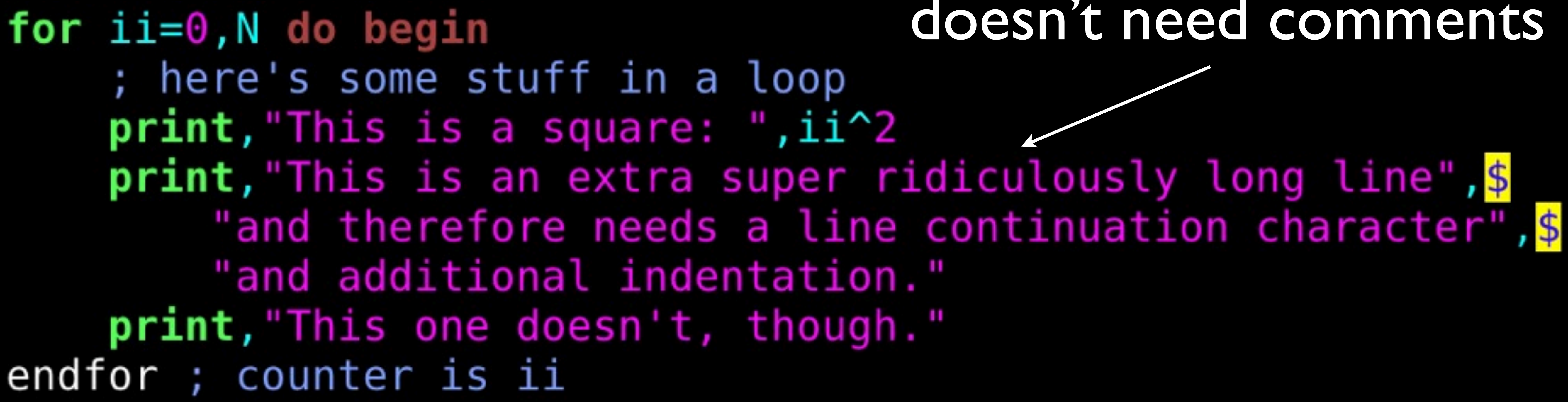
Loop Readability

Self-commenting code doesn't need comments



```
for ii=0,N do begin
    ; here's some stuff in a loop
    print,"This is a square: ",ii^2
    print,"This is an extra super ridiculously long line",$
        "and therefore needs a line continuation character",$
        "and additional indentation."
    print,"This one doesn't, though."
endfor ; counter is ii
```

label your "endfor" so if it's on a different page than the start, you know which one it corresponds to

Homework Status: How’s it going?

A) Fine

B) Alright

C) Not so good

D) Terrible

E) Other

SURVEY: How long did Assignment 4 exercises & WDIDs take?

A) <~ 1 hour

B) ~ 2 hours

C) ~ 3 hours

D) > 3 hours

E) I didn’t do the exercises & WDIDs

SURVEY: How long did Assignment 3 homework take?

A) <~ 1 hour

B) ~ 2 hours

C) ~ 3 hours

D) > 3 hours

E) I didn’t do / haven’t finished the homework (but if you know how long it will take you, answer one of the others)

The List data type

- New in IDL 8
- Start with a reminder about arrays, though:

Evaluate: `print,[0,0e,0d]`

A)
```
     0.00000     0.00000     0.00000
```

B)
```
      0.0000000      0.0000000      0.0000000
```

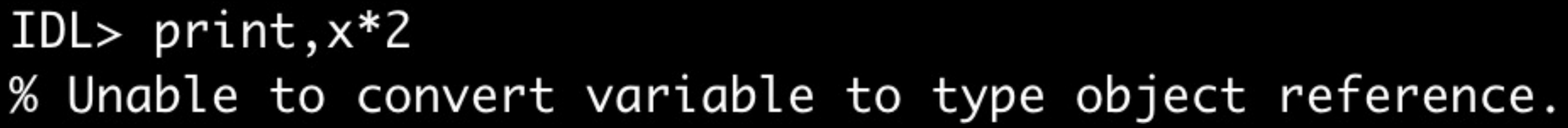
C)
```
           0       0       0
```

D)
```
 0 0 0
```

E) None of the above / I don’t know

List Data Type

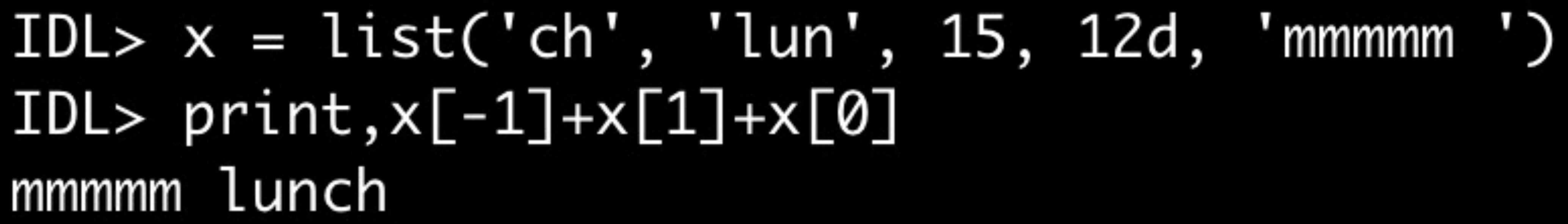
- `x = list(1,’1’,1.0,1d,1L)`
- Each element of `x` retains its data type
- Array operations (like multiplication) aren’t available for lists



```
IDL> print,x*2
% Unable to convert variable to type object reference.
```

List Data Type

- However, you can still access the list elements by index:



```
IDL> x = list('ch', 'lun', 15, 12d, 'mmmmm ')
IDL> print,x[-1]+x[1]+x[0]
mmmmm lunch
```

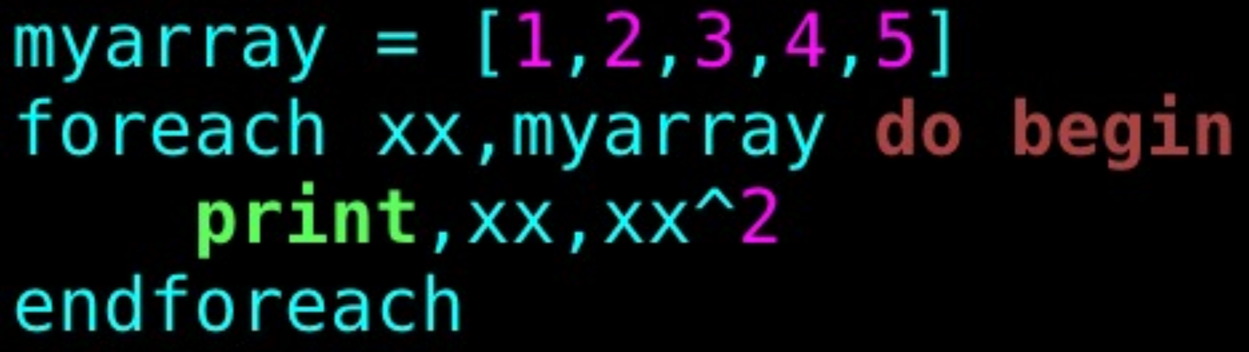
List Data Type

- You can loop through lists, but you can’t treat them as arrays
- What does it mean to “loop through” something?
- Bad old days:

  -
    ```
    for ii=0,n_elements(array)-1 do print,array[ii]
    ```

foreach

- New in IDL 8, much more sensible: the `foreach` loop
  - behaves like the python `for` loop
- Means “For each element of the array (or list), store that element in a variable, and do something with it.”



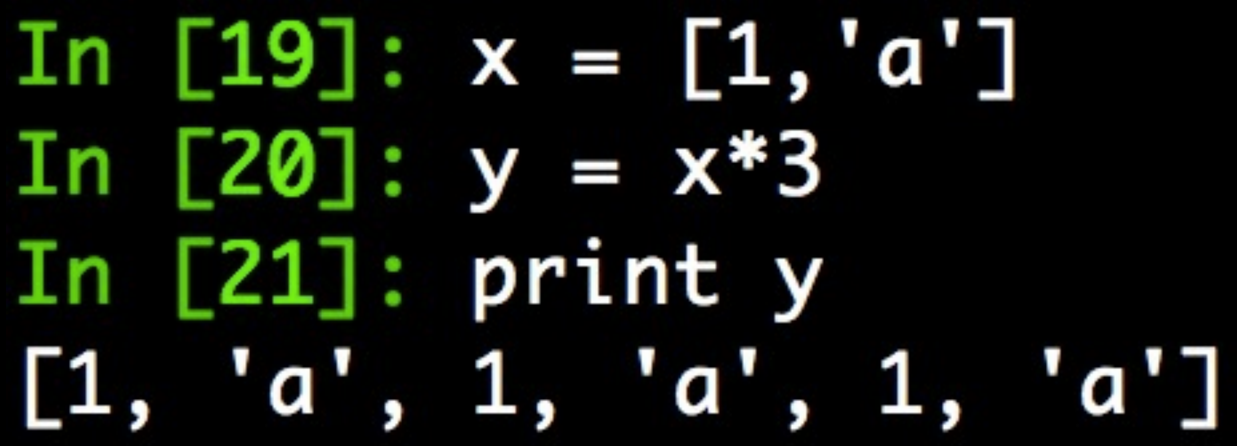
```
myarray = [1,2,3,4,5]
foreach xx,myarray do begin
    print,xx,xx^2
endforeach
```

Lists

- Lists can contain *any* variable type
  - lists can include arrays
  - lists can include lists!
- Lists are a default data type in python, but are new in IDL: they were added to make IDL a little more like python

Python lists vs IDL lists

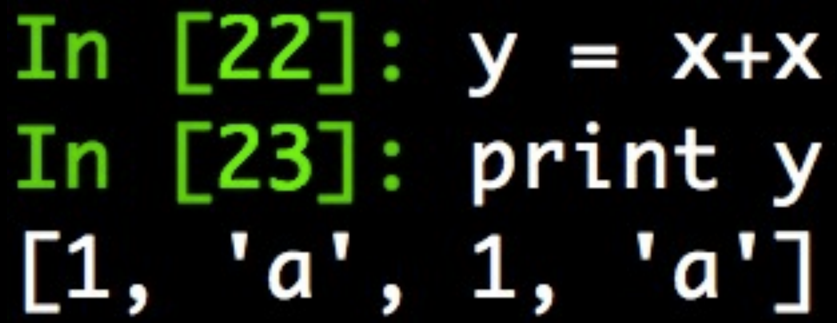
- Very similar, but you can do different things with python lists, e.g.:
  - multiply them - means “make 3 copies and stick them together”



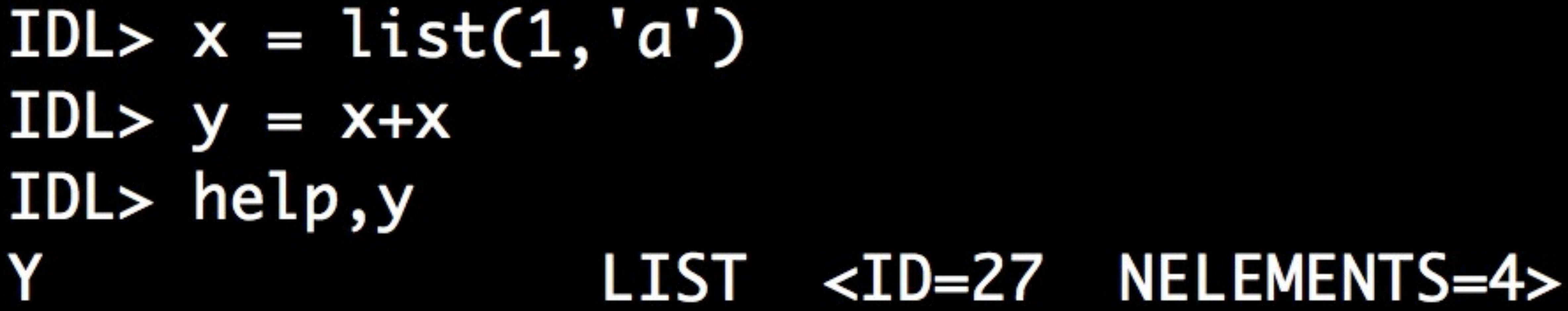
```
In [19]: x = [1,'a']
In [20]: y = x*3
In [21]: print y
[1, 'a', 1, 'a', 1, 'a']
```

List Operations

- BOTH IDL and python can concatenate lists using the + operator:

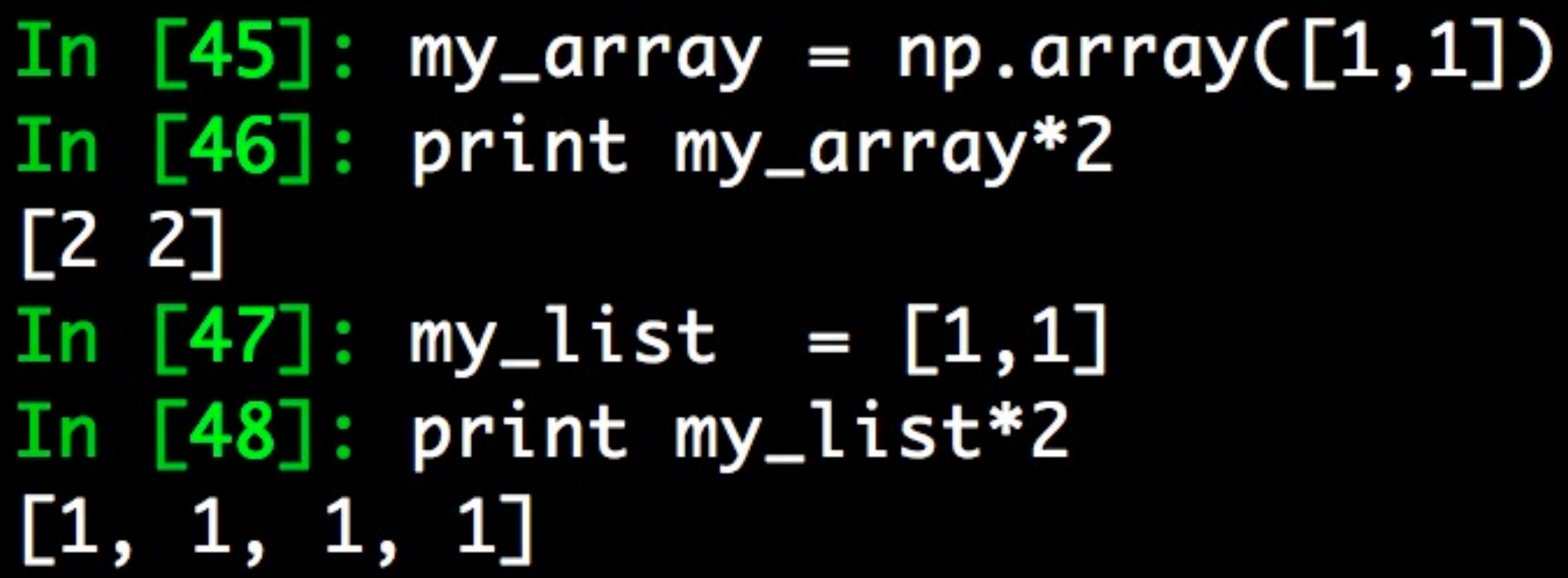


```
In [22]: y = x+x
In [23]: print y
[1, 'a', 1, 'a']
```



```
IDL> x = list(1,'a')
IDL> y = x+x
IDL> help,y
Y               LIST  <ID=27  NELEMENTS=4>
```

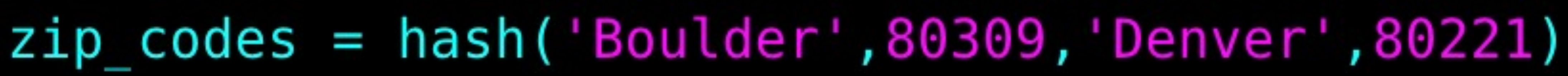
Arrays vs. Lists



```
In [45]: my_array = np.array([1,1])
In [46]: print my_array*2
[2 2]
In [47]: my_list  = [1,1]
In [48]: print my_list*2
[1, 1, 1, 1]
```

Hashes

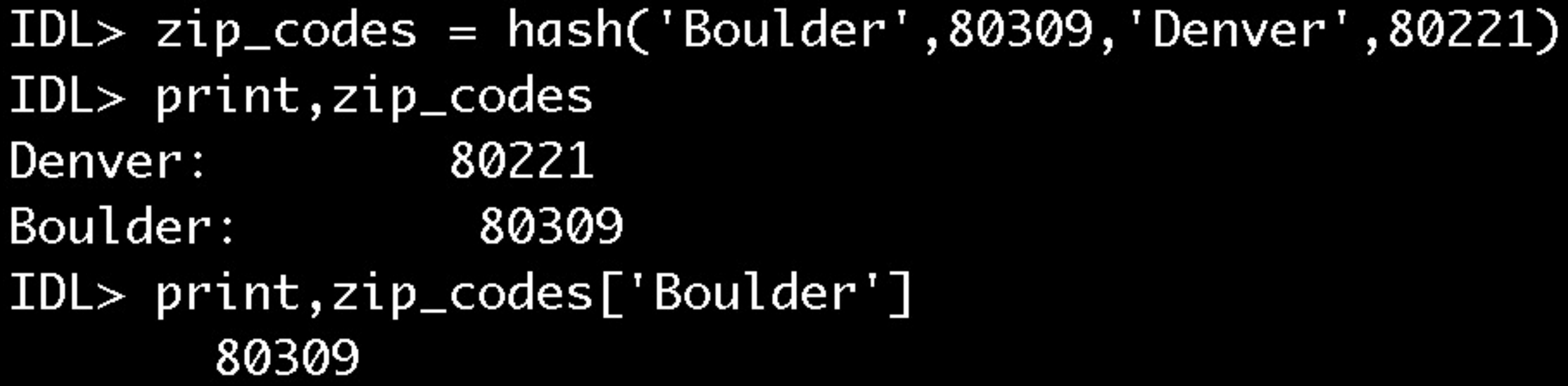
- In python, these are called “dictionaries”
- Declare them like you would a list, but a little differently:



```
zip_codes = hash('Boulder',80309,'Denver',80221)
```

- “hashes” contain a set of “keys” and associated “values”
- They are “look-up tables”

Hashes

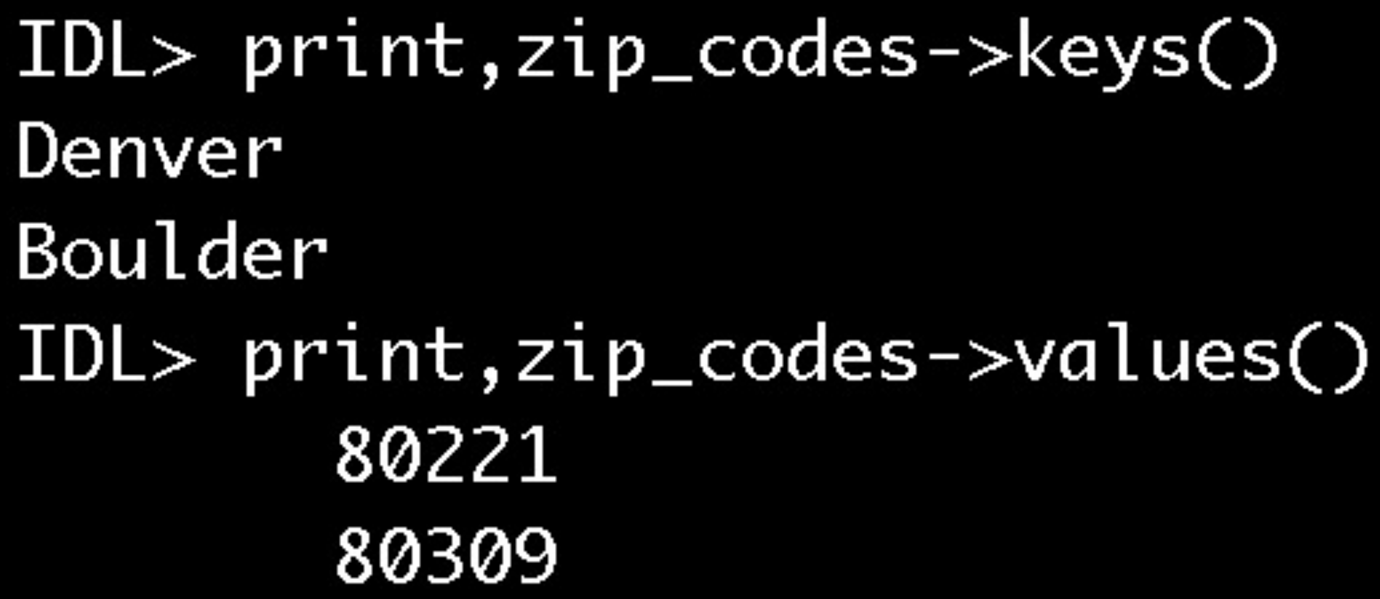


```
IDL> zip_codes = hash('Boulder',80309,'Denver',80221)
IDL> print,zip_codes
Denver:          80221
Boulder:          80309
IDL> print,zip_codes['Boulder']
       80309
```

- Indexed by a key (can be a number, string, or any type)
- Does not have an order

Hashes

- You can access just the keys or just the values:

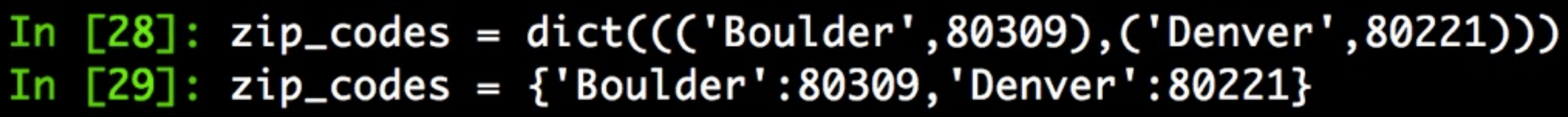


```
IDL> print,zip_codes->keys()
Denver
Boulder
IDL> print,zip_codes->values()
       80221
       80309
```

- The “->” syntax has special meaning we’ll get into much later

Dictionaries (like hashes)

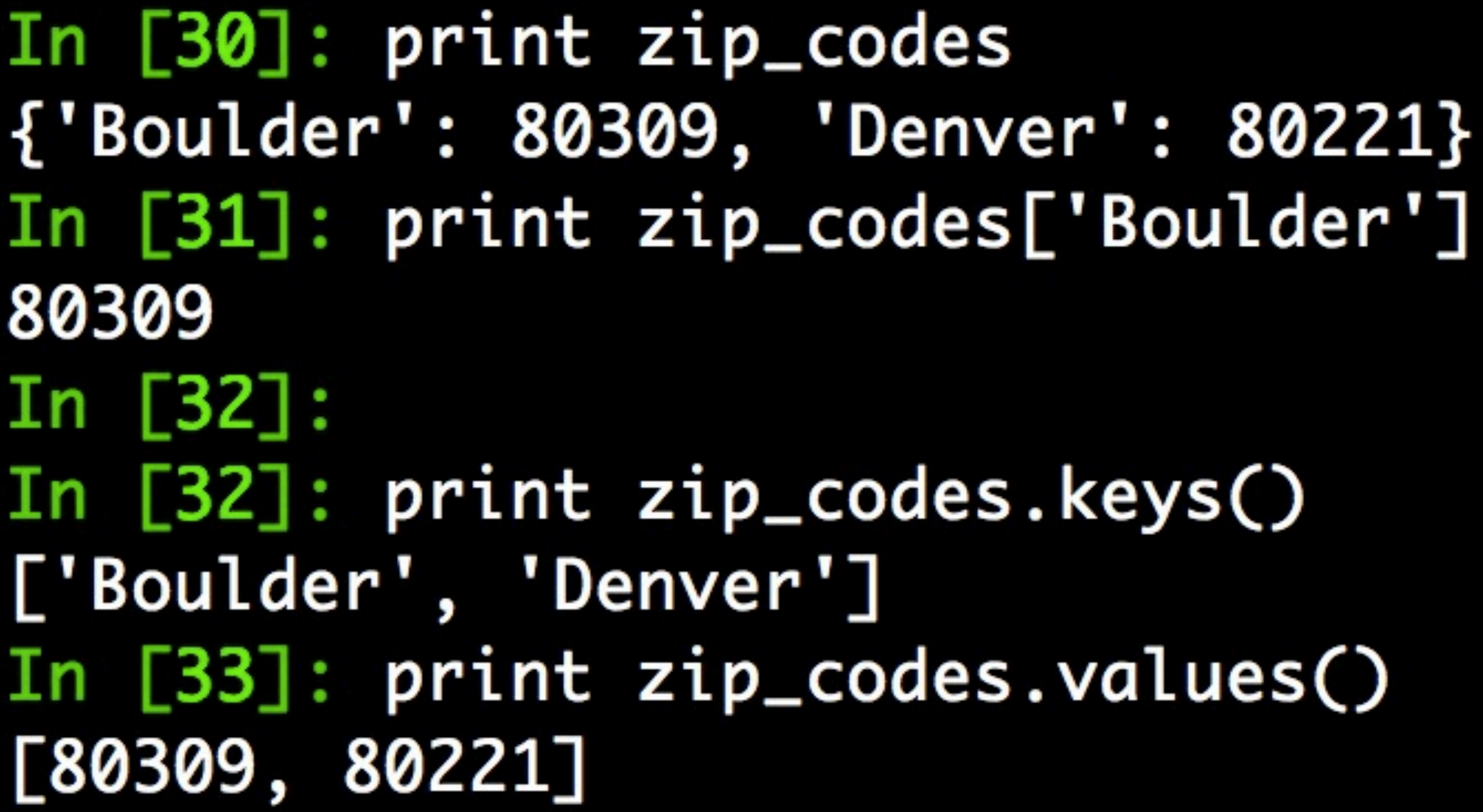
- Can use the IDL-like syntax, but it’s ugly
- Instead, curly braces are nicer:



```
In [28]: zip_codes = dict((('Boulder',80309),('Denver',80221)))
In [29]: zip_codes = {'Boulder':80309,'Denver':80221}
```

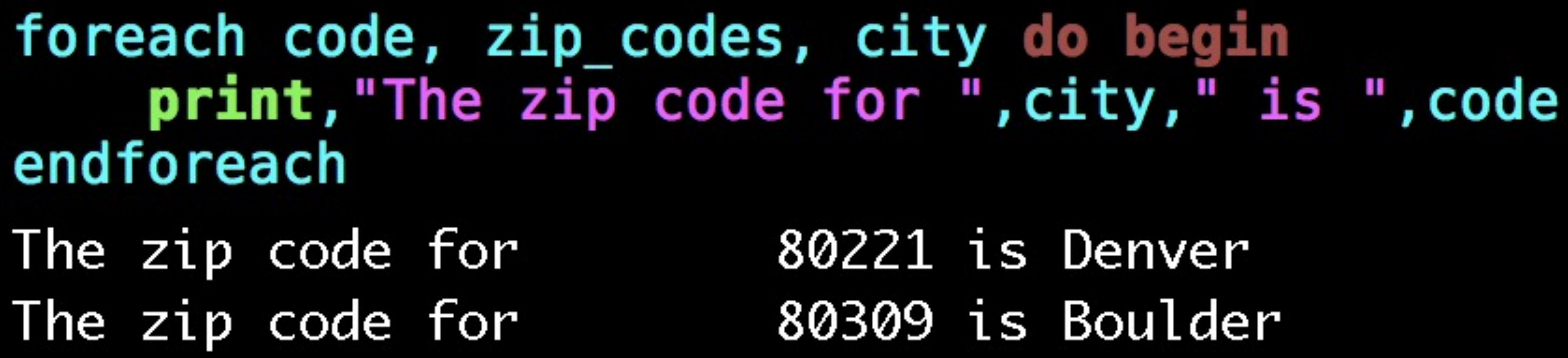
Dictionaries

- Accessing values is largely the same as in IDL, but with . instead of ->



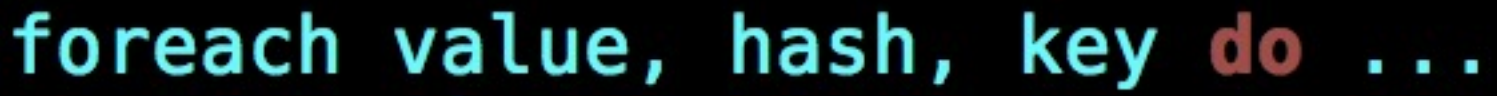
```
In [30]: print zip_codes
{'Boulder': 80309, 'Denver': 80221}
In [31]: print zip_codes['Boulder']
80309
In [32]:
In [32]: print zip_codes.keys()
['Boulder', 'Denver']
In [33]: print zip_codes.values()
[80309, 80221]
```

Foreach & Hashes



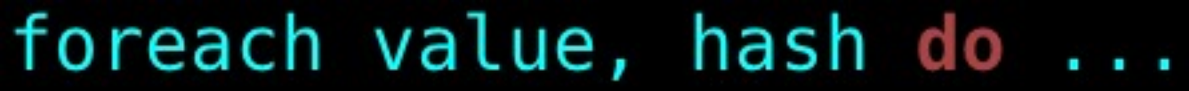
```
foreach code, zip_codes, city do begin
    print,"The zip code for ",city," is ",code
endforeach
The zip code for         80221 is Denver
The zip code for         80309 is Boulder
```

- So the syntax is:



```
foreach value, hash, key do ...
```

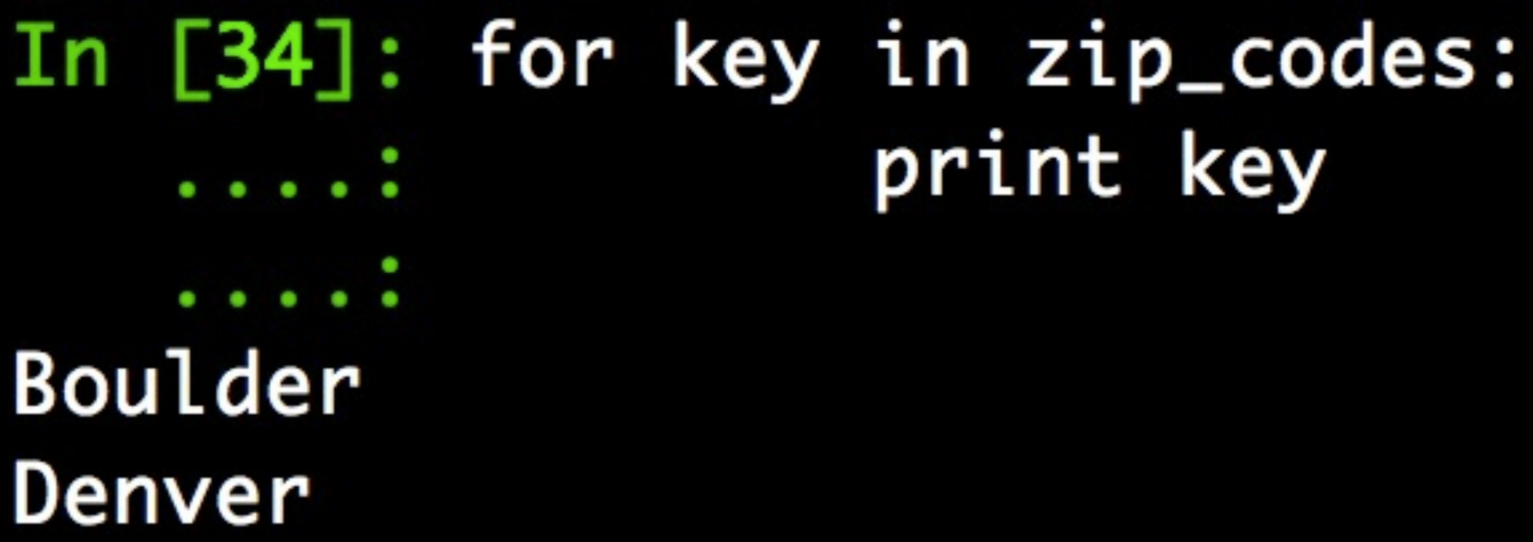
- You can also do:



```
foreach value, hash do ...
```

Python: for & dict

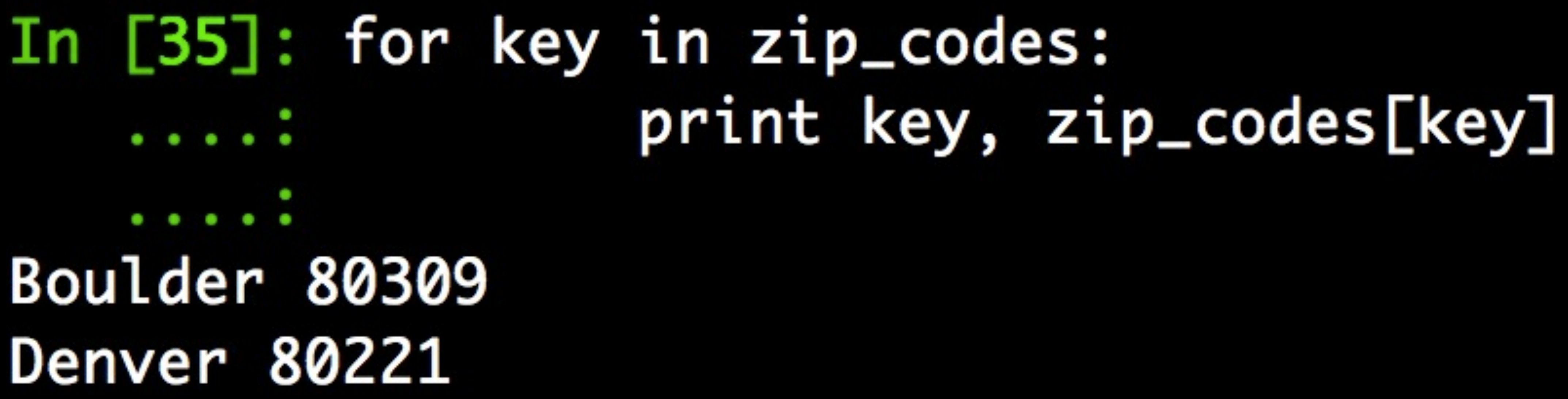
- Python’s for loops naturally work on dicts, BUT they only get you the keys:



```
In [34]: for key in zip_codes:
   ....:             print key
   ....:
Boulder
Denver
```

values?

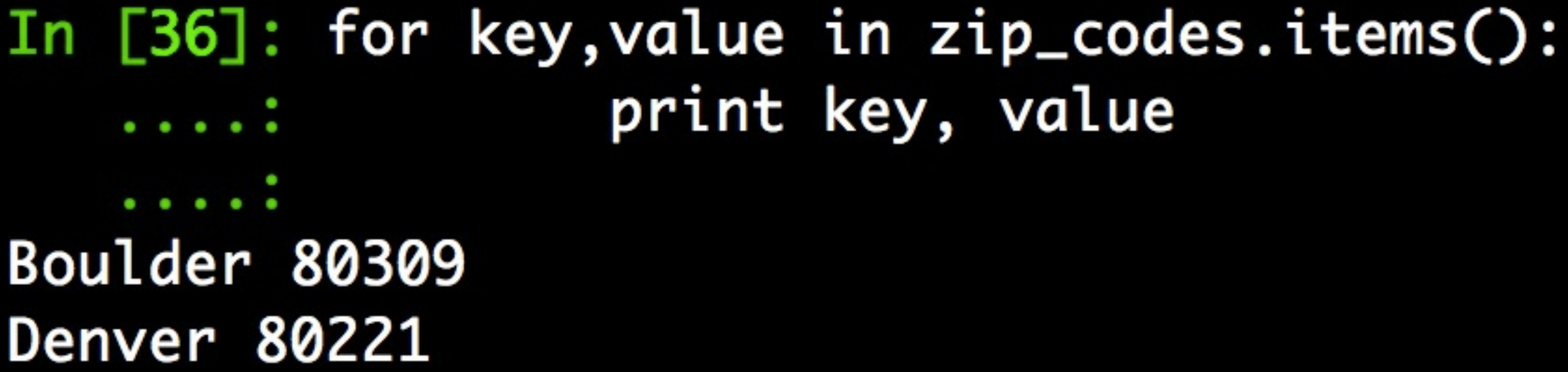
- There are two ways to get the values:
  - Easy:



```
In [35]: for key in zip_codes:
   ....:         print key, zip_codes[key]
   ....:
Boulder 80309
Denver 80221
```

values pythonically

- `.items()` returns key/value pairwise:



```
In [36]: for key,value in zip_codes.items():
   ....:             print key, value
   ....:
Boulder 80309
Denver 80221
```

Onto... Creating Functions

- Chapter 13: Writing Sub-Programs
- Sub-Programs include procedures and functions
- They are effectively re-useable shorthand for different code blocks

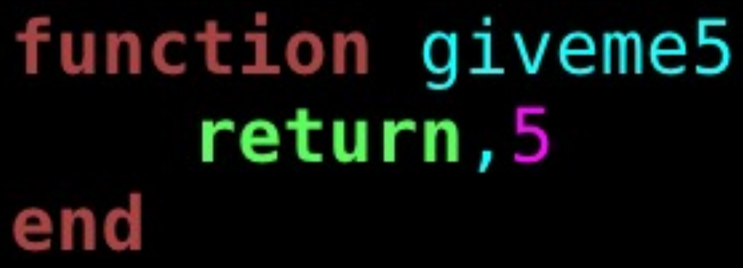
Sub-Programs

- Almost always want to put them in their own file with the same name as the program
- This allows IDL to automatically find and compile them

IDL Functions

- Must `return` something
- Are declared like:
- `function function_name,args...`
- end with an `end` statement

Example Function

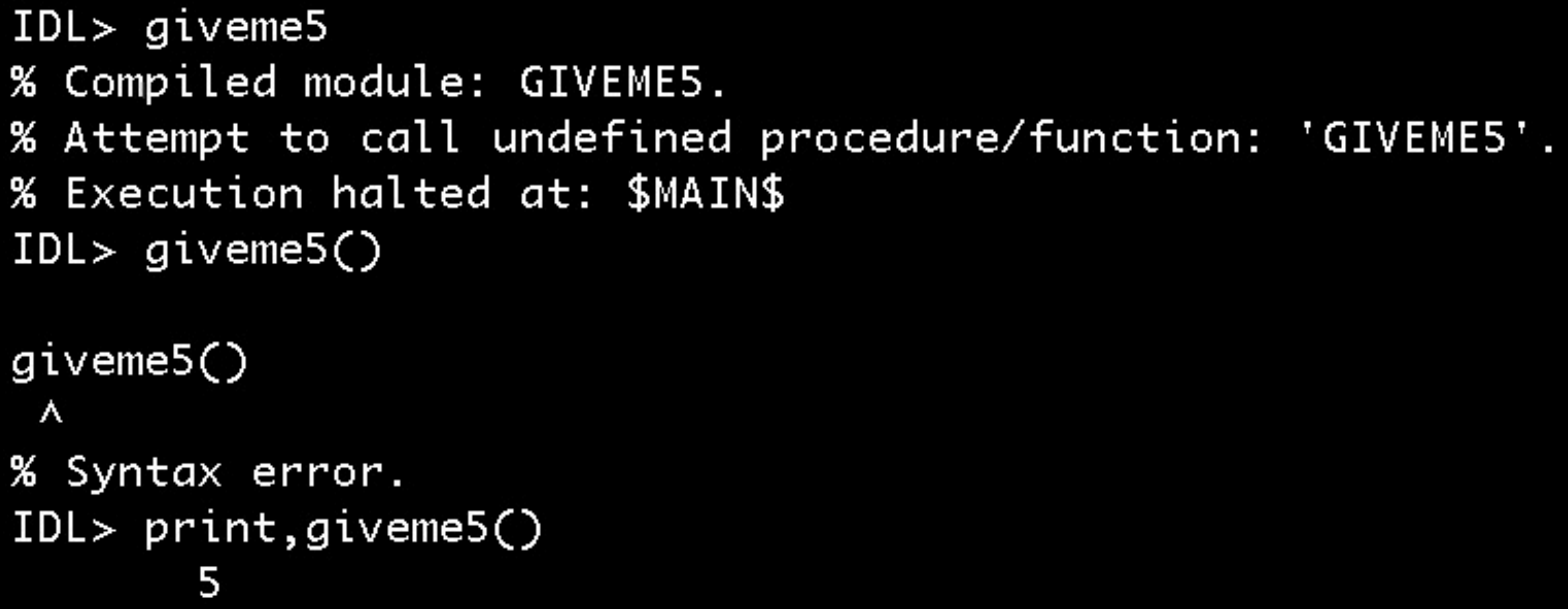


```
function giveme5
    return,5
end
```

- This function takes no input
- But it returns 5
- Also, note the explicit, clear, and obvious naming scheme

Using Functions

- Since they return something, you HAVE to either pass their return to a procedure or store it in a variable

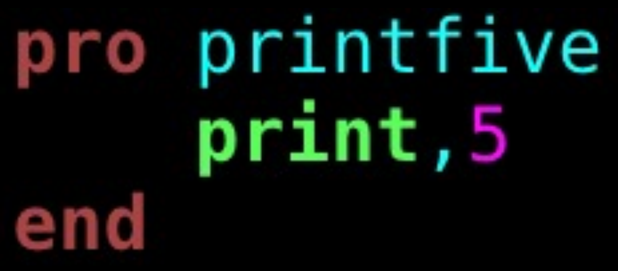


```
IDL> giveme5
% Compiled module: GIVEME5.
% Attempt to call undefined procedure/function: 'GIVEME5'.
% Execution halted at: $MAIN$
IDL> giveme5()

giveme5()
 ^
% Syntax error.
IDL> print,giveme5()
       5
```

Procedures

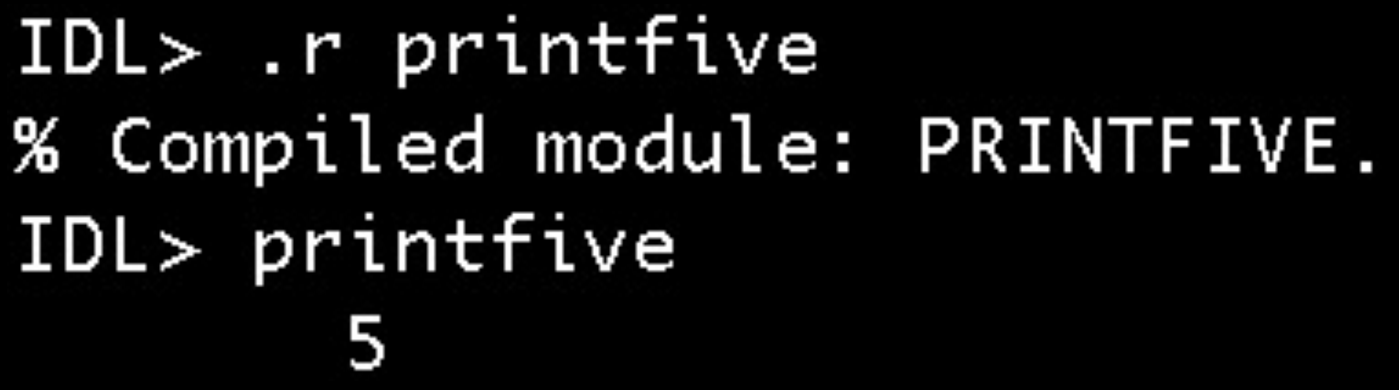
- Same general declaration:
-
  ```
  pro procedurename
      do something
  end
  ```
- no return required

Example Procedure



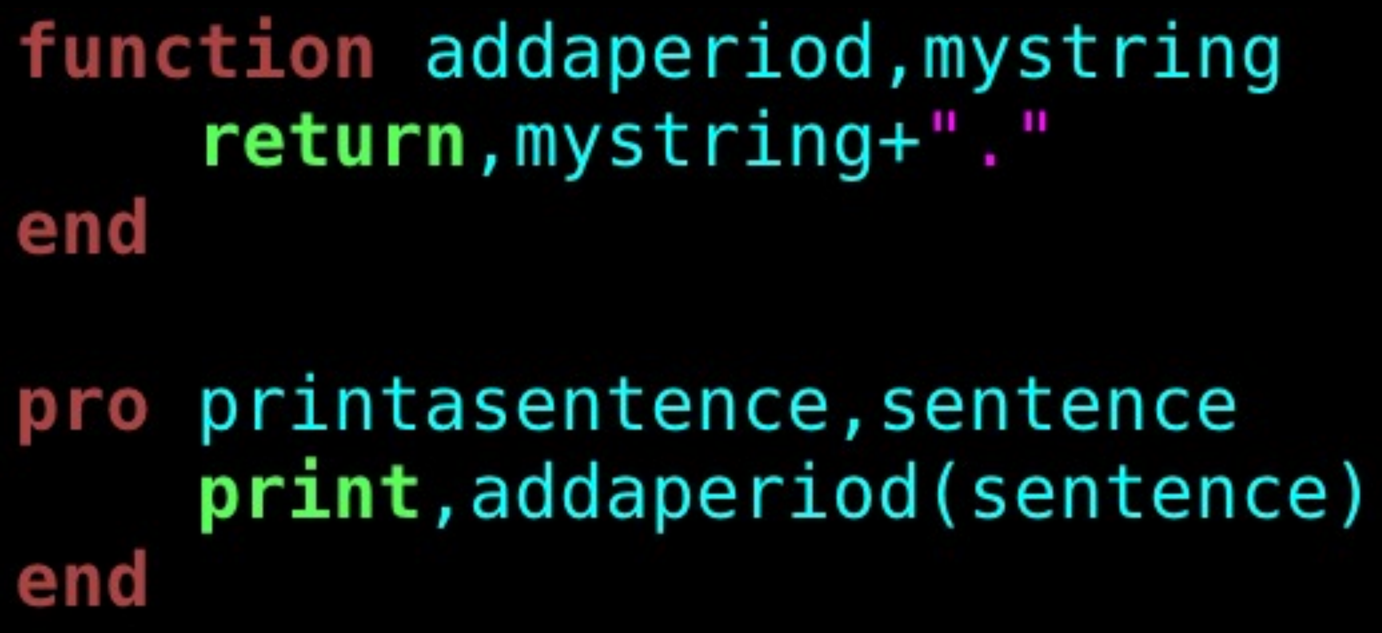
```
pro printfive
    print,5
end
```

- Again, no input, but it does something (something rather silly...)

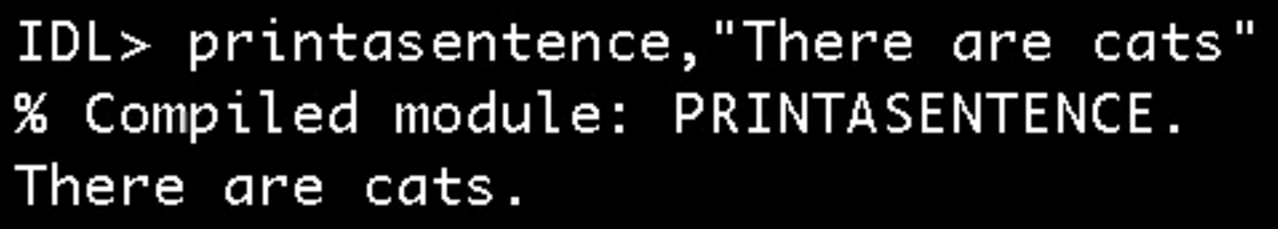


```
IDL> .r printfive
% Compiled module: PRINTFIVE.
IDL> printfive
       5
```

Example with Inputs

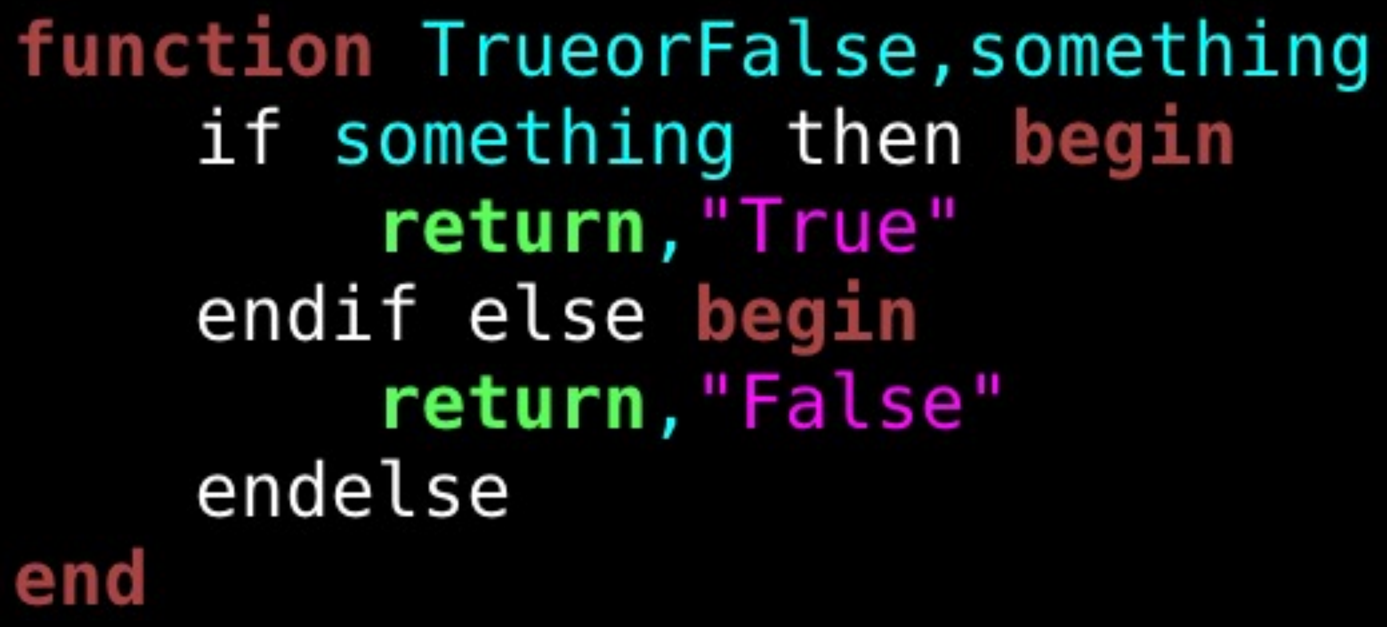


```
function addaperiod,mystring
    return,mystring+"."
end

pro printasentence,sentence
    print,addaperiod(sentence)
end
```



```
IDL> printasentence,"There are cats"
% Compiled module: PRINTASENTENCE.
There are cats.
```

Another simple example

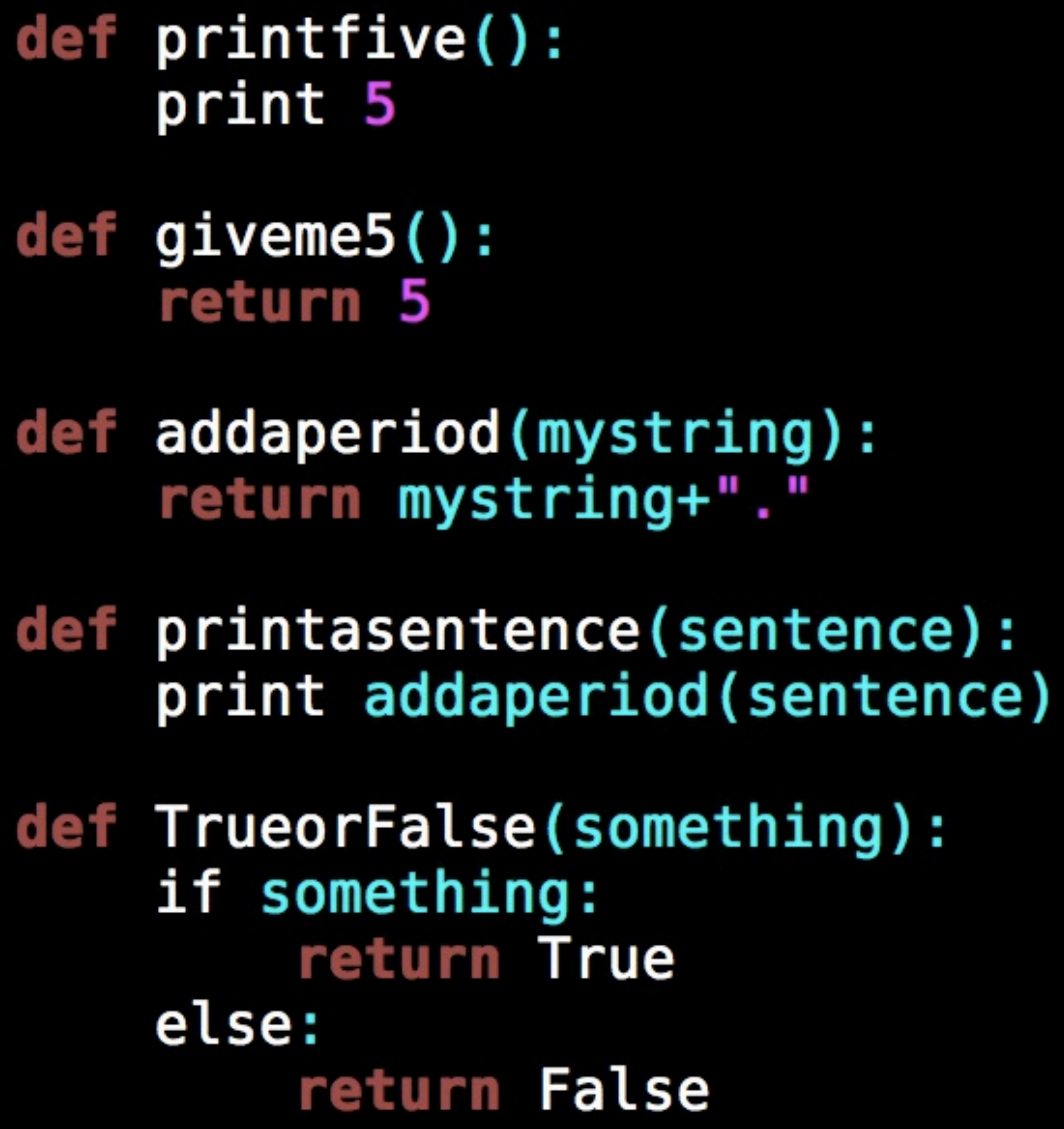


```
function TrueorFalse,something
    if something then begin
        return,"True"
    endif else begin
        return,"False"
    endelse
end
```

- Nice human-readable way to ask if a conditional is True

Procedures and functions?

- Python doesn't distinguish between procedures and functions
- Only real difference is that a "procedure" would be something with no **`return`** statement
  - In that case, it returns the special variable **`None`**



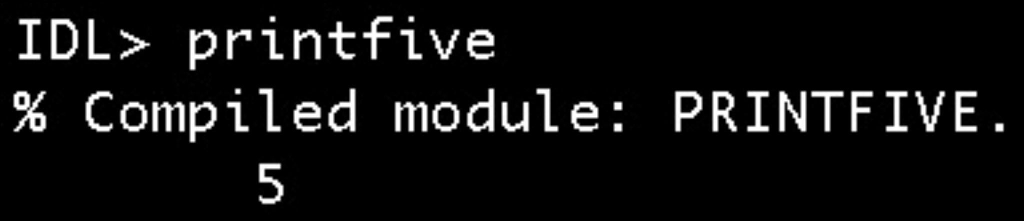
```python
def printfive():
    print 5

def giveme5():
    return 5

def addaperiod(mystring):
    return mystring+"."

def printasentence(sentence):
    print addaperiod(sentence)

def TrueorFalse(something):
    if something:
        return True
    else:
        return False
```

Philosophy

- (IDL ONLY!) I encourage you to have filenames with the same name as the contained program/function, and ONLY one program/function per file
- There will be exceptions to this
- The text disagrees, and it has good reason, but...

IDL and the !PATH

- If you call a procedure or function that has not been compiled, IDL will try to find a file with the exact same name
- i.e., if I ran “printfive” without compiling it first, it would do this:



```
IDL> printfive
% Compiled module: PRINTFIVE.
       5
```

- This is too convenient not to use

The `!PATH`

- IDL has a `!PATH` system variable
- It contains a list of UNIX paths separated by `:`'s
- These are the locations IDL will search for .pro files if a command name has not yet been compiled

Python doesn’t do that

- In python, it is common practice to have many functions per file
- So how do you access them?

Python `import`

- Python has a concept IDL lacks: `import`ing
- In IDL, you `.compile` a code, then have access to all functions defined in that program
- In python, you `import numpy`, then you can access its functions:
  - `numpy.linspace`
  - `numpy.sin`

`np.linspace?`

- You have already used this, perhaps unaware: when you run python with `ipython --pylab` it implicitly does an import for you: `import numpy as np`

Namespaces

- The reason for imports, import as, etc., is that there are a lot of functions, and their names can easily overlap
- We’ll see examples of this, but it’s really nice that you can use `x`, `y`, `z`, etc. in functions and not have to worry about whether you already used that variable

Code Development

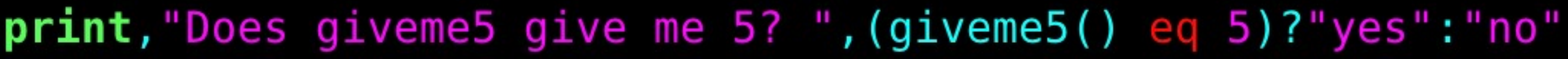
- (i.e., philosophy cont’d)
- There are many different strategies for writing code, AKA “Software Development”
- One of my favorite approaches is called “test-driven development” ...

Test-Driven Development

- In test driven development, you write the tests of your code before the code itself
    - i.e., you assume you know what you want the output to be, so you write a “test” to make sure you get that output
- This approach is great for the small pieces of a big project

Tests...

- An absurdly simple case, illustrating the general point:

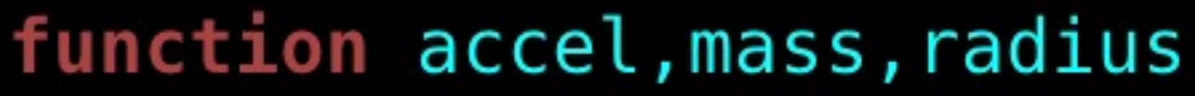


```
print,"Does giveme5 give me 5? ",(giveme5() eq 5)?"yes":"no"
```

- For more complicated math/physics, try doing the math by hand first

Example Physics case: Acceleration of Gravity

- The acceleration of gravity at Earth’s surface is about 9.8 m/s^2
- If we have a function that returns the acceleration as a function of mass and radius:

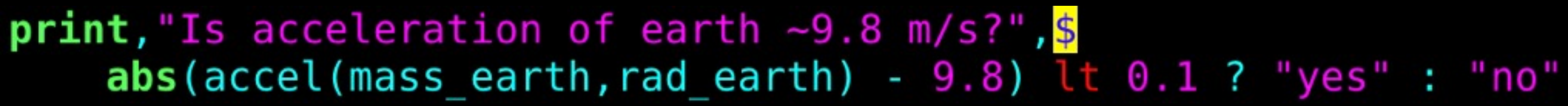


```
function accel,mass,radius
```

$$a = \frac{GM_\oplus}{r_\oplus^2}$$

- we can test it for Earth, even though it should apply anywhere in general

The Test



```
print,"Is acceleration of earth ~9.8 m/s?",$
    abs(accel(mass_earth,rad_earth) - 9.8) lt 0.1 ? "yes" : "no"
```

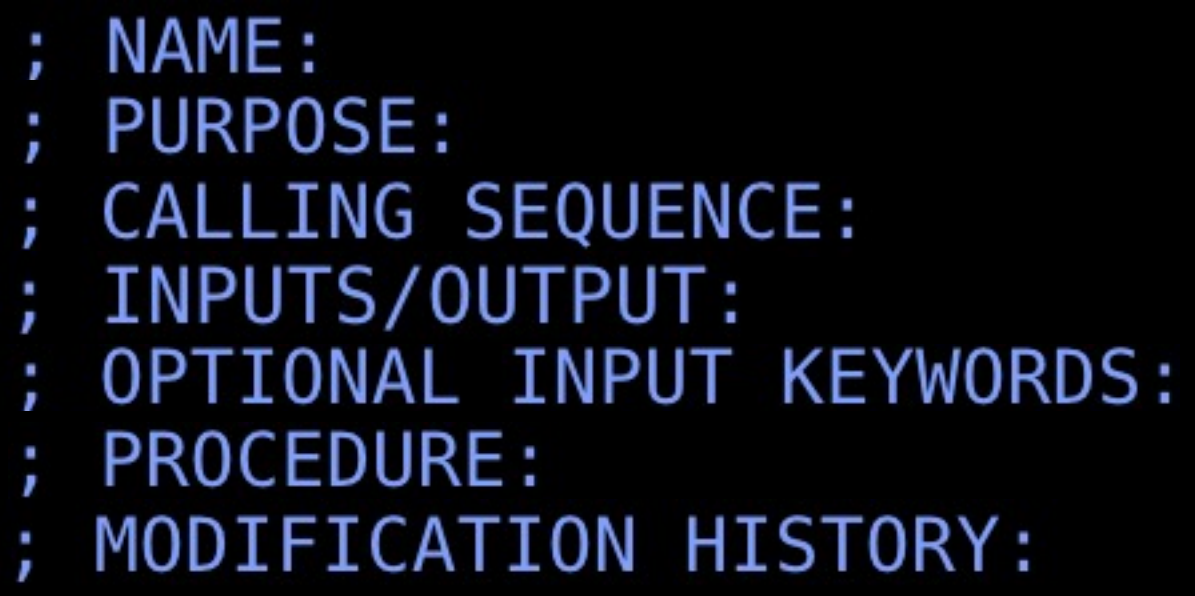
- Does a floating-point conditional test (make sure the acceleration is pretty close to 9.8)

Documenting Functions

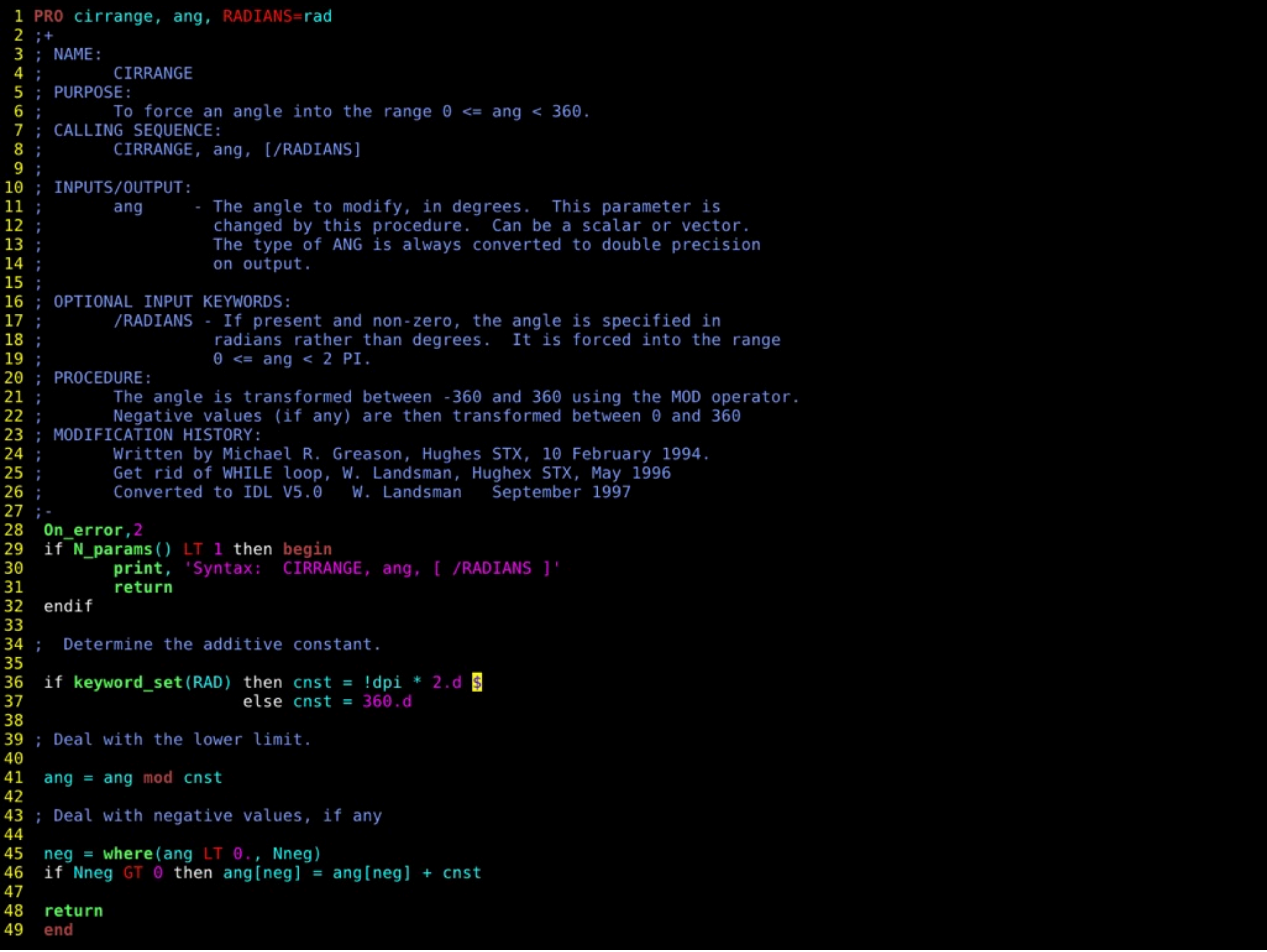
- While IDL doesn’t have clear standards for how to document code, NASA does
- For most functions, the documentation will end up being (much) longer than the code

Documenting Functions

- The “header” should include the following:



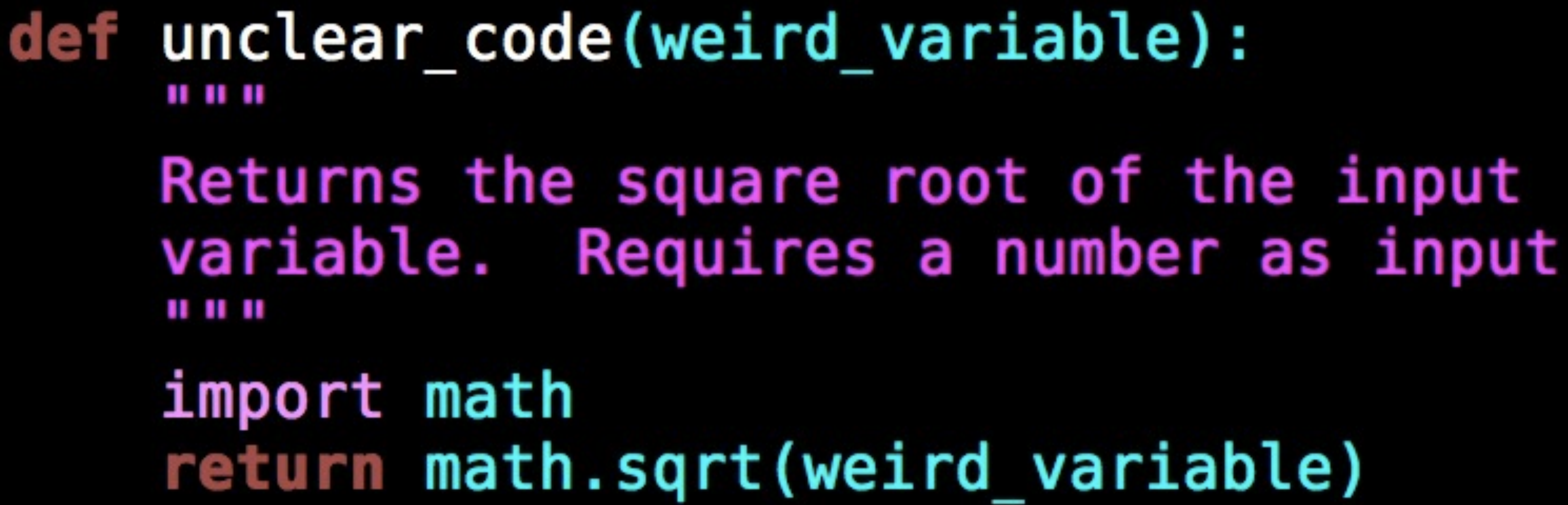
```
; NAME:
; PURPOSE:
; CALLING SEQUENCE:
; INPUTS/OUTPUT:
; OPTIONAL INPUT KEYWORDS:
; PROCEDURE:
; MODIFICATION HISTORY:
```



```
PRO cirrange, ang, RADIANS=rad
;+
; NAME:
;       CIRRANGE
; PURPOSE:
;       To force an angle into the range 0 <= ang < 360.
; CALLING SEQUENCE:
;       CIRRANGE, ang, [/RADIANS]
;
; INPUTS/OUTPUT:
;       ang     - The angle to modify, in degrees.  This parameter is
;                 changed by this procedure.  Can be a scalar or vector.
;                 The type of ANG is always converted to double precision
;                 on output.
;
; OPTIONAL INPUT KEYWORDS:
;       /RADIANS - If present and non-zero, the angle is specified in
;                 radians rather than degrees.  It is forced into the range
;                 0 <= ang < 2 PI.
; PROCEDURE:
;       The angle is transformed between -360 and 360 using the MOD operator.
;       Negative values (if any) are then transformed between 0 and 360
; MODIFICATION HISTORY:
;       Written by Michael R. Greason, Hughes STX, 10 February 1994.
;       Get rid of WHILE loop, W. Landsman, Hughex STX, May 1996
;       Converted to IDL V5.0   W. Landsman   September 1997
;-
 On_error,2
 if N_params() LT 1 then begin
        print, 'Syntax:  CIRRANGE, ang, [ /RADIANS ]'
        return
 endif

;  Determine the additive constant.

 if keyword_set(RAD) then cnst = !dpi * 2.d $
                     else cnst = 360.d

; Deal with the lower limit.

 ang = ang mod cnst

; Deal with negative values, if any

 neg = where(ang LT 0., Nneg)
 if Nneg GT 0 then ang[neg] = ang[neg] + cnst

 return
 end
```

Python docs

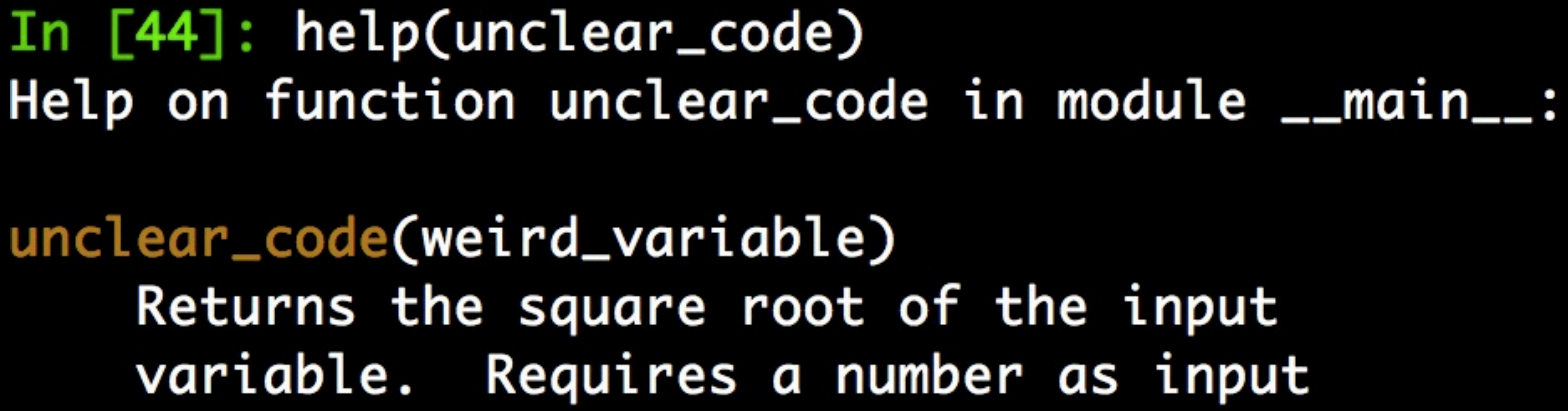
- Python has clear - and awesome - documentation standards:



```
def unclear_code(weird_variable):
    """
    Returns the square root of the input
    variable.  Requires a number as input
    """
    import math
    return math.sqrt(weird_variable)
```

Why are the docs awesome?

- “docstrings” are part of the function:



```
In [44]: help(unclear_code)
Help on function unclear_code in module __main__:

unclear_code(weird_variable)
    Returns the square root of the input
    variable.  Requires a number as input
```