# Which section of this *script* is (probably) wrong?

A
```
; plot_file.pro
; read a spectrum with name specified by the user
read,filename,prompt="Enter the file name you'd like to read"
openr,lun,filename,/get_lun
```

B
```
nn = 0L          ; first # is known to be long
readu,lun,nn ; read first 4 bytes of file into long(n)
nn = swap_endian(nn) ; swap the endianness [repeated for all vars]
```

C
```
yy = fltarr(nn)
readu, lun, yy   ; read the rest of the file into yy
yy = swap_endian(yy)
```

D
```
free_lun,lun ; close file

plot,yy
```

E None of the above

# Flow Control

- This is what programming is really all about
- Flow Control is how you tell the computer how to make decisions

# Choosing and Repeating

- Two different flow control “constructs”
- “choosing” what to do next
  - `if`, `else`
- “repeating” an operation
  - `for`, `while` loops

# capitalization

- The text suggests capitalizing IDL words (`for`, `while`, etc)
- I don’t like this approach because it is different from every other language I’ve used
- You cannot use capital words in python, therefore don’t use them in IDL either

# The “`if`” statement

- This is probably the most commonly used thing in programming (except maybe the “=” assignment operator)
- Do something only if something else is true
- `if condition then statement`

# Single-Line `if` statements

```
In [1]: if True: print "Hello"
Hello

In [2]: if False: print "Not going to happen"

In [3]:
```

```
IDL> if 1 then print,"Hello"
Hello
IDL> if 0 then print,"Not going to happen"
IDL>
```

# Statements

- Any IDL code, e.g. `print` statements, assignment - anything, really

```
IDL> if 1 then print,"The number 1 evaluates to True"
The number 1 evaluates to True
IDL> if 0 then print,"The number 0 evaluates to False"
IDL>
```

# Writing Code Blocks

- Text inside code blocks should always be indented (*must* be in python)
- Indentation should be 4 spaces, NOT a tab
  - `vim` lets you “map” the tab key to make 4 spaces instead of a tab
  - tabs are “variable length” = bad

# Conditions

- A “condition” is a statement that must evaluate to “true” or “false”
  - true = 1, false = 0
- Always should be *integer* 1 or 0
  - `‘0’` is actually True because it’s a string

# Multi-Line form

```
if 1 then begin
    x = 5
    print,"I still have ",x," frogs",format="(A,I1,A)"
endif
```

# Code Blocks

- All flow control statements allow `begin .. end` blocks, i.e.:

```
if true then begin
        print,"Hello World"
        x = 5
endif
```

```
if True:
    print "Hello"
```

# if then else

```
Ihave5frogs=0

if Ihave5frogs then print,"FROGS!" else print,"Dogs."
if Ihave5frogs then begin
    print,"I still have 5 frogs"
endif else begin
    print,"I do not have 5 frogs!"
endelse
```

The sinc function:

```
if (x eq 0) then y=1. else y=sin(x)/x
```

# if then else if

```
if (x eq 0) then begin
    print,"x is zero"
endif else if (x eq 1) then begin
    print,"x is one"
endif else begin
    print,"x is not zero or one"
endelse
```

# if elif else

```
if x==0:
    print "x is zero"
elif x==1:
    print "x is one"
else:
    print "x is not zero or one"
```

# conditionals and floats

- Floating point arithmetic has finite precision, i.e. it is not exact
- this can result in small errors that “break” conditional statements
- instead of doing
  `x == y` or `x eq y`
  use
  `(x-y) lt 1e-7` or `(x-y) < 1e-7`
  or some other small number

# floating point

- The reason to perform `(a-b) lt 1e-6` rather than `a eq b`:

```
IDL> print,0.1+0.2d eq 0.3
    0
IDL> print,0.1+0.2 eq 0.3
    1
IDL> print,0.2,0.2d,format="(2F30.20)"
        0.20000000298023223877          0.20000000000000001110
IDL> print,0.2 eq 0.2d
    0
```

However, this may not always work! See:
http://floating-point-gui.de/errors/comparison/

# Evaluate

```
print, 0.0 eq 0
```

A) 0

B) 0.0

C) 1

D) 1.0

E) None of the above

# Evaluate

```
print, 0.1 eq 0.1d
print, 0.5 eq 0.5d
```

A) 0,0

B) 0,1

C) 1,0

D) 1,1

E) I don’t know

# Evaluate

```
print, 0.1 eq 0.1d
print, 0.5 eq 0.5d
```

```
IDL> print,0.1,0.1d,format="(F25.20,F25.20)"
    0.10000000149011611938    0.10000000000000000555
IDL> print,0.5,0.5d,format="(F25.20,F25.20)"
    0.50000000000000000000    0.50000000000000000000
```

Compare to decimal:
1/3 can’t be expressed in decimals!
0.33, 0.33333, 0.3333333333 are all approximations
0.33 + 0.33 + 0.33 = 0.99, not 1, but it’s as close as you can get sometimes

# Floating Points

- If you ever try to look up floating point arithmetic, you'll likely land at this article: "What every computer scientist should know about floating-point arithmetic" http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- http://floating-point-gui.de/ has a nicer summary that is more comprehensible
- Short story: BE CAREFUL with floats

# The “ternary” operator

- #1 ? #2 : #3
- The ternary operator is represented by a ?: it means “return item #2 if item #1 is True, otherwise item #3”
- These are equivalent:

```
IDL> if (x eq 0) then y=1. else y=sin(x)/x
IDL> y = (x eq 0) ? 1. : sin(x)/x
```

(python version)

```
>>> y = 1. if x == 0 else sin(x)/x
```

# conditionals and ints

- You can use the “ternary operator” to determine if a statement is “True” or “False”

```
True = 1
False = 0
print, condition ? True : False
```

- For some peculiar reason, IDL evaluates EVEN integers as FALSE and ODD integers as TRUE.  So if you want “True”, ALWAYS use 1!

# conditionals and ints

```
True = 1
False = 0
print, condition ? True : False
```

```
IDL> for i=0,10 do print,i?1:0
       0
       1
       0
       1
       0
       1
       0
       1
       0
       1
       0
```

# Ternary Cont'd

- If you've ever used an "if" statement in Excel, it works just like the ternary:
  - `if(cell1>0,1,0)`
  - You can "nest" if statements:
    - if(cell1<0,1,if(cell2<0,1,0))
  - Same is true of ternaries

# Evaluate

```
In [2]: a = True
In [3]: b = False
In [4]: print 1 if b else 2 if a else c
```

A) 1

B) 2

C) 3

D) True

E) None of the Above

# Evaluate

```
x = 1
y = 0
print, x ? $
       (y ? 7 : 8) : $
       (y ? 2 : 1)
```

A) 1

B) 2

C) 7

D) 8

E) None of the Above

# Evaluate

```
x = 2
y = 3
print,x?y?2:3:y?4:5
```

A) 2

B) 3

C) 4

D) 5

E) None of the Above

# Evaluate

```
x = 2
y = 3
print,x?y?2:3:y?4:5
```

C) 4

LESSONS:

1) use 1 as True
2) use parentheses
3) use spacing

```
print (2 if y else 3) if x else (4 if y else 5)
```

# case statements

- Used primarily (exclusively?) for menus
- Could do menus like this:

```
read,userchoice
if userchoice eq 0 then begin
     ; do something
endif else if userchoice eq 1 then begin
     ; do something else
endif else if userchoice eq 2 then begin
     ; do something elser
endif else
     ; do something ad infinitum
endelse
```

# `case` statements

- Used primarily (exclusively?) for menus
- But this is somewhat simpler:

```
read,userchoice
case userchoice of
    0:; do something
    1:; do something else
    2:; do something elser
    else:; do something ad infinitum
endcase ; userchoice [to let you know
        ; which case you're ending]
```

# case statements

- Can use “blocks” again

```
read,userchoice
case userchoice of
    0: print,"One Thing"
    1: begin
        print,"One Thing"
    end
    2: begin
        print,"One Thing"
        print,"Two Thing"
    end
    else: print,"RedthingBluething"
endcase ; userchoice [to let you know
        ; which case you're ending]
```

# Switch Statement

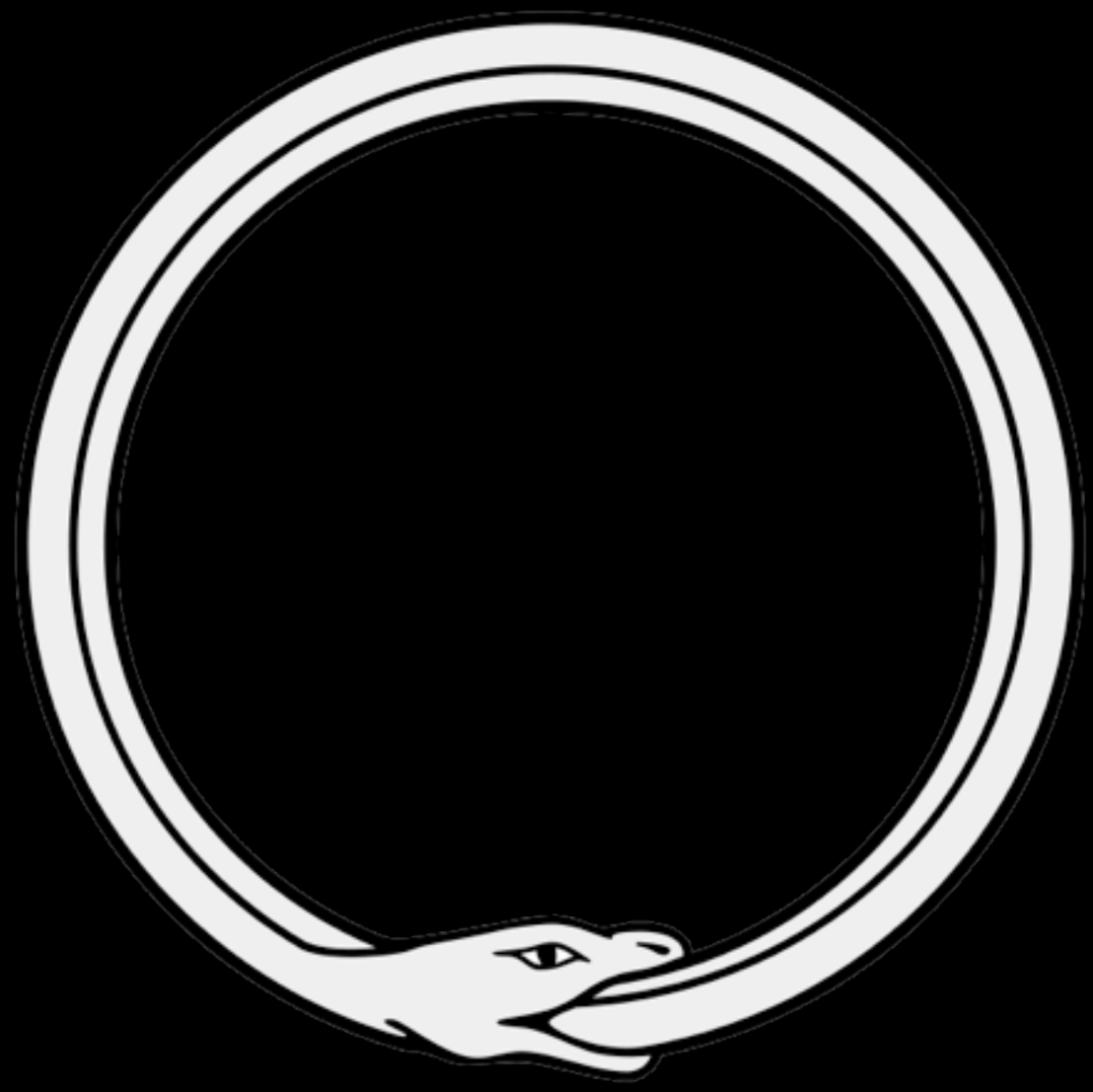
- Just like case, but EVERYTHING after the first matched case is executed
  - Good for having multiple options yield the same result (i.e., “Press 0,1, or 2 to continue”)
    - Use “`break`” statements to stop somewhere along the way

# Python doesn't have

- a `case` or `switch` statement
- instead, just use the `if/else` construction
- or, you can use a dictionary (but that's for later)

# Repetition...ition...ition

- Loops:
- Do the same thing multiple times (under different conditions)



# While

- Simplest loop
- Evaluates condition just like `if`, then goes again

```
while x le 10 begin
     x += 1
     y = y+x
endwhile ; x gt 10 now
```

```
if x le 10 then begin
     x += 1
     y = y+x
endif

if x le 10 then begin
     x += 1
     y = y+x
endif

if x le 10 then begin
     x += 1
     y = y+x
endif

...
```

# while

```
x = 0
while x <= 10:
    x += 1
print x
```

```
x = 0
while x < 10:
    x += 1
print x
```

A) 10
B) 9
C) 11
D) 10
E) None of the above

11
10
10
9

# Infinite Loops

- What if the "`until`" condition never happens? (e.g., "`while 1:`")
  - Your program will never finish
  - To stop it, you must press control-C to "break" the execution
  - Avoid infinite loops. Make sure there's a way for the loop to end.

# `for` loops

- Most useful and commonly used loop type
- Does some operation a fixed number of times
  - the `while` statements I showed also did this, but less efficiently

# `for` loops

- `for ii=start,end,stepsize do begin`
  `; do something`
  `endfor`
- These two are *almost* equivalent:

```
for ii=0,10 do begin
    print,"This is the ",ii,"th print statement"
endfor

ii = 0
while ii lt 10 do begin
    ii += 1
    print,"This is the ",ii,"th print statement"
endwhile
```

# for loops

```
for ii=start,finish do
```

- `ii` is initially set to the `start` value
- it is then incremented by 1 until

```
ii le finish
```

- After the for loop, `ii = finish + 1`

```
for ii=start,finish do print,ii
```

prints `0,1,...,10`

# `for` loops & type

```
for ii=start,finish,step do print,ii
```

- `ii` will ALWAYS have the type of `start`, even if `step` or `finish` are a different type
- This means all the caveats for data types (overflows especially) hold within for loops

# weirdness in loop finish

```
myarray = fltarr(10)
; evaluates n_elements(myarray) each time
while ii lt n_elements(myarray) do begin
    print,ii," is less than ",10
endwhile
```

```
; evaluates n_elements(myarray) only once
for ii=0,n_elements(myarray)-1 do begin
    print,ii," is less than ",10
endfor
```

- If the loop changes the length of an array, it can mess up

# Python for loops

- Very different from IDL for loops!
- python `for` loops “loop over” a list

```
a = [1,"two",3.0]
for x in a:
    print x

1
two
3.0
```

# What if you want #s?

```
for ii in xrange(2):
    print "This is the",ii,"th print statement"
print ii
```

```
This is the 0 th print statement
This is the 1 th print statement
1
```

```
for ii=0,1 do begin
    print,"This is the ",ii,"th print statement"
endfor
print,ii
```

# Loops are Slow

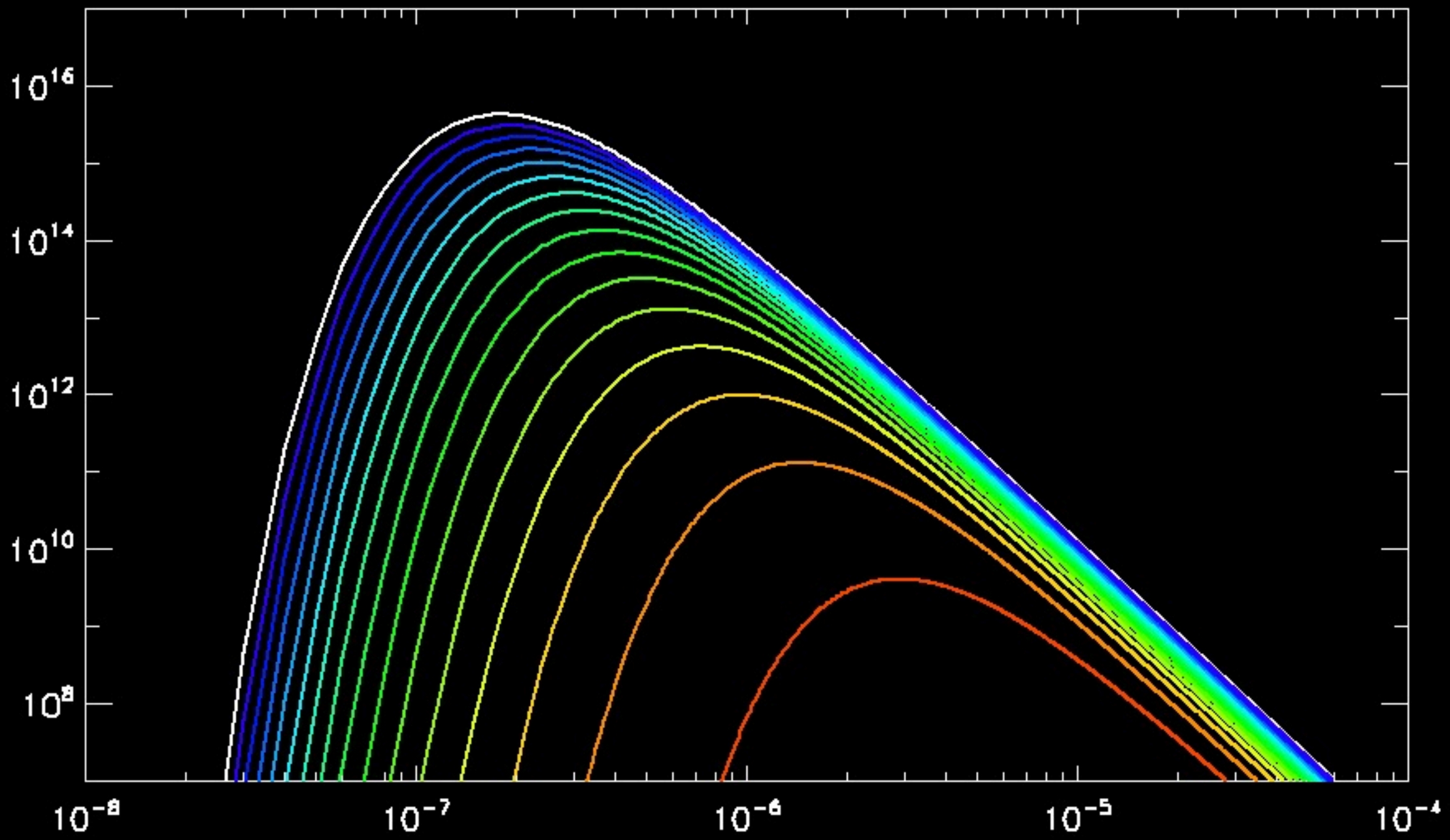
- Compared to array operations, loops in IDL (and python) are slow
- In C, C++, and FORTRAN, they are FAST
- Use array operations whenever possible, for loops only when necessary

# Practical Looping

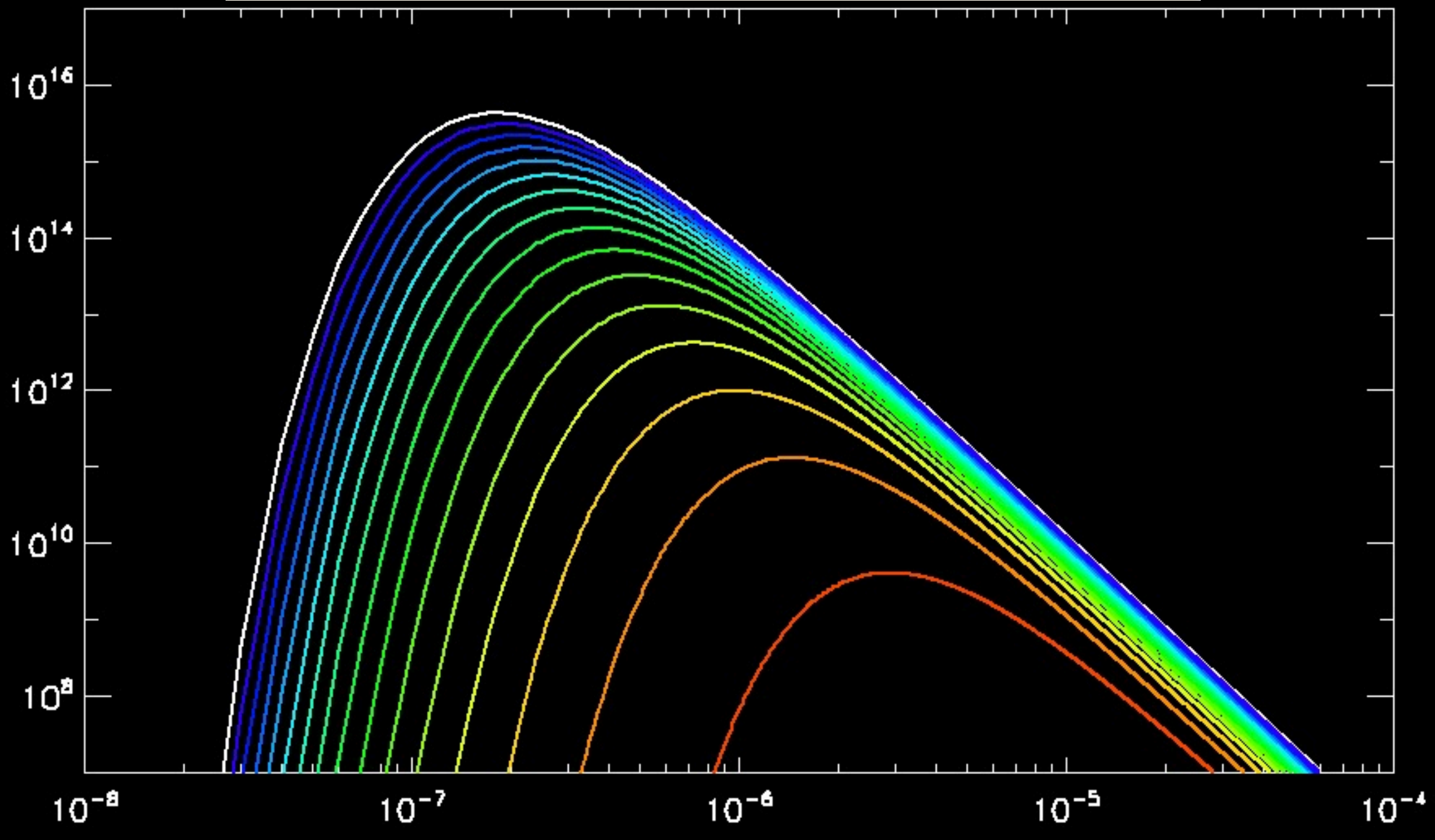
- Plot a bunch of Planck functions - color-coded!

```
lambda = findgen(10000)/9999. * 1000000 + 100
lambda = lambda * 1e-10 ; lambda in meters
plot,lambda,planck(lambda,16000),yrange=[1e7,1e17],$
    /ys,/xs,/ylog,/xlog,thick=2,charsize=2,charthick=2
for temperature=1000,15000,1000 do begin
    color_convert,temperature/15000. * 250.,1,1,$
        r,g,b,/hsv_rgb
    oplot,lambda,planck(lambda,temperature),$
        color=r+g*256L+b*256L^2L,thick=2
endfor
```

# Plancking




h=6.626x10-34 J·s


10^16
10^14
10^12
10^10
10^8
10^-8
10^-7
10^-6
10^-5
10^-4

# `break`

- No, not breaktime yet. `break` time.
- the `break` statement anywhere in a loop will stop the loop, preventing the next iteration
- the only time I use it is for “iterating to convergence”, which means you have some criterion that says “Looped enough times” (but you still want to limit the total # of loops)

(same in IDL and python)

```
 1 ; Taylor-approximate the sin(x)
 2 function taylor_sine,xx,maximum_iters=maximum_iters
 3
 4     if ~keyword_set(maximum_iters) then maximum_iters=50
 5
 6     y = 0d ; initialize the approximation
 7
 8     ; determine peak precision
 9     x_is_double = size(xx,/type) eq 5
10     machine_precision = machar(double=x_is_double)
11     epsilon = machine_precision.eps
12
13     ; 51 is arbitrarily selected
14     ; sin ~ x - x^3/3! + x^5/5! - ...
15     for ii=1,maximum_iters*2+1,2 do begin
16         ; separately determine the added term
17         ; so we know when to quit
18         new_term =  xx^ii / factorial(ii)
19         if new_term lt epsilon then break
20
21         sign = (ii+1) mod 4 eq 0 ? -1 : 1
22         y += sign * new_term
23
24         ; a debug statement
25         print,ii,sign,new_term
26     endfor
27     return,y
28 end
```

# GOTO statements

- For all practical purposes, these do not exist
- Should you ever encounter one, `goto` the internet and find out how hated they are



So then I typed GOTO 500 -and here I am!