Lab 09: Recursion

Goals for this lab:

- Design and implement recursive functions
- Identify multiple possible base cases in a recursive function
- Write and execute Python programs using the given instructions

General Guidelines:

- Use meaningful variable names
- Use appropriate indentation
- Use comments, especially for the header, variables, and blocks of code.

Example Header:

```
# Author: Your name
# Date: Today's date
# Description: A small description in your own words that describes what
the program does. Any additional information required for the program to
execute should also be provided.
```

Example Function Docstring:

```
"""
General function description
:param p1: DESCRIPTION
:type p1: TYPE
:param p2: DESCRIPTION
:type p2: TYPE
:return: DESCRIPTION.
"""
```

1. Recursion

Inheritance and polymorphism allow software developers to reuse code in multiple different ways. By defining superclasses, subclasses can reuse functions and variables that have been previously defined, and override other functions that may need more specific functionality for a particular subclass. We just need to be careful about how data is stored and accessed when using data structures that store objects of multiple different classes.

For this exercise you will be writing a program to calculate the result of a user specified base value being put to a user specified exponent.

To complete this program:

- In a file named **powers.py**,
  - Define a recursive function named `powers` that takes in two parameter variables, one the represents the base value and one that represents the exponent value, will return the product of the base and the recursive function call, and should:
    - Print out the following string "`powers(base,exponent)`" where base is the base parameter variable and exponent is the exponent parameter variable. This will help us see how the values are changing at each step of recursion
    - Specify a base case, such that if the value of exponent is 1, return the value of base
    - Specify the recursive step, such that product of the base and the result of a new call to the `powers` function is returned
      - Note you should pass the `powers` function the original base value and the exponent value – 1
  - Define a `main` function, which should:
    - Prompt the user for an integer representing the base value, and store it in a new variable
    - Prompt the user for an integer representing the exponent value, and store it in a new variable
    - Call the powers function, passing in the base and exponent values, and store the result in a new variable
    - Using an appropriate message, output to the user their base, their exponent, and the resulting value

Complete the program, making sure to execute it to verify that it produces the correct results. Note that you will submit the **powers.py** file to Canvas.

2. Iterative Palindromes

A palindrome is a word or phrase that reads the same both forwards and backwards. For example, “radar” can be read in either direction, as can “rotator”.

For this exercise you will be writing a program to determine if a user provided word is a palindrome using loops.

To complete this program:

- In a file named **palindromeIter.py**,
  - Define a function named `palinTest()` that takes in one parameter variable representing the string to test, will return a Boolean, and should:
    - Using a loop, iterate through the characters of the string to determine if it is a palindrome
      - Remember that a palindrome should have the same characters on both ends, so you should compare the first and last character, then the second and second to last characters, then the third and third to last characters, etc..
    - If any of the characters do not match, return `False`
    - Otherwise, return `True`
  - Define a `main` function, which should:
    - Prompt the user for word to test using an appropriate message
    - Call the `palinTest()` function, passing in the user’s word, and use the returned Boolean to output a message stating whether the word is or is not a palindrome.

Complete the program, making sure to execute it to verify that it produces the correct results. Note that you will submit the **palindromeIter.py** file to Canvas.

3. Recursive Palindromes

What we sometimes find is that the same problem can be solved either iteratively using loops, or recursively using functions. Palindrome detection is just such a problem, but we need to make sure we setup out base case and iterative step correctly. Starting with the base case, remember that a base case should be some small problem that is trivial to solve. So, what would be the smallest string that can be read forwards and backwards? Well, we have two candidates, 1. The empty string “” and 2. A string with a single character. In both cases, these strings can be read the same forwards and backwards, thus if we can eventually reduce our original string down to an empty string or a single letter string, we know that it must be a palindrome.

This leaves us with the iterative step. First we need to check if the first and last character are the same, just as we did iteratively, and if they are not we can immediately determine the string is not a palindrome. However, if the characters are the same, we need to reduce our problem size for the next recursive step. Here you will need to think about how to provide the next recursive function call a string that is slightly smaller, such that the current string’s second and second to last characters are the recursive function call’s first and last characters. Think about what techniques we have to covered so far for extracting portions of a string!

For this exercise you will be writing a new version of the palindrome testing program, but we can still use some of the old code. So, create a new file named **palindromeRec.py** and copy the contents of **palindromeIter.py** into it.

To complete this program:

- In **palindromeRec.py**,
    - Update your `palinTest()` function such that:
        - It now has a base case testing for single or zero length strings, and returns `True` such a string is passed to the functions
        - Returns `False` if the first and last characters do not match
        - Returns the result of a recursive call to `palinTest()`, passing in a slightly smaller version of the string
    - Do not change the `main` function:

Complete the program, making sure to execute it to verify that it produces the correct results. Note that you will submit the **palindromeRec.py** file to Canvas.

4. Balance Parenthesis

When developing a new programming language, one of the most important components is a language's grammar. Just like in a human language, a grammar determines how strings of characters should be organized and interpreted by the system. For example, in English, we capitalize the first letter in a sentence and usually end a sentence with punctuation such as ? ! or . . In Python, we use tabs to denote which code block a line of code belongs to, and when performing mathematical operators if we have an (we need a). Making sure that parentheses are balanced (i.e for every (it is eventually followed by a)) is actually a slightly more challenging problem than it may first appear, but it is one well suited recursive algorithms!

For this exercise you will be writing a program to determine if a series of (and) are balanced.

To complete this program:

- In a file named **parenthesis.py**,
  - Define a global variable named `count` and set it to 0
    - This will keep count of the number of (and) encountered
  - Define a recursive function named `parenTest()` that takes in two parameter variables, one the represents the line of () and one that represents the current position in that line, will return a Boolean, and should:
    - Design and implement an appropriate base case(s)
      - Remember that we must have the same number of) as (by the end of the string, AND that for a pair of ( ), the (must always come before the)
    - Additionally, as you move through the line of (and), you will want to keep count of how many (and) have been encountered using the global variable `count`
    - Return the result of the recursive call to `parenTest()` passing in the line of () and the new position in the line
      - The recursive function call should be the primary strategy for moving through the line of (and)
  - Define a `main` function, which should:
    - Prompt the user for series of (and) and store it in a new variable
    - Call the `parenTest()` function, passing in the user's string, and 0 for the position and use the returned Boolean to output a message stating whether the series of (and) are balanced or not.

Complete the program, making sure to execute it to verify that it produces the correct results. Note that you will submit the **parenthesis.py** file to Canvas.

Submission

Once you have completed this lab it is time to turn in your work. Please submit the following files to the Lab 9 assignment in Canvas:

- **powers.py**
- **palindromeIter.py**
- **palindromeRec.py**
- **parenthesis.py**

Sample Output

power.py

```
Please enter the base value: 2
Please enter the exponent value: 4
powers(2,4)
powers(2,3)
powers(2,2)
powers(2,1)
2^4 is 16
```

palindromeIter.py

```
Please enter a word to test if it is a palindrome: racecar
racecar is a palindrome!

Please enter a word to test if it is a palindrome: nascar
nascar is not a palindrome.
```

palindromeRec.py

```
Please enter a word to test if it is a palindrome: tacocat
tacocat is a palindrome!

Please enter a word to test if it is a palindrome: burritocat
burritocat is not a palindrome!
```

parenthesis.py

```
Please enter a series of parenthesis to see if they are balanced: ()
() is balanced.

Please enter a series of parenthesis to see if they are balanced: (())
(()) is balanced.

Please enter a series of parenthesis to see if they are balanced: ()()()
()()() is balanced.

Please enter a series of parenthesis to see if they are balanced: )
) is not balanced.
```

```
Please enter a series of parenthesis to see if they are balanced: ())(()
())(() is not balanced.

Please enter a series of parenthesis to see if they are balanced:
(((())())))
(((())()))) is not balanced.
```

Rubric

For each program in this lab, you are expected to make a good faith effort on all requested functionality. Each submitted program will receive a score of either 100, 75, 50, or 0 out of 100 based upon how much of the program you attempted, regardless of whether the final output is correct (See table below). The scores for all of your submitted programs for this lab will be averaged together to calculate your grade for this lab. Keep in mind that labs are practice both for the exams in this course and for coding professionally, so make sure you take the assignments seriously.

Score	Attempted Functionality
100	100% of the functionality was attempted
75	<100% and >=75% of the functionality was attempted
50	<75% and >=50% of the functionality was attempted
0	<50% of the functionality was attempted