

CS-119 Lab #5

Expected Learning Objectives

- Creating applications in Python.
- Python repetition (loop) structures.
- Selection structures.
- Arrays and lists.
- Parallel arrays.
- Searching an array for a value.
- Random numbers.

Overview

This lab provides an opportunity to apply the concepts and principles presented in PLD chapter 5.

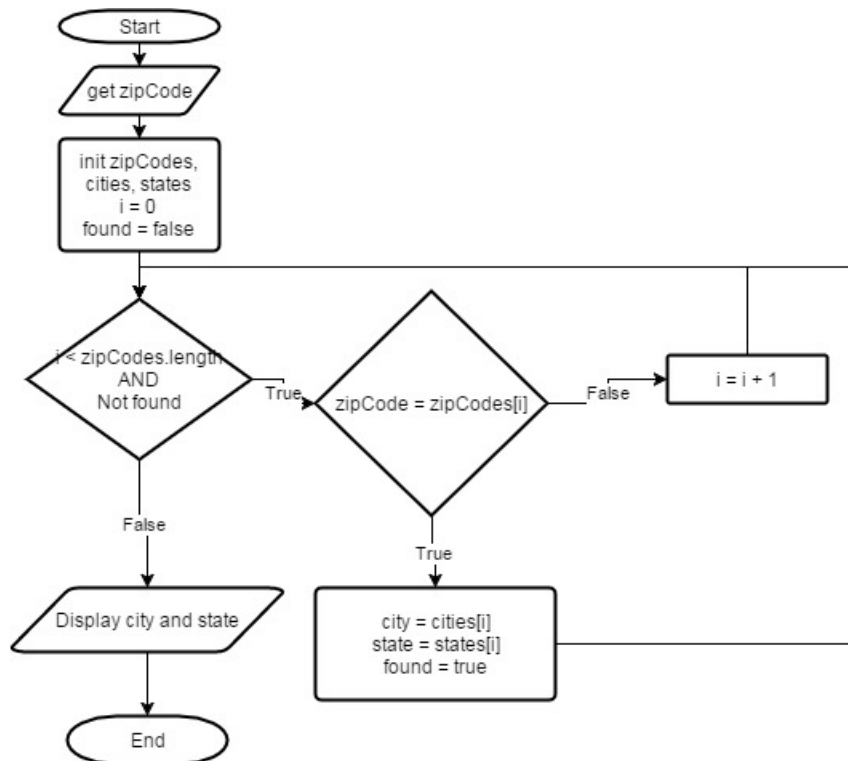
You are given the choice of doing either pseudocode or a flowchart for each exercise. In the guided exercises, we will mix it up and do the first exercise with a flow chart and the second with pseudocode.

Exercise 1 Guided Zip Code Lookup

The first case study is taken from page 199 in the PLD text. We're only going to do the part where the user enters a zip code. If the zip code is found, the program displays the city and state. This is something we've seen and done while ordering items on a web site or filling out an online form.

We're going to focus mainly on things that are new so there won't be a whole lot of detail on creating a program file in Python, etc. In this case, we'll focus on creating arrays, coding an efficient search loop, and accessing the elements of an array.

Analysis: The user enters a zip code. Search the array using a loop. If the zip code is found, return the corresponding city and state from the parallel arrays. Quick review from the book: If we search an array and find a match at subscript n , we can get the corresponding values from the parallel arrays at subscript n as well. We can do either pseudocode or a flowchart. Sample flowchart:



You can do this in pseudocode if you prefer.

Note in the loop condition, we have 2 conditions: one to stay in the bounds of the array and the other is a search flag called *found*. Note the found flag is initially set false and gets set to true if we get a match. This makes our search more efficient. Why continue searching for something if we already found it? To avoid a selection structure after the loop, we can initialize our city and state variables to something like “NOT FOUND” when we declare them.

In Python, create a new file called ArraySearchExercise1. Be sure to save in your lab 5 folder.

To save time and make things a little easier, we’ll do just 4 zip codes and the corresponding cities and states.

```
#!/user/bin/env python3

# Cuyamaca College CS-119
# Your name here!
# Lab 5 exercise 1 parallel array search

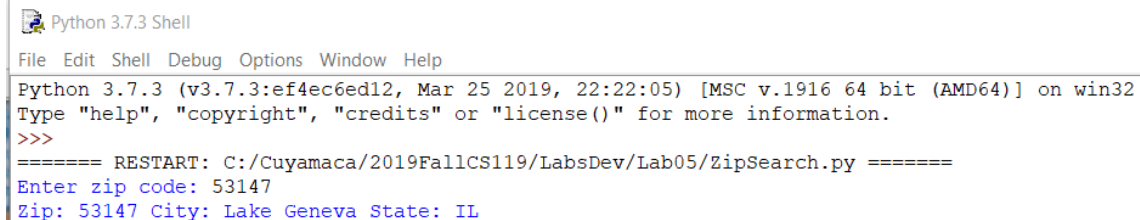
# Create parallel lists of zip codes and the corresponding cities & states
zips = [53115, 53125, 53147, 53184]
cities = ["Delevan", "Fontana", "Lake Geneva", "Walworth"]
states = ["WI", "WI", "IL", "IL"]

# other variables we need
zip_code = 0
city = "NOT FOUND"
state = ""
found = False
i = 0
list_len = len(zips) # get the array size for the upper bound of the loop

# input - get zip code
zip_code = int(input("Enter zip code: "))

# search the array
while i < list_len and not found: # don't forget the colon (:) on a loop condition
    # test for a zip code match and indent exactly 4 spaces for code in a loop body
    if zip_code == zips[i]: # Don't forget the colon (:) at the end if the if condition
        # found a match and be sure to indent exactly 4 spaces in the if true block
        city = cities[i]
        state = states[i]
        found = True
    i = i + 1 # be sure to increment the loop counter
# end while

# output
print("Zip: " + str(zip_code) + " City: " + city + " State: " + state)
```



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Cuyamaca/2019FallCS119/LabsDev/Lab05/ZipSearch.py =====
Enter zip code: 53147
Zip: 53147 City: Lake Geneva State: IL
```

Be sure to test with an invalid zip code:

```
===== RESTART: C:/Cuyamaca/2019FallCS119/LabsDev/Lab05/ZipSearch.py =====
Enter zip code: 91910
Zip: 91910 City: NOT FOUND State:
>>>
```

Notes on the code above:

- Initializing city to “NOT FOUND” and state to a blank string eliminates need for a selection structure after the loop.
- Note the compound condition in the while loop and use of the Python *not* operator.
- Watch your indentation closely with Python loops and selection structures
- Be sure to code the colon (:) after loop and selection structure conditions.

Exercise 2 Guided Dice Game

This one is for all you aspiring game programmers to give you a taste and appreciation of what it's like to program even a simple computer game. We're going to design and code a dice game or scaled down version of Yahtzee if you will. This exercise is based largely off exercise 15 on page 197 of the PLD text. Here's how it will work:

- Player (person) against the computer.
- Each player will roll 5 dice.
- Find highest dice combination rolled such as 5 of a kind, 4 of a kind, 3 of a kind, or pair.
- Display value of each roll.
- To keep things simple for now, play 1 round.

Random numbers are used in games for dice rolls, dealing cards, etc. so that's what will be used for the dice rolls.

Analysis

This time we'll use Pseudocode although you can do a flowchart if you'd like some extra challenge and practice with those. If you created a pseudocode file in exercise 1, you may add your pseudocode for this exercise. Otherwise, go ahead and create a file for your pseudocode now. Being a guided exercise, you are given a screen shot of the pseudocode.

```

// input

roll_types = { Junk, Pair, 3 of a kind, 4 of a kind, 5 of a kind }

# less detailed approach
get 5 dice rolls for the computer
get 5 dice rolls for the player

# OR more detailed pseudocode approach
for i = 0 to 4
    roll pDice[i] for player
    roll cDice[i] for computer
    display pDice[i] and cDice[i]
end for

# Which of the above would be easier to write code from?

# process - figure out what got rolled and who won

display what the player and computer rolled

# tally up the dice rolls so we can figure out pair, 3 of a kind, etc.
while i < MAX_ROLLS
    pTally[ pDice[i] - 1 ]++
    cTally[ cDice[i] - 1 ]++
end while

# now find the highest roll combination (pair, 3 of a kind, etc.)
pMax = pTally[0]
cMax = cTally[0]

i = 1
while i < MAX_DICE_VAL
    if pMax < pTally[i] then
        pMax = pTally[i]
    end if

    if cMax < cTally[i] then
        cMax = cTally[i]
    end if
    i = i + 1
end while

# output display the type of roll, who won the round
display rollTypes[ pMax - 1 ] for the player
display rollTypes[ cMax - 1 ] for the computer

if pMax > cMax then
    display player wins
else if cMax > pMax then
    display computer wins
else
    display tie
end

```

Code the Application

Create a new Python code file called DiceExercise2.py. Being a guided exercise, you are given screen shots of the code you need to write.

Things to watch out for as you write your code:

- Make sure you import random.
- Watch indentation (4 spaces) in loops and selection structures.
- In Python, we're actually using lists but we still need to watch the lower bound (0) and make sure we're staying within the upper bound.

```
#!/user/bin/env python3

# Cuyamaca College CS-119
# Your name here!
# Lab 5 dice game

import random

# variables and constants
MAX_ROLLS = 5
MAX_DICE_VAL = 6

# declare a list for roll types
roll_types = ["Junk", "Pair", "3 of a kind", "4 of a kind", "5 of a kind"]

# declare lists for dice rolls, and tallying up the dice rolls
pdice = [0,0,0,0,0]
cdice = [0,0,0,0,0]
# tally lists will be 6 in size since dice values run from 1 - 6
ptally = [0,0,0,0,0,0]
ctally = [0,0,0,0,0,0]

# roll the dice for player and computer
i = 0
while i < MAX_ROLLS:
    roll_val = random.randint(1,MAX_DICE_VAL)
    pdice[i] = roll_val
    roll_val = random.randint(1,MAX_DICE_VAL)
    cdice[i] = roll_val
    i += 1

# Print the rolls for player and computer
# This is good opportunity to test and debug what you have coded so far
# Make sure you have values for 5 dice and each value is in the range 1 - 6

# Display what the player rolled
i = 0
print("Player rolled: ", end=" ")
while i < MAX_ROLLS:
    print(pdice[i], end=" ")
    i += 1

# Display what the computer rolled
i = 0
print("\nComputer rolled: ", end=" ")
while i < MAX_ROLLS:
    print(cdice[i], end=" ")
    i += 1
```

```

# Load the tally list so we can determine pair, 3 of a kind, etc.
# Notice here how we use the dice value - 1 as an array subscript. Pretty clever!
# Good application from the chapter how we can use an array (well, list in Python)
# to replace a selection structure.
i = 0
while i < MAX_ROLLS:
    ptally[ pdice[i] - 1] += 1
    ctally[ cdice[i] - 1] += 1
    i += 1

# next step: find out how many 2's, 3's, 4's, etc. to determine pair, 3 of a kind, etc.
pmax = ptally[0]
cmax = ctally[0]

i = 1
while i < MAX_DICE_VAL:
    if pmax < ptally[i]:
        pmax = ptally[i]
    # end if

    if cmax < ctally[i]:
        cmax = ctally[i]
    # end if
    i += 1
# end loop

# output - display pair, 3 of a kind, etc. Once again, we use an array (list) to
# replace a selection structure.
print("\nPlayer rolled: " + roll_types[pmax - 1] )
print("Computer rolled: " + roll_types[cmax - 1] )

# determine the winner
if pmax > cmax:
    print("Player wins!")
elif cmax > pmax:
    print("Computer wins!")
else:
    print("TIE!")

```

Exercise 3 Shipping Charge Calculator

You've probably seen these in online shopping applications: Get an estimated shipping cost based on the zip code your order has to ship to. Price will vary depending on if it's shipped via Coyote Express or Coyote Ground.

Develop a Python application that will allow the user to enter a zip code and shipping method. You must then look up the zip code and return the correct city and shipping charge based on the user's shipping choice. You can shorten the shipping choice to "Express" and "Ground" if you prefer.

You *must* use either a Python array or list for the zip codes and shipping charges. If the user enters an invalid zip code, your application must display an appropriate "Zip code not found" message

Zip Code	92020	91901	92040	91976	92071
City	El Cajon	Alpine	Lakeside	Spring Valley	Santee
Express	\$5.00	\$10.00	7.00	\$6.00	\$7.00
Ground	\$3.00	\$5.00	\$4.00	\$3.00	\$4.00

Hint: This exercise is similar to guided exercise 1 so you will find quite a bit of code you can "borrow".

Before writing any code, be sure to complete either pseudocode or a flowchart. This is worth points!

Create a Python code file called ShippingChargeExercise3 for your program code.

Exercise 4 Dice Game (Revisited)

In this exercise, you will make the following modifications to the dice game from exercise 2:

1. Prompt the user for the number of rounds they want to play.
2. Code a loop to play the user entered number of rounds.
3. Keep score for the player and computer.
4. At the end, display who won the game after all rounds have been played (player, computer or tie).

You must complete exercise 2 before starting this. Save a copy of the DiceGameExercise2.py as DiceGameExercise4.py. You may copy the pseudocode or flowchart you did in exercise 2 and add/modify things as needed.

Grading Criteria:

Deliverable	Points	Breakdown
Exercise 1 Zip code guided pseudocode or flowchart	5	Completed. Inputs, processing and outputs are clearly defined. In general, logic “makes sense”. If flowchart, proper use of symbols and flow lines.
Exercise 1 Zip code guided code	5	Complete, code “makes sense”, appropriate use of comments, clear variable names and constants, results are correct.
Exercise 2 Dice game flowchart or pseudocode	5	Completed. Inputs, processing and outputs are clearly defined. In general, logic “makes sense”. If flowchart, proper use of symbols and flow lines.
Exercise 2 Dice game code and run	5	Complete, code “makes sense”, appropriate use of comments, clear variable names and constants, results are correct.
Exercise 3 Shipping charge flowchart or pseudocode	5	Completed. Inputs, processing and outputs are clearly defined. In general, logic “makes sense”. If flowchart, proper use of symbols and flow lines.
Exercise 3 Shipping charge code and run	10	Complete, code “makes sense”, appropriate use of comments, clear variable names and constants, results are correct.
Exercise 4 Dice version 2 flowchart or pseudocode	5	Completed. Inputs, processing and outputs are clearly defined. In general, logic “makes sense”. If flowchart, proper use of symbols and flow lines.
Exercise 4 Dice version 2 code and run	10	Complete, code “makes sense”, appropriate use of comments, clear variable names and constants, results are correct.
Lab Total	50	