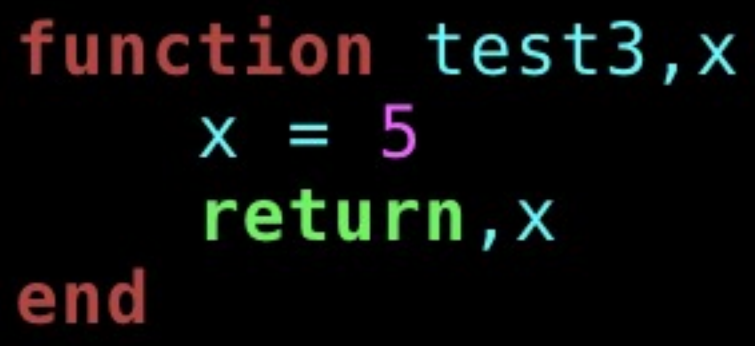
```
function test3,x
    x = 5
    return,x
end
```



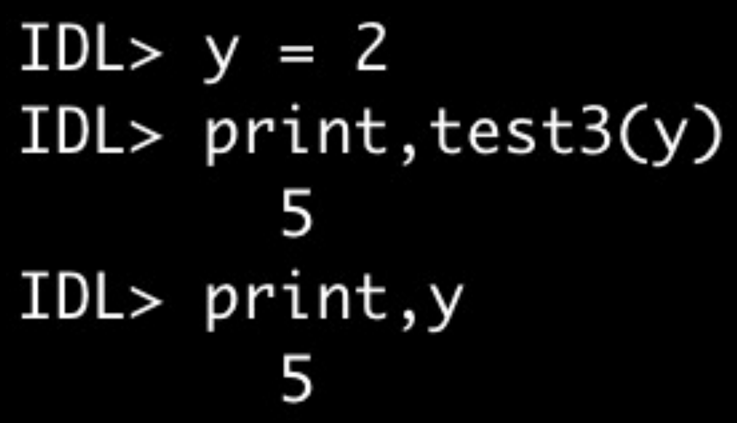
```
IDL> print,test3(3)
       5
IDL> print,3
       3
```

- test3 *does not* change the value of 3: 3 is not a named variable, it’s just a number

“Pass by Reference” example

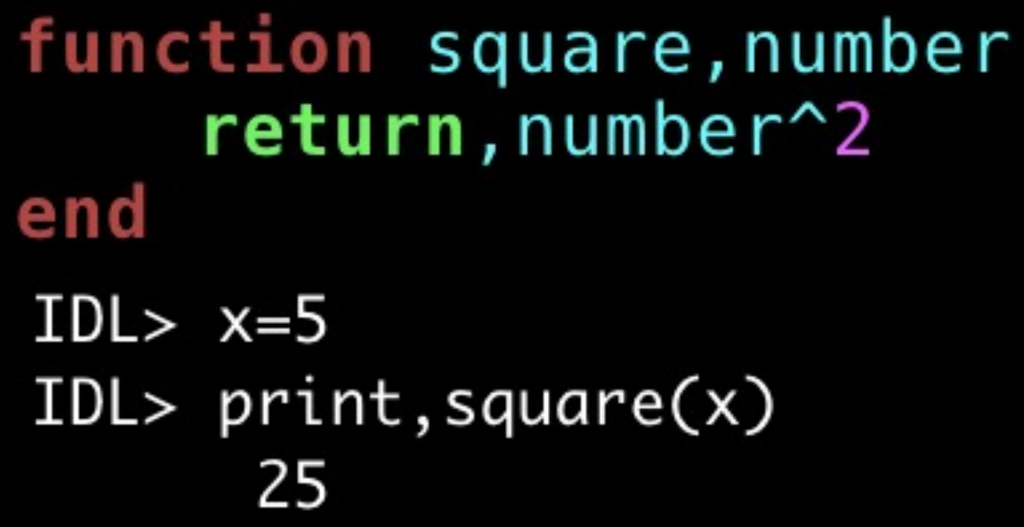


```
function test3,x
    x = 5
    return,x
end
```



```
IDL> y = 2
IDL> print,test3(y)
       5
IDL> print,y
       5
```

- test3 *changes the value of y* even though the variable name is different
- What was “y” outside the function is now “x” inside the function



```
function square,number
    return,number^2
end
```

```
IDL> x=5
IDL> print,square(x)
      25
```

What is `x` now?

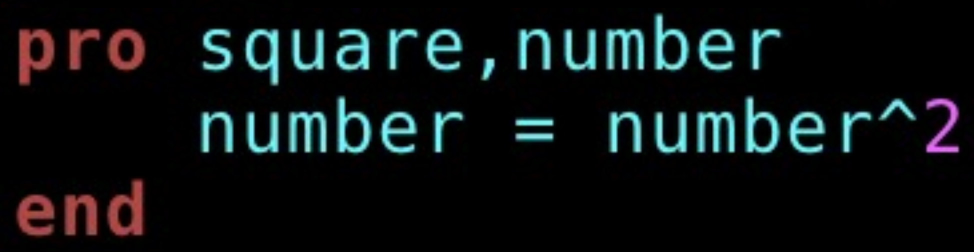
A) 5

B) 25

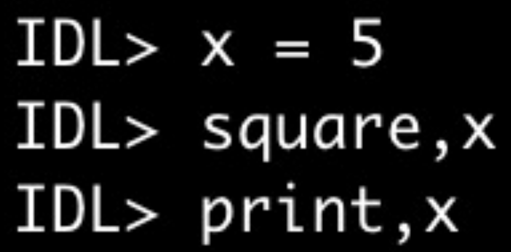
C) 625

D) “square”

E) None of the above



```
pro square,number
    number = number^2
end
```



```
IDL> x = 5
IDL> square,x
IDL> print,x
```

What is `x` now?

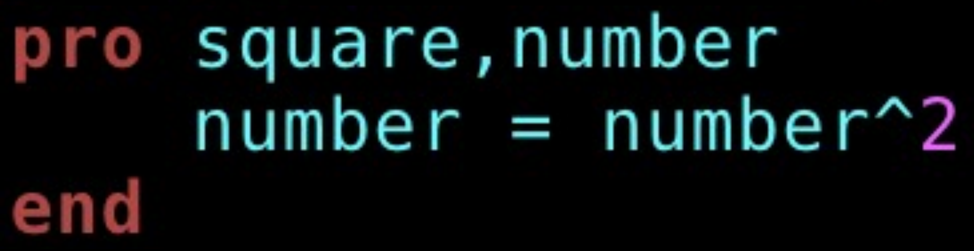
A) 5

B) 25

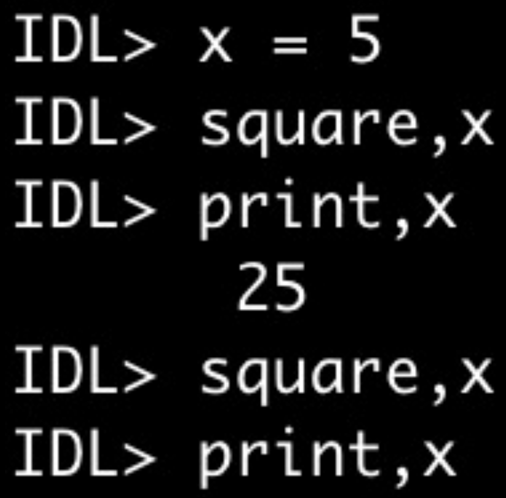
C) 625

D) “square”

E) None of the above



```
pro square,number
    number = number^2
end
```



```
IDL> x = 5
IDL> square,x
IDL> print,x
       25
IDL> square,x
IDL> print,x
```

What is x now?

A) 5

B) 25

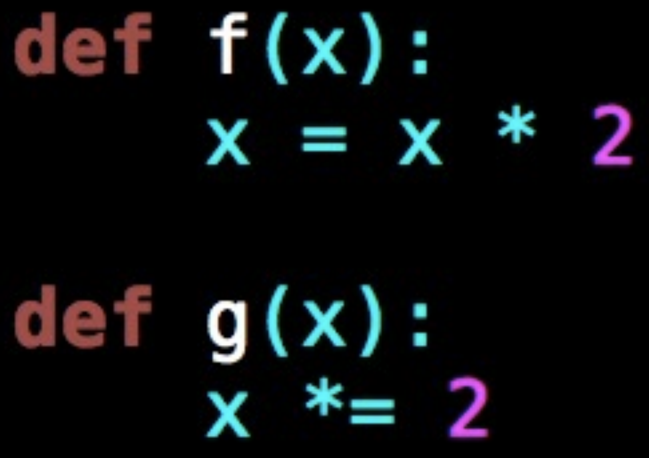
C) 625

D) “square”

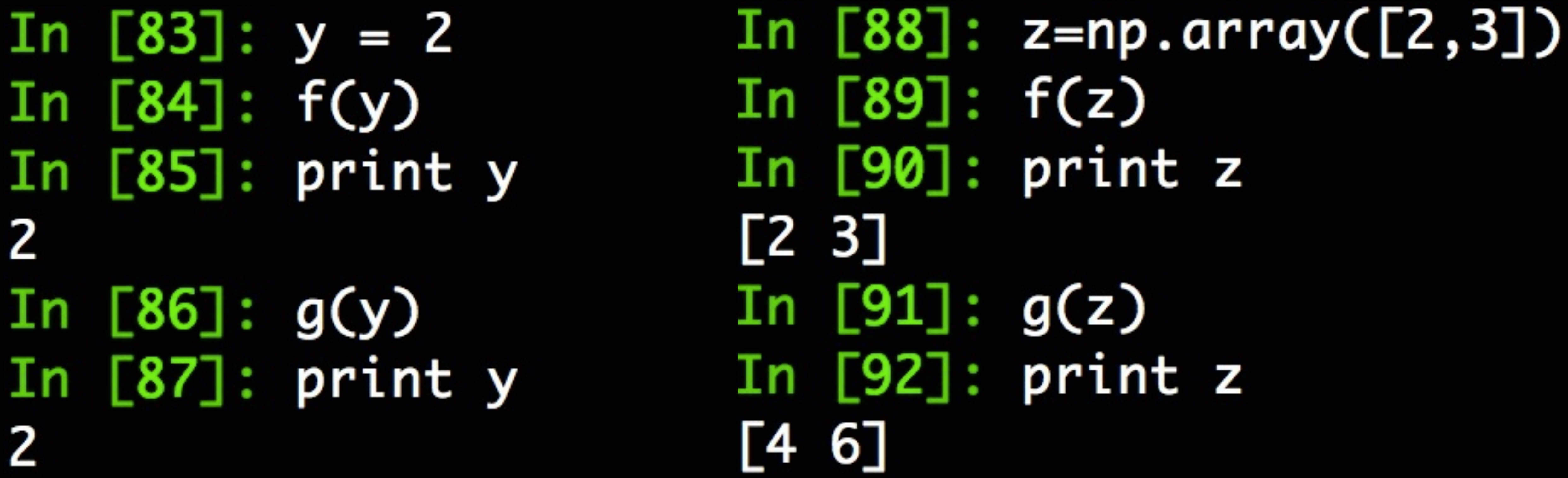
E) None of the above

Python is different

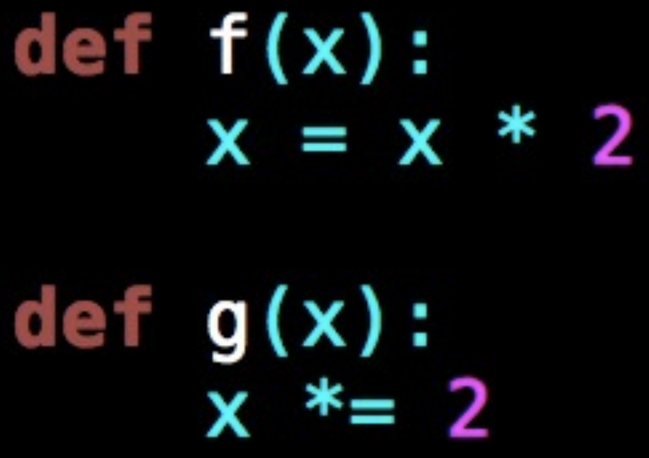
- Everything is “passed by reference”
- But only in-place modification changes the variable
  - and only if the variable supports in-place modification



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

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



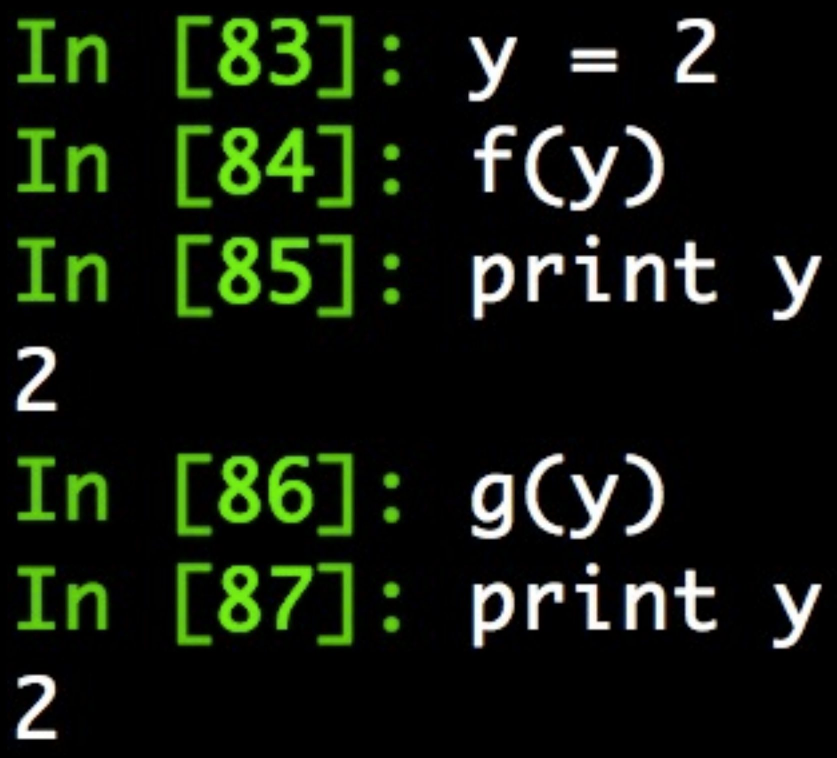
```
In [83]: y = 2                In [88]: z=np.array([2,3])
In [84]: f(y)                 In [89]: f(z)
In [85]: print y              In [90]: print z
2                             [2 3]
In [86]: g(y)                 In [91]: g(z)
In [87]: print y              In [92]: print z
2                             [4 6]
```



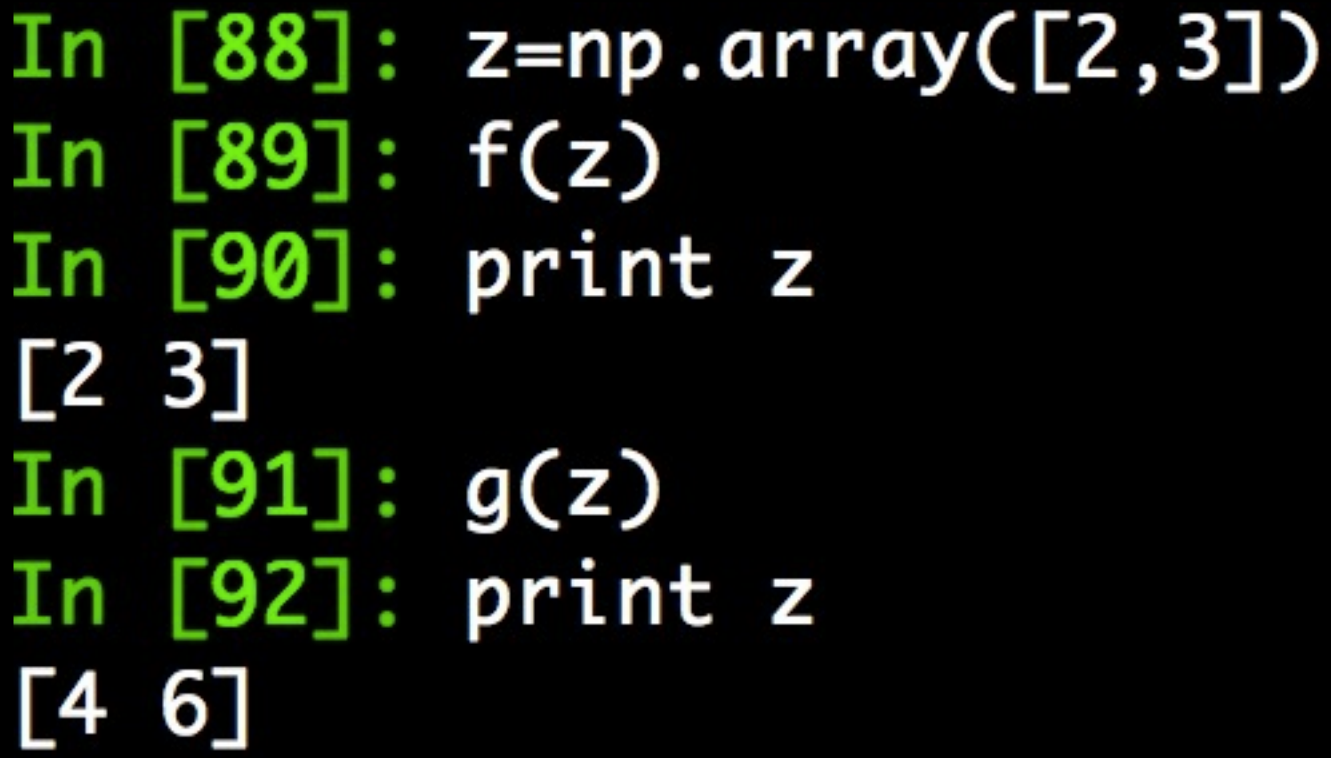
```
def f(x):
    x = x * 2

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

“assignment”

“in-place modification”

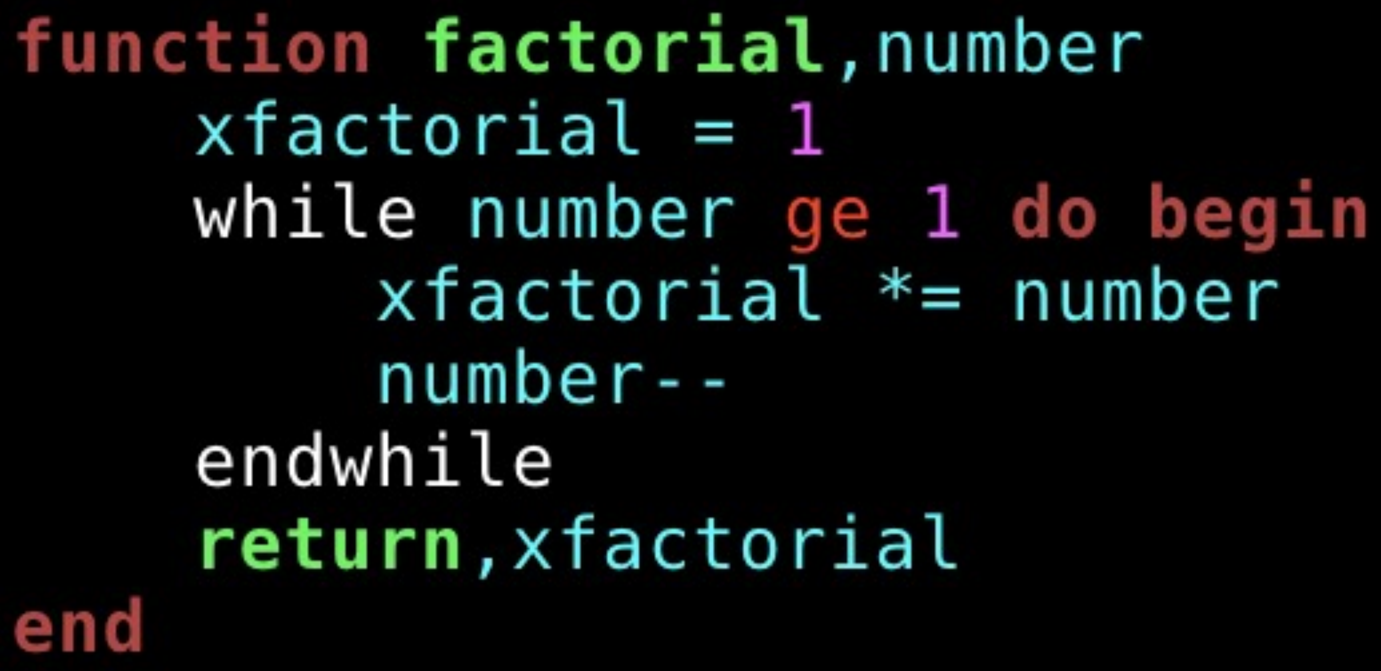


```
In [83]: y = 2
In [84]: f(y)
In [85]: print y
2
In [86]: g(y)
In [87]: print y
2
```



```
In [88]: z=np.array([2,3])
In [89]: f(z)
In [90]: print z
[2 3]
In [91]: g(z)
In [92]: print z
[4 6]
```

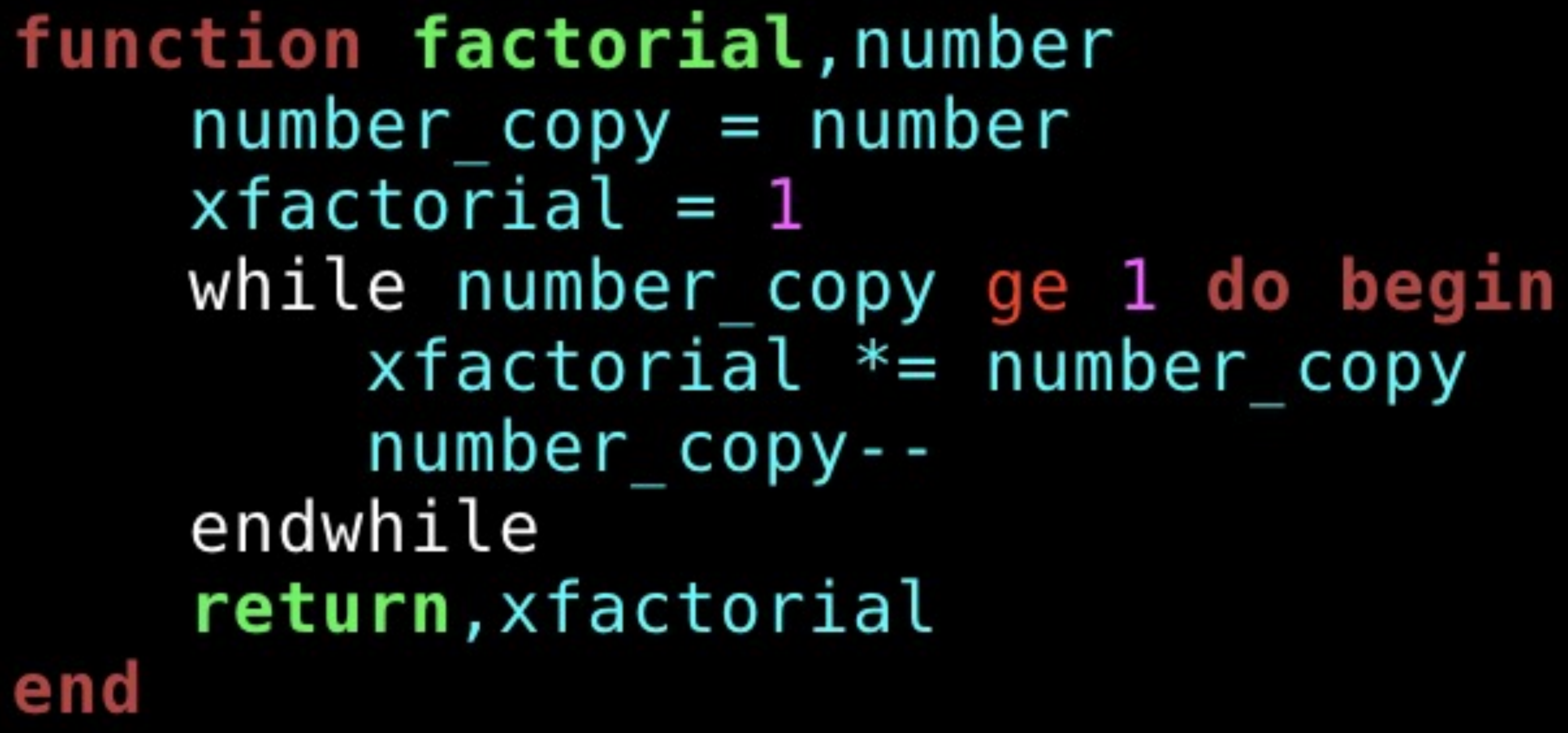
Variable Changes



```
function factorial,number
    xfactorial = 1
    while number ge 1 do begin
        xfactorial *= number
        number--
    endwhile
    return,xfactorial
end
```

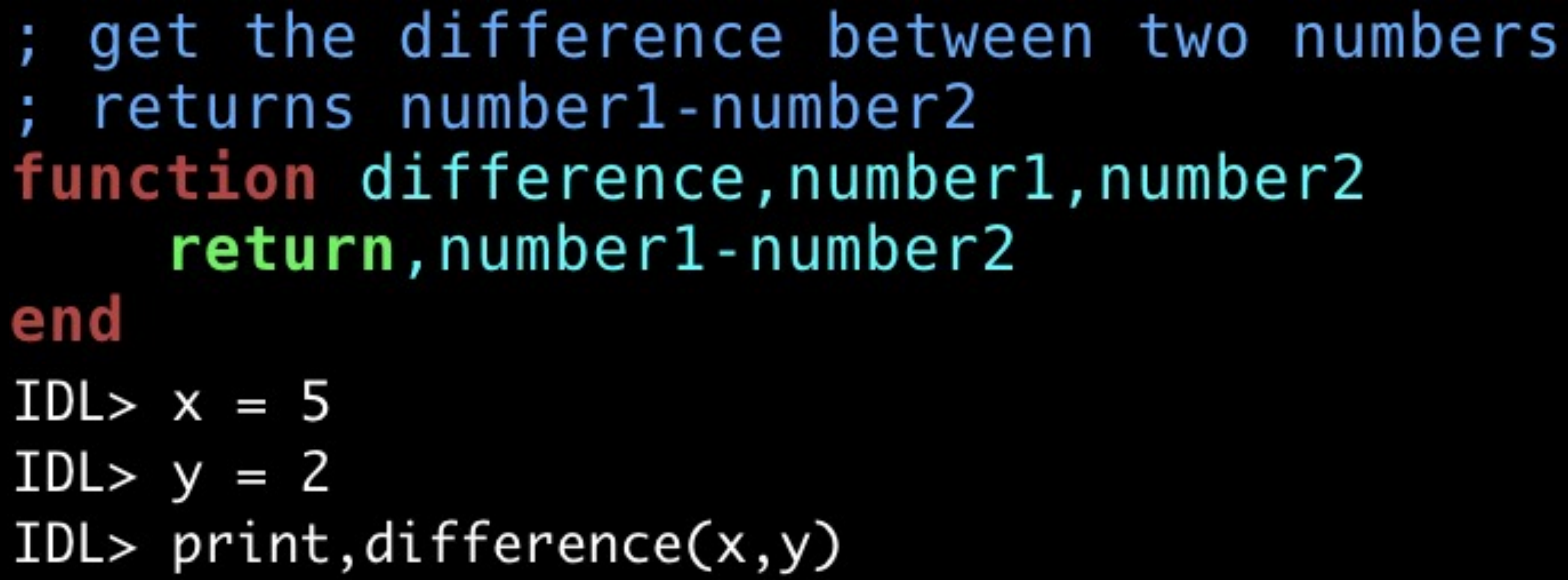
- This function needs to change the input in order to work
- But you don’t necessarily want to change it. What do you do?

Variable Changes



```
function factorial,number
    number_copy = number
    xfactorial = 1
    while number_copy ge 1 do begin
        xfactorial *= number_copy
        number_copy--
    endwhile
    return,xfactorial
end
```

- Copy the number before changing it



```
; get the difference between two numbers
; returns number1-number2
function difference,number1,number2
    return,number1-number2
end
IDL> x = 5
IDL> y = 2
IDL> print,difference(x,y)
```

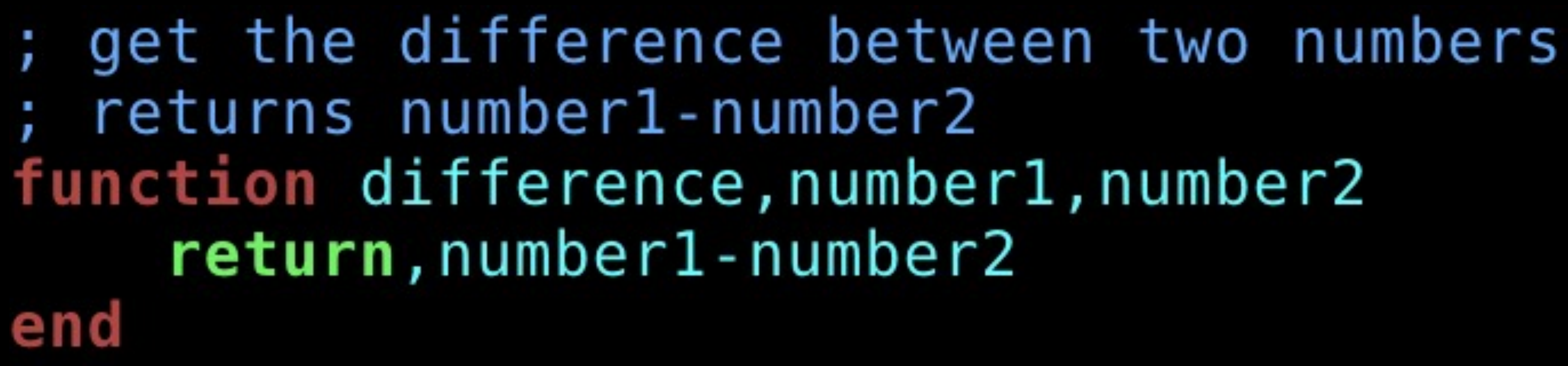
A) -3

B) 3

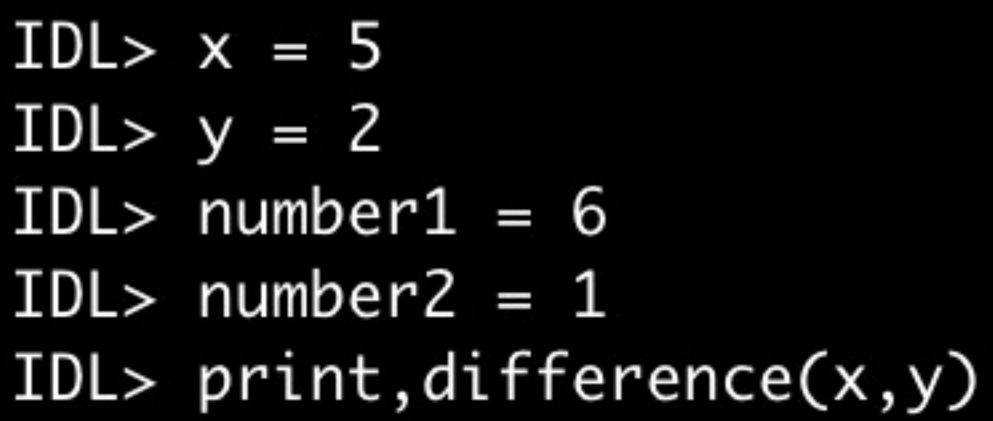
C) 0

D) 3,0

E) None of the above



```
; get the difference between two numbers
; returns number1-number2
function difference,number1,number2
    return,number1-number2
end
```



```
IDL> x = 5
IDL> y = 2
IDL> number1 = 6
IDL> number2 = 1
IDL> print,difference(x,y)
```

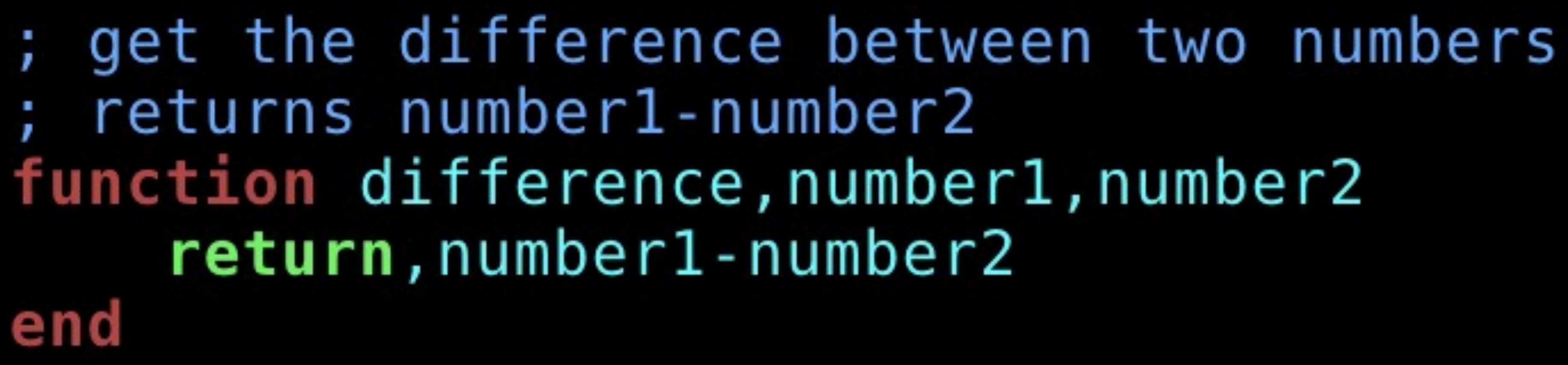
A) -3

B) 3

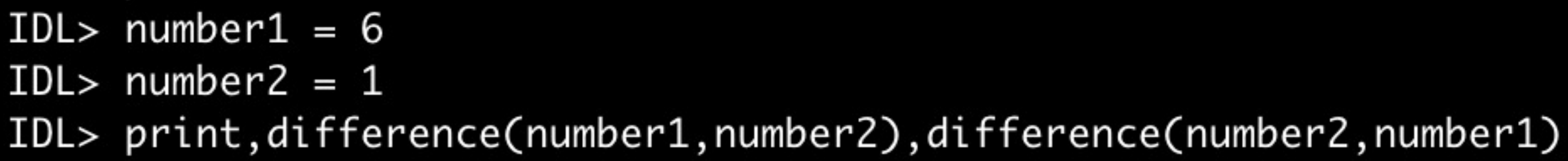
C) 5

D) -5

E) None of the above



```
; get the difference between two numbers
; returns number1-number2
function difference,number1,number2
    return,number1-number2
end
```



```
IDL> number1 = 6
IDL> number2 = 1
IDL> print,difference(number1,number2),difference(number2,number1)
```

A) 5,-5

B) -5,5

C) 5,5

D) -5,-5

E) None of the above

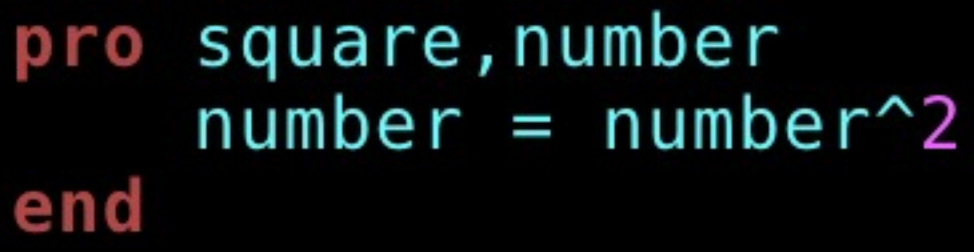
Expressions

- “Expressions” in mathematics are the things on either side of an equation, or in general any combination of numbers and operations
  - x+y
  - 2^3
  - 2+(3+4)

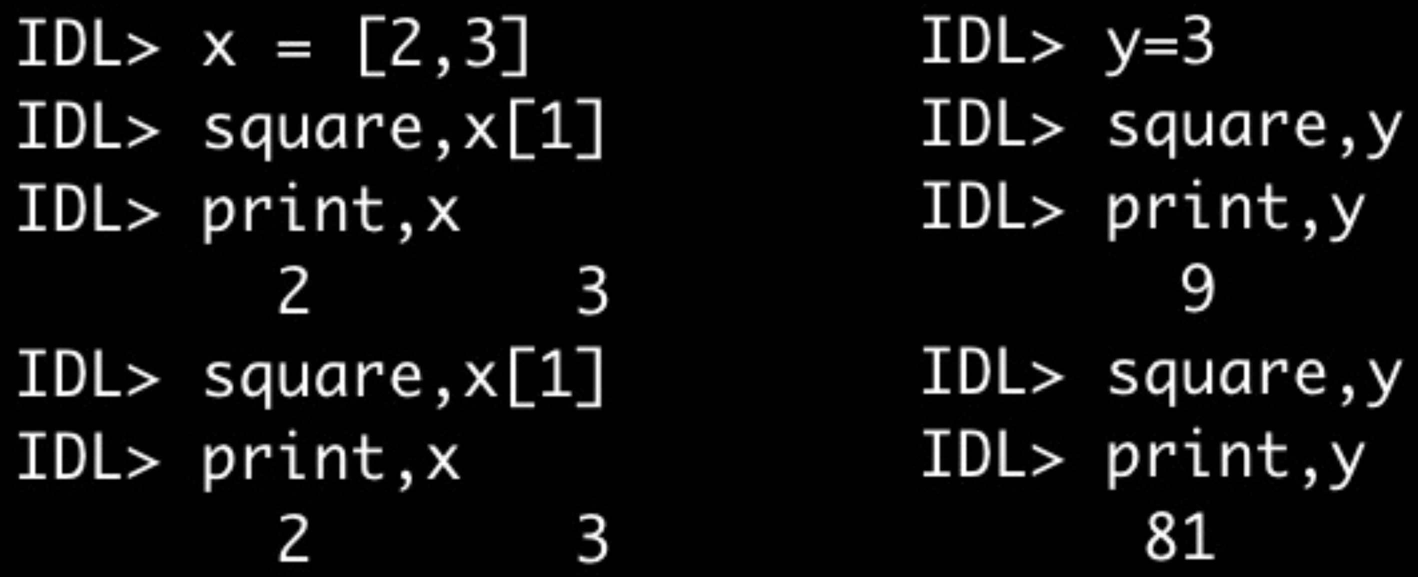
Expressions

- In IDL, you can make expressions
- any mathematical expression is OK
- `x[3]` is an expression
  - it is *not* a named variable
- `x` is an expression
  - it *is* a named variable

Only Named Variables can be modified by procedures

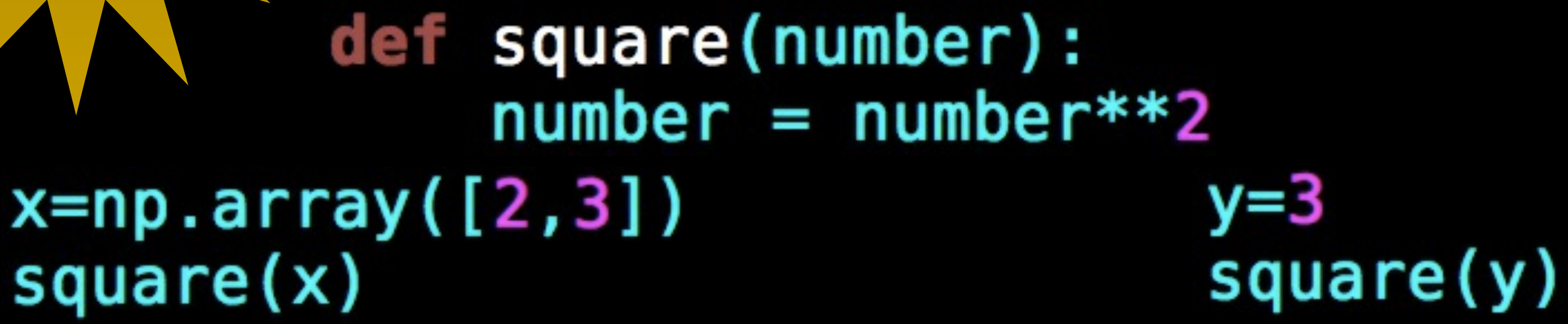


```
pro square,number
    number = number^2
end
```



```
IDL> x = [2,3]                IDL> y=3
IDL> square,x[1]              IDL> square,y
IDL> print,x                  IDL> print,y
       2       3                     9
IDL> square,x[1]              IDL> square,y
IDL> print,x                  IDL> print,y
       2       3                    81
```

What will python do?



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

```
x=np.array([2,3])
square(x)
```

```
y=3
square(y)
```

A) x=[2,3] y=3

B) x=[4,9] y=3

C) x=[2,3] y=9

D) x=[4,9] y=9

E) None of the above