CS 35L Fall 19 Section 3 Notes 1
Zhaowei Tan
Command line basics:

1. man: This command can be used to check the manual for a specific command. Usage: man [command name]

2. $HOME (= ~): The home directory (usually looks like /Users/[Your user name]). You can try cd ~ or cd $HOME.

3. ls: This command is used to check the contents under a directory. An option can be added after the command to unleash the power of a specific command. E.g. ls -l will display the detailed information of the current directory. You can always put multiple options together, e.g. ls –alG. Use man page to check the options.

4. cd: change directory. Usage: cd [path]. Your command line interface has a current path, which you could use command pwd to check. Using cd command, you are able to change the current directory.

5. Absolute path vs. relative path: the absolute paths begin with "/". When you specify the absolute paths, the system starts from the root directory ("/") and goes all the way to the location you specify. When you use relative path, e.g. you try to open "test/test.txt", the system will first try to enter the test directory under your current directory, and then try to find the file test.txt under that directory. In short, it tries to find [the result of pwd command]/test/test.txt

6. cp: short for copy; mv: short for move; mkdir: create a directory; rm: short for remove; rmdir: remove directory. Please check the usage of these commands using man page or Google them.

7. ln -s: create symbolic link. Using ls -l you can check the type of a file (the first symbol). Using ls -i you can see the inode information. Check more details on inode in slides.

8. chmod: use chmod [number] [filename] to change the permission of a file. How to get the number has been specified in the slides. There are ways other than number to specify the new permission, i.e. using chmod [group][option][permission] (see slides).

9. Find command
   The general form is
   find [starting directory] [options]
   E.g.
   find . -name "test" means that you try to find the files with name "test", under the currently directory (again, single dot means current directory!).

Note that the find command will search recursively under all the accessible subdirectories.
Other than name, you can also find files that satisfy the requirement of size, type… check the man page for detail

10. Wildcard:
    Unix wildcard could be thought of as a regular expression. To put it plainly, sometimes you want to search for something whose name follows a specific feature instead of knowing its exact name– this is when the wildcard could help.

    Here let's see two very simple examples:
    ? represents any single character
    * represents zero or more characters
    [] represents only allow the characters in the bracket

11. Use tab to autocomplete when you are typing in the command line interface. When there are multiple possible choices, the interface will inform you instead of autocompleting.

12. echo: print the value of a variable

13. Each Unix-like system has some package management tools
    Debian-like Unix system: apt-get, Mac OS: brew (you might need to download), CentOS: yum. For other systems, you could search online.

14. $PATH variable is important: when you try to execute a program, your terminal loops the path inside this variable, and search for the program that matches the name you indicate.
    To execute your own program, use ./[your_program] –options, or [/absolute path]/program –options. This will force the terminal to run the program you specified, instead of searching in the $PATH.
    Alternatively, you can add your own directory into the $PATH. Use export PATH=[new path]:$PATH to update the PATH as the intended directory + colon + the old $PATH variable. Now you can directly execute your program anywhere using [your_program] –options.
    Note: . = current directory, .. = parent directory. Check

15. Some commands to check the system information:
    uname: check system distribution
    df: disk information
    whoami: check the current user

16. Download: use wget or curl to download file from online.
    wget [options] [url]
    curl [options] [url]

17. You can use ">" sign to redirect your output from command line to the file. E.g. cat file1 > file2 will redirect the output of cat file1 to file2 (which effectively equals to cp)

18. cmd1| cmd2: Pipeline takes the output of cmd1 as input of cmd2. This is equivalent to cmd1 > tmp.txt && cmd2 < tmp.txt, but much easier.

19. sort command: Used to sort a file/input. Check man page for details!

20. cat, less, more, tail, head: used to display the file. Sometimes we only want to check the contents instead of editing a file.
    Usage: cat/less/more [filename]

21. diff: used to check the difference between two files.
    diff file1 file2

22. whereis/which: used to check where the actual command program is.
    Example: whereis cp

Emacs: https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf

23. Emacs basic: Use C-x, C-c to quit the Emacs. When you learn Emacs, C means Ctrl, C-x means press C and press x and then release both of them. M means Meta, which is usually your alt/option key on your keyboard. If you are using Mac, go to your preference of the terminal, and click the option to set the option key as your meta key.

24. Move around in the Emacs (can be used in terminal too)
    Character-wise: C-b move backward; C-f move forward; Del delete backward; C-d delete character forward
    Word-wise: M-b move one word backward; M-f move one word forward; M-Delete delete one word backward; M-d delete one word forward
    Line-wise: C-a move to the beginning of line, C-e move to the end of line, C-k kill forward to the end of line

25. Search: C-s search forward, C-r search backward. M-! or M-x shell used to type commands. M-x compile to type the compile command (like gcc).

26. C-x C-b will list all buffers in Emacs. Use C-x b to switch among buffers.

27. Other than emacs, you could also use vi or nano as your editors.
    Use vi [filename] or nano [filename]. You can find good (and interactive) courses on vim editor online.

Regular expressions

Character	BRE / ERE	Meaning in a pattern
\	Both	Usually, turn off the special meaning of the following character. Occasionally, enable a special meaning for the following character, such as for \(...\) and \{...\}.
.	Both	Match any single character except NULL. Individual programs may also disallow matching newline.
*	Both	Match any number (or none) of the single character that immediately precedes it. For EREs, the preceding character can instead be a regular expression. For example, since . (dot) means any character, .* means "match any number of any character." For BREs, * is not special if it's the first character of a regular expression.
^	Both	Match the following regular expression at the beginning of the line or string. BRE: special only at the beginning of a regular expression. ERE: special everywhere.

Regular Expressions (cont'd)

$	Both	Match the preceding regular expression at the end of the line or string. BRE: special only at the end of a regular expression. ERE: special everywhere.
[...]	Both	Termed a bracket expression, this matches any one of the enclosed characters. A hyphen (-) indicates a range of consecutive characters. (Caution: ranges are locale-sensitive, and thus not portable**.) A circumflex (^) as the first character in the brackets reverses the sense: it matches any one character not in the list.** A hyphen or close bracket (]) as the first character is treated as a member of the list. All other metacharacters are treated as members of the list (i.e., literally). Bracket expressions may contain collating symbols, equivalence classes, and character classes (described shortly).
\{*n*,*m*\}	BRE	Termed an *interval expression*, this matches a range of occurrences of the single character that immediately precedes it. \{*n*\} matches exactly n occurrences, \{*n*,\} matches at least n occurrences, and \{*n*,*m*\} matches any number of occurrences between n and m. n and m must be between 0 and RE_DUP_MAX (minimum value: 255), inclusive.
\(\)	BRE	Save the pattern enclosed between \(and \) in a special *holding space*. Up to nine sub patterns can be saved on a single pattern. The text matched by the sub patterns can be reused later in the same pattern, by the escape sequences \1 to \9. For example, **\(ab\).*\1** matches two occurrences of ab, with any number of characters in between.

Regular Expressions (cont'd)

\n	BRE	Replay the nth subpattern enclosed in \(and \) into the pattern at this point. n is a number from 1 to 9, with 1 starting on the left.
{n,m}	ERE	Just like the BRE \{n,m\} earlier, but without the backslashes in front of the braces.
+	ERE	Match one or more instances of the preceding regular expression.
?	ERE	Match zero or one instances of the preceding regular expression.
|	ERE	Match the regular expression specified before or after.
( )	ERE	Apply a match to the enclosed group of regular expressions.

Regular Expressions (cont'd)

*	Match zero or more of the preceding character
\{*n*\}	Exactly n occurrences of the preceding regular expression
\{*n*,\}	At least n occurrences of the preceding regular expression
\{*n*,*m*\}	Between n and m occurrences of the preceding regular expression

POSIX Bracket Expressions

Class	Matching characters	Class	Matching characters
[:alnum:]	Alphanumeric characters	**[:lower:]**	**Lowercase characters**
[:alpha:]	Alphabetic characters	[:print:]	Printable characters
[:blank:]	**Space and tab characters**	[:punct:]	Punctuation characters
[:cntrl:]	Control characters	[:space:]	Whitespace characters
[:digit:]	Numeric characters	**[:upper:]**	**Uppercase characters**
[:graph:]	Nonspace characters	[:ascii:]	**ASCII Characters**

Cheatography

Regular Expressions Cheat Sheet

by Dave Child (DaveChild) via cheatography.com/1/cs/5/

Anchors

^	Start of string, or start of line in multi-line pattern
\A	Start of string
$	End of string, or end of line in multi-line pattern
\Z	End of string
\b	Word boundary
\B	Not word boundary
\<	Start of word
\>	End of word

Character Classes

\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\x	Hexadecimal digit
\O	Octal digit

POSIX

[:upper:]	Upper case letters
[:lower:]	Lower case letters
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexadecimal digits
[:punct:]	Punctuation
[:blank:]	Space and tab
[:space:]	Blank characters
[:cntrl:]	Control characters
[:graph:]	Printed characters
[:print:]	Printed characters and spaces
[:word:]	Digits, letters and underscore

Assertions

?=	Lookahead assertion
?!	Negative lookahead
?<=	Lookbehind assertion
?!= or ?<!	Negative lookbehind
?>	Once-only Subexpression
?()	Condition [if then]
?()|	Condition [if then else]
?#	Comment

Quantifiers

*	0 or more	{3}	Exactly 3
+	1 or more	{3,}	3 or more
?	0 or 1	{3,5}	3, 4 or 5

Add a ? to a quantifier to make it ungreedy.

Escape Sequences

\	Escape following character
\Q	Begin literal sequence
\E	End literal sequence

"Escaping" is a way of treating characters which have a special meaning in regular expressions literally, rather than as special characters.

Common Metacharacters

^	[	.	$
{	*	(	\
+	)	|	?
<	>		

The escape character is usually \

Special Characters

\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed
\xxx	Octal character xxx
\xhh	Hex character hh

Groups and Ranges

.	Any character except new line (\n)
(a|b)	a or b
(...)	Group
(?:...)	Passive (non-capturing) group
[abc]	Range (a or b or c)
[^abc]	Not (a or b or c)
[a-q]	Lower case letter from a to q
[A-Q]	Upper case letter from A to Q
[0-7]	Digit from 0 to 7
\x	Group/subpattern number "x"

Ranges are inclusive.

Pattern Modifiers

g	Global match
i *	Case-insensitive
m *	Multiple lines
s *	Treat string as single line
x *	Allow comments and whitespace in pattern
e *	Evaluate replacement
U *	Ungreedy pattern

* PCRE modifier

String Replacement

$n	nth non-passive group
$2	"xyz" in /^(abc(xyz))$/
$1	"xyz" in /^(?:abc)(xyz)$/
$`	Before matched string
$'	After matched string
$+	Last matched string
$&	Entire matched string

Some regex implementations use \ instead of $.

By **Dave Child** (DaveChild)
cheatography.com/davechild/
www.getpostcookie.com

Published 19th October, 2011.
Last updated 12th May, 2016.

CS 35L Fall 19 Section 3 Notes W2 Wed
Zhaowei Tan

Regex:
Refer to the cheat sheet at:
http://www.regexlib.com/CheatSheet.aspx?AspxAutoDetectCookieSupport=1

Play around with this online tool:
https://www.regexpal.com/

Some highlights:

1. (Anchors) ^ start of a line, $ end of a line
2. (Character classes) . means any character except newline
   [abc] any of a, b, or c (only matches one occurrence)
   Or we can use [a-c], [a-g1-9] to represent a range of char/number
   We also have \w, \d, … for word, digit, …
3. (Quantification) After an expression (a single character class or a group), we can use *, + or ? to represent multiple occurrence of the expression
   * zero or multiple occurrence
   + one or more occurrence
   ? zero or one occurrence
   E.g. a*b will match b ; a+b will not match b
4. {} used to specify certain times of the occurrence (compared to * and +, which allows arbitrary many)
   E.g. [abc]{3} will match aaa but not bb
   [abc]{1,3} matches a string with 1-3 occurrence of [abc].
5. \ escape to escape special characters
6. (Grouping) () parenthesis used to capture group
   E.g. (abc){2} will match abcabc
7. Lazy vs. greedy, lookaround, back-reference: see slides
8. sed command:
   Substituting text:
   sed 's/regex/replacement/flags' file
   E.g. sed 's/^a/b/' file.txt
   sed 's/a$/b/g' file.txt (-g flag: substitute all the occurrence)
   The delimiter for sed can be another sign, e.g: sed 's_a_b_g'
9. To re-use the searched regex pattern use & sign. E.g. if you want to wrap every matching substring with a pair of parenthesis, do this: sed 's/ab+/(&)/g'
10. To delete with sed, use sed '/pattern to match/d'
11. Awk basics: https://www.geeksforgeeks.org/awk-command-unixlinux-examples/
12. Extended vs. basic regular expression. Some special signs need escape to have special meaning:
    https://www.gnu.org/software/sed/manual/html_node/BRE-vs-ERE.html

CS 35L Fall 19 Section 3 Notes W2 Mon
Zhaowei Tan

Shell basics:

1. We don't need to compile a shell script before we run it, unlike C or C++.

2. To declare the interpreter for the script, we start the file with #!/bin/sh (which can be a symbolic link to bash shell script), so that when we execute it, the system knows it is a shell script, and runs it with correct interpreter accordingly.

3. Comment in shell scripting: #

4. All the commands can be executed inside a bash script, just like when we execute them inside a terminal.

5. To assign a variable: var=5, no need to declare it beforehand. To print it, echo $var. Pay attention to when we need the dollar sign. Also pay attention to the space.

6. For arithmetic calculation, use $(()). Example: i=$(($i + 1))

7. In bash, in order to assign the output of commands to variables, we can wrap our command inside `. For example, a=`ls /usr/bin`. Alternatively, we could use a=$(ls /usr/bin)

8. String could be wrapped inside " or ', there's a major difference though. if we have lan=English
   echo "Language is $lan" and echo 'Language is $lan'
   have different outputs. You need the "" for the white space.

9. Some built-in variables:
   $0: the first command line argument (i.e. our script name)
   $1: the second command line argument (of script or function, depending on where you call it)
   $2: similar to $1
   …
   $#: the number of arguments (does not count $0)
   $*: all the input arguments
   $$: process ID

10. For loops, if, or function, check the script I upload.

11. If statement: (you need the space)

```
if [ condition ]
then
  commands
elif [ condition ]
then
```

```
 commands
else
 commands
fi
```

12. Some useful conditions:
    `string1 == string2` # if two strings are identical
    `integer_a –eq integer_b` # if a equals to b
    Similarly, we have `–gt, -ge, -lt, -le` for greater than, greater than or equals to, less than, less than or equals to, respectively.

13. Extra conditions used in `if` statement:
    ```
    -e file   # file exists
    -f file   # regular file exists
    -d file   # directory exists
    ```

14. Use `[ ! expression ]` returns true if expression is not true.

15. While loops:
    ```
    while [ condition ]
    do
     commands
    done
    ```

16. For loops:
    ```
    for i in list
    do
     commands
    done
    ```
    Here the list could be a string with items separated with space
    So this makes sense: `for i in `ls .``

17. Bash function:
    ```
    function function_name {
     commands
    }
    ```

18. Pass arguments to functions:
    `function_name argument1 argument2`
    Get the arguments inside the function: `$1, $2 …`
    Same as passing parameters from command lines

19. Get return value from the function:
    Inside the function, use `return` keyword for exit status.
    After calling the function, get return value: `$?`
    Also note that the variables you define inside the functions are usually global, which can be accessed outside the scope of the function

20. Integer list: `{start..end}`

e.g. for i in {1..10}

21. Get the length of the string
    ${#var}

22. Some other extras: Arrays, case, … In the script I upload!

CS 35L Fall 19 Section 3 Notes Week 3 Monday
Zhaowei Tan

Python Basics:

1. Installation and package management:
   sudo pip install module_name
   sudo pip3 install module_name
   Note that you might not be able to do that if you do not have root permission (i.e., unable to 'sudo')
2. Using interactive shell or script execution:
   python script.py
   python3 script.py
3. Modules are important parts of Python, probably one of the major reasons why people love Python. Use pip to install a new module in your computer. To import a module into the script:
   import module
   import module as module_alias
4. Variable types: You don't have to declare the variable type when you assign it – instead, Python would automatically know what type the variable is. E.g. the following all work:
   a = "123"
   a = 3
   a = 4.0
   Note that:
   1) To transfer from one type to another, use something explicitly like
      a = 4.0
      a = int(a)
   2) In calculation, the Python would not do the auto type transfer for you
      a = 4 / 3 # equals 1 in python 2
      a = 4 / 3 # equals 1.333 in python 3
   3) To check the type, use type(var)
5. Use print (…) to print things to standard output.
   Print can recognize any type of the variable, int, float, list, etc.
   Python 3 print special: sep and end
   print('G','F','G', sep='')
   print("Welcome to" , end = ' ')
6. String:
   a. Access a char: s[index]; index can be negative, s[start:end] for slicing
   b. Split the string: s.split('delimiter')
   c. Find substring within a string: s.find("substring", start_position). Returns index, or negative if not found. Other method: replace, upper, …
   d. Concatenate: c = a + b
7. List:
   To create a list
   my_list = ['a', 4, [1, 3]]
   The items in Python list are not necessarily be of same length or same type. Use list[a] to fetch an item. Use list[a:b] to fetch a subset. Index a is inclusive, but b is exclusive.

Useful list functions: append, extend, index, remove, pop, count…

8. Dictionary
   To initiate a list/directory
   my_dict = {"name": "Zhaowei", "major": "CS", "status": 0}
   Access value by key: my_dict["name"]
   Useful dictionary functions: keys, values, items…
9. Tuples: Not mutable. Old tradition of keeping heterogeneous stuffs. Can be used as key for dictionary.
10. if condition:
      do something
    You can use and, or, not to connect the different statements.
    Use indentation to wrap the block.
    To enable else if:
    if condition1:
      do something
    elif condition2:
      do something else
    else:
      do something different
    Some special conditions: if key in dict (check if key is in the keys of dict); if ele in list
11. Loops:
    while condition:
      do something
    or
    for i in [list]:
      do something for i
    Note:
    1) Be careful that Python uses indentation to indicate different levels, instead of using brackets (C) or keywords (Bash). I recommend using spaces instead of tabs as your indentation for compatibility.
    2) You could use range(min, max, step) to generate a number list and use for loop to iterate. The list generated includes min, but not max (similar to list[a:b]). We can use a negative step to generate numbers in a reversed order.
    3) To terminate a while/for loop, use break. To skip the current pass of loop, use continue.
12. Functions in Python:
    def func_name(arg1, arg2, …):
      do something
      return something
    Call this function
    func_name(arg1, arg2, …)
    You don't have to specify the type for arguments.
13. Python class
    class className:
      define your variables
      define your functions
    A special function is called __init__, which will be called after you instantiate a new variable.

The functions in the class should have an extra argument, self
Inside the function, to call the variable inside the class, use self.varName

When you put double underscore (__) before the variable name, the name of the variable changes from __varName to _className__varName outside this class. This is a common practice to declare the variable as a private variable, however, we could still access this variable using the transferred variable name. But never do that! The same for function here.
A class example:

```
class myClass:
  __num = 0

  def __init__(self, num):
       self.__num = 100
  def printNum (self):
    print self.__num
  def _add_people (self, num):
    self.__num += num

t = myClass(100)
t. printNum()
t._add_people(10)
print t._myClass__num #don't do this in reality!
```

14. Exception handling.
    We could use try/except to capture the error and continue the program.

```
try:
  f = open("testfile", "w")
  f.write("This is a test file.")
except IOError:
  print "Error: cannot find file or read data"
else:
  print "Written content in the file successfully"
  f.close()
```

Note: this can only catch runtime error, not syntax error

1. File I/O:
   f = open('filename', mode)
   The mode could be: r(read), w(write), a(append) …
   To write to a file: f.write("content")
   write mode will override everything in the file; append mode will add the content after the end of the file
   A complete list of mode could be found here:
   https://www.tutorialspoint.com/python/python_files_io.htm
   To read lines in file:
   ```
   for line in f:
     do something on this line
   ```
   or
   f.readline()
   Close the file by f.close() after you finish dealing with the file.
   To write and read together: r+, w+, a+
   a. r+ cannot override the previous stuff
   b. check w+ a+ and see what they do
2. Read input from users
   s = raw_input("Please input: ")

3. System related operations
   Call command: os.system(cmd)
   To save the output, might need import subprocess.
   https://stackoverflow.com/questions/4760215/running-shell-command-and-capturing-the-output
   You can deal with the file system by importing the module os
   os.mkdir(""), os.chdir(""), …
   A more complete list could be found at:
   https://www.tutorialspoint.com/python/os_file_methods.htm

4. Assignment related: OptionParser usage
   https://docs.python.org/2/library/optparse.html
   For Python 3, Argparse is recommended
   https://docs.python.org/3/howto/argparse.html

Shell Scripting:

```
#!/bin/sh

## Variables, Strings, Calculation, Command##
var1=123
var2=345
# varstring='456$var'  varstringstringstring="1 2 3"
# echo $var
# echo $varstring
# echo $varstringstring
# echo $(($var1+$var2))
# ret=$(ls -al)
# echo "The result is"
# echo $ret

# if [ $var2 == 123 ]
# then
#   echo equal
# elif [ $var2 == 345 ]
# then
#   echo "equals to second"
# else
#   echo "not equal"
# fi

# if [ -d "/var/log/wifi.log" ]
# then
#   echo "file exist"
# else
#   echo "not exist"
# fi

### While loop ###
# i=0
# output=""
# while [ $i -lt 3 ]
# do
#   i=$(($i+1))
#   output="$output $i"
# done
# echo $output

### For loop ###
# l=`ls /`
# for i in $l
# do
#   if [ ! -d "/$i" ]
#   then
#     echo "regular file"
#   else
#     echo "dirctory"
#   fi
# done

### Functions ###
# function function_name {
#   a="test"
#   echo $1 $2
#   echo $a
#   b="return"
# }
# function_name arg1_func arg2_func
# echo $b
# echo $# # #of arguments
# echo $*
# echo $$

### Case ###
# varcase=1
# case $varcase in
#   1)
#     echo "Case 1"
#     ;;
#   2)
#     echo "Case 2"
#     ;;
#   *)
#     echo "default"
#     ;;
# esac

### More Arrays ###
# arr=(value1 value2  .... valueN)
# echo ${arr[0]}
# arr[0]=valuenew
# echo ${arr[0]}

### Extras ###
# for i in {1..10}
# do
#   echo $i
# done

# teststr=12345
# teststr2=123456
# echo ${#teststr2}
```

Python Script:

```
# print("new string 1", end="")
# print("new string 2")

# import os
# os.func_name()

a = "this is a string"
# print(type(a))
# print(a[-1])
# print(a[len(a)-1])
# print(len(a))

# print(a[1:-1])

ret = a.find("string")
# print(ret)
# print(a.upper())

# test_list = ["element 1", "element b", 1.0,
5, [1,2]]
# # print(test_list[-1][0])
# print(test_list[1:3])
# print(len(test_list))
```

```
# test_list[0] = "element A"
# print(test_list)
# test_list.append("new through append")
# test_list.append("element A")
# test_list.remove('element A')
# print(test_list)
# test_list.pop(0)
# print(test_list)
# test_list.pop(-1)
# print(test_list)

# s = "one,two,three,4"
# print(s.split(','))

# test_tuple = (1,3.0,"test")
# test_tuple[2] = "new value"

# a = (1,2)
# test_dict = {"course name": "35L", "number":
36, "instructor": "prof. eggert", a: "234" }
# print(test_dict[a])
# test_dict['new key'] = "new value'"
# print(test_dict)
# test_dict['new key'] = [test_dict['new
key'], "new value 2"]
# print(test_dict)
# print(test_dict.keys())
# print(test_dict.values())
# print(test_dict.items())
# d = {"1" : "100", "2": "200"}
# if "1" in d:
#   print("1 in d")
# # elif 2 in l:
# #   print("2 in l")
# else:
#   print("not in d")

# a = 0
# while a < 10:
#   print(a)
#   a += 2

l = [1, 2, 3, 4]
d = {"1" : "100", "2": "200"}
# for i in l:
#   print(i + 2)
# for i in d.keys():
#   print(i, d[i])

# a = 3
# b = 10
# for i in range(a, b, 2):
#   if i == 3:
#     continue
#   if i == 9:
#     break
#   print(i)



def func_name(arg1, arg2):
  if arg1 < 0:
    return 0
  tmp = arg1 + arg2
  return tmp

# print(func_name(1,2))

# class className:
#   __var2 = 5
#   def __init__(self, v):
#     self.var1 = v
#   def function_name(self, arg1, arg2):
#     if arg1 < 0:
#       return 0
#     tmp = arg1 + arg2 + self.var1
#     return tmp

# t = className(10)
# ret = t.function_name(1,2)
# print(ret)

# print(t._className__var2)

try:
  func_name(1,2,3)
except TypeError:
  print("error")
```

C Program:

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

// Macro
#define BUFFER_SIZE 1024
#define min(X, Y) ((X) < (Y) ? (X) : (Y))

void f ( int ** a , int ** b);
int f1 (int a);
int f2 (int a);

int main(int argc, char *argv[]) {
  int a = 10;
  int b = 15;

  // Pointer to pointer
  int * a_p = &a;
  int * b_p = &b;

  printf("before f: a_p points to %d\n",
*a_p);
  printf("%p\n", (void *) a_p);

  // Pass pointer to pointer to function
  f(&a_p, &b_p);
  printf("%p\n", (char *) a_p);

  printf("after f: a_p points to %d\n", *a_p);
  // printf ("Number of arguments %d, the
first argument is %s\n", argc, argv[1]);
  int arr1[10];
  // printf("%lu \n", sizeof(arr1));
```

```
  // Put int i outside the loop for some older
version of C
  // int i;
  // for ( i =0; i<5; i+=1){printf("%d", i);}

  // Function Pointer
  int (*fn_ptr)(int);
  fn_ptr = f2;
  printf("Return is: %d \n",(*fn_ptr)(1));

  char * p_c;
  printf("pointer: %p\n", p_c);
  p_c = malloc(1);
  printf("pointer: %p\n", p_c);
  *p_c = 'a';
  // Don't try this!! You only allocate 1
bytes but trying to access 2 bytes
  // This might work on your computer, but
actually undefined behavior
  *(p_c+1) = 'b';
  printf("charactor is %c \n", *p_c);
  free(p_c);
  // Don't try this!! You free the memory but
try to access again
  // This might work on your computer, but
actually undefined behavior
  printf("charactor is %c \n", *p_c);
  return 0;
}

void f ( int ** a_ptr , int ** b_ptr) {
  printf("%p\n", *a_ptr);
  // printf("%p\n", a_ptr);
  *a_ptr = *b_ptr;
}

int f1 (int a) {return a+1;}

int f2 (int a) {return a-1;}
```

System Calls:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main() {
  int fd = open("file.txt", O_RDONLY);
  if (fd < 0) {
    // exit(1);
    perror("open error! ");
  }

  char buf[40];
  int ret = read(fd, buf, 8);
  if (ret < 0) {perror("read error!");}
  // printf("%s\n", buf);
  write(1, buf, 8);
  write(1,"\n",1);
  read(fd, buf, 8);
  write(1, buf, 8);
  write(1, "\n",1);

  lseek(fd, 0, SEEK_SET);
  read(fd, buf, 8);
  write(1, buf, 8);
  write(1, "\n",1);

  lseek(fd, -2, SEEK_CUR);
  read(fd, buf, 8);
  write(1, buf, 8);
  write(1, "\n",1);

  struct stat st;
  fstat(fd, &st);
  printf("The size of this file is %d\n",
st.st_size);

  close(fd);
  return 0;
}

Makefile:
CC=gcc
CFLAGS=-ldl -Wl,-rpath=.

all: main-load libmylib-d.so
main-load: main-load.c
      $(CC) $^ -o $@ $(CFLAGS)
libmylib-d.so: mylib.h mylib.c
      $(CC) -fPIC -c mylib.c -o mylib.o
      $(CC) -shared mylib.o -o libmylib-d.so
clean:
      rm *.o libmylib-d.so main-load

.PHONY: clean all
```

Dynamic Loading:

```
#include "mylib.h"
#include <dlfcn.h>
#include <stdio.h>

int main() {
  void * handle;
  void (*g)();
  handle = dlopen("./libmylib-d.so",
RTLD_LAZY);
  if (dlerror() != NULL) {
    printf ("error! %s \n", dlerror());
  }
  g = dlsym(handle, "f");
  (*g)();
  dlclose(handle);
  // check with dlerror()
  return 0;
}
```

Revision Selection - Commit Ranges

```
git log master..experiment
```

will show commits in experiment not reachable from master
so the output will be

```
D
C
```

On the other hand,

```
git log experiment..master
```

will output

```
F
E
```