

**Python Programming**

**Computational Physics**

**Python Programming**

# Outline

- Useful Programming Tools
  - Conditional Execution
  - Loops
  - Input/Output
- Python Scripts, Modules, and Packages

# Conditional Execution

- Sometimes we only want to execute a segment of our program when a certain condition occurs.

- Conditional Execution is done with *if* statements.

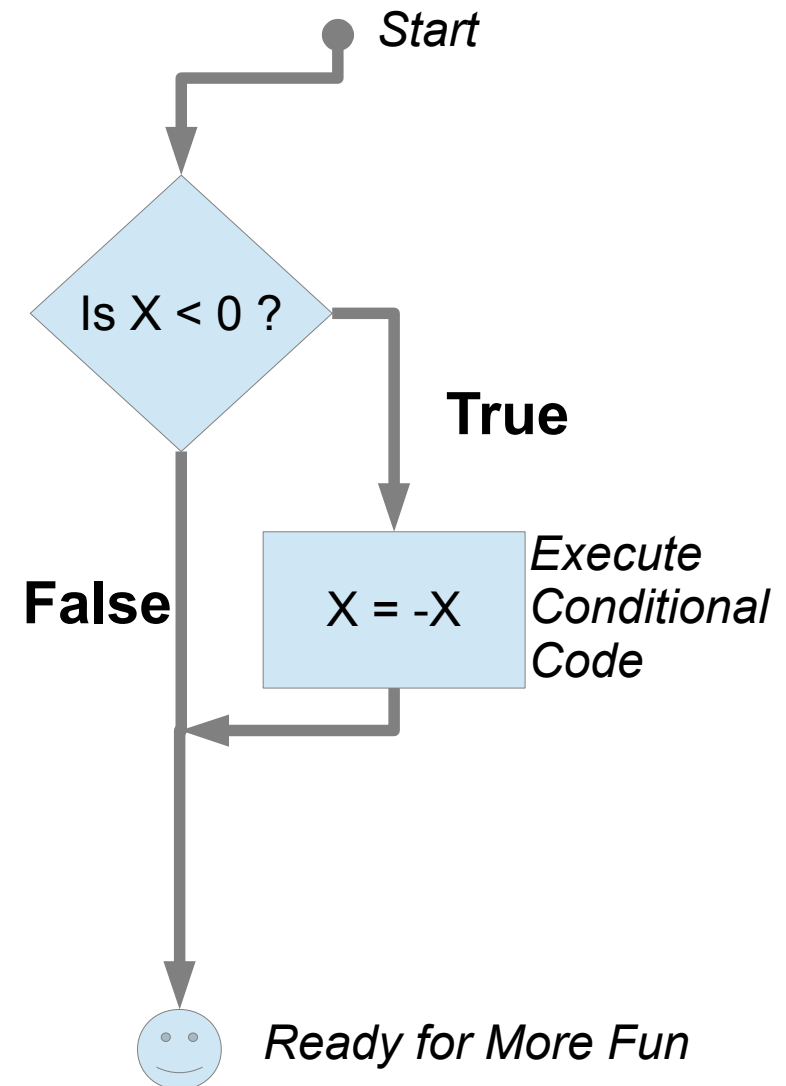
if some\_logical\_condition :

    statements\_to\_execute\_if\_true

    statements\_are\_indented

if ends with end of indentation ...

no special symbols or words



# *if...elif...else* statement example

```
# enter a value to test

X = input('Enter Value of X: ')

# now we do the test with if statement
if X < 0:
    print 'X is less than 0!'
elif X == 0:
    print 'X is zero!'
elif X == 1:
    print 'X is one!'
else:
    print 'X = ',X,' is not a special case'

# another example of if...else
if X < 0 and X > -2:
    print 'X < 0 but X > -2 !'
else:
    print 'X is not between -2 and 0.'
```

If X is less than 0  
we do this

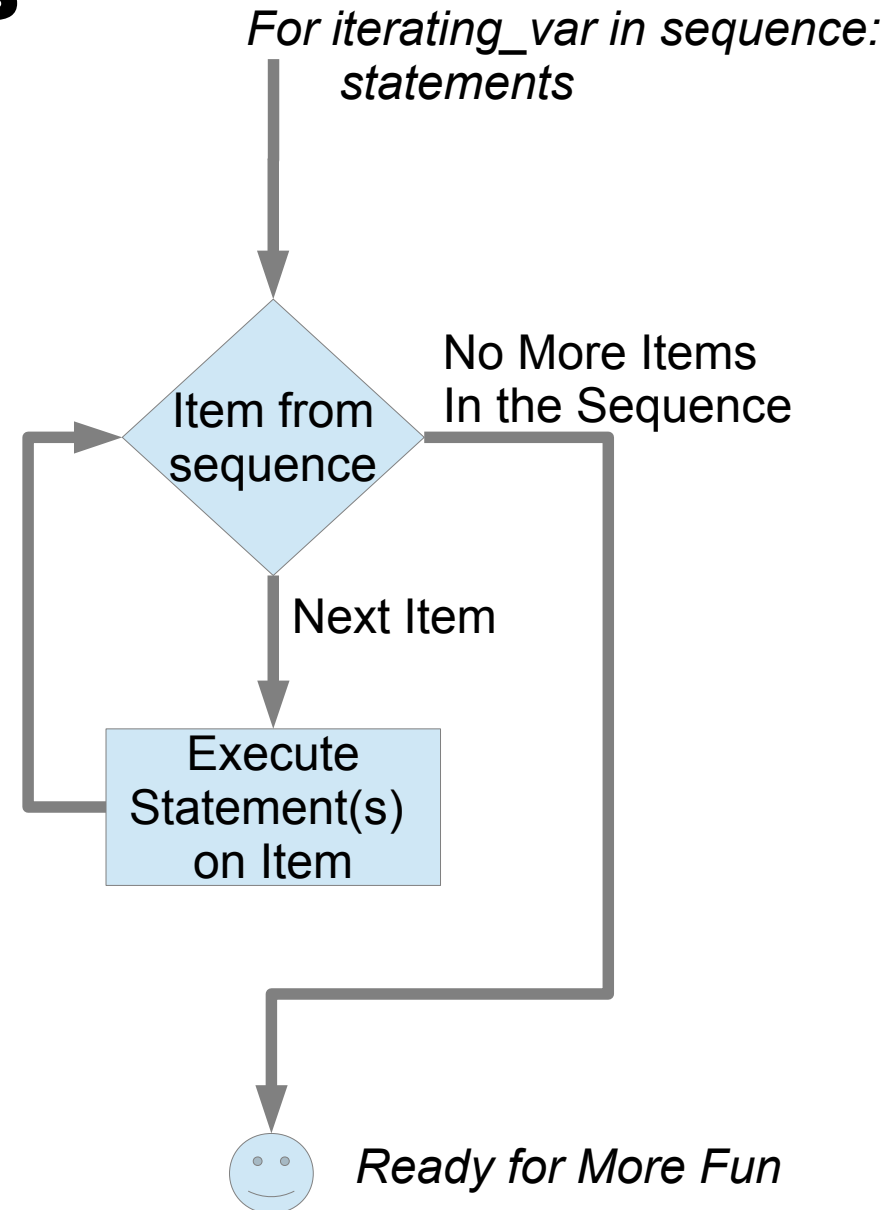
Otherwise, if X is  
equal  
To 0 we do this.  
== means "is equal to"

Otherwise, if X is  
equal  
To 1 we do this.

Otherwise, if X doesn't  
Match any of above,  
Condition can be any  
Logical Statement


# Loops

- Loops are a special type of conditional execution.
- Loops iterate over a sequence of items.
- In python, the items can be any items in a list.
- We will often iterate through the indices that point to items in NumPy arrays.



# *for* loop example

```
# enter an array for example  
t = np.linspace(0.,1.,11)  
  
# use for loop to iterate through t array  
  
for x in t:  
    print x  
  
# loop on INDEX to the t array  
  
for i in range(len(t)):  
    print 'index = ',i, ' Value = ',t[i]
```



In this example, each value in the t array is considered sequentially in the loop.

Output is:

0.0  
0.1  
0.2  
0.3  
0.4  
0.5  
0.6  
0.7  
0.8  
0.9  
1.0

# for loop example

```
# enter an array for example

t = np.linspace(0.,1.,11)

# use for loop to iterate through array

for x in t:
    print x

# loop on INDEX to the t array

for i in range(len(t)):
    print 'index = ',i, ' Value = ',t[i]
```

In this example, we consider each possible index in the t array.

*len(t)* gives the number of elements.

In this case *len(t)* = 11.

*range(t)* makes a list starting at 0 that has *len(t)* elements. In this case:

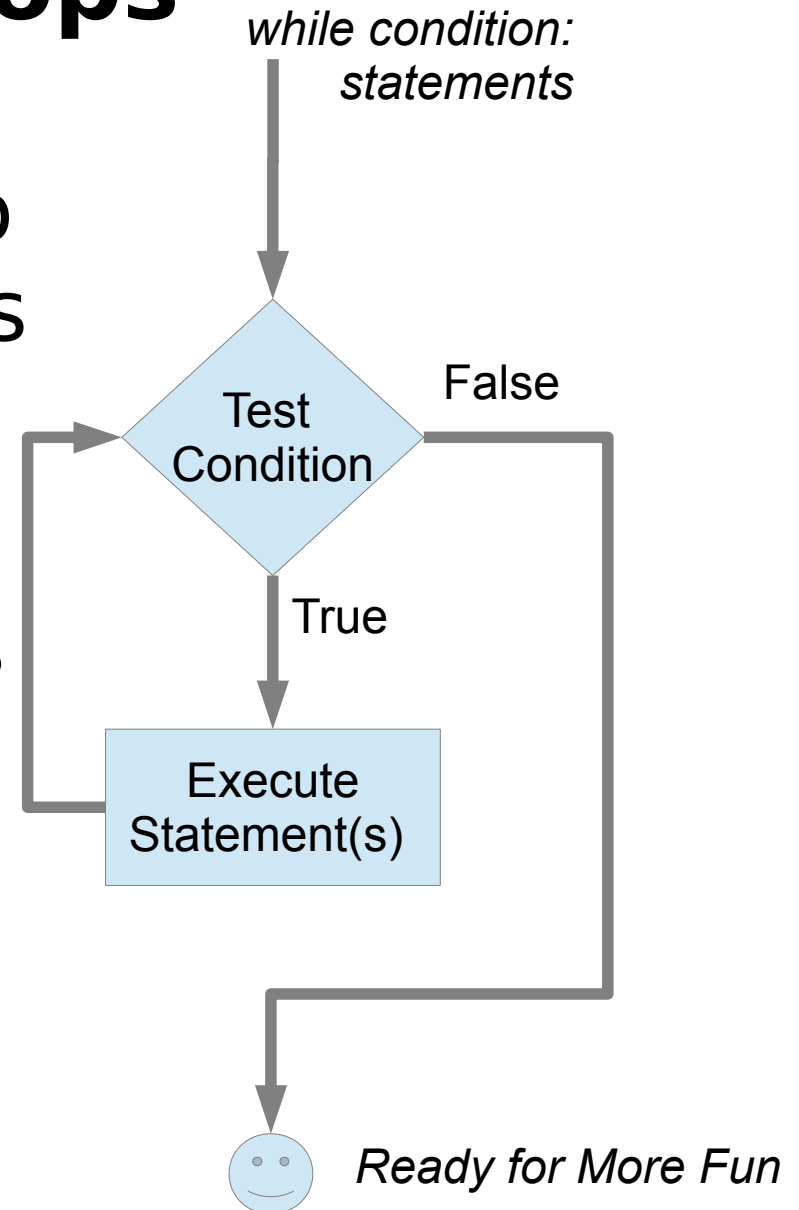
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Output is:

index = 0 Value = 0.0  
index = 1 Value = 0.1  
index = 2 Value = 0.2  
index = 3 Value = 0.3  
index = 4 Value = 0.4  
index = 5 Value = 0.5  
index = 6 Value = 0.6  
index = 7 Value = 0.7  
index = 8 Value = 0.8  
index = 9 Value = 0.9  
index = 10 Value = 1.0

# *while* loops

- While loops continue to execute the statements in the loop as long as a condition is True.
- Note that if statements do not change the condition, the loop will continue forever.





# *while* example

```
# here is our starting point for calculation

t = 0      # value we will operate on
dt = 0.1   # increment to be added to t

# use while to consider values of t
# as long as t is less than or equal to 1.0

while t <= 1.0:
    print t
    t = t + dt  # increment t

print 'All Done!'
```

Initialize the variables for the loop.

As long as t is less than or equal to 1.0, we continue around the loop.

Add dt to t each time around the loop

Output (value of t each time around loop):

0  
0.1  
0.2  
0.3  
0.4  
0.5  
0.6  
0.7  
0.8  
0.9  
1.0  
All Done!

# Input and Output

- Input

- We have already seen “input” in action.

*X = input('Set the value of X: ')*

- NumPy provides a simple way to read in a 2D array of values: `np.loadtxt('filename')`

*A = np.loadtxt('mydata.dat')*

mydata.dat is a text file with a 2D array arranged in rows and columns. A will be a NumPy array with the data arranged in rows and columns.

# Input and Output

- Input (continued)
  - `np.loadtxt` can also read a *csv* text file, such as those made by Excel.
  - In a *csv* file, individual values are separated by a “delimiter” ... often a `<TAB>`, semicolon (`;`) or comma (`,`)  
*A = np.loadtxt('mydata.csv', delimiter=';')*
  - Some rules about `loadtxt`:
    - 2D only
    - All data of same type (as in NumPy)
    - Number of items in each row must be the same.

# Input and Output

- Output

- We have already encountered *print*  
*print 'Hello World!'*

*print X # just prints the variable X*

- NumPy has a `savetxt` method which will write out a 2D array to a file.

*np.savetxt('o.dat',A)*

will write the array `A` to a file named `'o.dat'`

*np.savetxt('o.csv',A,delimiter=';')*

will write `A` as a csv file with delimiter `';`

# Input and Output

- NumPy `loadtxt()` and `savetxt()` are very useful for quickly loading and saving simple array data for our programs.
- There is an equivalent `load()` and `save()` that deal with NumPy arrays in binary form.
- Sometimes we need to read and write data according to some more specific format. Maybe we want to mix types....
- We can do this by reading and writing from files.

# File I/O

**SEE: [docs.python.org/2/tutorial/inputoutput.html](https://docs.python.org/2/tutorial/inputoutput.html)**

- Steps:
  - Open the file with `open()` method
  - Read or Write to the file with `read()` or `write()` method
  - Close the file with `close()` method
- Open the file “f.dat”
  - For writing  
*`F = open('f.dat','w')`*
  - For reading  
*`F = open('f.dat','r')`*

# Reading Data from File

*Z = F.read()*

Reads ALL the data as a single string into Z no matter how big. (Tough to process this.)

*Z = F.readline()*

Reads a line from the file. Z will be a string. Subsequent calls read subsequent lines.

*Z = F.readlines()*

Reads all lines. Z is an array of strings, one element for each line in the file.

In all cases your data is a string. For numeric data you must convert the string to a number with *eval()*.

# Read Example

f.dat:

```
1.1  
2.1  
3.2  
4.555  
6
```

```
# open f.dat for reading  
F = open('f.dat', 'r')  
  
# read all the lines  
Z = F.readlines()  
  
# use print to show contents  
for x in Z:  
    print eval(x)  
  
F.close()
```

Open the file

Read all the lines into an array named Z

Print each element of Z after evaluating it

Close the file

Output:

```
1.1  
2.1  
3.2  
4.555  
6
```



# Writing Data to File

*F.write(s)*

Writes the string *s* to open file *F*.

Note that if you wish to write a number, you must convert it to a string.

Let *a* be a float and *b* be an int:

Old Way: *F.write( '%5.3f %4d' % (a,b) )*

New Way:

*F.write( '{0:5.3f} {1:4d}'.format(a,b) )*

**5.3f** => write float in 5 spaces with 3 digits after the decimal point.

**4d** => write int in 4 spaces

# Write Example

```
# make some data for writing
A = np.array([1.1,2.1,3.14,4.55])

# open f.dat for writing
F = open('f.dat', 'w')

# write one value at a time
for x in A:
    F.write('{0:6.3f}\n'.format(x))

# close the file
F.close()
```

Open the file f.dat for writing

Write one value at a time.

The `\n` means end the line after each value is written.

The values will be written in 6 spaces with three values after the decimal point.

Close the file

Output written in f.dat:

```
1.100
2.100
3.140
4.550
```

# Scripts, Modules, Packages

- We write “programs” in python using text files. We may distinguish:
  - **Scripts**: a file with a set of python statements that we wish to run. It's the same as typing them into *ipython*.
  - **Modules**: a file that defines functions, classes, and/or other variables that we want to use with other pieces of python code.
  - **Packages**: a package is a set of Modules which are related and maintained together in the same directory.

# Why?

- We use Modules to try to stay organized. Functions and classes are separate from the scripts that call them.
  - They can be used by MANY different scripts
  - We don't risk changing them as we edit new scripts.
- Packages keep related Modules together.
  - Keep individual modules from getting too big to be easily maintained.
  - Easy to gather the whole group together for others to use.

# Example with Modules

```
"""my module example
"""

import math

def sind(arg):
    """computes sin with degree argument
    """
    theta = arg/180.*math.pi
    return( math.sin(theta) )
```

This is the module, which is in file: my\_module.py

It just defines the sind() function.

```
import my_module

print my_module.sind(90.)
```

Here is a script that imports the Module from my\_module.py

# ***import and reload***

- Once you import a module, all the new functions are defined.
- Suppose you make a change in the module and try to import it again....  
*Python sees that there already is a defined function with that name and does not over write it with the new one!*
- To force python to use the newer version use:

*reload(module\_name)*

# Reload example

```
In [144]: import my_module
```

```
In [145]: run my_script.py
```

```
In [146]: reload(my_module)
```

```
In [147]: run my_script.py
```

OOPS, When I run the script I discover there is a problem with one of the functions in my\_module

So I fix it. Now must reload my\_module

Next time, my\_script gets the corrected Program and runs correctly.