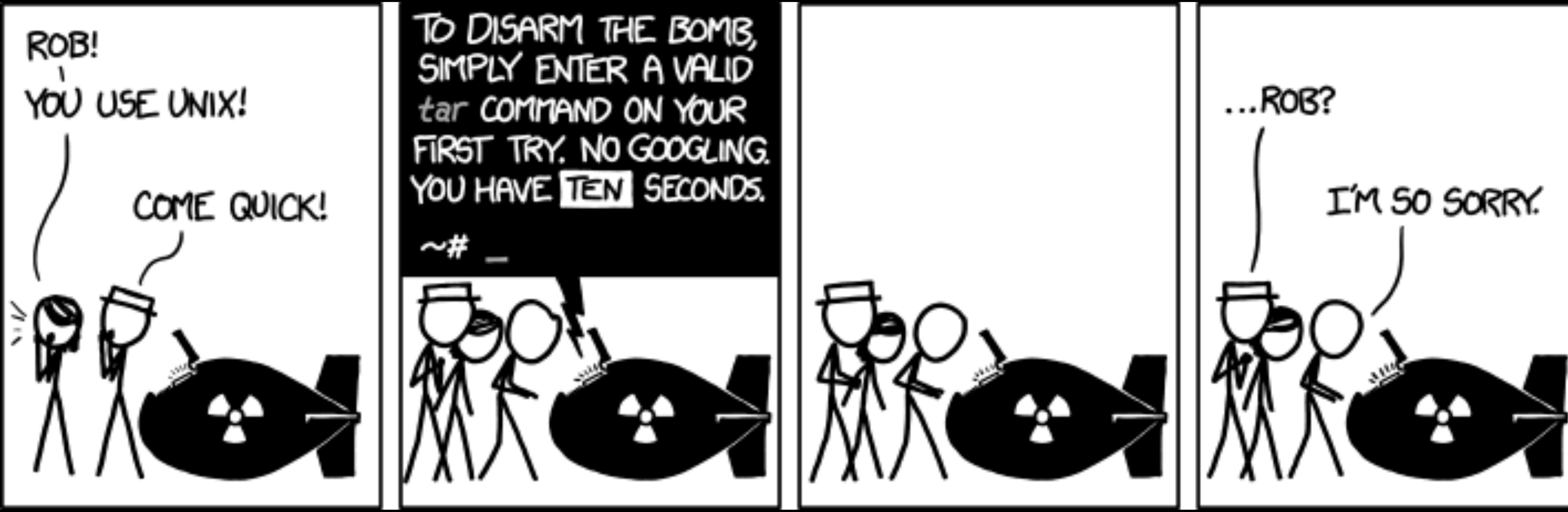
Reading

- Chapter 7

Tarballs

- Remember those at all? I mentioned them once but you never used them
- They’re the standard way to package source code (so you can compile it into a program yourself)

Tarballs

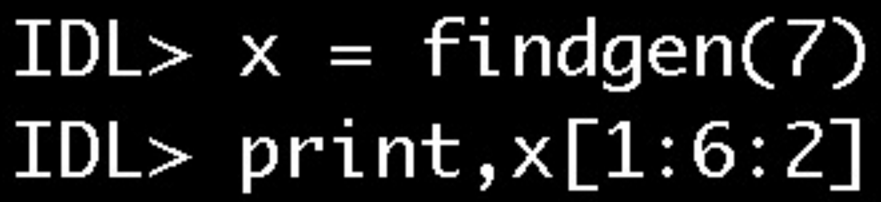


ROB!
YOU USE UNIX!
COME QUICK!
TO DISARM THE BOMB, SIMPLY ENTER A VALID tar COMMAND ON YOUR FIRST TRY. NO GOOGLING. YOU HAVE TEN SECONDS.
~# _
...ROB?
I'M SO SORRY.

- Flavor txt: I don't know what's worse--the fact that after 15 years of using tar I still can't keep the flags straight, or that after 15 years of technological advancement I'm still mucking with tar flags that were 15 years old when I started.

Array Indexing Review

Evaluate:



```
IDL> x = findgen(7)
IDL> print,x[1:6:2]
```

A)
```
1.00000      3.00000      5.00000
```

B)
```
0.00000      2.00000      4.00000      6.00000
```

C)
```
0.00000      1.00000      2.00000
```

D)
```
0.00000      1.00000      2.00000      3.00000
4.00000      5.00000      6.00000
```

E) None of the above/I don't know

Python vs IDL indexing

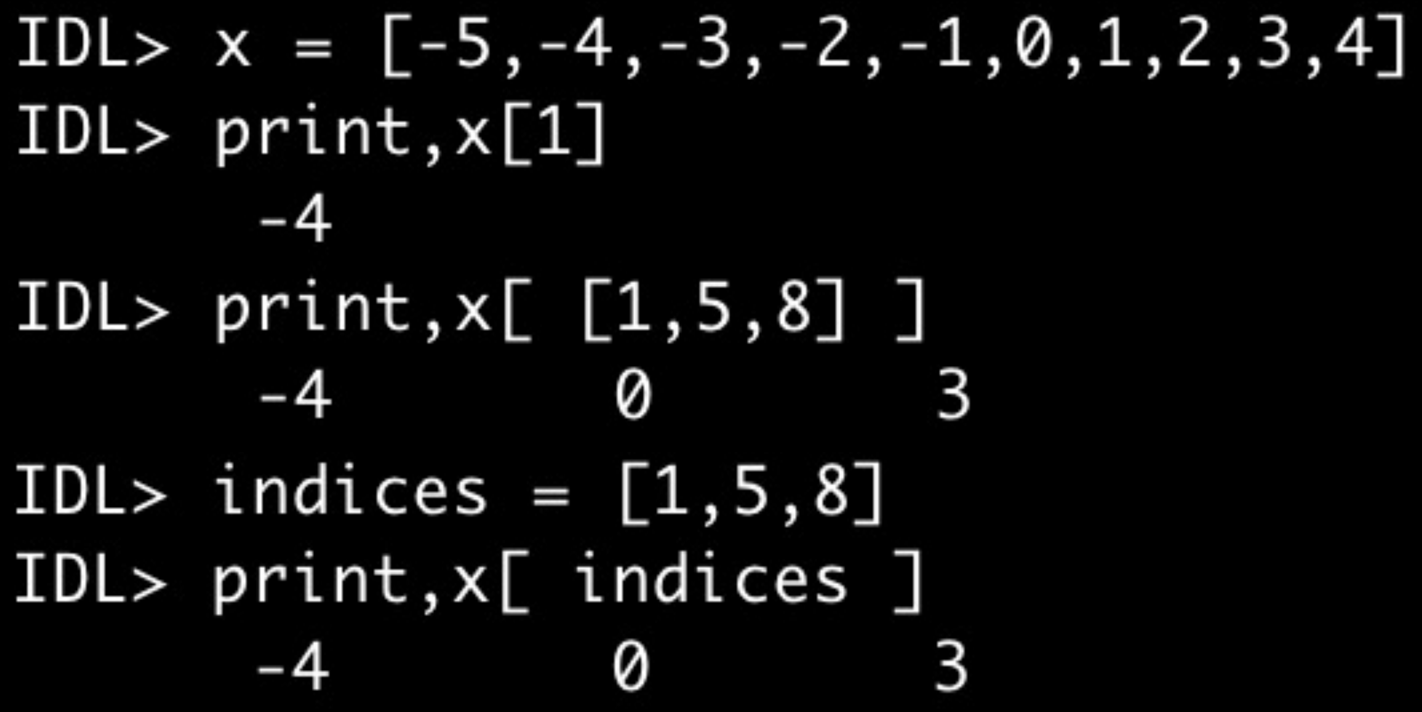
- IDL is “inclusive” while Python is not
  - In math terms, IDL ranges look like [0,2] and python ranges look like [0,2)
- x[0:5] - in IDL, 6 elements, in python, 5
  - 5-0 = stop-start = 5 in python
- `x = x[:i] + x[i:]` (python - lists)
- `x = [x[*:i-1],x[i:*]]` (IDL)

Array Access, Sorting, and Searching

- Sometimes you only want parts of arrays
- Sometimes those parts can’t be specified by simple “slices” (e.g., a[1:5])
- Sometimes you want to change the order of arrays

Indexing an array with an array

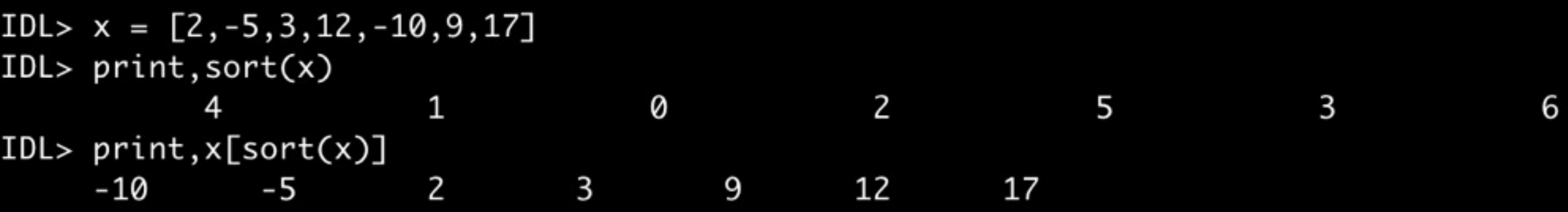
- You can index with an array! (same in python)



```
IDL> x = [-5,-4,-3,-2,-1,0,1,2,3,4]
IDL> print,x[1]
      -4
IDL> print,x[ [1,5,8] ]
      -4       0       3
IDL> indices = [1,5,8]
IDL> print,x[ indices ]
      -4       0       3
```

Sorting

- You can sort arrays in ascending order

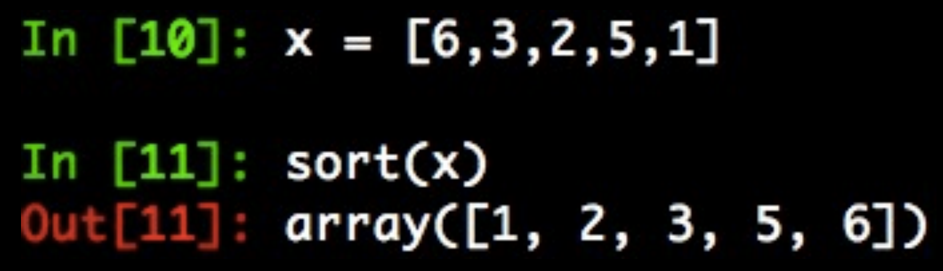


```
IDL> x = [2,-5,3,12,-10,9,17]
IDL> print,sort(x)
           4           1           0           2           5           3           6
IDL> print,x[sort(x)]
     -10      -5       2       3       9      12      17
```

- `sort(x)` returns the *indices* of `x` in the correctly sorted order
- to get a sorted version of `x`, take `x[sort(x)]`

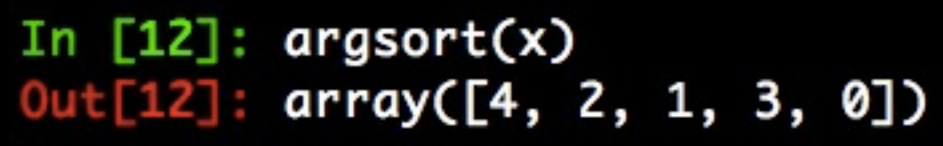
python sorting

- sort(x) returns the sorted array



```
In [10]: x = [6,3,2,5,1]

In [11]: sort(x)
Out[11]: array([1, 2, 3, 5, 6])
```

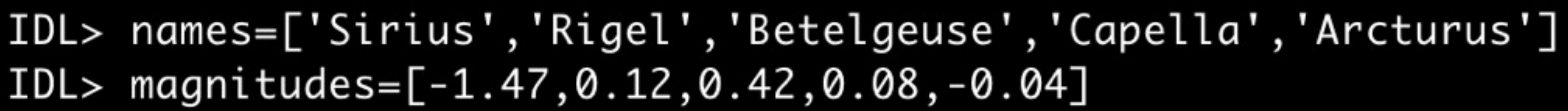
- argsort(x) returns the indices



```
In [12]: argsort(x)
Out[12]: array([4, 2, 1, 3, 0])
```

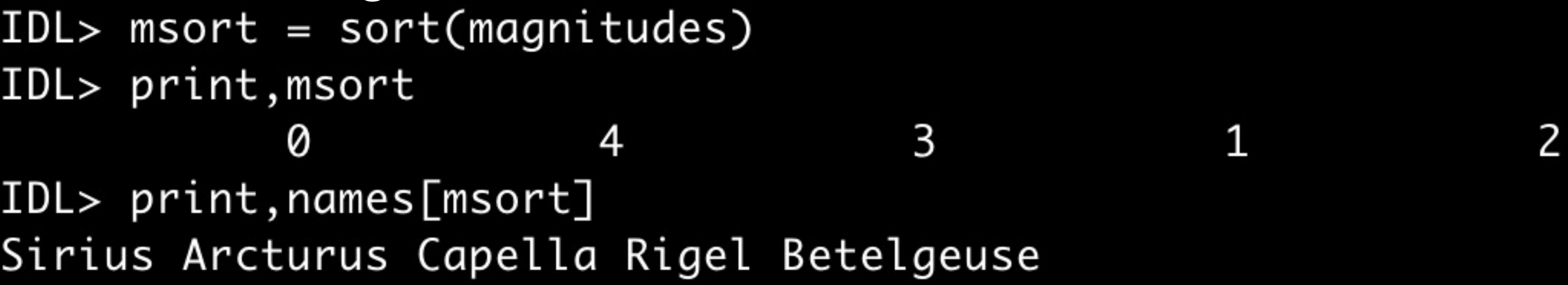
Using Indices

- Say you have an array of names and one of magnitudes:



```
IDL> names=['Sirius','Rigel','Betelgeuse','Capella','Arcturus']
IDL> magnitudes=[-1.47,0.12,0.42,0.08,-0.04]
```

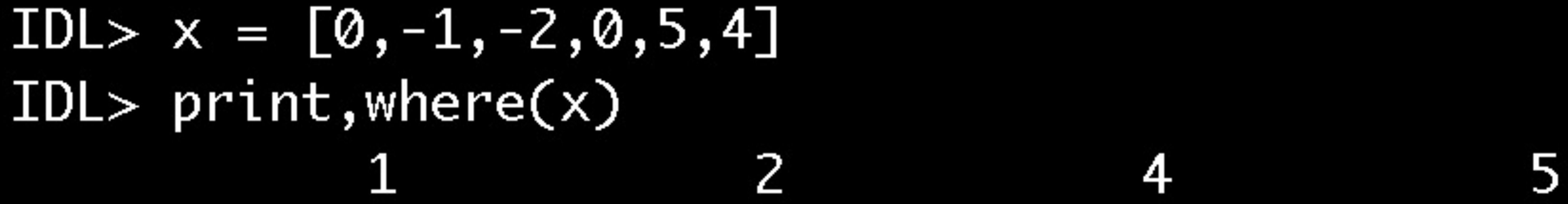
- You can get the names sorted by their magnitudes



```
IDL> msort = sort(magnitudes)
IDL> print,msort
           0           4           3           1           2
IDL> print,names[msort]
Sirius Arcturus Capella Rigel Betelgeuse
```

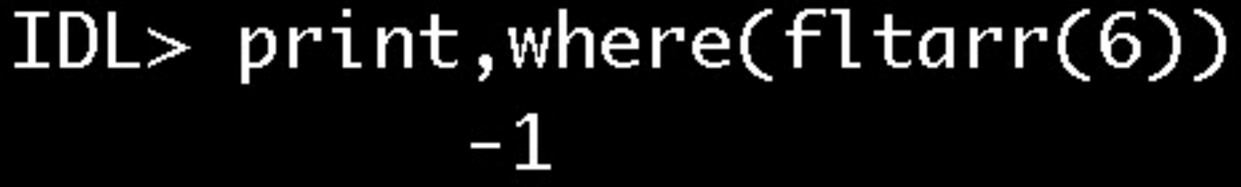
Searching Arrays

- You can search for data within arrays using the `where` function
  - `where` takes in an array, and returns the indices wherever the array is non-zero



```
IDL> x = [0,-1,-2,0,5,4]
IDL> print,where(x)
           1           2           4           5
```

  - it returns -1 if there are none



```
IDL> print,where(fltarr(6))
          -1
```

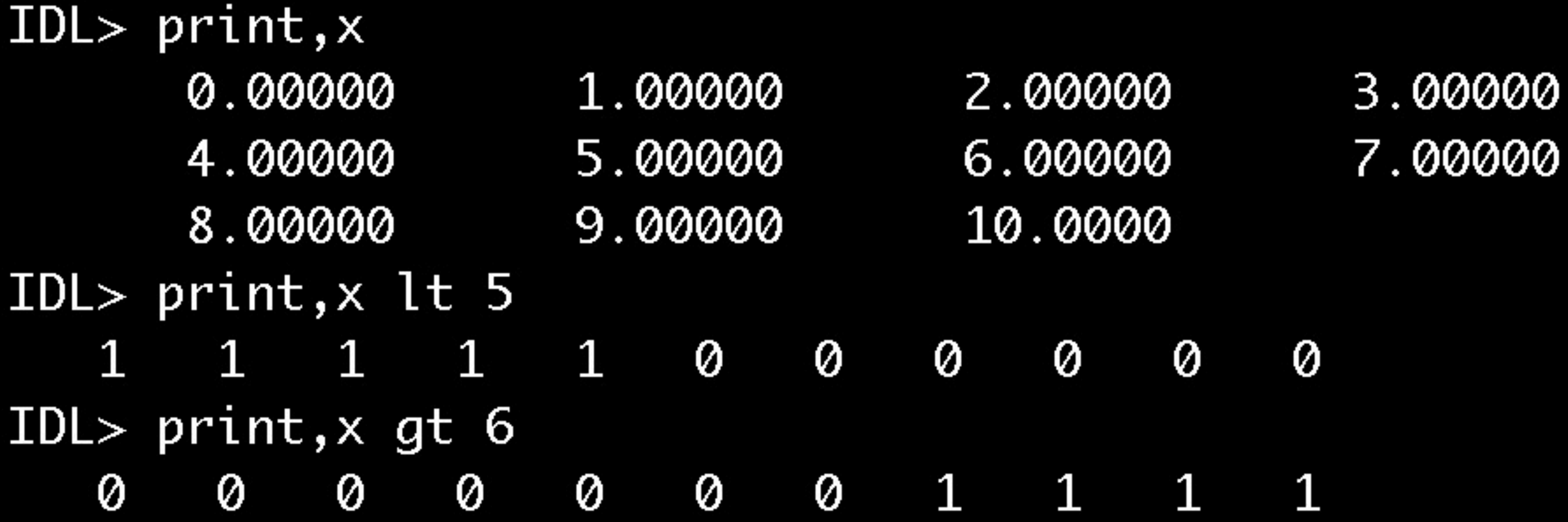
Boolean Operations on Arrays

- There are many boolean operations that work on arrays
  - greater than (equal) = `gt (ge)`
  - less than (equal) = `lt (le)`
  - equals = `eq`
  - not equal = `ne`

Boolean Operations on Arrays

- There are many boolean operations that work on arrays: python version
  - greater than (equal): > (>=)
  - less than (equal): < (<=)
  - equals: ==
  - not equal: !=

Boolean Operations on Arrays



```
IDL> print,x
      0.00000      1.00000      2.00000      3.00000
      4.00000      5.00000      6.00000      7.00000
      8.00000      9.00000      10.0000
IDL> print,x lt 5
   1   1   1   1   1   0   0   0   0   0   0
IDL> print,x gt 6
   0   0   0   0   0   0   0   1   1   1   1
```

Evaluate: `print,[0,1,2,3] le [2,2,2,2]`

A)
```
1     1     0     0
```

B)
```
0     0     0     1
```

C)
```
1     1     1     0
```

D)
```
0     0     1     1
```

E) None of the above

Evaluate:

```
print,[0,1,2,3] le [2,2,2,2]
```

A) 1   1   0   0   lt

B) 0   0   0   1   gt

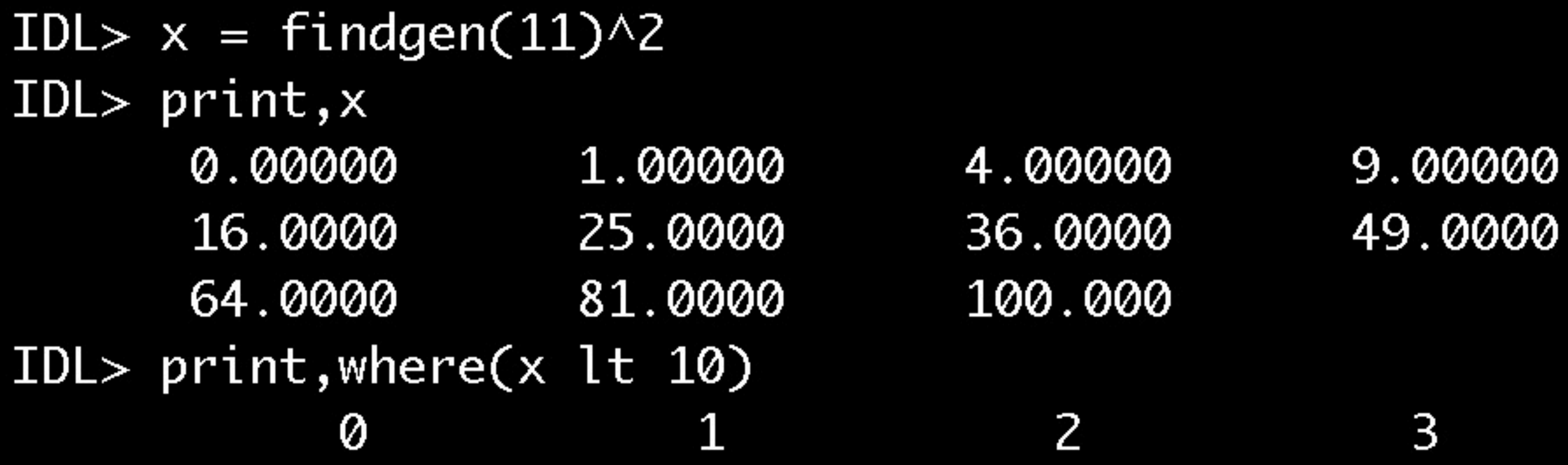
C) 1   1   1   0   le

D) 0   0   1   1   ge

E) None of the above

Boolean Operations on Arrays

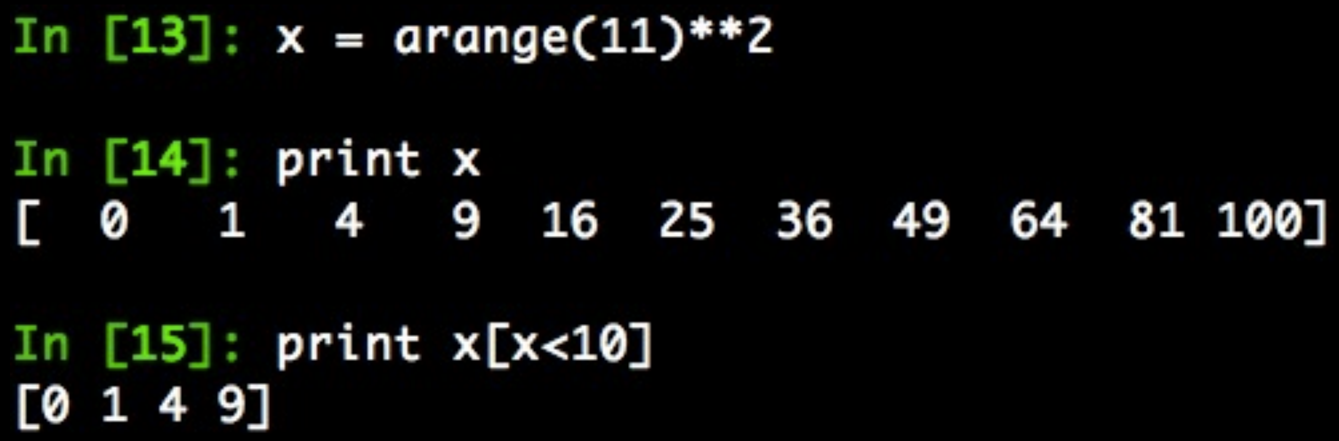
- You can use these with where



```
IDL> x = findgen(11)^2
IDL> print,x
      0.00000      1.00000      4.00000      9.00000
      16.0000      25.0000      36.0000      49.0000
      64.0000      81.0000      100.000
IDL> print,where(x lt 10)
           0           1           2           3
```

Boolean Operations on Arrays

- python version:



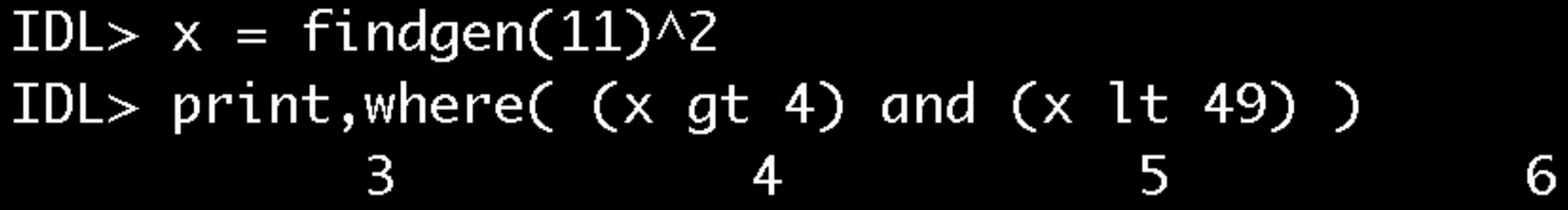
```
In [13]: x = arange(11)**2

In [14]: print x
[  0   1   4   9  16  25  36  49  64  81 100]

In [15]: print x[x<10]
[0 1 4 9]
```

Boolean Operators

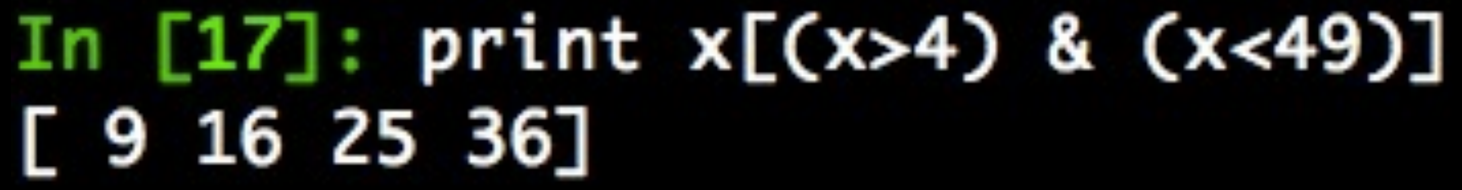
- You can do two operations by combining with `or`, `and`, and `xor` (python doesn't have xor)
  - `1 or 0 = 1   1 or 1 = 1`
  - `1 and 0 = 0  1 and 1 = 1`
  - `1 xor 0 = 1  1 xor 1 = 0`

Boolean Operators

- Useful if you want numbers between two end points



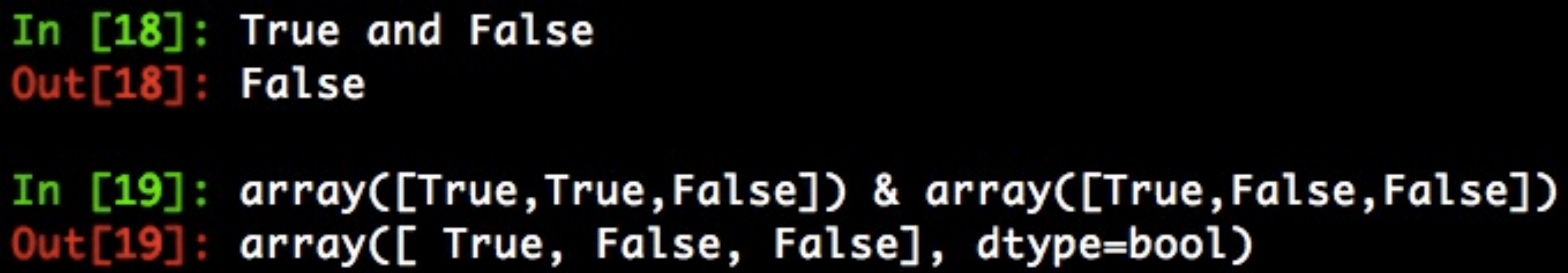
```
IDL> x = findgen(11)^2
IDL> print,where( (x gt 4) and (x lt 49) )
           3           4           5           6
```



```
In [17]: print x[(x>4) & (x<49)]
[ 9 16 25 36]
```

Python’s special bools

- “and” is for single-valued things
- “&” is for arrays



```
In [18]: True and False
Out[18]: False

In [19]: array([True,True,False]) & array([True,False,False])
Out[19]: array([ True, False, False], dtype=bool)
```

where

```
IDL> x=findgen(10)
```

Evaluate:

```
IDL> print,where((x lt 4) and (x gt 1))
```

A)

```
           2           3           4
```

B)

```
           1           2           3           4
```

C)

```
           1           2           3
```

D)

```
           2           3
```

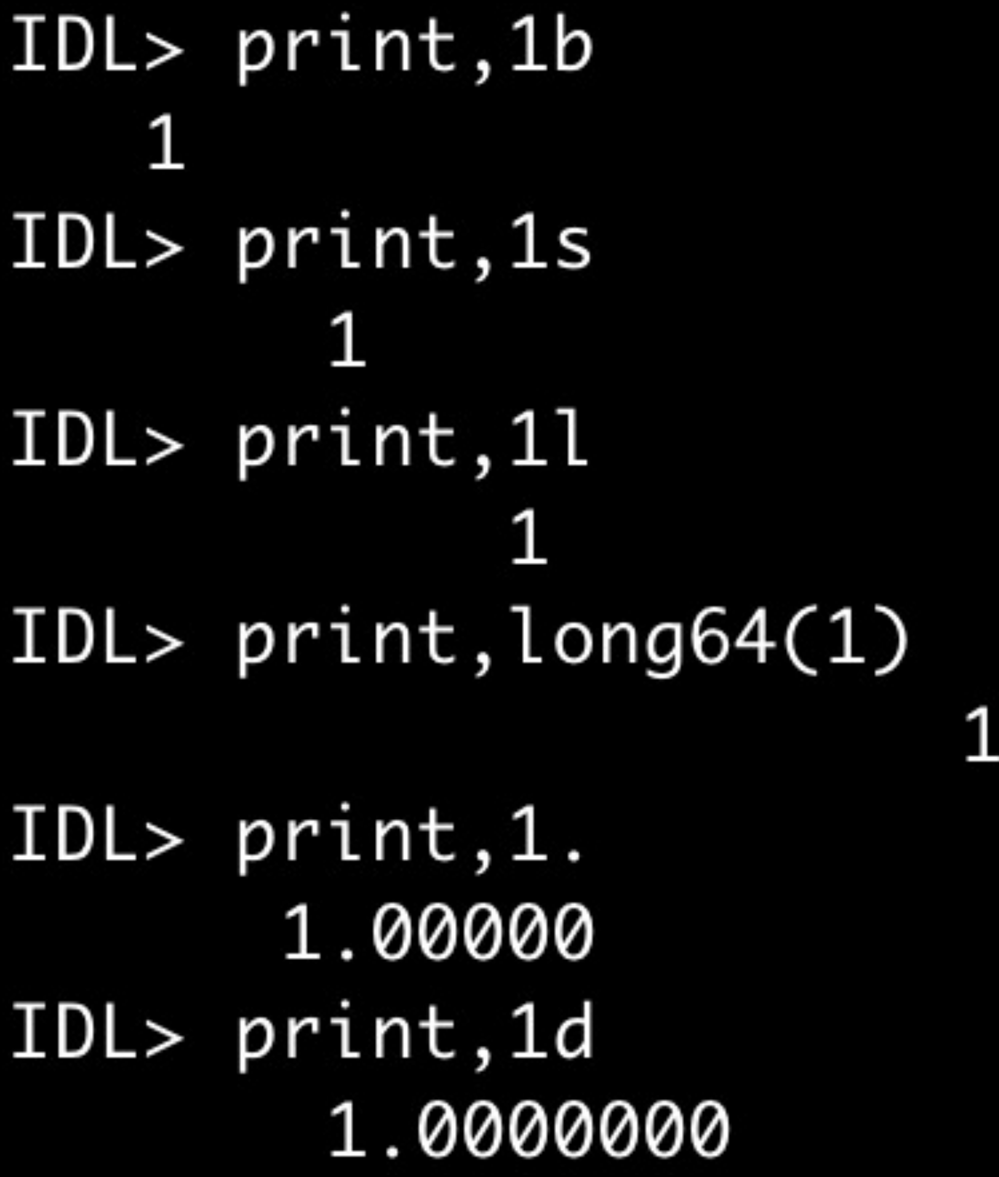
E) None of the above / I don’t know

Text input & output

- We've used the `print` command but only in its simplest form
- we can also `read` text from the command line
- and we'll use the `string` command to format text nicely

Printing and Data Types

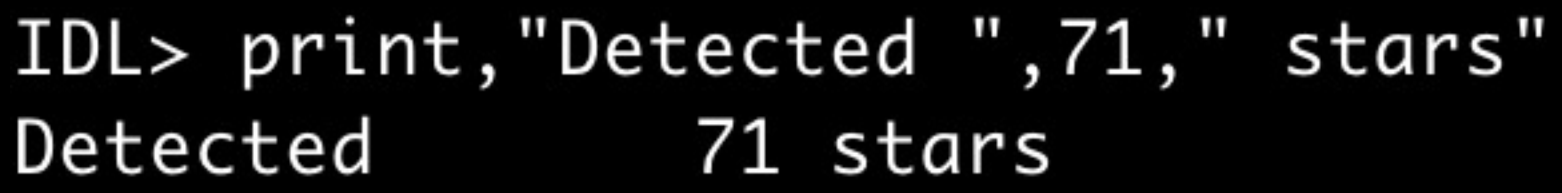
- IDL has a different default number of spaces depending on the *type* of the value being printed



```
IDL> print,1b
   1
IDL> print,1s
       1
IDL> print,1l
           1
IDL> print,long64(1)
                     1

IDL> print,1.
      1.00000
IDL> print,1d
       1.0000000
```

Printing

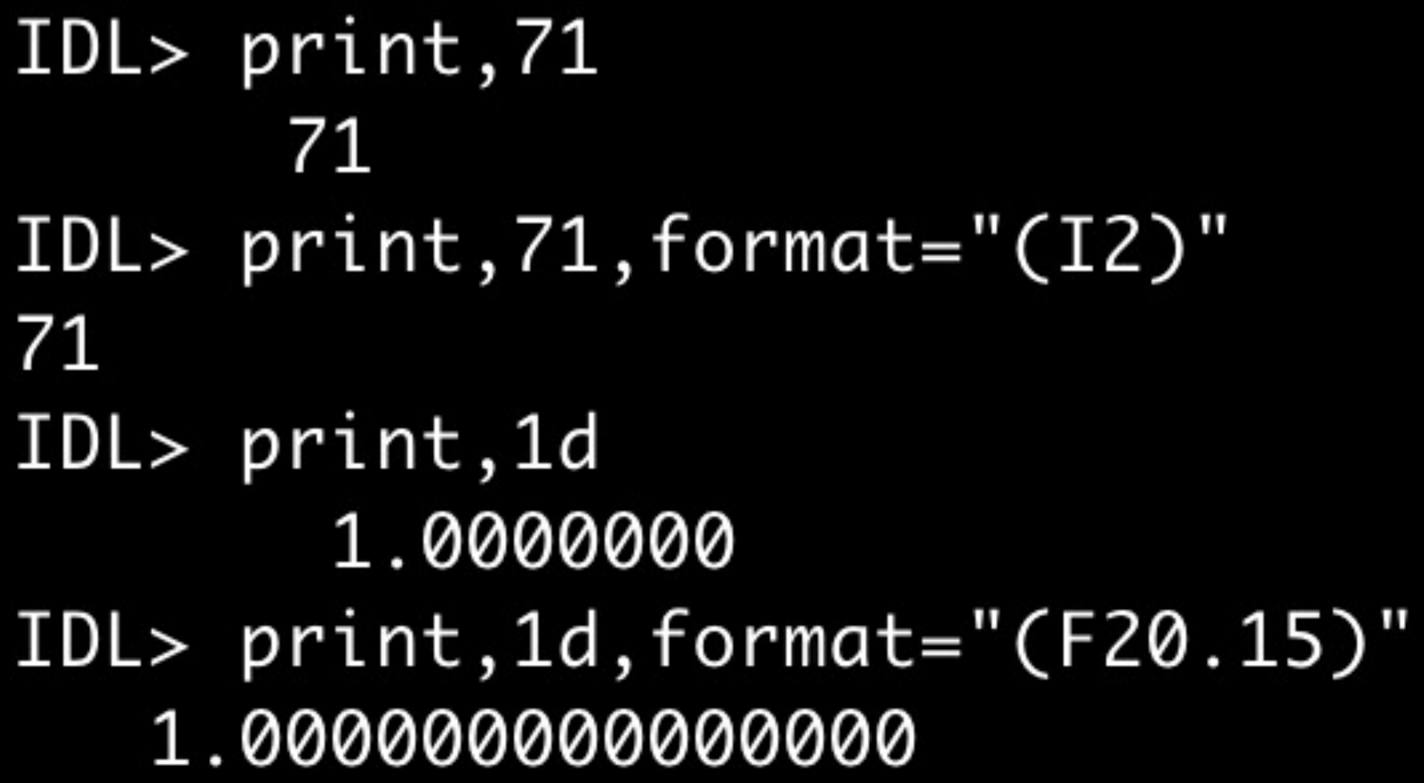
- This can be kind of ugly:



```
IDL> print,"Detected ",71," stars"
Detected        71 stars
```

- so it would be nice to format what we print

Print Formatting



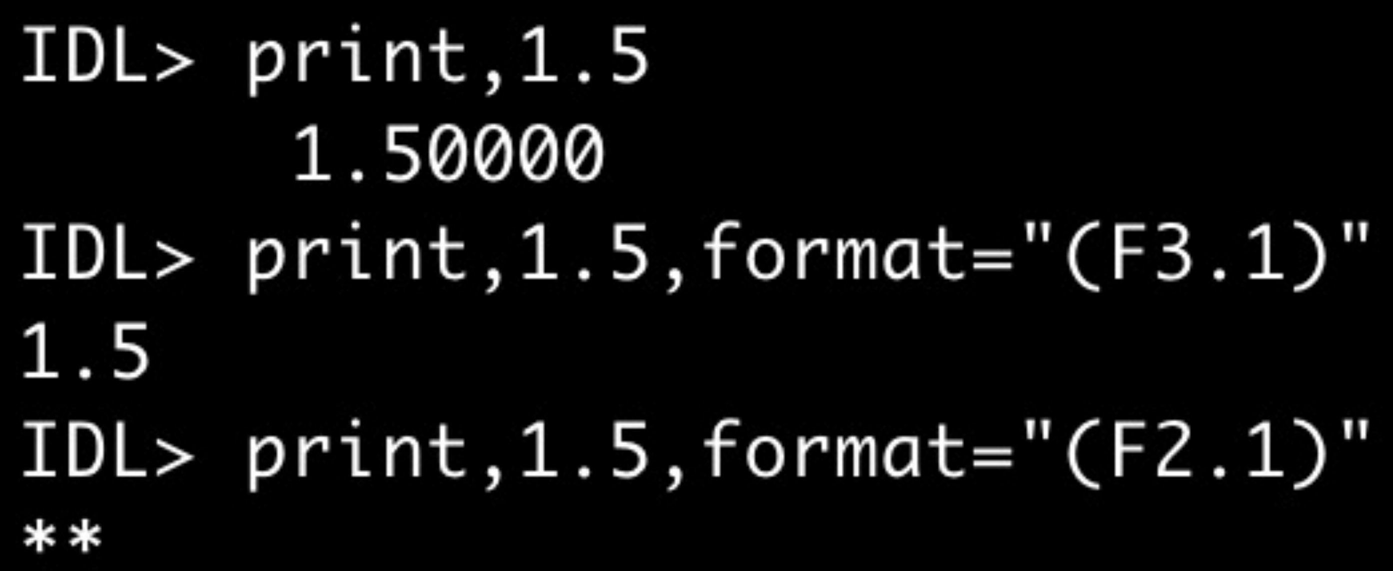
```
IDL> print,71
      71
IDL> print,71,format="(I2)"
71
IDL> print,1d
       1.0000000
IDL> print,1d,format="(F20.15)"
   1.000000000000000
```

Print Formatting

- A “format string” tells IDL how to print out numbers and strings, and each formatting code is type-specific
  - `I9` = Integer with 9 *total* characters
  - `I09` = integer with 9 total characters, where instead of leading spaces, there are leading zeros

Float Formats

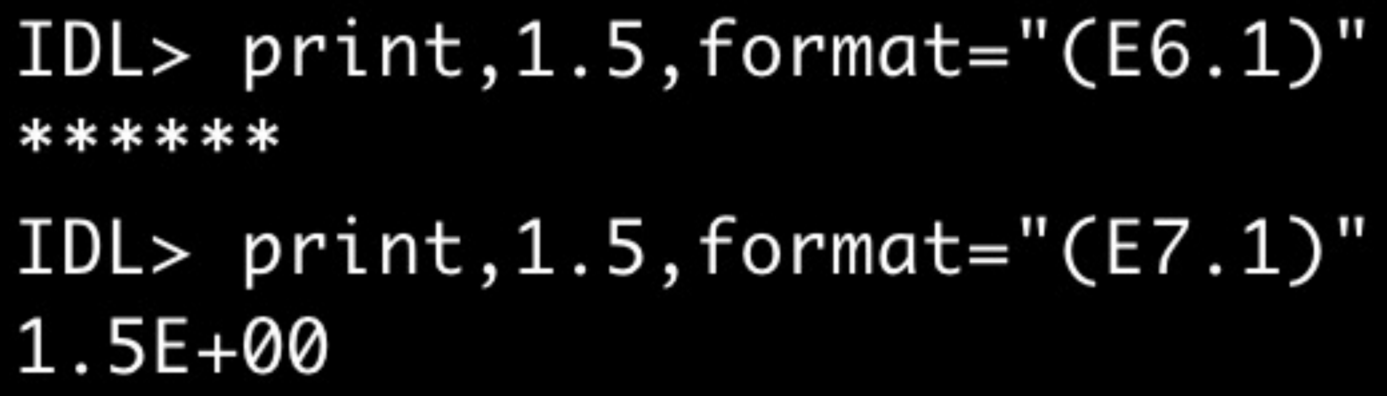
- `(F6.2)` indicates 6 *total* characters and two numbers after the decimal point
  - the 6 characters *include* the decimal!



```
IDL> print,1.5
      1.50000
IDL> print,1.5,format="(F3.1)"
1.5
IDL> print,1.5,format="(F2.1)"
**
```

Float Formats

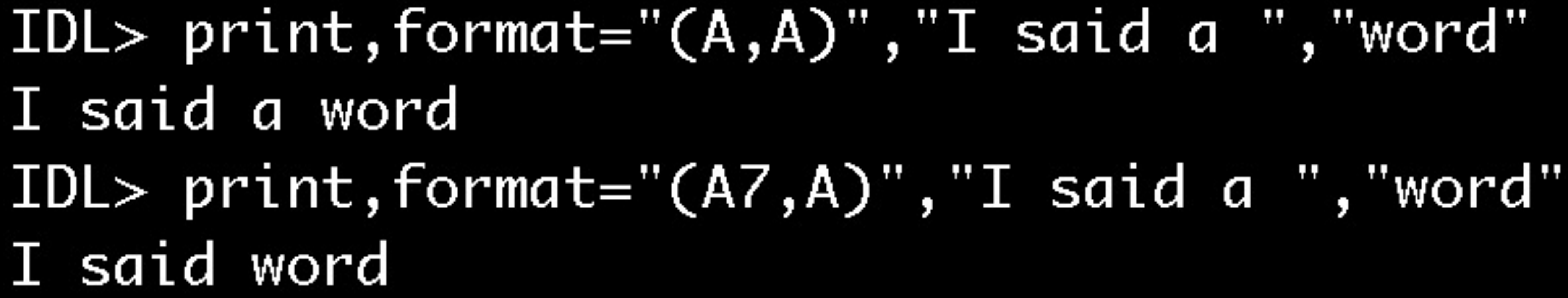
- Exponentials take even more space
- `(E7.1)` means 7 characters, which MUST include the trailing characters "`E+##`"



```
IDL> print,1.5,format="(E6.1)"
******
IDL> print,1.5,format="(E7.1)"
1.5E+00
```

String Formatting

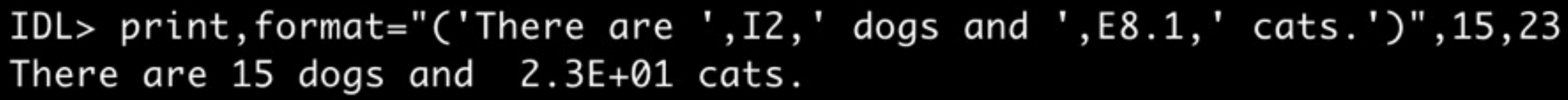
- There is also an “`A`” format for strings
  - I don’t know why it’s A, not S. Maybe for ASCII?
- It can be used to truncate (chop off) text by specifying the allowed number of characters



```
IDL> print,format="(A,A)","I said a ","word"
I said a word
IDL> print,format="(A7,A)","I said a ","word"
I said word
```

Formatting...

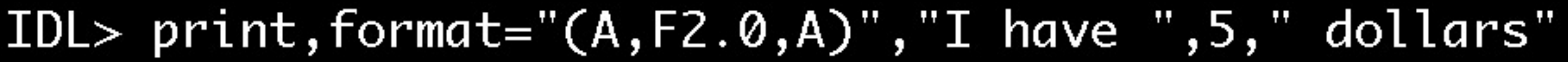
- Can use multiple format specifications:



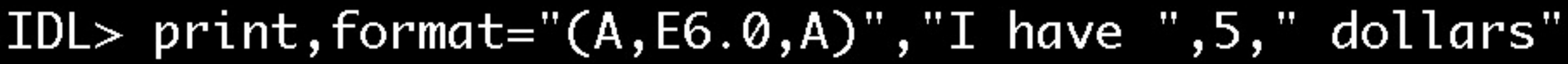
```
IDL> print,format="('There are ',I2,' dogs and ',E8.1,' cats.')",15,23
There are 15 dogs and  2.3E+01 cats.
```

  - You can include “literal strings” within the format specification
  - You can have multiple different format specifications

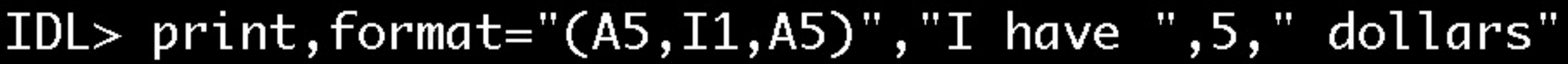
Which of these lines is *not* valid?

A) 

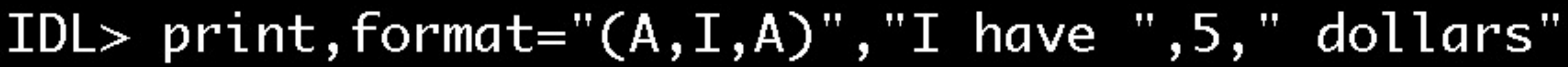
```
IDL> print,format="(A,F2.0,A)","I have ",5," dollars"
```

B) 

```
IDL> print,format="(A,E6.0,A)","I have ",5," dollars"
```

C) 

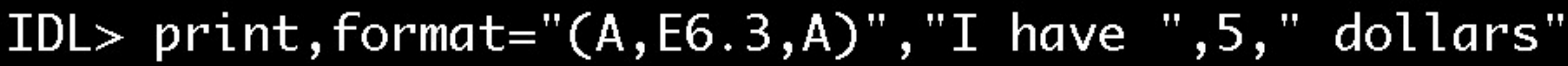
```
IDL> print,format="(A5,I1,A5)","I have ",5," dollars"
```

D) 

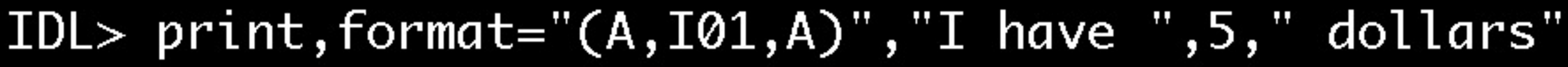
```
IDL> print,format="(A,I,A)","I have ",5," dollars"
```

E) None of the above / I don’t know

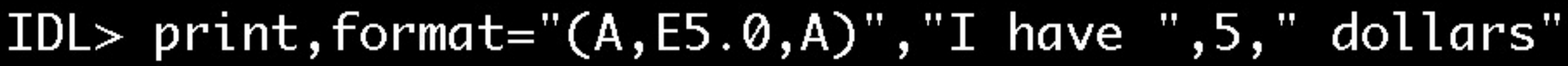
Which of these lines will include the number 5 when printed?

A) 

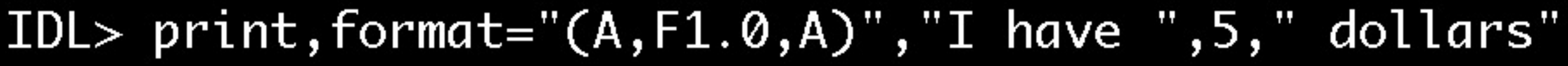
```
IDL> print,format="(A,E6.3,A)","I have ",5," dollars"
```

B) 

```
IDL> print,format="(A,I01,A)","I have ",5," dollars"
```

C) 

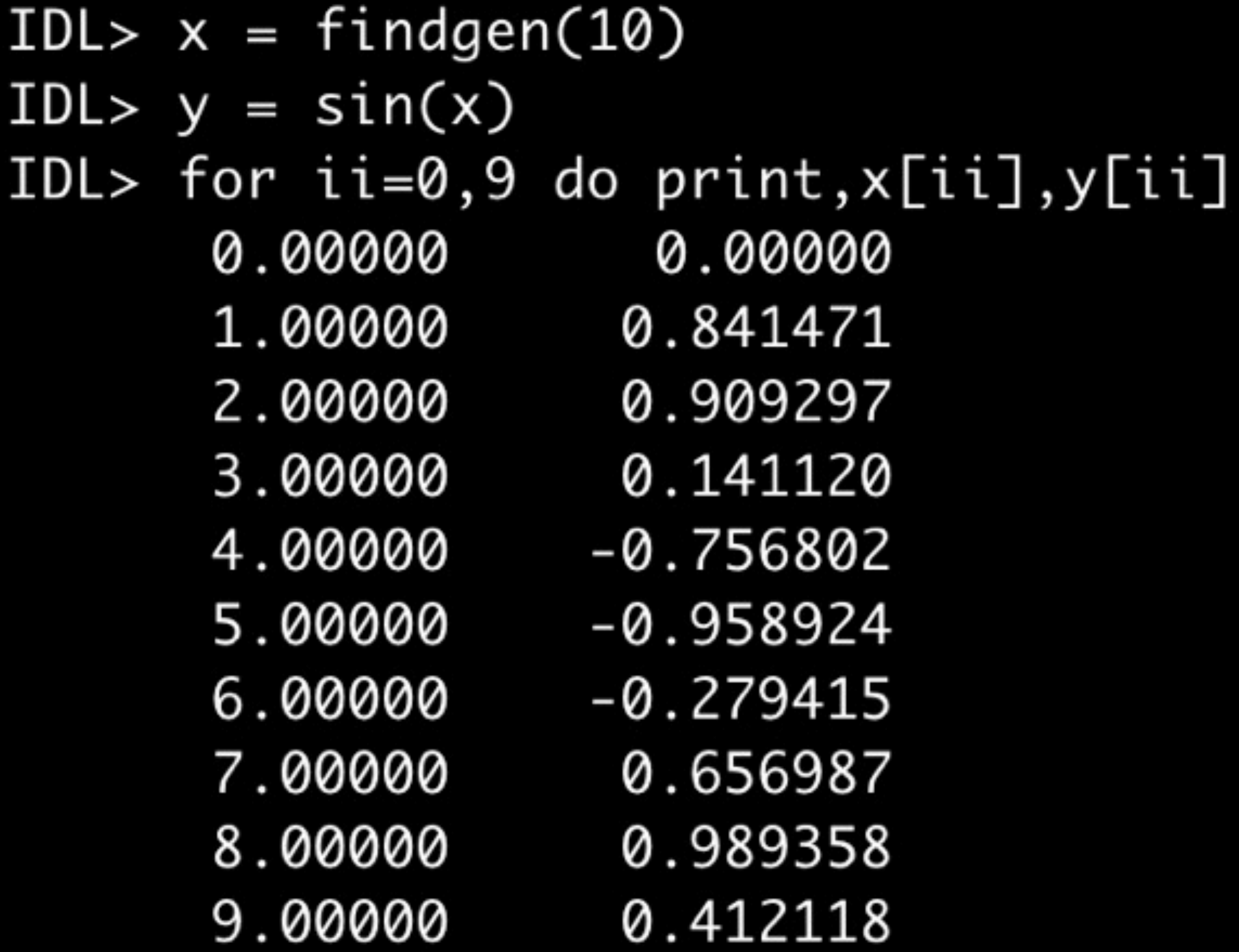
```
IDL> print,format="(A,E5.0,A)","I have ",5," dollars"
```

D) 

```
IDL> print,format="(A,F1.0,A)","I have ",5," dollars"
```

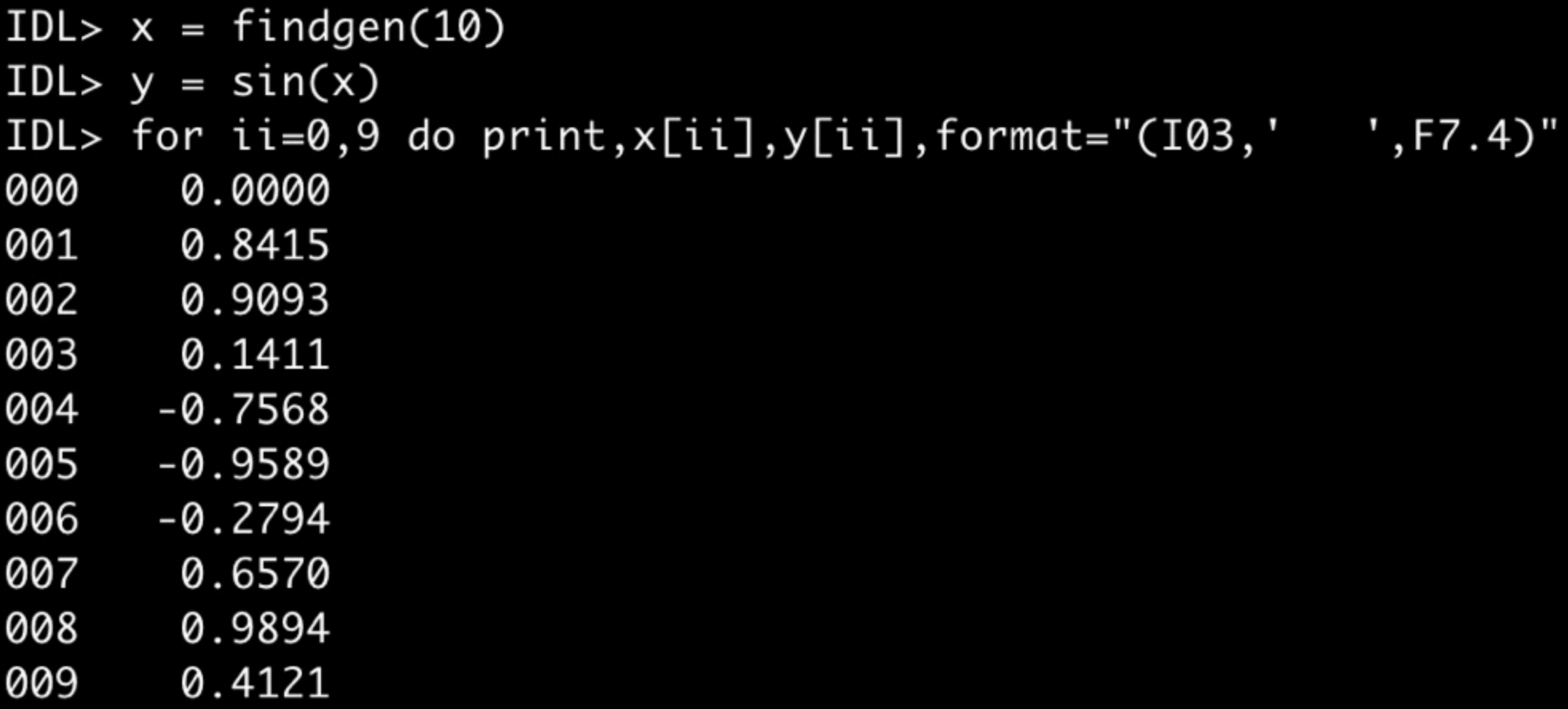
E) None of the above / I don’t know

Formatting to make Tables



```
IDL> x = findgen(10)
IDL> y = sin(x)
IDL> for ii=0,9 do print,x[ii],y[ii]
      0.00000      0.00000
      1.00000     0.841471
      2.00000     0.909297
      3.00000     0.141120
      4.00000    -0.756802
      5.00000    -0.958924
      6.00000    -0.279415
      7.00000     0.656987
      8.00000     0.989358
      9.00000     0.412118
```

Formatting to make Tables



```
IDL> x = findgen(10)
IDL> y = sin(x)
IDL> for ii=0,9 do print,x[ii],y[ii],format="(I03,'   ',F7.4)"
000    0.0000
001    0.8415
002    0.9093
003    0.1411
004   -0.7568
005   -0.9589
006   -0.2794
007    0.6570
008    0.9894
009    0.4121
```

Format strings

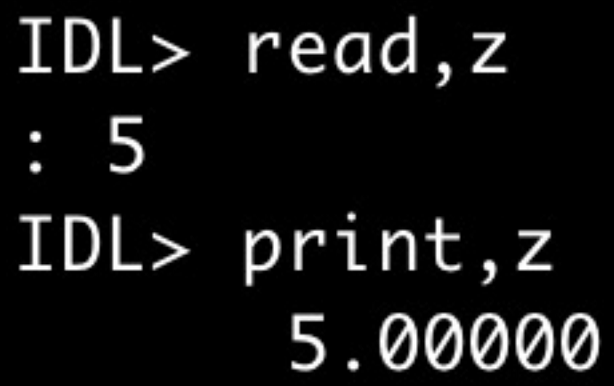
- Format strings are just strings:
  - `“(I5,F6,‘the dog’,E12)”`
- They can therefore be stored in variables
  - `format=“(I5,F6,‘the dog’,E12)”`
  - `print,format=format,1,2,3`

The `string` function

- Behaves quite a lot like `print`, but it is a function rather than a procedure
  - `newstring = string(“twelve”,5,format=”(A,I1)”)`
- Great for making sequential filenames
  - `filename = string(format=”(‘testfile’,I02,’.pro’,1)`

The `read` procedure

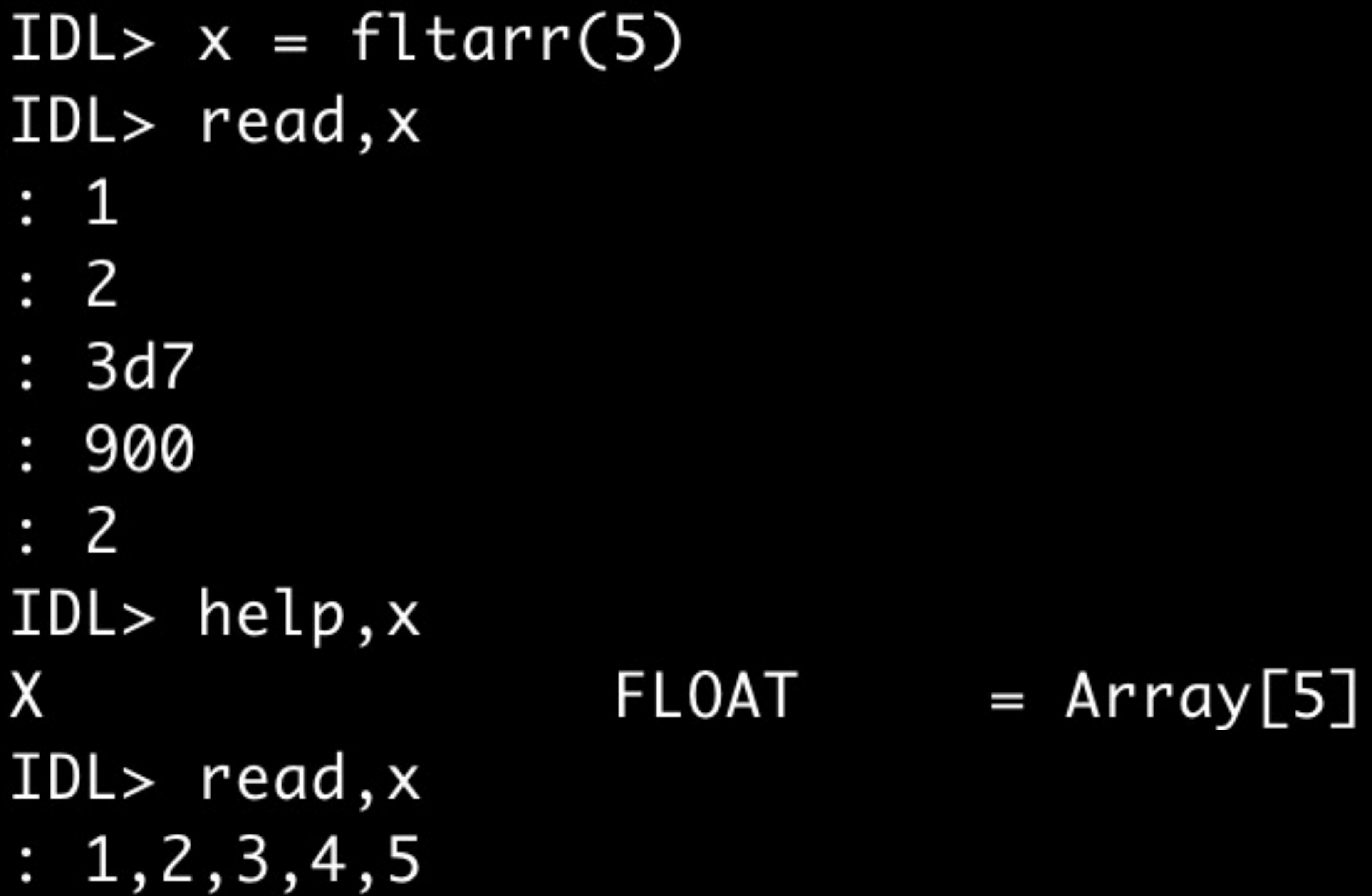
- Allows you to read values entered from the command line into variables
- Not the most useful procedure



```
IDL> read,z
: 5
IDL> print,z
      5.00000
```

z is assumed float

read



```
IDL> x = fltarr(5)
IDL> read,x
: 1
: 2
: 3d7
: 900
: 2
IDL> help,x
X               FLOAT     = Array[5]
IDL> read,x
: 1,2,3,4,5
```

File I/O

- Kind of a big deal - this is how you'll interact with all sorts of data
- It's also how you'll save and maybe eventually publish your work

Opening Files

- Most languages make you open a file in one of a few modes; IDL has the first 3
  - ‘r’ - read
  - ‘w’ - write
  - ‘u’ - both
  - (‘a’ - append)
  - (‘b’ - binary)

Opening Files

- Good practice to only use ‘r’ and ‘w’: that way you can’t accidentally overwrite something you were reading from
- commands `openr`, `openw`

Opening Files

- `openw, lun, filename, /get_lun`
  - `lun`: “Logical Unit Number”, an integer
  - `filename`: e.g. ‘data.dat’
  - `/get_lun`: flag to say you need a new lun defined for you

Using your file

- `printf` - write formatted ASCII text data to a file
- `readf` - read text
- `writeu` - write binary data to a file
- `readu` - read binary data

printf

- printf, lun, “text to send”, “to the file”
  - lun is the second parameter
  - otherwise, same as print (accepts format keyword)

Closing Files

- You MUST close files when you’re done with them
  - until you close the file, it remains empty
- Weird command for “`close`”:
  - `free_lun, lun`

IDL is weird

- I would have thought you would do:
  - `openw,lun,‘file.txt’,/get_lun`
  - `close,lun,/free_lun`
- But no! That is not right!

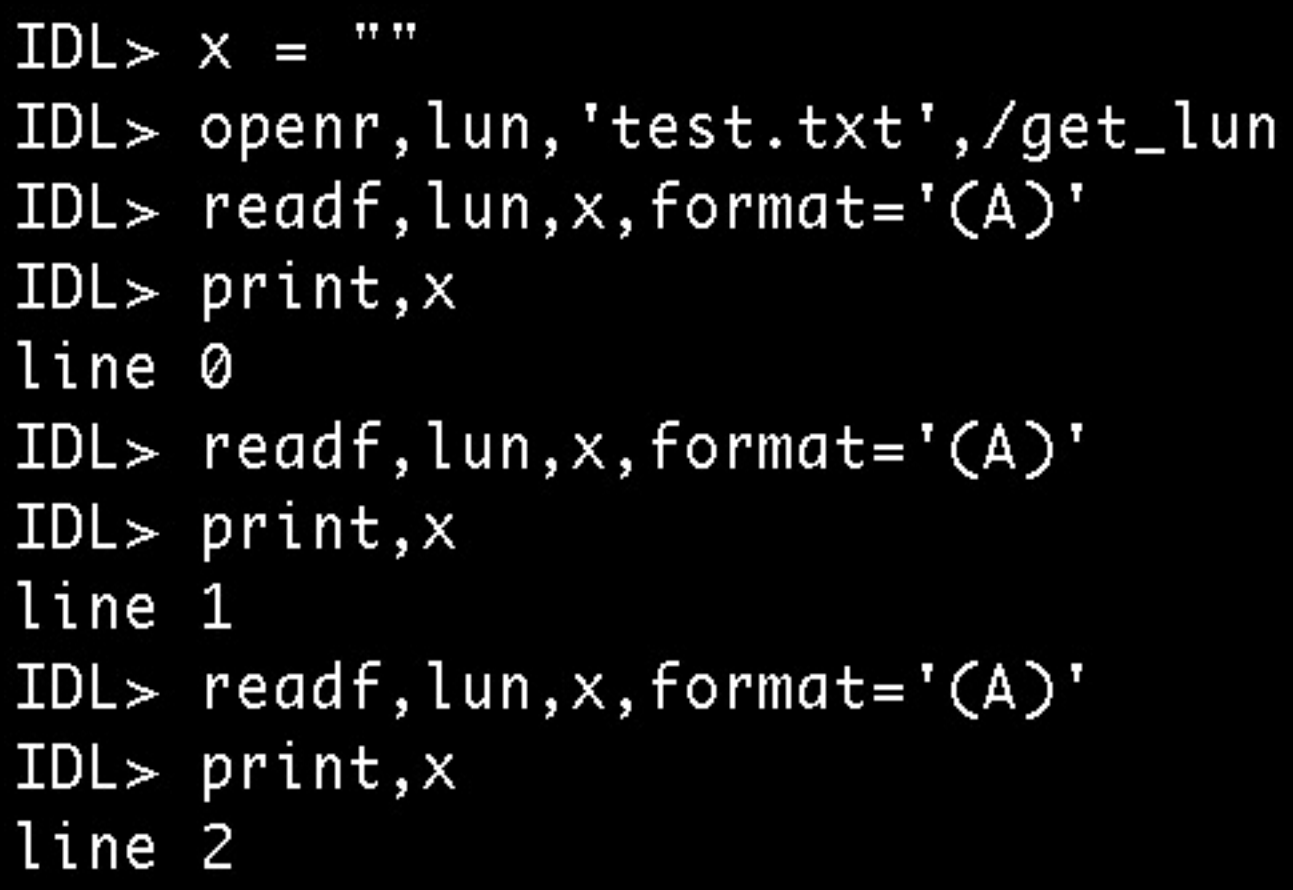
LUNs

- `close,lun` will close your file, but it won’t give you permission to use that LUN
- There are only 32 LUNs available in IDL
- If you close without freeing them, you can only open a total of 32 files in a session, then you’re stuck....
    - but `close,/all` closes everything and frees all your LUNs

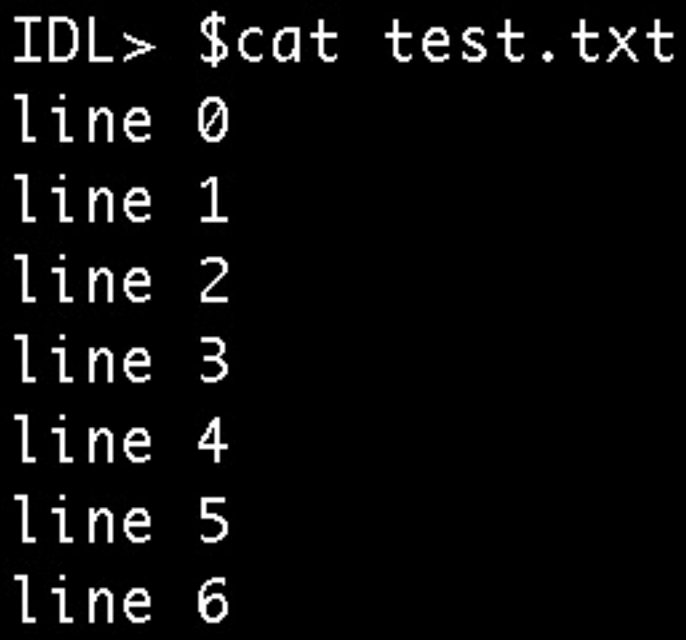
Reading

- `readf,lun,variable`
  - WARNING: if the file contains a string, you MUST “declare” the variable to be of string type first! This is in contradiction to the general philosophy of IDL
  - This strikes me as crazy

Reading



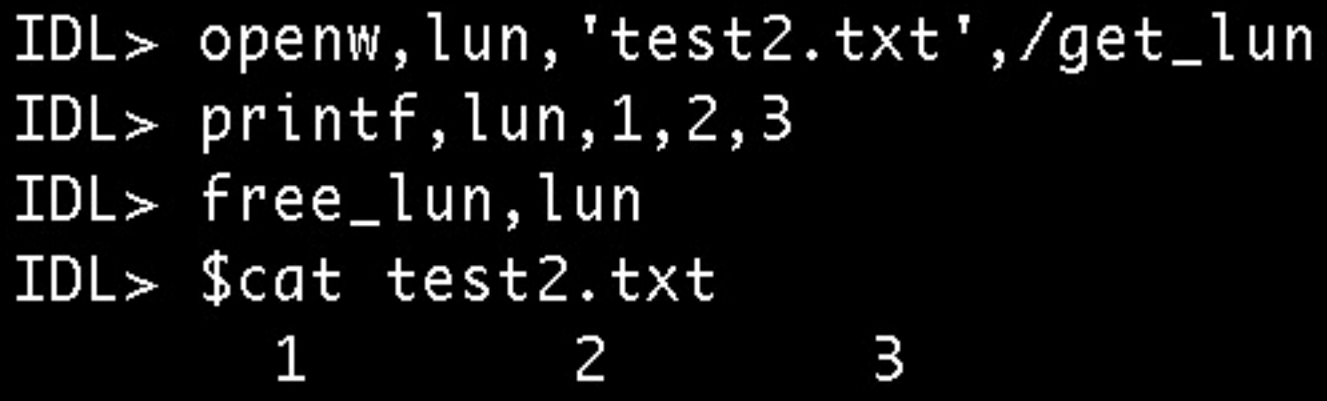
```
IDL> x = ""
IDL> openr,lun,'test.txt',/get_lun
IDL> readf,lun,x,format='(A)'
IDL> print,x
line 0
IDL> readf,lun,x,format='(A)'
IDL> print,x
line 1
IDL> readf,lun,x,format='(A)'
IDL> print,x
line 2
```



```
IDL> $cat test.txt
line 0
line 1
line 2
line 3
line 4
line 5
line 6
```

- You can call shell commands from IDL if you precede them with `$`

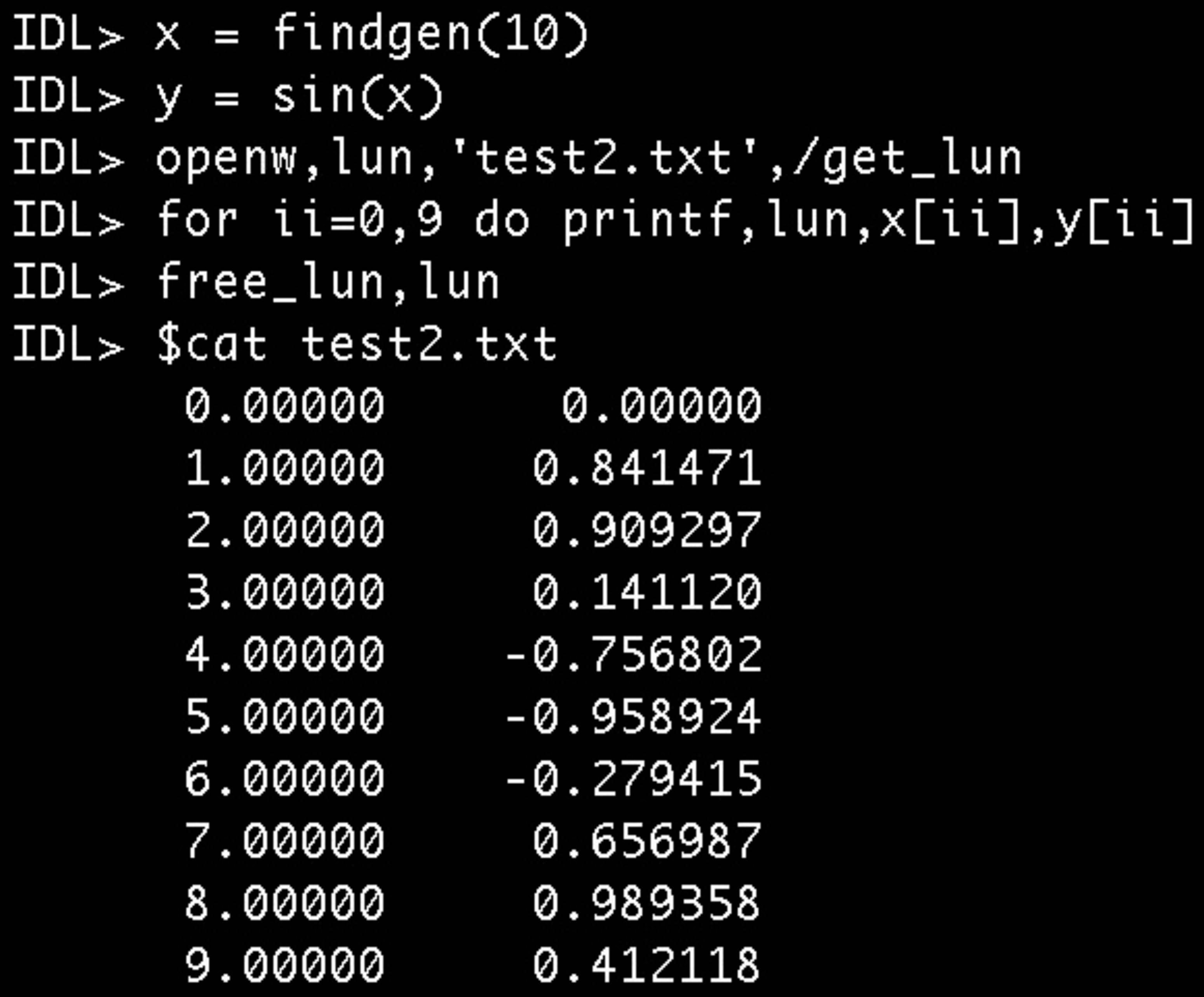
Writing



```
IDL> openw,lun,'test2.txt',/get_lun
IDL> printf,lun,1,2,3
IDL> free_lun,lun
IDL> $cat test2.txt
       1       2       3
```

- Pretty straightforward
- but not really easy to write columns this way
  - a new line is added after each `print` statement

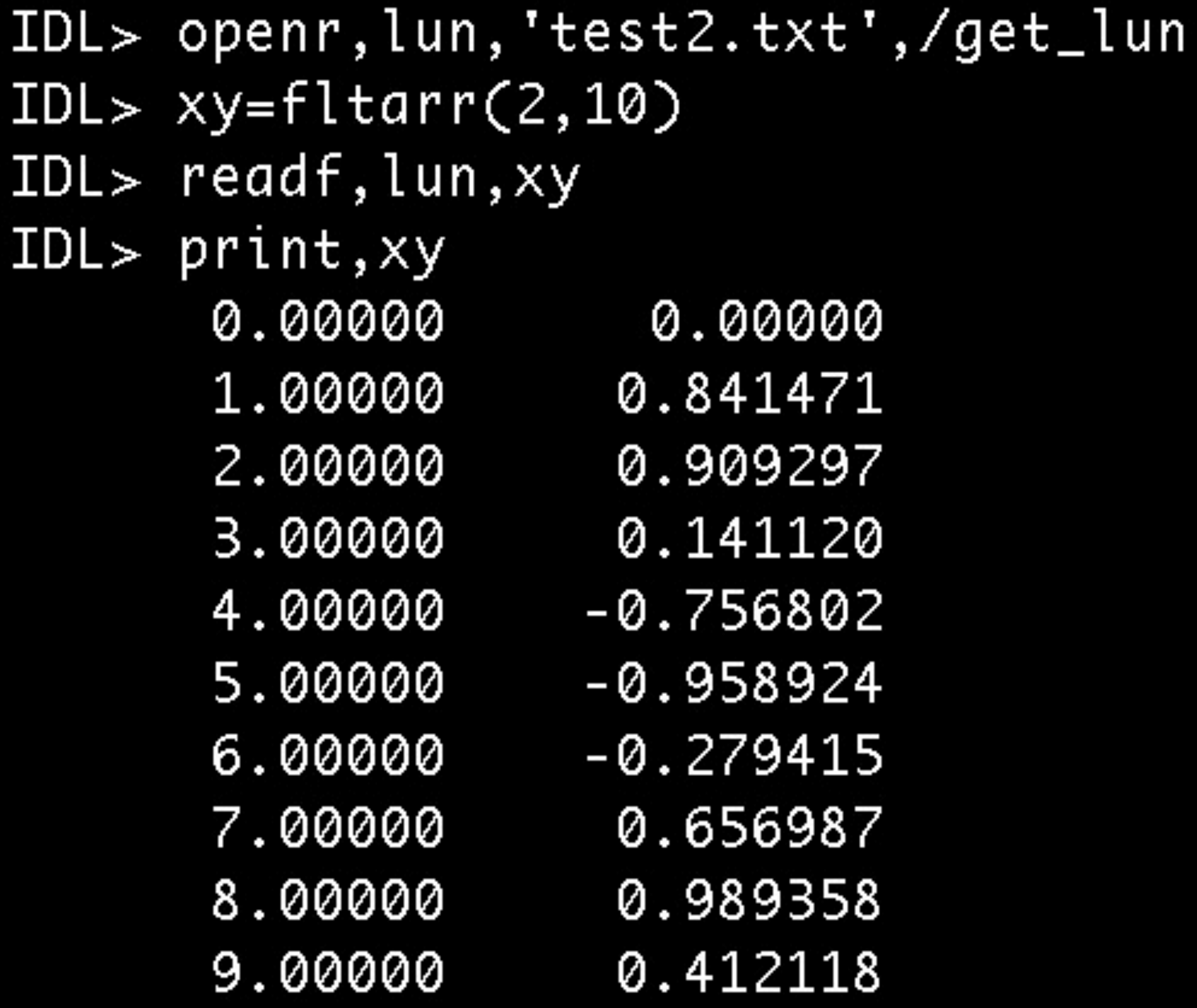
Writing Columns



```
IDL> x = findgen(10)
IDL> y = sin(x)
IDL> openw,lun,'test2.txt',/get_lun
IDL> for ii=0,9 do printf,lun,x[ii],y[ii]
IDL> free_lun,lun
IDL> $cat test2.txt
      0.00000      0.00000
      1.00000     0.841471
      2.00000     0.909297
      3.00000     0.141120
      4.00000    -0.756802
      5.00000    -0.958924
      6.00000    -0.279415
      7.00000     0.656987
      8.00000     0.989358
      9.00000     0.412118
```

Reading Columns

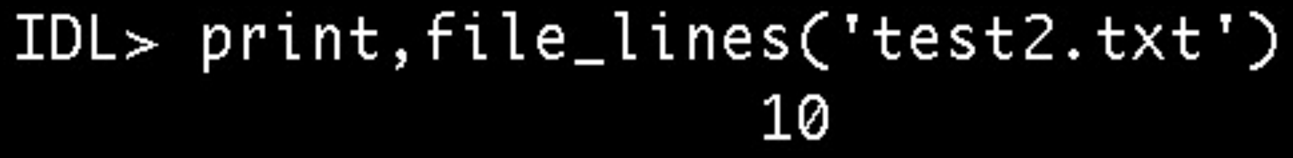
- If you know the *exact* format of the file you’re reading in, you can read columns



```
IDL> openr,lun,'test2.txt',/get_lun
IDL> xy=fltarr(2,10)
IDL> readf,lun,xy
IDL> print,xy
      0.00000      0.00000
      1.00000     0.841471
      2.00000     0.909297
      3.00000     0.141120
      4.00000    -0.756802
      5.00000    -0.958924
      6.00000    -0.279415
      7.00000     0.656987
      8.00000     0.989358
      9.00000     0.412118
```

How many lines in a file?

- file_lines
- Convenient if you know how many *columns* but not how many *rows*



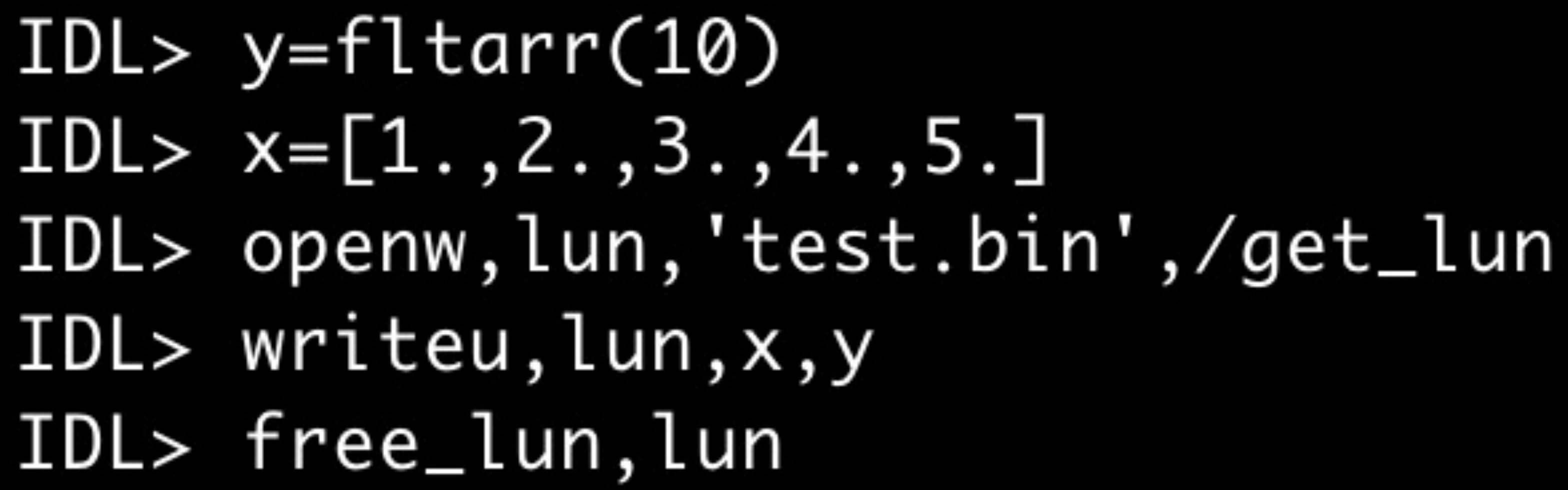
```
IDL> print,file_lines('test2.txt')
                      10
```

Binary Files

- Store the data more efficiently:
- -32676 is 6 characters (or 6 bytes) if stored as ascii
- But the number, if we know it’s a short integer, can be represented by 2 bytes

Writing Binary Files

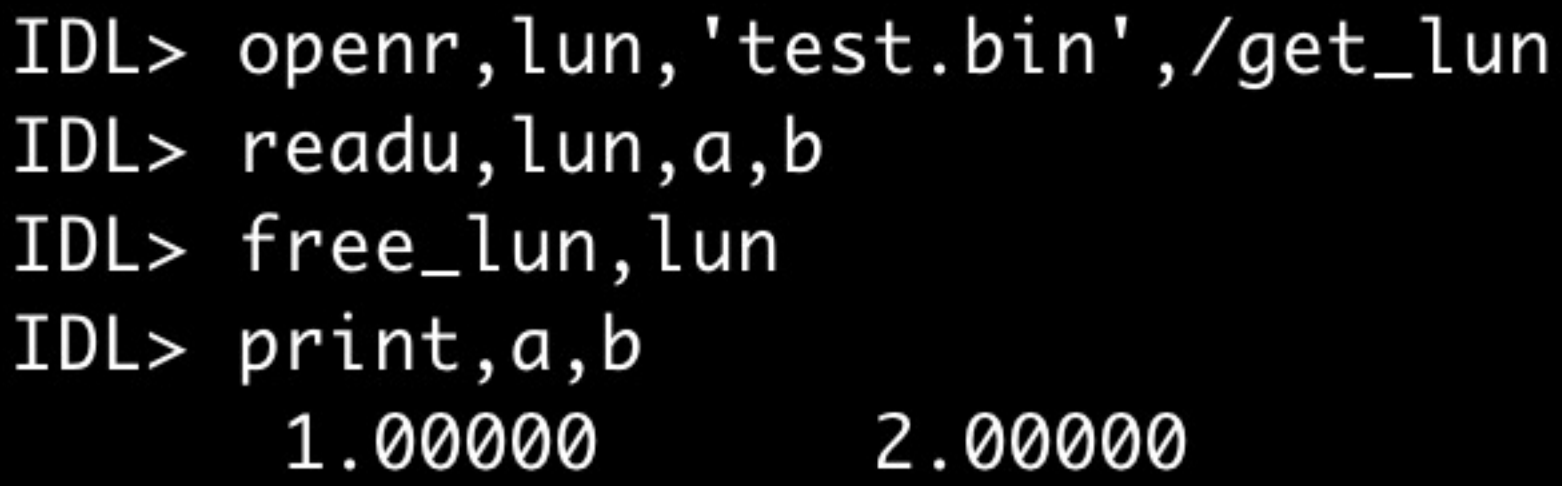
- Straightforward: Write the variables, don’t worry about formatting



```
IDL> y=fltarr(10)
IDL> x=[1.,2.,3.,4.,5.]
IDL> openw,lun,'test.bin',/get_lun
IDL> writeu,lun,x,y
IDL> free_lun,lun
```

Reading Binary Files

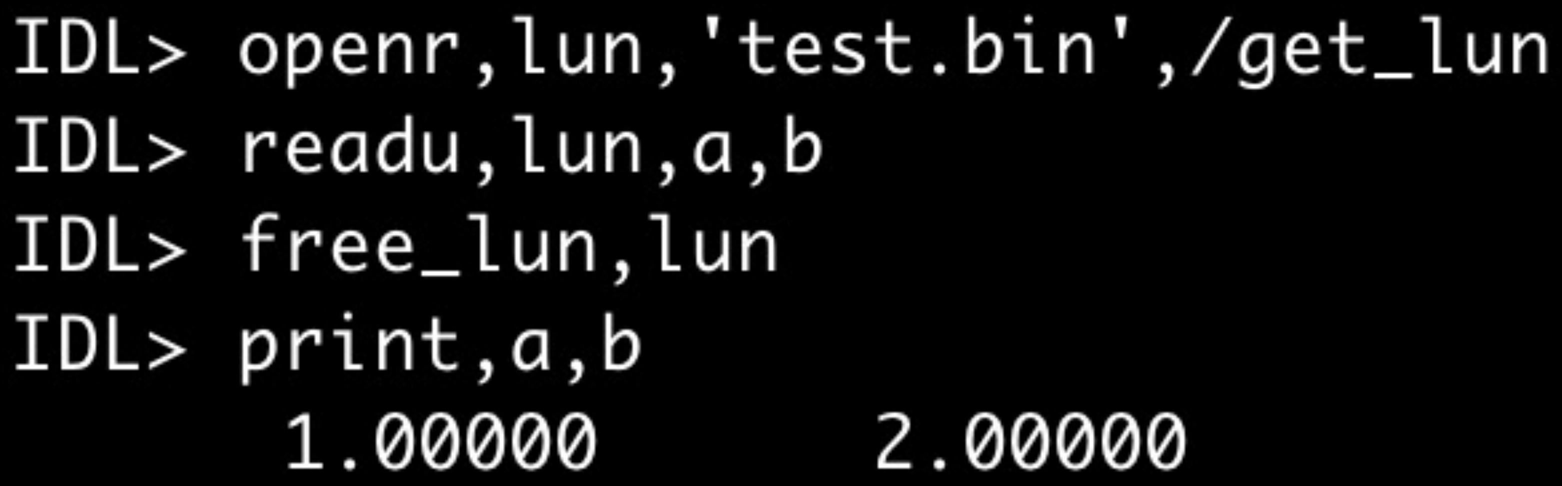
- Difficult: You must know exactly what’s in them



```
IDL> openr,lun,'test.bin',/get_lun
IDL> readu,lun,a,b
IDL> free_lun,lun
IDL> print,a,b
      1.00000      2.00000
```

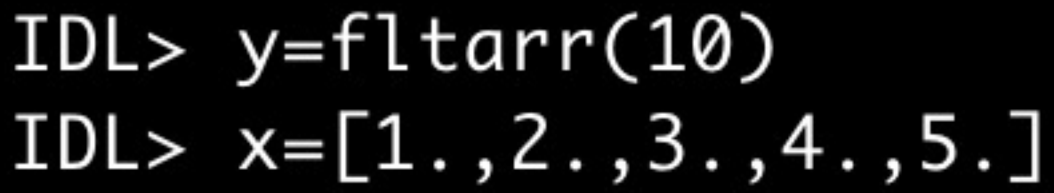
Reading Binary Files

- Difficult: You must know exactly what’s in them



```
IDL> openr,lun,'test.bin',/get_lun
IDL> readu,lun,a,b
IDL> free_lun,lun
IDL> print,a,b
      1.00000      2.00000
```

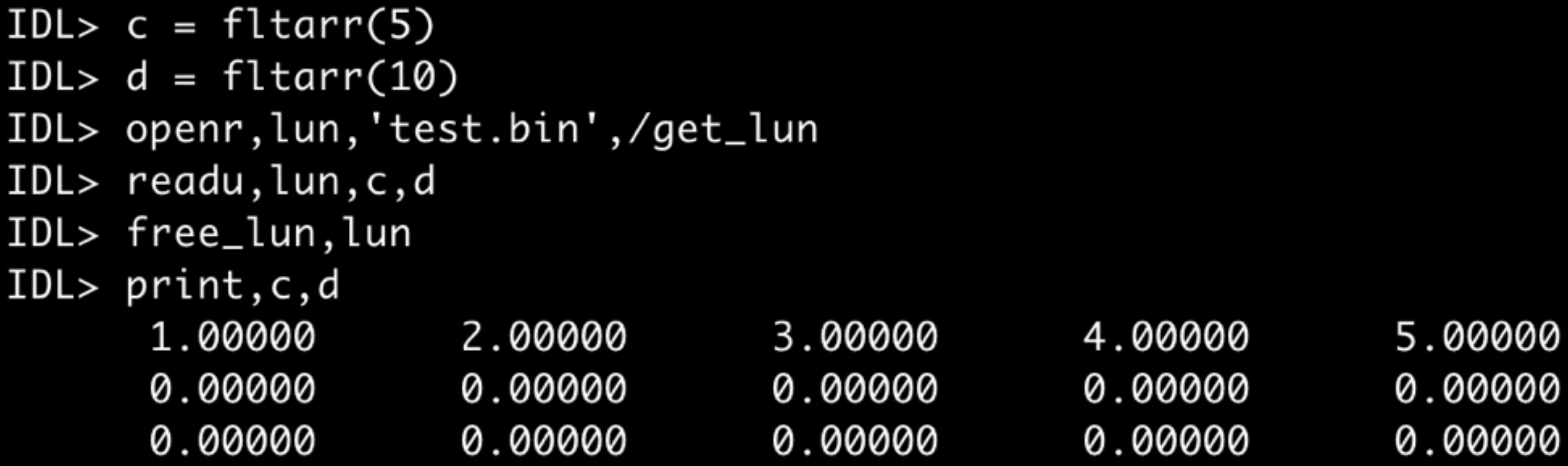
You probably expected:



```
IDL> y=fltarr(10)
IDL> x=[1.,2.,3.,4.,5.]
```

Reading Binary Files

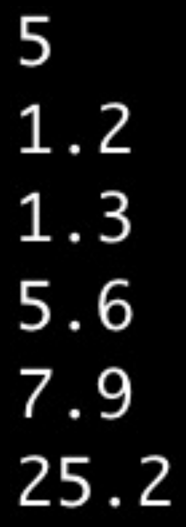
- Must “declare” variables before reading them



```
IDL> c = fltarr(5)
IDL> d = fltarr(10)
IDL> openr,lun,'test.bin',/get_lun
IDL> readu,lun,c,d
IDL> free_lun,lun
IDL> print,c,d
      1.00000      2.00000      3.00000      4.00000      5.00000
      0.00000      0.00000      0.00000      0.00000      0.00000
      0.00000      0.00000      0.00000      0.00000      0.00000
```

Self-Describing Binary

- First N bytes of the file describe what is in the rest of the file
    - e.g., first 4 bytes are a single Long Integer telling how many floats there are in the file



```
5
1.2
1.3
5.6
7.9
25.2
```

Endianness

- Short Integers are represented by two bytes. e.g., 50:
- Big Endian:

```
0 0 0 0 0 0 0 0   0 0 1 1 0 0 1 0
```

- Little Endian:

```
0 0 1 1 0 0 1 0   0 0 0 0 0 0 0 0
```

If you read this wrong, you'd get `12544`

Endianness

- Intel processors are “little endian”
- If you ever get total nonsense numbers, it’s possible the endianness is off
  - I’ve never encountered a need to worry about endianness in my career, but there is a possibility you’ll run into it