Lab 7: Python

What is Python?

- A dynamic language that supports multiple programming paradigms
- There are two concurrent versions: 2.* and 3.*
  - **We'll be using Python 3.4.3**
- Zen of Python
  - Beautiful is better than ugly
  - Explicit is better than implicit
  - Simple is better than complex
  - Complex is better than complicated
  - Readability counts
- Import everything!
  - There are LOTS of modules so you don't have to reinvent the wheel
  - We will **NOT** be using NumPy

How does Python differ from C?

- No brackets, no braces, no semi-colons, no problem!
  - Uses indentation and colon (:) to indicate code sections
- No explicit declaration of datatypes required
  - `foo = 0`
  - `bar = “i am a string”`
- You can return multiple types from a function!
  - No more pointers
- Use words and not symbols in conditionals
  - &, ||, ! → and, or, not
- Refer to cheatsheet included with your git repo

How does Python differ from C?

- You can run and compile the code in one command!
  - Python interpreters
    - What we'll use: python3
    - Other options: pypy, jython
- To run a python file
  - `$ python3 file.py`
- Make it an executable
  - Add **#!/usr/bin/env python3** to the **FIRST** line of the python
  - Make it executable: `$ chmod +x <python_file.py>`
  - Then you can run like you would a C executable
    - `$ ./file.py`

How does Python differ from C?

- Like most interpretive languages, Python has an CLI
  - You can enter the interactive CLI python environment by just typing **$python3**
    - To exit, type **exit()** or **CTRL+D**
    - Follow along in the CLI for the rest of this lab!

- All this sounds awesome, so why did we have to struggle through C?
  - Python is slower than C
    - Since Python is dynamic language, many compiler optimization tricks don't work
  - It becomes tricky when you need to work with very specific data types
  - You can override built-in Python datatypes and variables by accident

Indentation and Whitespaces

- The indentation level you are on, determines the scope of you code
  - No indentation means you are on the global level
  - Always change your indentation level after a condition statement and a (**:**), for example.
    - for i in array:
    - while(1):
    - if a == True:
- **WHITESPACES MATTER!**
  - In vim you can view hidden characters using the command **:set list**
  - **Do not mix tabs (\t) and spaces.**
- You can run python in one of 2 ways:
  - As a module
  - As a script/program

Let's Try It

- Create a file called **helloworld.py** and add in lines:

```
if __name__ == '__main__':
    print("{} was run from the command line!".format(__name__))
else:
    print("Hello World from {}!".format(__name__))
```

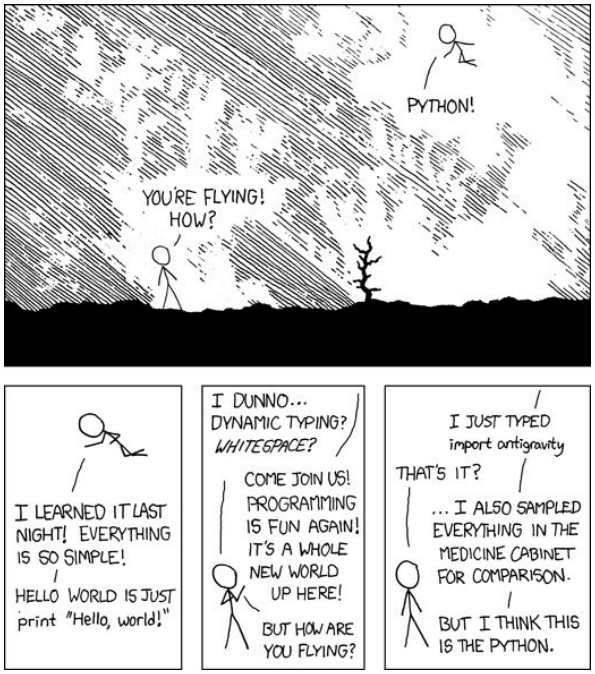
- Type `$ python3 ;` to enter python CLI
  - `$ import helloworld`
    - What gets printed to the screen?
  - Type exit() to leave the CLI and run:
    - `$ python3 helloworld.py`
    - What gets printed to the screen this time?

```c
//For comments use: // or /* */

//Import required libraries
#include <stdio.h>
#include <string.h>

//Main function
int main(int argc, char* argv[]){
    //Format print my name
    printf("My program name is %s\n", argv[0]);
    //Format print the number of arguments
    printf("I have %d arguements\n", argc);

    //In C, you need to declare variable types
    int i = 0;
    char sentence[50];

    //In C, to for-loop through array you need direct indexing
    for (i = 1; i < argc; i++) {
        //In C, you need to use the <string.h> functions for string manipulation
        strncat(sentence, argv[i], strlen(argv[i]));
    }

    //Since you concatenating to a empty char array, this removes the first bad char
    char *p = &sentence[1];
    //Print our string
    printf("%s\n", p);

    return 0;
}
```

```python
#!/usr/bin/env python3

#For comments use: # or """

#Import required libraries
import sys


#In python, you can run a program or load it as a module
#If you run it, then __name__ is set to '__main__'
if __name__=='__main__':
    #Format print my name
    print("My program name is {}".format(sys.argv[0]))
    #Format print the number of arguments
    print("I have {} arguments".format(len(sys.argv)))

    #In python, you don't need to declare variable types
    sentence = ""

    #In python, you can for-loop through lists with indexing direc
    #and you don't need to define 'word' before hand.
    for word in sys.argv[1:]:
        #In python, string manipulation is built in!
        sentence.append(word)



    #Print our string
    print(sentence)
```

Importing modules

- You can import libraries (and rename them!)
  - Rename the library **argparse** as **ap**
  - `import argparse as ap`
- You can also import just the stuff you want
  - Only load **argv** from the library **sys**
  - `from sys import argv`
- Remember, if the module is not installed in ECS 348 you can't use it!
- Google will be your friend
  - Python is very popular, there are alot of online tutorials and guides



Command Line Arguments & User Input

- **sys.argv**
  - This module works exactly like C argv works. Except you don't pass it as an argument, it is available through the **sys** module
    - **sys.argv** is a list
  - There is no *argc* variable, however, you can just get the size of the list:
    - `argc = len(sys.argv)`
  - The module **argparse** is a fancier version of sys.argv (Google it!)
- **input()**
  - To get user input for your program, you can use **input()**
    - user_input = input("How old are you? ")
  - The result of **input()** is always a string, so you will be to cast to another type if required

Function

- Any code that exists at the outermost indentation level is considered *global*
  - Better idea, use functions!
- Functions are defined using **`def`**

```
def main():                    ← ‘:’ indicates a change in scope
    print(“I ran from command line”)
if “__name__” == “__main__”:    ← Everything on leftmost line is global!!!
    main()
```

Functions

- Order matters because there are no function prototypes
  - The function you want to use my be defined earlier in the file before you use it
- You can return more than one value (of any datatype)
- You can set default values for your function arguments

```
def my_function_name(arg1,arg2=”default_val”):
    printf(“I am in the function scope”)
    foo = arg2
    bar = arg1
    return foo, bar
```

Python Lists

- Arrays have 2 types:
  - Dictionaries: { key: value } (We'll talk about these next lab)
  - Lists: (can be mixed types!)
    - `OneD = [ 1, "two", 3 ]` `TwoD = [[1,2],[2,3]]`
    - Lists are **always** passed by reference to functions
- Indexing
  - Negative indexing
    - `A = [1, 2, 3] ;  A[-1]` (Give these a try!)
  - Range indexing
    - `B = [1, 2, 3, 4, 5]  ;  B[1:3];  B[:-2];  B[1:];`
- Mutable
  - Add an element: `A.append(6)`
  - Combine 2 lists into 1: `A.extend(B)`

Pythonic Looping

- Python supports **for-loops** and **while-loops**

e.g.

```
while True:
    doSomething()
```

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

- In-line looping
  - `A = [ x for x in range(5) ]`
    - -> `[0, 1 ,2 , 3, 4]`
  - This is faster than using a standard for-loop to create a list!
- Enumerate: also include the the index
  - `A = [ (idx, v) for idx, v in enumerate(range(5,10)) ]`
    - -> `[(0,5), (1,6), (2,7), (3,8), (4,9)]`

Python Tuples

- Tuple
  - `A = (1,2)` (Note that we're using parenthesis and not brackets!)
- Unlike lists, tuples are **immutable**
  - You can't change them once they're created (so no adding or deleting elements)
- Why use them?
  - They are faster than lists
  - Can be used as dictionary keys
  - You can check if a tuple is inside a list using **in** or **not in:**

```
tuple_list = [("hello", 1), ("world", 1)]
if ("world", 1) in tuple_list:
     print("World already in list!")
else:
     print("World not in list!")
```

(What is output? What if you change the tuple to ("world", 2)?)

String Manipulation

- Just like in C, you can access a character on string using array indexing
  - `foo = “I am a string”`
  - `foo[3] ; foo[-3]; foo[2:4]` ; foo[::-1]
- You can treat them as lists in other ways:
  - Get the length of a string is the same as a list
    - `len(foo)`
  - You can multiply a string the same as a list
    - `“Boo” * 3 → “BooBooBoo”`
- Format strings using **format**
  - print(“{} works {} great!”.format(“This”, “really”))
  - print(“{1} comes before {0}”.format(3,1))

String Manipulation

- You can split up a string into a list based on a substring
  - `bar = foo.split(“ “) ;  bar`
- You can create strings from lists
  - `“-”.join(bar)`
- Count the number of instances of substring in string
  - `foo.count(“am”)`
  - `foo.count(“a”)`
- Replace substring in string with another string
  - `foo.replace(“string”, “word”)`

String Manipulation

- Get FIRST index of substring in string
  - `foo.find(“a”)`
- Get LAST index of substring in string
  - `foo.rfind(“a”)`
- Does string start or end with substring?
  - `foo.startswith(“I”);  foo.startswith(“X”)`
  - `foo.endswith(“word”);  foo.endswith(“I”)`
- For more examples:
  - http://www.pythonforbeginners.com/basics/string-manipulation-in-python

print() - Python's printf()

- print() adds a '\n' to the end of the output
  - To change that you can use the optional parameter, **end**
    - print("Hello", end='', '')
    - print("World")
- print() sends to stdout by default, you can change this by using the optional parameter, **file**
  - import sys ; print("Error message", file=sys.stderr)
- You can print an entire list using the **format** function
  - A = [1,2,3,4]
  - fmt = "{:02} " * len(A) ← You can use similar formatting string operators
  - fmt.format(*A) → 01 02 03 04 #The *A unravels the list A

File IO - Opening

- To read/write a file you use the following syntax:

```
with open(filename, params) as file:
    <do stuff with the file at this indentation level>
<file will close automatically here>
```

- The code under the *with* indentation is run, if and only if, the file is opened successfully. You should stick all file-specific code in this indentation level
- *params:* These are how you want to handle the file
  - ‘r’: read ‘rt’: read as text ‘rb’: read as binary ‘r+’: read and write
  - ‘w’: write ‘wt’: write as text ‘wb’: write as binary
  - ‘a’: append

File IO

- You can loop through the lines of a file directly:

```
with open(filename, “rt”) as file:
    for line in file: #Does not remove ‘\n’
        Line = line.rstrip() #Remove trailing ‘\n’
        output = handleFileRow(line) ← Do something!
```

- Or use the file reading functions
  - file.readline() ← Read one line at a time (ending in \n)
  - file.readlines() ← Read entire file, but as a list of lines (ending in \n)
  - file.read() ← Read the entire file

File IO - Writing

- To write to a file:
  - Note: This will erase the file if it exists, use ‘a’ to append

```
with open(filename, “w”) as file:
    file.write(“I am something\n”)
    file.write(“{} is easier with {} variables\n”.format(“This”, 2))
print(“All done with file. It closes automatically!”)
```

- Important note, if the **with** statement fails, NO error is produced, it just doesn’t go into the **with** statement.

Practise: Word Counter

- Using everything you've learned so far to create a program called **word_counter.py** that:
  - Takes a file as a command line parameter
    - ./word_counter.py input.txt
  - Count the number of words in the file and output
  - Count the number of times each word appears in the file and output