TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

# Programming with Python

Southern University and A&M College

April 1st


**PRESENTED BY:**

Antonio Gomez Iglesias

Texas Advanced Computing Center

HPC Group

agomez@tacc.utexas.edu

http://hpcuniversity.org/trainingMaterials/237/

# XSEDE

- **Single virtual system** that researchers can use to interactively share computing resources, data and expertise.
- **People around the world** use these resources and services - things like supercomputers, collections of data and new tools - to improve our planet.
- **Access to resources** that include HPC machines, High Throughput Computing (HTC) machines, visualization, data storage, test-beds, & services
- **Science Gateways** enable entire communities of users associated with a common discipline to use national resources through a common interface that is configured for optimal use.
- **Extended Collaborative Support Service (ECSS)** through which researchers can request to be paired with expert staff members for an extended period (weeks up to a year).

**https://www.xsede.org**

# XSEDE Resources

## XSEDE Resource User Guides

Below are links to each resource's user guide. Each guide provides information and instructions on system access, computing environment and running jobs specific to that resource. Resources are listed alphabetically within each resource type: High Performance Computing, High Throughput Computing, Visualization, Storage systems, Special Purpose systems and Software

XSEDE is committed to providing quality, useful documentation to its users. Please feel free to leave your suggestions and comments at the bottom of each user guide.

### High Performance Computing

Bridges (PSC)

Bridges GPU (PSC) *Coming soon...*

Comet (SDSC)

Comet GPU (SDSC) *Coming soon...*

Gordon (SDSC) *decommission April, 2017*

Gordon ION (SDSC) *decommission April, 2017*

Jetstream (IU/TACC)

Stampede (TACC) *decommission Summer 2017*

Stampede 2 (TACC) *Coming soon...*

SuperMIC (LSU)

Wrangler Analytics (TACC)

XStream (Stanford)

### High Throughput Computing

Open Science Grid

### Scientific Visualization

Maverick (TACC)

### Storage Systems

Bridges Pylon (PSC)

Data Oasis (SDSC)

Ranch (TACC)

Wrangler Storage (TACC)

**https://portal.xsede.org/user-guides**

# Maverick

# Ready?

- Did you sign in?
- Do you have a username/password?
- Can you connect to the Internet?

http://hpcuniversity.org/trainingMaterials/237/

# Getting Started

- Go to TACC vis. portal: **https://vis.tacc.utexas.edu/**
- Select "*TACC User Portal User*"
- Use your username/password
- Go to the *Jobs* tab
- On "*Session type*", click on **iPython/Jupyter Notebook**
- Click on **Set iPython Password** and choose something you like
- Click on **iPython/Jupyter Notebook** (again)
- Click on **Start Job**
- Click on **Open in Browser**
- Create a new **Python 2 Notebook** ("**New**" button)

# Part I

- My first program
- Data types
- Variables
- Arithmetic operations
- Relational operations

# My first program

- In your Jupyter Notebook, write this program:

```
"""
This is my first program
"""
print "Hello from my first program"
```

- Shift+Enter will run this code
- You can also click on the "Run" button

# Our first program

```
"""
This is my first program
"""
```

The text between “”” is a comment ➔ it helps you to know what the code does, but it’s not executed. You can also use #, and all of text to the right of the # symbol will be a comment

```
print “Hello from my first program”
```

This is a Python instruction (**print**). This is the code that will be executed

# What happens when you run your program?

- There is a Python interpreter
- The interpreter understand Python code
- Converts this code to something that the computer understands
- Any "computer" with a Python interpreter installed will be able to run your program!
- Jupyter calls this interpreter for you
  - If you are not using Jupyter, from a Console, simply run:

```
"$ python your_program.py"
```

# Let's print a number

```
print "This is my first program!"
print 5
print 1+1
```

# Using variables

- You will need to store data into variables
- You can use those variables later on
- You can perform operations with those variables
- Variables are declared with a **name**, followed by '**=**' and a **value**
  - An integer, string,…
- When declaring a variable, **capitalization** is important:
  - 'A' <> 'a'

# Using variables

```
five = 5
one = 1
print five
print one + one
message = "This is a string"
print message
```

# Scope of variables

- Not all variables are accessible from all parts of the program (we'll see it with functions)
- Variables in the main body (what you have so far) are global: accessible everywhere in the code
- Variables in one indented block are accessible in that block and dependent blocks (blocks inside the block): we are going to see this (**indentation**)

# Data types

```
integer_variable = 100
floating_point_variable = 100.0
string_variable = "Name"
```

# Checking/changing types

- Variables have a type
- Check the type of a variable using the **`type()`** function:

```
print type(integer_variable)
```

- It is also possible to change the type of some basic types:
  - **`str(int/float)`**: converts an integer/float to a string
  - **`int(str)`**: converts a string to an integer
  - **`float(str)`**: converts a string to a float
- Be careful: you can only convert data that actually makes sense to be transformed

# Arithmetic operations

| Symbol | Task Performed | Example |
|---|---|---|
| + | Addition | 1 + 1 = 2 |
| - | Subtraction | 5 – 3 = 2 |
| / | Division | 4 / 2 = 2 |
| % | Modulo | 5 % 2 = 1 |
| * | Multiplication | 5 * 2 = 10 |
| // | Floor division | 5 // 2 = 2 |
| ** | To the power of | 2 ** 3 = 8 |

# Arithmetic operations

- What happens when you try 5 / 2 ?
- And 5.0 / 2 ?
- Can you do string + string?
- And string + number?
- String * number?
- What's the result of  3+5*2 ?

# Order of operators

1. () [] {}
2. **
3. *, /, % , //
4. +, -
5. <, >, <=, >=
6. ==, !=

So, if 3+5*2 = 13. How can you get 16 with the same sequence of numbers? (You can add more operators)

# Order of operators

- 3+5*2 = 13
  - 5*2 = 10
  - 3+10 = 13
- (3+5)*2 = 16
  - 3+5 = 8
  - 8*2 = 16

# Relational operations

| Symbol | Task Performed | Example |
|---|---|---|
| == | True if equal | 1 == 1 |
| != | True if not equal | 1 != 2 |
| < | Less than | 1 < 2 |
| > | Greater than | 2 > 1 |
| <= | Less than or equal to | 2 <= 2 |
| >= | Greater than or equal to | 3 >= 2 |

# Part 2

- Input and output
- Control flow:
  - Loops
  - Conditions
- More data types
- Functions

# Reading something from the keyboard

```
var = input("Write a number: ")
print "The number that you wrote was : ", var
```

Try to read a string

# Writing on the screen

- We have already seen `print`
- `print` has some tricks to put together strings and numbers:

```
print "The number that you wrote was : ", var
print "The number that you wrote was : %d" % var
```

| Symbol | Meaning |
|---|---|
| %d | integer |
| %f | floating point |
| %s | string |

# Reading and writing files

- Files are permanently stored on disk
- You can create data, store it on a file, and reuse it later on
- When working with files you have to:
  1. **Open a file**
  2. **Read/write**
  3. **Close the file**

# Writing to a file

```
my_file = open("output_file.txt",'w')
vars = "This is a string"
my_file.write(vars)
var1 = 10
my_file.write(str(var1))
var2 = 20.0
my_file.write(str(var2))
my_file.close()
```

**What's happening with the output?**

# Some special characters

| Symbol | Meaning |
|---|---|
| \n | New line |
| \t | Insert tab |

- Fix the previous program to write each variable in one line.

```
my_file = open("output_file.txt",'w')
vars = "This is a string"
my_file.write(vars)
my_file.write("\n")
var1 = 10
my_file.write(str(var1))
my_file.write("\n")
var2 = 20.0
my_file.write(str(var2))
my_file.write("\n")
my_file.close()
```

# Appending to a file / Rewriting a file / Open as read only

- When opening a file, you need to decide "how" you want to open it:
  - Just read?
  - Are you going to write to the file?
  - If the file already exists, what do you want to do with it?

| Character | Meaning |
|---|---|
| r | Read only file (default) |
| w | Write mode: file will be erased if it exists |
| a | Write mode: new content will be appended to the end of the file |

# Read the file you created before

```
my_file = open("output_file.txt",'r')
content = my_file.read()
print content
my_file.close()
```

# Read the file you created before (line by line)

```
my_file = open("output_file.txt",'r')
vars = my_file.readline()
var1 = my_file.readline()
var2 = my_file.readline()
print "String: ", vars
print "Integer: ", var1
print "Float: ", var2
my_file.close()
```

# Control flow

- So far we have been writing instruction after instruction
- Every instruction is executed in order
- What happens if we want to have instructions that are only executed if a given condition is true?

# if/else/elif

- The if/else construction allows you to define conditions in your program
- The syntax is as follows:

```
if conditionA:      #Remember to put the colon
    statementA          #Remember indentation
elif conditionB:    #Colon here too
    statementB
else:               #And another colon
    statementD
this line will be always executed (after the if/else)
```

# if/else

- There can be many instructions in each part of the if/else
- Remember the scope of the variables: if a variable is defined inside of if/else, it won't be available outside the if/else
- **Remember to keep the 4 spaces**: that's the delimiter of a block. All the lines with the same indentation belong to the same block

# Exercise

- Read a number from the keyboard
- If the number is greater or equal than 10, show "This number is greater or equal than 10"
- Else, show the message "This number is smaller than 10"

**5 minutes for this exercise**

# Solution

```
number = input("Write a number: ")
if number >= 10:
    print "This number is greater or equal than 10"
else:
    print "This number is smaller than 10"
print "Bye!"
```

# Nested if

- You can nest different blocks:

```
if condition1:
    statement1
    if condition2:
        statement2
    else:
        if condition3:
            statement3 # when is this statement executed?
else:  # which 'if' does this 'else' belong to?
    statement4  # when is this statement executed?
```

# For loops

- **For loops** allow you to iterate over a sequence

```
my_file = open("output_file.txt",'r')
vars = my_file.readline()
var1 = my_file.readline()
var2 = my_file.readline()
for i in vars, var1, var2:
    print i
my_file.close()
```

# range()

- This function creates a list containing arithmetic progression

```
range(5)
[0, 1, 2, 3, 4]
```

- This is very useful in loops

```
for i in range(5):
    print i
```

# Exercise

- Part 1:
  - Read a number from the keyboard
  - Show the list of all positive integers from 0 to that number
- Part 2:
  - Only show **even** numbers

**5 minutes for this exercise**

# While loops

- Another iterative construct
- Gives you a different type of control

```
while (condition is True):
    do something
```

- Example: simulate a for loop with a while loop

```
idx = 0          # Initialization
while (idx < 10):    # Condition
    do_something
    idx = idx + 1    # Why do we need this?
```

# Exercise

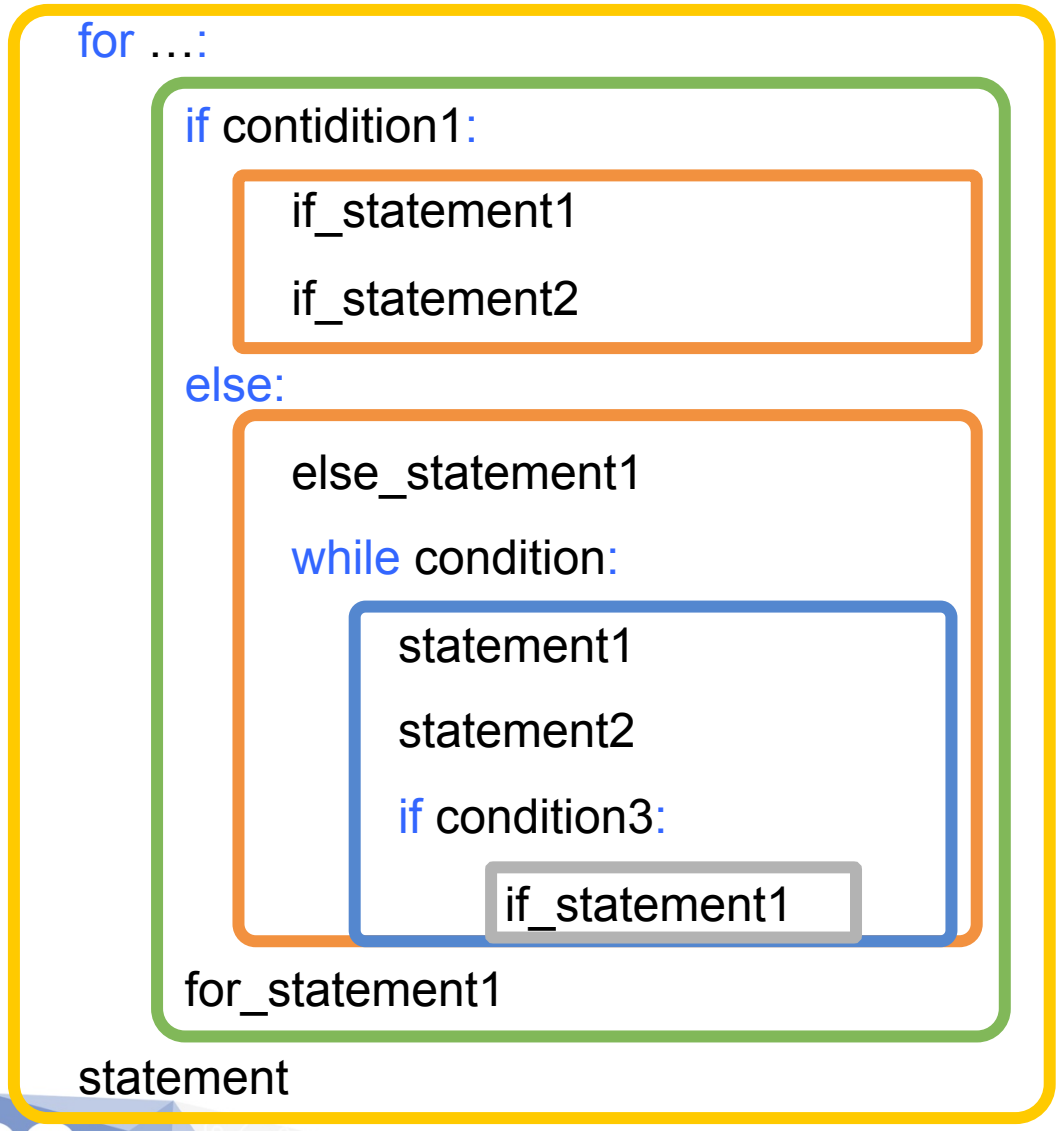
- Use a while loop to read a number from the keyboard
- Stop when the number read is one you previously decided (i.e. 10)

**3 minutes for this exercise (3!!!)**

# Solution

```
var = 0
while (var != 10):
    var = input("Write a number: ")
```

# Nesting


for …:
if contidition1:
if_statement1
if_statement2
else:
else_statement1
while condition:
statement1
statement2
if condition3:
if_statement1
for_statement1
statement

# lists

- A list is a sequence, where each element is assigned a position (index)
- You can access each position using []. **First position is 0**
- Elements in the list can be of different type

```
mylist1 = [“first item”, “second item”]
mylist2 = [1, 2, 3, 4]
mylist3 = [“first”, “second”, 3]
print mylist1[0], mylist1[1]
print mylist2[0]
print mylist3
print mylist3[0], mylist3[1], mylist3[2]
print mylist2[0] + mylist3[2]
```

# lists

- It's possible to use slicing:

```
print mylist3[0:3]
print mylist3
```

- To change the value of an element in a list, simply assign it a new value:

```
mylist3[0] = 10
print mylist3
```

# lists

- There's a function that returns the number of elements in a list

```
len(mylist2)
```

- Check if a value exists in a list:

```
1 in mylist2
```

- Delete an element

```
len(mylist2)
del mylist2[0]
print mylist2
```

- Iterate over the elements of a list:

```
for x in mylist2:
    print x
```

# Exercise

- Create a list of 10 elements
- Find the maximum number in the list

**5 minutes for this exercise**

# Exercise

```
mylist = [1, 99, 51, 43, 112, 7, 64, 11, 16, 81]
mymax = 0
for i in mylist:
    if (mymax < i):
        mymax = i
print mymax
```

# lists

- There are more functions

```
max(mylist), min(mylist)
```

- It's possible to add new elements to a list:

```
my_list.append(new_item)
```

- We know how to find if an element exists, but there's a way to return the position of that element:

```
my_list.index(item)
```

- Or how many times a given item appears in the list:

```
my_list.count(item)
```

# break/continue

- `break`: terminates a loop
- `continue`: skip the remainder of the loop and return to the beginning of the loop

```
for x in my_list:
    if x == something:
        break
statement
```

```
for x in my_list:
   if x == something:
       continue
   statement
```

# Functions

- The code can (will) get too complicated
- Group the same functionality in a function:
  - Reusable code
  - Provides modularity to your code
  - Easier to develop, make changes, …
  - Ideally a function does one thing
- You can use any of the control flow options that we already know
- Functions are executed when they are called from the code currently in execution
- They need to be declared **before** they are called

# Functions

- To declare a function:

```
def name_of_the_function ( [arguments] ):
```

- The code inside the function is indented
- Functions normally end with a return statement:

```
    return [expression]
```

- The variables that are declared within the function, are not accessible from outside the function (scope)

# Functions

```
def function_name(): #The function is only executed when called
    statement1
    statement2
    control_flow_statement:
        stament
    return

statement_before_the_function
function_name() # we call the function here
statement_after_the_function
```

# Declaring our first function

```
def my_first_function():
    print "Hello from the function"
    return

print "This is before the function"
my_first_function()
print "This is after the function"
```

# Passing values to functions

- Define a list of arguments separated by commas

```
def name_of_the_function (arg1, arg2, arg3):
    print arg1, arg2, arg3
```

- When calling the function, your variables don't need to be called arg1, arg2, arg3:

```
name_of_the_function(var1, varY, VaRZ)
```

# Passing a string and an integer

```
def function_with_args(my_str, my_int):
    print "This is the string: ", my_str
    print "This is the integer: ", my_int
    return
var_str = "Hi"
var_int = 5
function_with_args(var_str, var_int)
```

# Returning a value from a function

- Use the return statement to return a variable or set of variables to the caller
- Assign the function to a variable or set of variables on the caller

```
def my_function():
    …
    return a,b,c
mya, myb, myc = my_function()
```

# Exercise

- Read a number from the keyboard
- Pass the number to a function
- In the function:
  - If the number is greater than 10, show "This number is greater than 10"
  - Else, show the message "This number is smaller than 10"

# Special functions

- What code is executed when the program starts?
  - The interpreter will execute all the code that it finds in the file
- It is sometimes useful (modules) to define a "**main**" function, the entry point to your program, and put all the statements inside **main**
- Everything inside **main**, is local to **main** (remember the scope of variables)

```
if __name__ == "__main__":
    first_statement_of_your_program
```

# Remember the scope

```
my_str = "hi"

def example():
    a = 2
    print my_str                              #can I do this? why?
    print "This is the function: ", a    #which a is this one?
    return

if __name__ == "__main__":
    a = 1
    example()
    print "This is main: ", a  #what is this going to print? 1? 2?
```

# Part 3

- Modules
- Using your own modules

# Modules

- So far we have seen some functions:
  - `len()`, `range()`, `max()`, `min()`,…
- Python includes many external libraries or modules that provide additional functionality
  - Mathematical functions
  - System interaction
  - Plotting
- You can also define your own modules
  - Helpful to group a lot of functionality together and reuse it

# import

- To use a module, first you have to tell Python that you want to use it

```
import math
import string
```

- You now have access to the functionality provided by math by using "**math.**" plus the name of the function that you want

```
math.floor(math.pi)
math.sqrt(9)
math.pow(3,2)
```

- https://docs.python.org/2/library/math.html
- https://docs.python.org/2/library/string.html

# from X import Y

- When you import a module, you still need to put the name of the **`module + "." + function name`**
- If you know that you only need the "floor" function from math, you can simply import like:

```
from math import floor
```

- Now you can call **`floor(3.14)`**, but you can't call **`floor(math.pi)`**
- You can also import everything from a module (this might be dangerous)

```
from math import *
```

# Creating your own module

- Create a .py file (`my_module.py`):
  - New -> Tex file (File - Rename)
- Define the functions that you need:

```
def function1(…):
def function2(…):
```

- From your notebook, import the module:

```
import my_module
```

- You should now be able to call:

```
my_module.function1(…)
```

# Part 4

- Plotting

# matplotlib

- **matplotlib** is the most popular plotting library in Python

```
import matplotlib.pyplot as plt
```

- You need to tell matplotlib what is that you want to plot (the data):

```
plt.plot(x, y, [style])
```

- **plot** takes at least one parameter
- **plot** can be used to plot several different series

# matplotlib

- You normally plot lists:

```
#We need the next line for Jupyter
%matplotlib inline
import matplotlib.pyplot as plt
myx = [1,2,3,4]
myy = [1,2,3,4]
plt.plot(myx, myy)
```

- But, sometimes, this doesn't show anything!!

  **`plt.show()`** ➔ this function displays an image

# Adding more things

- We already know who to create basic plots
- When presenting data, you need to give more information: axis, units, legend
- Adding labels to the axis:

```
plt.xlabel("put xlabel here")
plt.ylabel("put ylabel here")
```

- Adding a title:

```
plt.title("title of the plot")
```

# Changing the plot style

```
plt.plot(myx, myy, [format])
```

- Combine colors and markers to create different styles

| Symbol | Color |
|---|---|
| ‘r’ | red |
| ‘b’ | blue |
| ‘g’ | green |
| ‘c’ | cyan |
| ‘m’ | magenta |
| ‘y’ | yellow |
| ‘k’ | black |
| ‘w’ | white |

| Symbol | Marker/style |
|---|---|
| ‘o’ | Use circles |
| ‘^’ ‘v’ ‘<‘ ‘>’ | Use triangles |
| ‘s’ | Use squares |
| ‘*’ | Use stars |
| ‘-’ | Single dashes |
| ‘--’ | Double dashed line |

# Plotting more than one series

- Many times you need to compare two different series in a single plot
- You can use more than one **`plot`** and then **`show`** them all together:

```
plt.plot(x1, y1, "bo")
plot.plot(x2, y2, "r")
plt.show()
```

# Setting limits, adding ticks

- It is possible to set limits to both x and y-axis
- Useful in cases where matplotlib might not be doing the best job

```
plt.xlim(xmin, xmax)
plt.ylim(ymin, ymax)
```

- It is sometimes also useful to add more detail between ticks, so that it's easier to visualize the data

```
minorticks_on()
```

# Limits and ticks

```
import matplotlib.pyplot as plt


plt.plot([1,2,3,4])
plt.ylabel("some numbers")
plt.title("my plot")
plt.xlim(0,2)
plt.ylim(0,3)
plt.minorticks_on()
plt.show()
```

# More advanced plotting

- It is sometimes better to explicitly create the figure and the axis as independent elements

```
myx1 = range(5)
myx2 = range(10)
fig, ax = plt.subplots()
ax.plot(myx1, "bo", label="label1")
ax.plot(myx2, "r", label="label2")
ax.set_xlim(1,5)
ax.set_ylim(1,5)
ax.set_xlabel("x label")
ax.set_ylabel("y label")
ax.minorticks_on()
ax.legend(loc='upper left', shadow=True)
plt.show()
```

# More plotting

- Feel free to explore more:
  - http://matplotlib.org/gallery.html
- Very often you will see code with something like:

```
import numpy as np
```

- NumPy is a numerical library, designed to provide data structures that are very fast (arrays)
  - https://docs.scipy.org/doc/numpy-dev/user/quickstart.html
- There are other plotting libraries
  - http://seaborn.pydata.org/index.html
  - http://ggplot.yhathq.com/
  - http://bokeh.pydata.org
  - https://plot.ly/python/

# Finally

- Your feedback is important
  - What went well, what didn't
  - What else can we do?
- Please take a few minutes to complete the survey!

http://bit.ly/xsedesouthern

TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

# Programming with Python

Southern University and A&M College

April 1st


**PRESENTED BY:**

Antonio Gomez Iglesias

Texas Advanced Computing Center

HPC Group

agomez@tacc.utexas.edu

http://hpcuniversity.org/trainingMaterials/237/