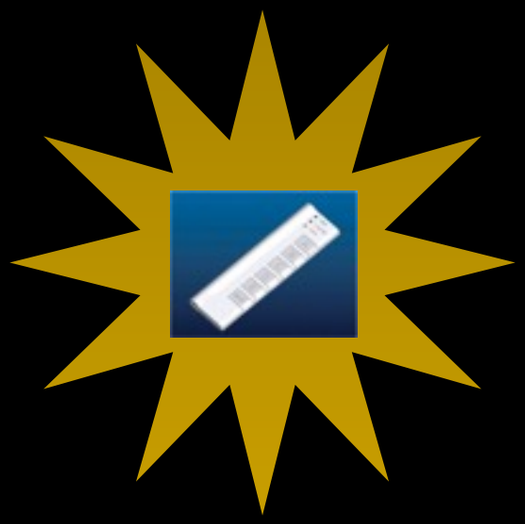
Animation

- We did a very simple “live” animation in `snake.pro`
- I also showed a sample of one of Cameron’s assignments using `xinteranimate`. We’ll use that now

Xinteranimate

- X for X-windows, the graphic system for linux
- inter - interactive
- animate. Like cartoons.



```
ii = 6
print,ii++
print,ii
```

What will be printed?

A) 5,6

B) 6,6

C) 6,7

D) 7,7

E) None of the above

Xinteranimate

1. Set up the plotting area

```
xinteranimate,set=[500,500,90],title='title',/showload
```

2. Add frames to the animation

```
xinteranimate,image=img,frame=iframe
```

3. Animate!

```
xinteranimate,30
```

Xinteranimate

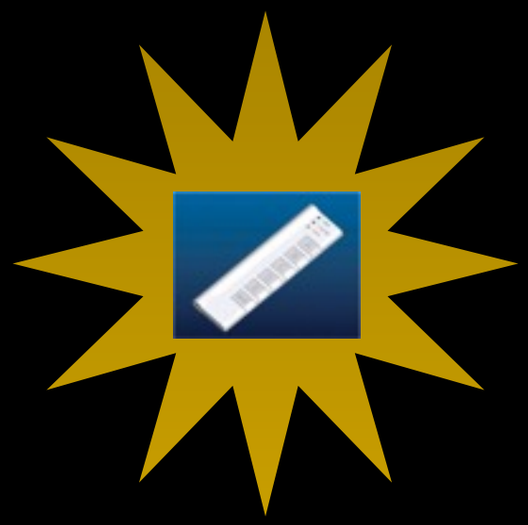
- Single task does many things depending on parameters you pass it
- in general, I don’t like this approach - it’s easier to make mistakes when re-using the same command

```
xinteranimate,set=[500,500,90],title='title',/showload
```

`set=[500,500,90]` Set the frame size to be 500x500 pixels
There will be 90 frames

`title='title'` Set the plot title (pick something descriptive)

`/showload` Show each frame as it is loaded
-good if you're at cosmos
-turn this off if you're working remotely



How much memory will it take to store a 500x500x90 movie?
(500x500x90 = 22,500,000)

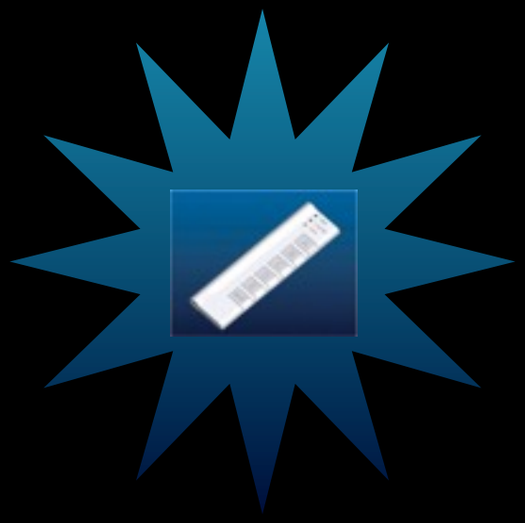
A) 500x500x90x2 = 45 million bytes ~ 45 MB

B) 500x500x90x500 = 11.25 billion bytes ~ 11.25 GB

C) 500x500x90x90 = 2.025 billion bytes ~ 2.025 GB

D) All of the above

E) None of the above



How much memory will it take to store a 2 hour movie at 1080p resolution at 30 frames per second?

A) 700 megabytes

B) 700 gigabytes

C) 700 terabytes

D) 700 petabytes

E) 700 kilobytes

```
xinteranimate,set=[500,500,90],title='title',/showload
```

`set=[500,500,90]` Set the frame size to be 500x500 pixels
There will be 90 frames

`title='title'` Set the plot title (pick something descriptive)

`/showload` Show each frame as it is loaded
-good if you're at cosmos
-turn this off if you're working remotely

```
xinteranimate,image=img,frame=iframe
```

```
image=img,
```

`img` must be a 500x500 image
-can becolor or monochrome
-500x500 was set on last slide!

```
frame=iframe
```

`iframe` is the frame number, and must be specified
-technically, you could add/change frames out of order, but why?

```
xinteranimate,30
```

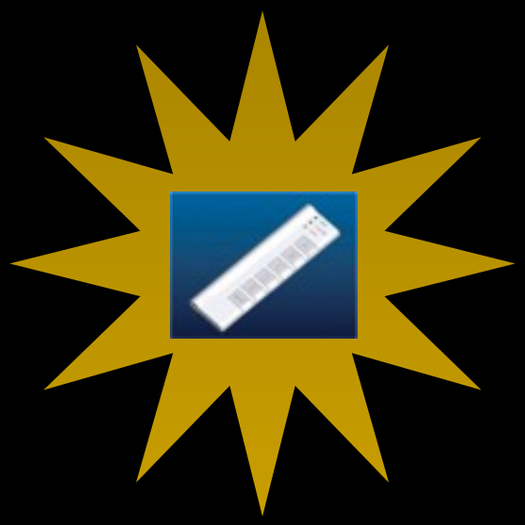
30 specifies the initial framerate in frames per second

-it can be changed once the animation has started

```
nframes = 100
imsize=300
xinteranimate,set=[imsize,imsize,nframes],title='title',/showload

plot,[0,nframes*5],[0,nframes*5],/nodata,xstyle=4,ystyle=4
for iframe=0,nframes-1 do begin
    oplot,[iframe*5,0],[iframe*5,nframes*5]
    img = tvrd()
    xinteranimate, image=img, frame=iframe
endfor

xinteranimate,30
end
```



How do you read a *color* image into data?

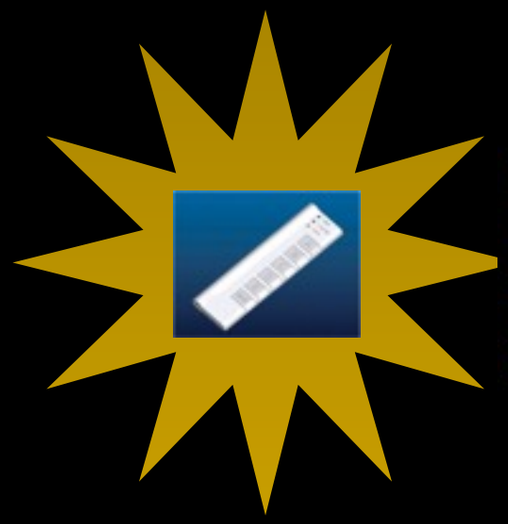
A) `img = color_tvrd()`

B) `img = tvrd(/color)`

C) `img = tvrd()`

D) `img = tvrd(/true)`

E) None of the above



```
nframes = 360+180
imsize=500
xinteranimate,set=[imsize,imsize,nframes]
```

If you have set up your plot window as above, what size will `img` be once you do `img=tvrd(/true)`?

A)`[imsize,imsize,3]`

B)`[nframes,nframes,3]`

C)`[imsize,imsize]`

D)`[nframes,nframes]`

E) None of the above

Skipping Ahead:
Ch 18 now, 17 later

COMMON blocks

- `COMMON` blocks allow you to create “global” variables, i.e. things you can access from anywhere
  - kind of like `!pi`, except no ! and you get to write them
- However, they are easy to abuse and misuse. Their misuse will be punished severely! (i.e. your code will probably crash)

Using a Common Block

```
COMMON [name], variable1, variable2
```

- Each COMMON block has a name (just like a procedure or function
- It also has a list of named variables

When to use Common blocks

- COMMON blocks are useful when you want some numbers that will be common to all of your procedures and functions, but will never change
- e.g., constants

When *not* to use COMMON blocks

- If you have variables you want to change
  - instead, use ‘output parameters’
- When you want to pass variables between functions

Declaring Common Blocks

- COMMON blocks can be declared anywhere, in any “namespace” or “scope”
- But, I want you to have a separate location that *only* declares the common block

Declaring a COMMON block in a program

```
COMMON units, AU, kmpersec, year, parsec

      AU = 1.496e11                ; [m]
      kmpersec = 1e3               ; [m/s]
      year = 365L*24L*3600L        ; [s]
      parsec = 3.08567758e16       ; [m]

end
```

Declaring a COMMON block in a procedure

```
pro common_units

    COMMON units, AU, kmpersec, year, parsec
    AU = 1.496e11                ; [m]
    kmpersec = 1e3               ; [m/s]
    year = 365L*24L*3600L        ; [s]
    parsec = 3.08567758e16       ; [m]

end ; common_units
```

Program vs Procedure

- The *procedure* has the advantage that IDL will auto-find it
- You can also re-call the procedure to re-set the variables

Advantages of COMMON blocks

```
IDL> help,au,year
AU (UNITS)      FLOAT     =   1.49600e+11
YEAR (UNITS)    LONG      =     31536000
```

- `help` tells you which block it's in

A “feature” of common blocks

- You can “recall” a declared common block:

```
COMMON units
```

- This will set `AU, kmpersec, year, and parsec` in your local namespace
- But it doesn’t tell you what variables are being declared! You have to hunt down the definition to find out!

“Correct” COMMON usage

- So Dewey says instead, you should always do this:

```
COMMON units, AU, kmpersec, year, parsec
```

- That way, it’s always clear what variables you’re declaring

```
COMMON units, AU, kmpersec, year, parsec

AU = 1.496e11                    ; [m]
kmpersec = 1e3                   ; [m/s]
```

With the common block definition above, what will happen if I “recall” the common block using

```
COMMON units, kmpersec, AU
```

A) AU ~ 1.5e11, kmpersec ~ 1e3, year = 0

B) AU ~ 1e3, kmpersec ~ 1.5e11, year = 0

C) AU = 0, kmpersec = 0, year = 0

D) AU ~ 1.5e11, kmpersec ~ 1e3, year ~ !pi*10^7

E) None of the above

But wait!

- What if you change the COMMON block, and add a new variable?
  - You then have to change ALL references to the common block, everywhere! This is not so good
- Also, every “reference” to the COMMON block looks like a “definition”

Two alternative solutions, using structs

- You can put a structure in your COMMON block, then only have one *structure* declaration
- removes the risk of “overwriting” a common block

```
pro common_mks_units

    COMMON mks_units, mks_units

    mks_units = {MKS_UNITS,$
        AU : 1.496e11,                $; [m]
        kmpersec : 1e3,               $; [m/s]
        year : 365L*24L*3600L,        $; [s]
        parsec : 3.08567758e16,       $; [m]
        }

end ; common_mks_units
```

- Good, but doesn’t resolve the “reference” vs “definition” controversy:

```
COMMON mks_units     COMMON mks_units, mks_units
```

Defining System Variables

- (not covered in the book)
- Remember `!pi` and `!path`? We can make our own
- `!pi` can't be changed, which is good!

```
IDL> !pi=5
% Attempt to write to a readonly variable: !PI.
```

defsysv

DEFSYSV, *Name*, *Value* [, *Read_Only*] [, EXISTS=*variable*]

- name: “`!pi`” or similar (should be a string
- value: any variable!
- read_only: 1 or 0 - NOT a keyword!
- exists: Can be used to check if the variable already exists (safety check)

```
pro define_mks_units

    mks_units = {MKS_UNITS,$
        AU : 1.496e11,                $; [m]
        kmpersec : 1e3,               $; [m/s]
        year : 365L*24L*3600L,        $; [s]
        parsec : 3.08567758e16        $; [m]
        }

    defsysv,'!MKS_UNITS',mks_units,1

  end ; define_mks_units
```

This is a great approach:

-no common block stuff

-clear procedure name

-can never accidentally overwrite a unit

```
IDL> help,!mks_units
** Structure MKS_UNITS, 4 tags, length=16, data length=16:
   AU              FLOAT       1.49600e+11
   KMPERSEC        FLOAT            1000.00
   YEAR            LONG           31536000
   PARSEC          FLOAT       3.08568e+16
IDL> print,mks_units.au
  1.49600e+11
```

More goodness:

-`help` tells you about each component

-accessing the variable tells you both the name of the constant and the unit system

Naming

- Why did I use `mks_units`? Why not just `units`?
  - What if you wanted to use cgs units sometimes?
  - What if you use someone else's code, and they define `units` to be something different?
    - Introduces risk & conflict

Is there another way to do the same thing?

- Of course, this is programming!
- You could make a *function* that defines and returns the unit *structure*.

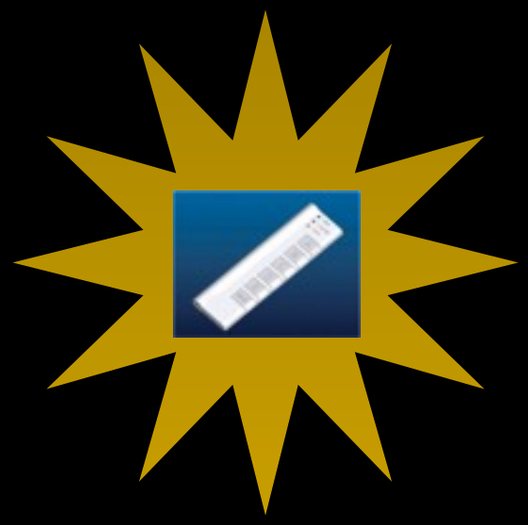
- Can never overwrite!
- Only disadvantage is that it's a little awkward to use
- Also, re-defines the struct each time

```
function mks_units

    mks_units = {MKS_UNITS,$
        AU : 1.496e11,                  $; [m]
        kmpersec : 1e3,                 $; [m/s]
        year : 365L*24L*3600L,          $; [s]
        parsec : 3.08567758e16          $; [m]
        }

    return,mks_units

end ; mks_units
IDL> help,mks_units()
** Structure MKS_UNITS, 4 tags, length=16, data length=16:
   AU              FLOAT       1.49600e+11
   KMPERSEC        FLOAT            1000.00
   YEAR            LONG           31536000
   PARSEC          FLOAT       3.08568e+16
IDL> print,(mks_units()).au
  1.49600e+11
```



Given *all* of the previous definitions, which of these will work (not crash)?

```
!mks_units.au = 5          ; A
mks_units.au = 5           ; B
(mks_units()).au = 5       ; C
mks_units().au = 5         ; D
; None of the above        ; E
```

```
IDL> !mks_units.au = 5    ; A
% Attempt to write to a readonly variable: Structure referen
% Execution halted at: $MAIN$
IDL> mks_units.au = 5     ; B
IDL> (mks_units()).au = 5 ; C
% Attempt to store into an expression: Structure reference.
% Execution halted at: $MAIN$
IDL> mks_units().au = 5   ; D
```

```
mks_units().au = 5   ; D
  ^
% Syntax error.
```

TESTING

- Not covered in the book!
- There's a LOT of ugly depth here; I'm going to give you some code to work with and try to help you understand it rather than building from scratch

TEST: Make sure the code *works* and is *right*

- It is nice to have, for each procedure or function, some other procedure/function that tells you whether it does what you want!

```
IDL> test_mks_units
          Tag Name                Command                   Value
                au                 Passed                  Passed
           kmpersec                Passed                  Passed
             parsec                Passed                X Failed X
               year                Passed                  Passed
```

Testing Tricks

- You need to have “code that writes and calls code” because you don’t want to try to write EVERY possible use of your function/procedure
- The `execute` function will execute code!
  - Pass it a string containing valid IDL code

Execute

- `exec_status = execute(command)`
- If the command succeeds, `exec_status=1`
- If the command fails, `exec_status=0`
- This is neat! It means you can ask IDL to tell you whether some code works!
  - execute is also a valid command in python

Small but complex example

```
pro test_mks_units

    ; make a hash storing the appropriate values
    tagvals = hash('au',1.496e11,$
                   'kmpersec',1e3,$
                   'year',365L*24*3600,$
                   'parsec',3.08567758e16)

    ; make a nicely formatted table
    print,"Tag Name","Command","Value",format="(3A20)"
    foreach val,tagvals,tag do begin
        cmd = 'OK = ((mks_units()).'+ tag +') eq '+string(val)
        test_status = execute(cmd)
        msg1 = test_status ? "Passed" : "X Failed X"
        msg2 = OK ? "Passed" : "X Failed X"
        print,tag,msg1,msg2,format="(3A20)"
    endforeach

end ; test_mks_units
```

(this code is available, with better comments, on github)