Lab 1: July 10, 2019

Subjects to review this week: basics of Python, variables, expressions, data/variable types, branching/conditional expressions, loops

Learning outcomes:

1. Define key terms and ideas from Ch. 1-5 of course lectures
2. Describe and use different data types and understand when/where they are useful
3. Confirm understanding of how Python objects are held in memory
4. Gain experience trouble-shooting code
5. Develop code using basic Python skills to solve simple programming problems

Big picture: the goal this week is to get you comfortable coding in Python, with a proper understanding of the building blocks of object-oriented programming. We will build on this foundation in the coming weeks, using Python as a tool to accomplish other applied data tasks (data munging, statistical analysis, web-scraping, etc.)

Part I: Key terms (~10 minutes)

Please write out (on paper, in .txt file, etc.) a couple of sentences highlighting the most important or course-relevant aspects of these terms:

- Compiled vs. interpreted code
- Syntax error vs. runtime error vs. logic error
- Variable
- Expression
- Object
- Module (vs. main script)
- Mutable vs. immutable object
- Comparison operators (equality vs. relational)
- While loop
- For loop

Part II: Data types (~20 minutes)

This series of short exercises[1] aims to ensure you're familiar with the basics of each main data type discussed in lecture. These include strings, numeric values, Booleans, lists, tuples, and dictionaries. You will write a python script that accomplishes the following:

1. Create a tuple called *my_tuple* with 5 elements: (i) a string, (ii) a Boolean, (iii) an integer, (iv) a float numeric, and (v) a nested tuple that contains 5 integers.
2. Use the print function to display the 3rd element of *my_tuple*'s nested tuple.
3. In a single line of code, unpack *my_tuple* into five separate variables (*var1, …, var5)*
4. Create *short_tuple*, a tuple that only contains elements (i), (iii), and (iv) of *my_tuple*. Do this by extracting the relevant elements, not manually creating a new tuple.
5. Now create a list, *my_list*, that contains the following 6 tuples: (*"a"*, 1), (*"b"*, 2), (*"c"*, 3), (*"d"*, 1), (*"e"*, 2), (*"f"*, 1).
6. Convert *my_list* into a dictionary, *my_dict*, with each tuple's letter representing a key, and its number a value (*Hint: use Python's setdefault method*)
7. Create a new 6-element list (called *letter_list)* that contains only the string elements of my_list; do this by extracting the relevant elements, not manually creating a new list.

[1] Adapted from w3resource.

8. Next, you are given the following code that produces a dictionary, *student_data*:

```
student_data = {'id1':
   {'name': ['Sara'],
    'class': ['V'],
    'subject_integration': ['english, math, science']
   },
'id2':
  {'name': ['David'],
    'class': ['V'],
    'subject_integration': ['english, math, science']
   },
 'id3':
    {'name': ['Sara'],
    'class': ['V'],
    'subject_integration': ['english, math, science']
   },
 'id4':
   {'name': ['Surya'],
    'class': ['V'],
    'subject_integration': ['english, math, science']
   },
}
```

   a. What are the keys for *student_data*?
   b. Write code to produce a duplicate-free version of *student_data*

9. Finally, create two sets.
   a. Let *set_a* contain the elements "green", "yellow", "red", "black", and "pink".
   b. Let *set_b* contain the elements "purple", "green", "brown", "black", "beige", and "yellow".
   c. Extracting from these two original sets, create a new set, *set_c*, that is the intersection of *set_a* and *set_b*.

Part III Tracing code – loops (~10 minutes)

You have two snippets of Python code below. Your task is simple, **mentally work through the code and determine what the printed output will be**. While completing this, think carefully about how objects are being stored in the memory.

1. Simple while loops:

```
n = 1
itr = 0
while n < 5:
    m = n + 1
    while m < 4:
        m += 1
    if m == 5:
        n = 7
    n += 1
    itr += 1
print("The final product of n and m is", n*m)
print("This code ran for", itr, "iterations")
```

2. Sorting a list using for loops

```
grade_list = [89, 45, 85, 81, 77, 94, 22, 79, 92, 91]
n = len(grade_list)     # makes it easy to denote the
length of the list

for i in range(n - 1):
    smallest = i

    for j in range(i, n):
        if grade_list[j] < grade_list[smallest]:
            smallest = j


    temp = grade_list[i]
    grade_list[i] = grade_list[smallest]
    grade_list[smallest] = temp

print("Grades after sorting: " + str(grade_list[::-1]) )
```

Part IV Trouble-shooting code (~20 minutes)

In these exercises[2], you are provided with some snippets of code. They are full of *missttakes*! Your job is copy the code over to your IDE of choice, debug it, and then run it. Take care to understand what your code is supposed to be doing… remember, if only a logical error exists in the code, the IDE will not report an error!

1. Pirates (6 errors?)

```
greeting = input("Hello, possible pirate! What's the password?)
if greeting in ["Arrr!")
  print "Go away, pirate."
elif
print("Greetings, hater of pirates!")
```

2. Collections (11 errors?)

```
authrs = {
    "Charles Dickens": "1870";
    "William Thackeray": "1863";
    "Anthony Trollope": "1882";
    "Gerard Manley Hopkins": "1889"

for author date in authors.items{}:
    print "%s" % authors + " died in " + "%d." % Date
}
```

[2] Code examples taken from Humanities Programming.

3. Grading (More than 10 errors, count them yourself!)

```
# Write a program that will average 3 numeric exam grades,
return an average test score, a corresponding letter grade, and
a message stating whether the student is passing.

# 90+ (A), 80-89 (B), 70-79 (C), 60-69 (D), 0-59 (F)

exam_one = int(input("Input exam grade one: "))
exam_two = input("Input exam grade two: "))
exam_3 = str(input("Input exam grade three: "))

grades = [exam_one exam_two exam_three]
sum_gr = 0
for grade in grade:
sum_gr = sum_gr + grade

avg = sum_gr / len(grdes)

if avg >= 90:
    letter_grade = "A"
elif avg >= 80 and avg < 90
    letter_grade = "B"
elif avg > 69 and avg < 80:
    letter_grade = "C'
elif avg <= 69 and avg >= 65:
    letter_grade = "D"
elif:
    letter_grade = "F"

for grade in grades:
    print("Exam: " + str(grade))
    print("Average: " + str(avg))
    print("Grade: " + letter_grade)

if letter-grade is "F":
    print "Student is failing."
else:
    print "Student is passing."
```

Part V: Project Euler (~1 hour)

Project Euler is a set of mathematical problems that many programmers use to develop their coding skills in a new language. These 650+ problems range in complexity: the easiest require a single line of code while others may require several dozen lines of code on a coder's first solution attempt. You can find out more here: https://projecteuler.net/about.

What is nice about these problems is that there are many potential code solutions. As long as your code gets you to the answer, you're doing all right! Of course, there are more and less elegant ways to use a programming language's logic to your advantage, and as you improve in a language, the method you use to solve a problem is likely to evolve. (Note: you can see why this standardized set of problems is useful to programmers- solving a familiar problem in a new language makes it easy to quickly understand a language's nuances).

Today, we are going to solve 6 of the more straightforward problems. They proceed from easier to harder, and can be solved using only the programming skills we've developed so far in lectures – ideally, none should require more than 10 lines of code or 10 minutes of thought.

PLEASE DO NOT LOOK UP SOLUTIONS! TRY TO WRITE YOUR OWN CODE TO SOLVE THESE PROBLEMS… you will greatly improve your programming intuition in the next ~60 minutes trying to solve these problems yourself, even if it requires a bit of struggle. First develop pseudo-code (in your head or on screen), then try to implement it in Python. Troubleshoot as needed until you get to the correct answer.

1. (P.E. Question 1) If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

   Find the sum of all the multiples of 3 or 5 below 1000.

2. (P.E. Question 6) The sum of the squares of the first ten natural numbers is

   $$1^2 + 2^2 + ... + 10^2 = 385.$$

   The square of the sum of the first ten natural numbers is

   $$(1 + 2 + ... + 10)^2 = 55^2 = 3025$$

   Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$.

   Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

3. (P.E. Question 48) The series, $1^1 + 2^2 + 3^3 + ... + 10^{10} = 10405071317$.

   Find the last ten digits of the series, $1^1 + 2^2 + 3^3 + ... + 1000^{1000}$.

4. (P.E. Question 9) A Pythagorean triplet is a set of three natural numbers, $a < b < c$, for which

   $$a^2 + b^2 = c^2$$

   For example, $3^2 + 4^2 = 9 + 16 = 25 = 5^2$.

   There exists exactly one Pythagorean triplet for which $a + b + c = 1000$.

   For this triplet, find the product abc.

5. (P.E. Question 4) A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.

   Find the largest palindrome made from the product of two 3-digit numbers.

6. (P.E. Question 19) ***How many Sundays fell on the first of the month during the twentieth century (1 Jan 1901 to 31 Dec 2000)?***

Hint: use Python's calendar method (type import calender at the head of your code). The function monthcalender requires you to input (year, month) and returns as output a usable matrix list from which you can extract the 1st of the month's day of week.