

Programming and Scripting Tutorial

June 4th, 2013

Andrew Kail

Pragnesh Patel

Programming Languages

- A programming language is designed to communicate a series of instructions to a machine
- Split into two components, Syntax and Symantics
 - Syntax – a set of rules defining the combinations of symbols
 - Symantics – provides the rule for interpreting the syntax

Language Types

There are four categories of programming languages:

- Dynamic

- Data types are inferred and determined during runtime

- Static

- All Data types must be explicitly defined during declaration

- Functional

- Emphasizes the application of function as the main form of computation

- Object Oriented

- Focuses on the use of “objects” as containers for data values and associated procedures call methods

Dynamic Vs. Static

Python

```
data1 = 100.53;
```

legal

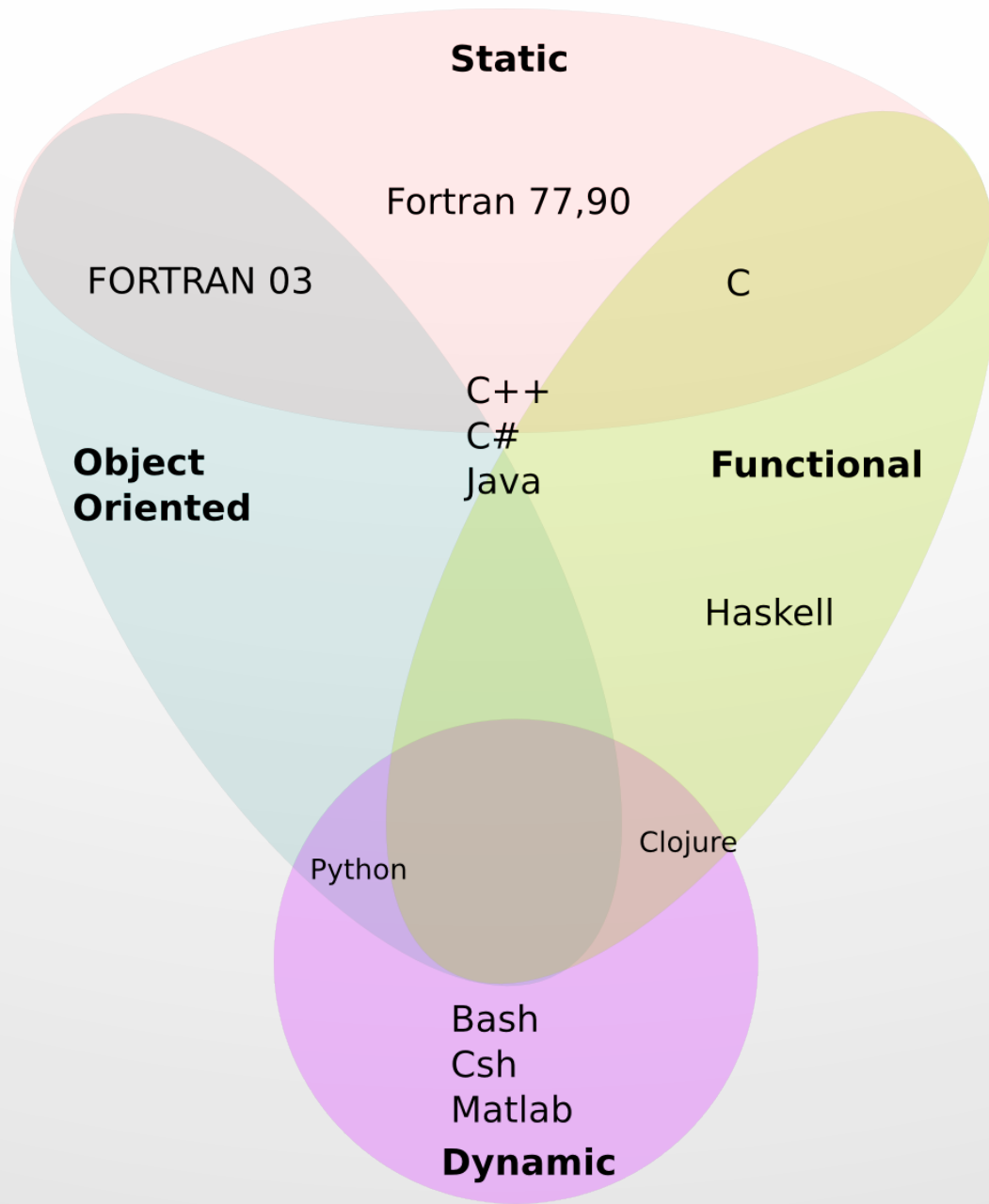
```
data1 = "hahaha";
```

C

```
double data1 = 100.53;
```

illegal/not valid

```
data1 = "hahaha"
```



Compiled Languages

- C, C++ and Fortran are all compiled programming languages
- Each source code file is compiled to a binary object file
 - Ex. *gcc -c myfunction.c*
- These object files can then be “linked” together to form an executable
 - Ex. *gcc file1.o file2.o -o myprog*

Scripting Languages

- Scripting languages do not require a compiler
- Are instead interpreted during runtime
- Sometimes much slower than compiled programming languages do to runtime overhead
- Example Languages
 - Any shell language (bash,csh,zsh,tcsh,fish)
 - Python, Perl, Ruby, Java Script

Compiling

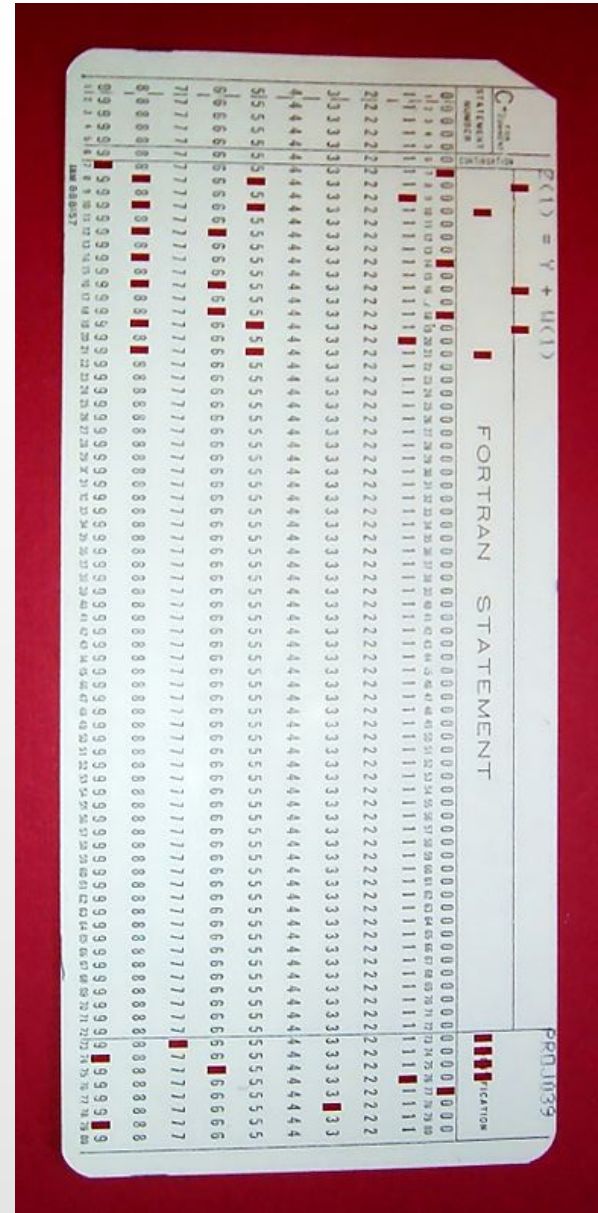
- A compiler converts ASCII based source code into an executable binary file
- Compiling is a two step process
 - Compile step converts source code to binary object files
 - Linker step “links” these objects together to form the executable
- EX.
 - Compile: `gcc -c hello.c => hello.o`
 - Link: `gcc hello.o -o hello => hello`

Compilers cont...

- There are many compilers with the most common being the GNU family of compilers
 - gcc, g++, gfortran
- Other compilers are developed and in some cases optimized for particular systems and architectures
 - Intel, PGI, and Cray

Fortran

- Developed in 50's as a replacement to Assembly Language by IBM with the first FORTRAN compiler built in 1957
- FORTRAN stands for Formula Translating system
- Originally programs were written in punch cards and inserted one at a time into the machine. NO MONITORS!
- Fortran is now an Object-Oriented language
- Still in use on systems and programs for FEM, CFD and other computational fields



FORTRAN Example

```
PROGRAM ComputeMeans
  IMPLICIT NONE

  REAL :: X = 1.0, Y = 2.0, Z = 3.0
  REAL :: ArithMean, GeoMean, HarmMean

  WRITE(*,*) 'Data items: ', X, Y, Z
  WRITE(*,*)

  ArithMean = (X + Y + Z)/3.0
  GeoMean = (X * Y * Z)**(1.0/3.0)
  HarmMean = 3.0/(1.0/X + 1.0/Y + 1.0/Z)

  WRITE(*,*) 'Arithmetic mean = ', ArithMean
  WRITE(*,*) 'Geometric mean = ', GeoMean
  WRITE(*,*) 'Harmonic Mean = ', HarmMean

END PROGRAM ComputeMeans
```

- Begins by defining the name of the program
- IMPLICIT NONE – all data types must be explicitly declared
- REAL – floating point data type
- WRITE(*,*) Prints to screen
- Perform arithmetic functions
- END PROGRAM

Crash Course in C

C is a statically typed and functional programming language developed by Bell Labs in the early 70's.

Runtime execution ALWAYS begins at the start of a main function.

```
int main( int argc, char** argv)
{
    return(0);
}
```

Every function in C and C++ has a series of inputs following the function name and a data type to return.

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    printf( "I am alive! Beware.\n" );
```

```
    return 0;
```

```
}
```



I am alive! Beware

- First line contains a pre-processor directive telling the compiler to include standard I/O functions
- Define the main function with a return type of int (No arguments required)
- Call function *printf* to print character string enclosed
- Return a value of 0 to the operating system to indicate success

Variables

- Variables are declared with a type.
- Initialization can occur at the point of declaration or the value can be stored at a later point in the code

```
int main()
{
    int x = 3; // Variable x declared and initialized
    int y; // Variable y is defined
    y = 2; // The value 2 is stored in variable y

    double a,b,c,d; // Multiple variables can be declared simultaneously
    return 0;
}
```

- All variables can be modified using several operators
 - *, -, +, /, =, ==, >, <

Basic C Data Types

Type	Memory Size	Explanation
char	2 Bytes	Integer type interpreted as a character data set
int	2 Bytes	Integer type value
float	4 Bytes	Single precision floating point value
double	8 Bytes	Double precision floating point value
bool	1 Byte	Boolean. Defined as 1 (true) or 0 (false)
void	N/A	Empty return or input data type
pointer	4-8 Bytes (system dependent)	A reference that records the location of an object or function in memory

Functions

- A Function is a group of statements that can be executed when called from the program.
- Reduces the need to repeat a series of statements multiple times in a program
- Must be declared before main function
- Function structure
 - Return type
 - Function name
 - Input variables
 - Code to execute

```
// function example
```

```
#include <stdio.h>
```

```
int addition (int a, int b)
```

```
{
```

```
    int r;
```

```
    r=a+b;
```

```
    return (r);
```

```
}
```

```
int main ()
```

```
{
```

```
    int z;
```

```
    z = addition (5,3);
```

```
    printf("The result is %d",z);
```

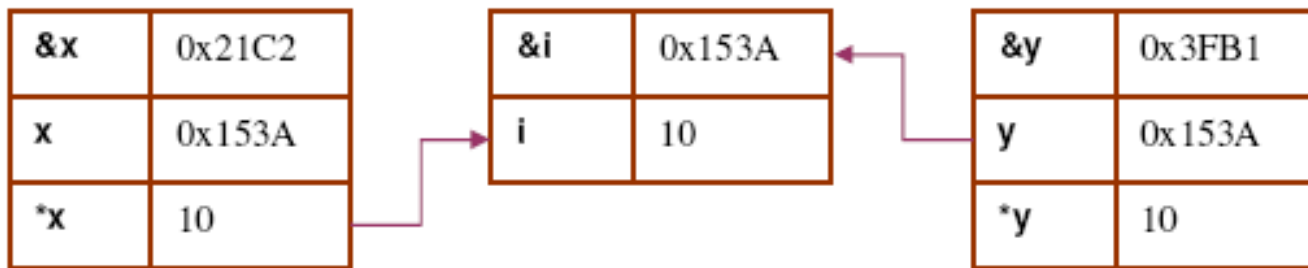
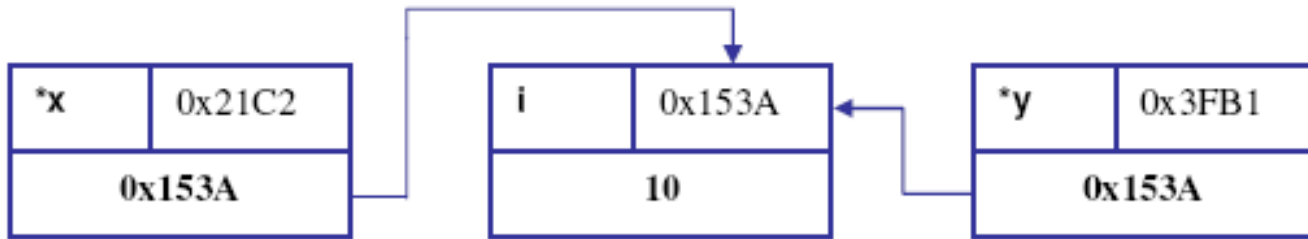
```
    return 0;
```

```
}
```


Pointers

- All variables are stored in memory and have a location where that memory starts
- Pointers are used in C based languages as variables that “point” to that memory location.
- When a variable is declared, a section of memory associated with that variable is set aside. This is called allocating memory and is dependent on the data type and if present the size of an array or struct.
 - Ex. `int x = 2;` only utilizes 2 bytes of memory
 - Ex. `int x[4];` is allocated 8 bytes of consecutive memory
- A pointer variable can be assigned a memory address from an existing variable
 - Ex. `int * x = &y;`
 - * denotes variable x is a pointer, pointing to an integer value
 - & denotes returning the memory address of variable y
 - If we were to output the value of x it would look like -> 0x7fff9c5089d4
- Pointers can be used to conveniently pass data to functions

Pointers cont...



variable	Memory address (hex)
Value of variable	

Ex 2: Illustration of & and * Operators with Pointers

C Input and Output

- Under stdio.h
- `printf("string");`
 - Prints to std out
- `scanf("%d", &number);`
 - Reads from std in and assigns value to number
- `FILE *fp = fopen("filename", "rw");`
 - Opens file name "filename" with read and write properties
- `fscanf(fp, "%d", &number);`
 - Reads file fp and stores first variable in number
- `fprintf(fp, "string");`
 - Prints string value to file fp

C I/O cont...

Standard in/out

```
#include <stdio.h>
int main()
{
    char input[80];

    scanf("%s", input);
    printf("%s",input);

    return 0;
}
```

File I/O

```
#include <stdio.h>
int main()
{
    char input[80];
    FILE * fin,fout;

    fin = fopen("infile","r");
    fout = fopen("outfile","w");

    fscanf(fin,"%s",input);
    fprintf(fout,"%s",input);

    fclose(fin); fclose(fout);

    return 0;
}
```

Control Structures

For Loops

```
for(int i=0; i<6; i++)  
{  
    x = x + i;  
}
```

X = ?

While Loops

```
while(x <= 4)  
{  
    x = x + 1;  
}
```

X = ?

If statements

```
if( x == 0)  
{  
    do something  
} else if (x == 2)  
{  
    do something  
} else {  
    do something  
}
```

Exercise - 1

- Using printf and scanf, read in data from the terminal and print it back to the screen

Exercise - 1

- Using printf and scanf, read in data from the terminal and print it back to the screen

```
#include <stdio.h>
int main()
{
    char* str;
    scanf("%s",str);
    printf( "%s",str );
}
```

Exercise 2

- Perform Exercise 1 with a file instead

Exercise 2

- Perform Exercise 1 with a file instead

```
#include <stdio.h>
int main()
{
    int x;

    FILE * file1,file2;
    file1 = fopen("filename","r");

    fscanf(file1,"%d",x);

    file2 = fopen("filename","w");

    fprintf(file2, "%d",x);

    fclose(file1);
    fclose(file2);
}
```

Exercise - 3

- Write a function that will calculate the arithmetic mean of two input variables and print to the screen

Exercise - 3

```
double calcmean(double x, double y)
{ return (x+y)/2;
};
int main()
{
    double x,y,z;
    x=2; y=4;
    calcmean(x,y);

    return(0);
}
```

Exercise 4

- Write 4 different functions and have each function call the preceding function

Exercise 4

```
#include <stdio.h>
int function1( int x )
{
    return function2(x) + 1;
};
int function2( int x )
{
    return function3(x) + 1;
};
int function3( int x )
{
    return function4(x) + 1;
};
int function4( int x )
{
    return 21;
};

int main()
{
    printf( "%d",  function1(2) );
    return 0;
}
```

C++

- C++ is structurally the same as C and meant as an improvement to C (hence the ++)
- All functions used in C are also implemented in C++
- Main difference is use of object oriented programming, the STL and template programming.
- This will covering the basics of the STL and high level overview of Object Oriented programming

STD Namespace

- Namespaces allow classes, objects and functions to be grouped under one name
- All C++ functions, classes and STL containers are grouped under the STD namespace
- In order to call them they must be prepended with `std::`
 - Ex. `std::cout` `std::list`
- For beginners adding *using namespace std;* to the beginning of the source code of the main function allows one to access this functionality without having to include `std::`
- It is preferred to use `std::` to prevent other functions of the same name from being accidentally used by the compiler

Input/Output


- Input and output is handled differently than C
- *#include <iostream>* brings in all the functionality for std I/O in c++
 - Provides cout & cin amongst others
- File I/O depends on *<fstream>*
- Insertion operators “<<” and “>>” are used to transfer the stream objects
- Examples
 - *std::cout << “Hello”;*
 - *infile >> string1;*

Hello World

```
#include <iostream>
```

```
int main(int argc, char** argv)
```

```
{
```

```
    std::cout << "Hey buddy" << std::endl;  Hey buddy
```

```
    return 0;
```

```
}
```

- <iostream> includes functions cout and endl
- String “Hey buddy” is inserted to std::cout
- std::endl acts as a carriage return

File I/O

```
#include <iostream>
#include <fstream>
#include <string>
```

```
int main(int argc, char** argv)
{
    std::string readtxt;
    std::ifstream infile("input.txt");

    infile >> readtxt;

    infile.close();

    std::cout << "The first item of the file reads " << readtxt << std::endl;

    std::ofstream outfile("output.txt");

    outfile << readtxt;

    outfile.close();
    return 0;
}
```

Exercise - 1

- Using `std::cout` and `std::cin` read in data from the terminal and print it back to the screen
 1. Add in proper header files
 2. Create input string
 3. Read in data
 4. Print out data

Exercise - 1

```
#include <iostream>
#include <string>
int main()
{
    std::string inputstr;
    std::cin >> inputstr;

    std::cout << inputstr;

    return 0;
}
```

Exercise 2

- Perform Exercise 1 with a file instead using `fstream`
 1. Create `ifstream` object and open file
 2. Read contents of file to a string
 3. Close File
 4. Create `ofstream` object and open file
 5. Print string to file
 6. Close File

Exercise 2

```
#include <fstream>
#include <string>
int main()
{
    std::ifstream infile("inputfile");
    std::string instr;
    infile >> instr;
    infile.close();

    std::ofstream outfile("outputfile");
    outfile << instr;
    outfile.close();
}
```

Follow Up

For further practice take a look at:
projecteuler.net

C and C++ programming tutorials

- cplusplus.com
- cprogramming.com

GNU Make

What is Make?

- Make is a utility that allows for automatic builds of executable programs and links external libraries
- Can rebuild things for you automatically: timestamp
- Its behavior is dependent on rules defined by the user
- Executing the make command begins by searching the current directory for a file name “Makefile” or “makefile” and reads that file.
- Different makefiles can be specified with a `-f` option

Make Rules

A simple makefile consists of “rules” with the following shape:

```
target ... : prerequisites ...  
    recipe  
    ...  
    ...
```

- Target – usually a name of a file generated by the rule, or the name of an action to carry out
 - Can be executable or object name
- Prerequisites – a file or action that the target is dependent upon.
 - Can be another target
- Recipe – A series of commands to execute for completion of the target

Variables

- Makefiles can also make use of variables
 - Variables can be a list of files
 - SOURCES = main.c function1. function2.c
 - Can be an executable command
 - CC = g++
 - Options to pass to a compiler
 - CFLAGS = -g -Wall O3
- Variables are referenced using a \$ sign
 - “\$(CC) -c \$(SOURCES)”
 - Equivalent to “g++ -c main.c function1.c function2.c”

Executing a Makefile

- A makefile will execute the first target of a makefile
 - `$ make` (*first target*)
- Other targets can be executed by specifying the specific target name
 - `$ make clean`

Exercise 1

Compile Exercise 1 with a makefile:

makefile

```
myexe: exercise1.o
```

```
    gcc exercise1.o -o myexe
```

```
exercise1.o: exercise1.c
```

```
    gcc -c exercise1.c
```

Clean Target

Try typing make again. What is the result?

If we want to recompile everything, we must remove all object files and the executable.

Clean Target:

clean:

```
rm -rf *.o myexe
```

Improving Our Makefile

Wouldn't it be nice if we could easily change compilers? Lets use a variable.

Add: "cc= gcc" to the beginning of the makefile

Replace: "gcc" later on with \$(cc)

Makefile

```
cc=gcc
```

```
myexe: exercise1.o
```

```
    gcc exercise1.o -o myexe
```

```
exercise1.o: exercise1.c
```

```
    $(cc) -c exercise1.c
```

Adding FLAGS

- Adding a flags variable allows you to quickly change the compiler options.
- After the cc variable on the top add a new variable “CFLAGS = -Wall”
- Add “\$(CFLAGS)” to the compile line
 - “\$(cc) -c \$(CFLAGS) exercise1.c”

Makefile

```
cc=gcc
```

```
CFLAGS= -Wall
```

```
myexe: exercise1.o
```

```
$(cc) exercise1.o -o myexe
```

```
exercise1.o: exercise1.c
```

```
$(cc) -c $(CFLAGS) exercise1.c
```

Source Code Management

- Use list of sources instead
 - “SOURCES=exercise1.c”
- Create a list of object files from the source list
 - “OBJECTS= \$(SOURCES:.cpp=.o)”
- Add target to convert source code into a target file
 - “%.o: %.c”
 - “\$(cc) -c \$(CFLAGS) \$<”
- “\$<” references all prerequisites

New Makefile

```
cc=gcc
```

```
CFLAGS=-Wall
```

```
SOURCES=exercise1.c
```

```
OBJECTS=$(SOURCES:.c=.o)
```

```
%.o: %.c
```

```
    $(cc) -c $(CFLAGS) $<
```

```
myexe: $(OBJECTS)
```

```
    $(cc) $(OBJECTS) -o myexe
```

Makefile Template

```
# Assign Compilers
CC=g++
cc=gcc
ftn=gfortran

# Executable Name
EXECUTABLE=

# List of Sources
SOURCES=

# Compiler Flags
CFLAGS=
FFLAGS=

# Make Targets #
all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    #$(CC) $(OBJECTS) -o $@
    #$(cc) $(OBJECTS) -o $@
    #$(ftn) $(OBJECTS) -o $@

$(OBJECTS):
    #$(CC) -c $(CFLAGS) $
(INCLUDE_FLAGS) $(SOURCES)
    #$(cc) -c $(CFLAGS) $
(INCLUDE_FLAGS) $(SOURCES)
    #$(ftn) -c $(FFLAGS) $
(INCLUDE_FLAGS) $(SOURCES)

clean:
    rm -rf $(OBJECTS) $(EXECUTABLE)
```

Further Reading for GNU Make

<http://www.gnu.org/software/make/manual/make.html>

Bash Tutorial

- Bash stands for Bourne Again Shell and can be found on almost any Linux/Unix based computer system.
- Scripting language as well command line interface

Basic commands

Command	Description
cd	Change directory
ls	Lists the contents of a directory
mkdir	Makes a directory
sed	Stream edit. Used to edit ascii text
grep	Searches files or input for a pattern match
	Pipes output of a command directly into another
man	Returns the manual page of a given command
pwd	Print working directory
chmod	Change file mode bits
env	Print out Linux environment
time	Will time an executed command/program

Command	Description
touch	Change file timestamps
finger	Returns information on the user
id	Return the information on a specified user
alias	Alias a command to another value
tar	Tape archive extraction
echo	Displays a line of text
exit	Exits the login/terminal session
pkill	Kills a process
ps	Process status
ssh	Used to log into remote machines
su/sudo	Substitute user/ execute with root permissions
who	List all currently logged in users
awk	Pattern scanning and processing language

Bash Scripting

- Simple text file begun with `#!/bin/bash`
 - Tells the OS the script will be interpreted by the bash shell located under the `/bin` directory
- All following bash commands can be executed as if on the command line

Ex.

```
#!/bin/bash  
mkdir bashtest  
cd bashtest  
touch bashexample
```

Executing Bash Scripts

- Not inherently executable. Only a text file.
- Must use “chmod u+x”

Bash Scripts can handle arguments.

Ex. myls

```
#!/bin/bash
```

```
ls -la $1
```

execute: “./myls exercise1.c”

- \$1 represents first arguments following the script executin. \$2, \$3 etc.. would follow
 - \$@ will pass in all arguments following

Bash Loops

- For loop

```
#!/bin/bash
for i in $( ls ); do
echo item: $i
done
```

- While Loop

```
#!/bin/bash
count=1
while [ $count -le 9 ]
do
    echo "$count"
    sleep 1
    (( count++ ))
done
```


Python

- Python is a high-level (typically) scripting language
- Object Oriented
- Uses white-space indentation to delimit coding blocks like {}

Ex.

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

```
#
print "Enter two integers and I will tell you"
print "the relations they satisfy"

number1 = raw_input( "Please enter the first integer: " )
number1 = int(number1)

number2 = raw_input( "Please enter the second integer:" )
number2 = int(number2)

if number1 == number2:
    print "%d is equal to %d" % (number1, number2)

if number1 != number2:
    print "%d is not equal to %d" % (number1, number2)

if number1 < number2:
    print "%d is less than %d" % (number1, number2)

if number1 > number2:
    print "%d is greater than %d" % (number1, number2)

if number1 <= number2:
    print "%d is less than or equal to %d" % (number1, number2)

if number1 >= number2:
    print "%d is greater than or equal to %d" % ( number1, number2 )

dummy=raw_input()
```

```
print "Simple for loop using a range variable"
print
for x in range(10):
    print "Burp!"
```

```
print
print "Counting by 5s"
for x in range(0,10,5):
    print str(x) + " Errruppppp!"
```

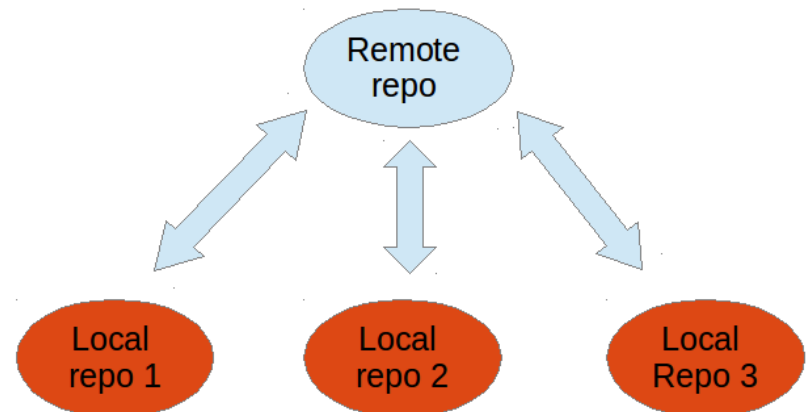
```
print
print "The White Knight (from Alice in Wonderland) counts backwards:"
for x in range (10, 0, -1):
    print str(x) + " Feed your head"
```

```
print
print "Breaking up is hard to do. Well, actually, it's pretty easy when you're a string."
word = raw_input("Enter a word: ")
for letter in word:
    print letter
```

Git

What is Version Control?

- Version control allows one to store multiple copies of a file and simultaneously track their history
- Able to capture snapshot of the system which one can revert back to
- Git allows for the data to be stored in a repository, either locally or remotely
- Allows for branching



Go to Github

Explore Features Enterprise Blog [Sign in](#)

Pick a username

Your email

Create a password

Tip: use at least one number and at least 7 characters.

By clicking on "Sign up for free" below, you agree to the [Terms of Service](#) and the [Privacy Policy](#).

[Sign up for free](#)

[See plans and pricing](#)

- Go to github.com and create an account
- Write down user name on paper passed around

for

Clone our first directory

- Make a directory to house the training git repository
 - “mkdir gitrepo”
- Change to the new directory and then clone the repo
 - “cd”
 - “git clone <https://github.com/jics-csure/training.git>”

Git Commands

- git add
- git rm
- git status
- git mv
- git push
- git merge

All these commands can be found with “git –help”

Resources

- FORTRAN - <http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/intro.html>
- C – cprogramming.com
- C++ - cplusplus.com
- Bash
- Python
- Git