COEN 146 Lab 1

100 Points

1 Goal

The goal of Lab 1 is to familiarize you with Python and some of the basic aspects of the language that we will be using throughout the quarter. This lab can serve as a starting point for most of the programming tasks that you will tackle throughout the quarter.

2 Description

This lab assumes that you are already proficient in one programming language. If you are not, please talk with your TA and attend office hours. Python is an interpreted language which can be used in many ways for many different purposes. The implementation of Python which we will be using is CPython, the most common one. As you might guess from the name, it is based on C and you can write valid C code if you would like. Writing C code and using it in Python is outside the scope of this class and will not be covered.

Before you get started, please follow the directions to sideload Python3 onto your computer. This should only have to be done once this quarter and from then on, you should have access to the command `python3` from your terminal.

This first lab will introduce you to the language's syntax and some of the tooling. This is not a full Python tutorial, only the necessary components for the lab this quarter. If you are interested, there are much more in depth tutorials on the language. Through this lab you will become familiar with Python as a language as well as some of the standard libraries. For your reference, here is a list of what we will cover:

1. Variables and Types
2. Conditionals and Loops
3. Strings
4. Tuples and Lists
5. Functions
6. Input and Output
7. Additional Resources

3 Basic Python

3.1 Variables and Types

One of the main differences between Python and C/C++ is that Python is interpreted and dynamically typed. When you declare variables, you just assign them a value which gives the variable its initial type. There is not even a need for a semicolon. A newline character (pressing Enter on your keyboard) separates statements. Although you can use a semicolon to denote the end of a statement, it's generally best not to have two statements on the same line so it is usually avoided. Here is an example.

```
>>> hello = "Hello World"
"Hello World"
>>> counter = 0
0
>>> counter += 1
1
>>> type(counter)
int
```

You can use the built in method `type()` to get the type of a variable which can be very useful for debugging purposes.

Python uses the uppercase reserved word `True` or `False` instead of lowercase for boolean values even though there is no boolean type. Values can be tested as "truthy" (evaluating to True) if they follow certain conventions such as being an empty list or string, being None, or evaluating to 0 for a numeric type. Although Python has quite a few different types, for the purposes of this lab, you will only need to use `int, str, list,` and `tuple`

To create Python programs (in our case scripts), just create a new file with the `.py` extension. Just type `python3 script.py` to start executing your program. It'll start running anything defined at the lowest level of indentation.

3.2 Conditionals and Loops

In Python, whitespace matters. In C/C++ blocks of code are grouped by their enclosing braces. In Python, they're grouped by whitespace. So statements at the same level of indentation are considered in the same block. Here are some differences between Python and C/C++ when creating blocks of code:

- No curly braces, just proper indentation
- No parentheses around condition for if and while statements
- Switch statements are non-existent
- No assignments in conditionals
- Use `elif` instead of `else if`

Python uses the abbreviated `elif` instead of `else if` for brevity's sake. In addition, it helps keep code succinct so that you don't accidentally type an else with no condition followed by an if statement and have some nasty bugs. Conditionals are pretty much exactly the same as you're used to. For comparison, we use the keywords `and, or,` and `not` to make Python read more clearly. Be mindful of the indentation in the next block of code. Notice that we start blocks using a colon

```
x = 0
y = 0

# nested conditional
if x == 0 and y == 0:
  x += 1
  y += 1
  if y == 1:
    y += 1

# y == 2 at this point
```

While-loops are very similar to C/C++. There is no `do...while` statement in Python and so code will often be written in the following manner:

```
x = 0

while True:
  if x < 5:
    x += 1
  elif x <= 15:
    x += 2
  else:
    break

# x == 17 at the end of this loop
```

For loops are a different beast in Python. A C style loop has three parts, but a for loop in python only does has a comparison.

```
// C For loop
int i, x = 0;
for (i = 0; i < 10; i++){
  x += 10;
}

# Python For loop
x = 10
for i in range(0, 10, 1):
  x += 10
```

Although not all arguments are needed for the built in method `range()`, to parallel C, we explicitly state each argument. Python uses this style of for loops so that iterating over anything (such as a list or a string) becomes much cleaner to read. We'll see an example of this in the next section. In this manner, while loops have a much clearer use case, to iterate over a sequence arbitrarily. For loops iterate over a sequence at a fixed range.

3.3 Strings

Strings in python are of the type `str`. You should name your string variables more meaningfully than `str`, but for Python, `str` is actually a reserved word. You can just think of strings as a list of characters, much like strings are an array of characters in C/C++. Strings can be defined either using the single or double quote notation. You can also use a triple single or triple double quote, but for our purposes we will only use that for docstrings. We use the built-in method `print()` in order to print things to the console.

```
>>> department = "COEN "
>>> course = "146"

>>> print (department + course)
"COEN 146"

# convert the string to an integer
>>> x = int(course) + 1
>>> print (x)
147
>>> print (department + x)  # ERROR: must explicitly cast int -> str
>>> print (department + str(x))
"COEN 147"

# testing for membership
if "CO" in department:
  print ("Right department")
else:
  print ("Wrong department")

if "146" not in department:
  print ("Wrong class")
else:
  print ("Right class")

>>> print (department[0])
"C"
>>> print (department[-1])
" "
```

```
>>> print (department.strip() + course)
"COEN146"
if department[0:2] == "CO":
 print("Great department")

>>>print ("I am in {}L right now".format(department + course)
I am in COEN 146L right now
```

Notice the use for the string method `format()` in the last statement. It's similar to the method `printf()` in C. The difference is that we don't specify the type so we can just use curly braces as the placeholders. There are tons of built in methods that you can check out for strings.

3.4 Lists and Tuples

The closest thing to arrays from C are lists in Python. They are mutable (can be changed) and don't have a fixed size. Lists, just like strings have a quite a few useful methods available to them. They include methods like `find()`, `append()`, `pop()`, and `count()`. These can be looked up in your own time. The main syntax that we will review is working with ranges and iterating over lists.

```
>>> class_grades = [90, 100, 83, 92, 97, 85, 83, 98] #defining a list
>>> class_grades[0:5]
[90, 100, 83, 92, 97]
>>>class_grades[:-3]
[90, 100, 83, 92, 97]
>>> class_grades[-1:-5]

# safe iteration over list without using ranges (No off by one errors)
for grade in grades:
 if grade < 90:
     grade += 5
print (grades)
```

A similar concept is tuples, which are immutable. You cannot change a tuple once it's been created. This is especially useful when you don't want someone messing with values accidentally. Quite a few functions we will use in this lab return tuples. They are positionally based just like lists and all the indexing and iteration covered in the last example works just the same with tuples. Notice that tuples are created using `()` instead of `[]` like lists.

```
>>> class_grades = (90, 100, 83, 92, 97, 85, 83, 98) #defining a tuple
>>> class_grades[0:5]
[90, 100, 83, 92, 97]
>>>class_grades[:-3]
[90, 100, 83, 92, 97]
>>> class_grades[-1:-5]
```

```
# safe iteration over list without using ranges (No off by one errors)
for grade in class_grades:
if grade < 90:
    grade += 5
print(grade)         # okay, but does not save changes back into class_grades

# illegal, cannot change values of a tuple
class_grades[1] += 1       # ERROR
```

3.5 Functions

In using Python in a procedural manner, functions are not much different than C functions. Python is a pass by argument language. This means that mutable types (like lists) are pass by reference. Immutable types, you won't be able to change them regardless (like strings and tuples). One caveat is that if you shadow variable names, the outer scope will have no idea about the changes you supposedly made to the argument passed in. If you don't want to mess with this, you can always just return tuples out of your function.

To start creating a function, we choose its scope first based on whitespace indentation. Then use the `def` keyword, followed by the name, the list of arguments and then a colon indicating the start of the block. For our cases, you'll probably create everything in the global scope. Notice the triple double quotes which start the docstring for our function. Documentation is important. You'll get marked down if your code doesn't have documentation. The functions we have are pretty simple and so one line docstrings and good variable names explain most of our code

```
# nice and pure function, doesn't mess with arguments
def add(x, y):
 """Add x and y."""
 return x + y

# lists are mutable so the function knows what to do
def add_to_list(int_list, x):
 """Add x to every integer in int_list."""
    for i in int_list:
     i += x

# we try to rebind the name (shadowing occurs) so this function has no effect
def mess_up_list(int_list):
 """Assign new values to the given list."""
 int_list = [1, 2, 3, 4]

# can't change immutable types
def add_to_tuple(point_tuple, x):
```

```
    """Add x to every integer in point_tuple."""
    for dimension in point_tuple:
     dimension += x
```

With all the boring stuff out of the way, Python has some interesting features with functions. First off, there are two types of arguments:

1. Positional Arguments - values are assigned based on their position in the list of parameters
2. Keyword Arguments - values are assigned either by position or by explicitly binding variables. Default values must be provided.

So if we have a bunch of required parameters, we'd could make them all positional. If we have a bunch of optional arguments or things that can easily be defaulted, we can use keyword arguments. Because positional arguments are required, they must always come before keyword arguments.

```
# only positional arguments
def add(x, y):
"""Add x and y."""
    return x + y

# a keyword argument mixed with positional
def increment_list(int_list, x = 1):
"""Add x to every integer in int_list."""
for i in int_list:
     i += x

# illegal function, keyword arguments must come after positional
def illegal(x = 1, int_list):
"""Add x to every integer in int_list."""
for i in int_list:
     i += x

# valid function calls
z = add(3, 4)

sequence = [4, 5, 6]
increment_list(sequence) # [5, 6, 7]
increment_list(sequence, x = 2) # [7, 8, 9]
increment_list(sequence, 2) # [9, 10, 11]
```

The other thing to touch on is that return tuples is really common practice in Python. This is similar to passing in a struct by reference in C. It's just way easier. In this way, we can return multiple values out of a function or we can create tuples out of new ones. In the following function, you'll see a slightly larger docstring since the function is a little bit more complex to understand.

```
def rotate_point(point):
    """
    Rotate a point in 3 dimensional space.

    Positional Arguments:
     point - a 3-tuple containing the points to be rotate

    Returns:
     rotated_point - 3-tuple, respecting order of original point.
         If point cannot be rotated, the same point will be returned
    """

    # entire tuple is wrapped in parentheses and comma separated
    if len(point) > 2:
      return (point[0] * -1, point[1] * -2, point[2] * -2)
    else:
      return point
```

3.6 Input and Output

3.6.1 Command Line

Since everyone is using Python3 for this lab, we can just use the `input()` method. This function prompts the user on the command line given some arguments. The return value of this function is a string denoting what the user input. If you expect it to be an integer, you should safely convert it to one.

You can print to the command line using the `print()` method. Remember that you cannot print a string and int together by adding them together, you need to explicitly cast everything to a string if you're combining types. You can also use the `format()` method. Most built-in types are printable although printing functions might give you a pointer index.

```
>>> name = input("What is your name?")
>>> age = input('What is your age?')
>>> print (name + " " + str(age))
>>> print ("Hi {}. You are {} years old".format(name, age))
```

To access command line arguments when executing your Python scripts. You'll need to import the `sys` library. There are a lot of modules in the Python standard library. You won't need to use most of them for this quarter, mainly the `socket` library.

```
"""arguments.py prints all given arguments to the command line"""
import sys #importing a standard library

# the 0th argument in sys.argv will be the script name
```

```
for argument in sys.argv:
print (argument)

$ python arguments.py first second third
arguments.py first second third
```

3.6.2 File IO

Instead of checking if File pointers are null like in C, in Python, the preferred method is to wrap everything in a `with...as` block. You can iterate through the given file line by line just using a for loop and everything's magic. Yes, there are other ways to do File IO, but please just use this and make life easy on everyone. The `with function as name` block wraps the opening and closing of the file and assigns the given name to the return value of the supplied function.

```
# open(path_to_file, flags) --> flags are the same as in C
with open('file_name', 'r') as f:
for line in f:
     print(line)

# writing to a file
classes = [146, 171, 174, 175]
with open('file_name', 'w') as f:
for class in classes:
     f.write('COEN {}'.format(class))
```

3.7 Additional Resources

Here are some additional tutorials on the basics of Python:

1. Google's Python Class - `https://developers.google.com/edu/python/`
2. Learn Python the Hard Way - `https://learnpythonthehardway.org/book/`
3. Hitchiker's Guide to Python - `http://docs.python-guide.org/en/latest/intro/learning/`
4. Official Python Resources - `https://wiki.python.org/moin/BeginnersGuide/Programmers`

4 Multiplexing and Demultiplexing

Multiplexing is a useful way to combine multiple streams of data into just one. Demultiplexing is the reverse process, taking one stream of data and breaking it into multiple. When we communicate over a network, we'll often only have one interface to use such as your Wireless adapter or your Ethernet connection.

This one interface is responsible for sending and receiving all types of data from different applications on your computer across a network link.

The coding section of this lab will consist of you writing a multiplexing function and demultiplexing function. Your multiplexing function will take in a list of messages and write all the messages to a file. Your demultiplexing function will reverse this operation by taking in a file name and then outputting lists of messages.

4.1 Multiplexing

You can implement one of two types of multiplexing: Time Division Multiplexing or Statistical Time Division Multiplexing. Both types of multiplexing allot a certain amount of time (we'll call it "ticks") to each machine. Each machine writes out as much data as possible in each tick. The difference in the two methods comes when a machine does not need to write any messages. In Time Division Multiplexing, the machine will still use up the allocated time on the network interface. In Statistical Time Division Multiplexing, the machine will inform the network interface that it has no messages.

Your input will be a list of list of strings. Your output will be the number of messages written. Each list of strings will represent some messages that need to be sent by a Linux DC Machine. Each "tick" will be the writing of one message to the file. So we share this file (our single interface for communcation) by allowing each machine to write only at certain times.

In order to distinguish which machine sent this message, each line in the file should begin with the 5 digit number of the machine followed by a colon. The machine number is assigned as '604xx' where 'xx' is the index of the list of messages from the function input. After the colon, the actual message sent by that machine will be written. This should continue only until all messages from all machines have been written. This is common to both types of multiplexing you can write for this lab. The difference between the two types of multiplexing comes when a machine has "ticks" leftover.

1. In **Time Division Multiplexing**: Each machine will always write 5 messages regardless of it a message needs to be sent or not. When a message doesn't need to be sent, the machine will just write out "No messages" for every tick it does not use.

2. In **Statistical Time Division Multiplexing**: Each machine will write up to 5 messages. When a message doesn't need to be sent, the machine will write out "No messages" just once, and the program will move on to writing out the next machines messages.

That may be a bit confusing, so let's go through an example really quickly. If we have the following input:

```
[
    ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l"],
```

```
    ["1", "2", "3"]
]
```

Notice that the list of letters is at the 0th index of the array. All lines with letters should therefore be prefixed with "60400:". All lines with numbers should be prefixed with "60401:".

We'll get the following output written to the file for **Time Division Multiplexing**:

```
60400: a
60400: b
60400: c
60400: d
60400: e
60401: 1
60401: 2
60401: 3
60401: No messages
60401: No messages
60400: f
60400: g
60400: h
60400: i
60400: j
60401: No messages
60401: No messages
60401: No messages
60401: No messages
60401: No messages
60400: k
60400: l
60400: No messages
60400: No messages
60400: No messages
60401: No messages
60401: No messages
60401: No messages
60401: No messages
60401: No messages
```

We'll get the following output written to the file for **Statistical Time Division Multiplexing**:

```
60400: a
60400: b
60400: c
```

```
60400: d
60400: e
60401: 1
60401: 2
60401: 3
60401: No messages
60400: f
60400: g
60400: h
60400: i
60400: j
60401: No messages
60400: k
60400: l
60400: No messages
60401: No messages
```

As you can see, Statistical Time Division Multiplexing only allots as much time as needed for each machine. The output is a lot shorter, and the longer list of letters is transferred much more quickly.

4.2 Demultiplexing

The type of demlutiplexing that we will use is based off a Frequency Division Multiplexing. In Frequency Division Multiplexing, the data is divided into different non-overlapping channels. Think of your radio, where 99.7 FM transmits over a different frequency than 94.9 FM. They both use the same radio, but you only get the audio that comes at a certain frequency. For the lab, you can imagine that each of the machine numbers is a different channel.

Your input will be just a file name. Your output will be a tuple of lists of messages. You should safely open this file and then determine which machine it should go to. Once you start reading the file, you should store messages to machines 60400, 60401, 60402, 60403 in 4 separate lists. Any messages belonging to other machines should go into a 5th list. The machine number of a message is the first 5 characters of the line (which should be 604xx). If the message sent is 'No messages', then no message should be added to any list. In all other cases, the message should go to one of the bins. All 5 lists should be returned at the end.

Given the following input:

```
60400: 0
60400: 0
60400: 0
60400: 0
60400: 0
60401: 1
```

```
60410: 10
60405: 5
60405: 5
60406: 6
60400: No messages
```

You should return the following tuple:

```
(
    ["0", "0", "0", "0", "0"],
    ["1"],
    [],
    [],
    [10, 5, 5, 6]
)
```

Notice that the we returned empty lists for machines 60402 and 60403. They did not have any messages. All the messages from machines 60405, 60406, and 60410 were thrown into the last bin.

5 Submission Guidelines

For this lab, you will have the following deliverable:

1. lab1.py - Your code from section 4 of this handout

Please make sure that you have proper indentation and are using Python3. The Linux computers by default have Python 2.6 installed, which may be different from Python 2.7 which your Mac might have installed. For information on sideloading Python3 onto the Linux computers, please consult your TA.

You code should look organized and have docstrings for all functions. In addition, your variable names should be meaningful. If you're looking for a good coding standard for Python that makes sure you'll get full points on style, Google for PEP 8.