Data Structures and Algorithms with Python

Workshop 1

This workshop's goal is to allow you to continue developing your Python coding skills. The workshop has four sections:

1. Conditionals, loops, and functions recap
2. Iteration and square roots
3. Recursion and Fibonacci numbers
4. String operations and palindromes

A 'skeleton' code for these exercises is provided in the file available on the Hub. Open this file in Spyder and you're ready to go. This is the core part of the workshop.

The main workshop is followed by a set of **Challenge** problems. These are more open and complex coding exercises intended for those of you who wish to challenge themselves and go further than the required course material. The exercises sometimes involve new concepts that we haven't gone through in class — Google may be your friend here.

1 Conditionals, loops, and functions recap

Conditionals allow us to check the *state* of the program and determine its path based on this state. The structure is as follows:

```python
score = 54
if score >= 50:
    print('pass')
else:
    print('fail')
```

Here we're first assigning the value 54 to the variable score, and then printing out something based on a condition on this value. The condition after the if-keyword is a Boolean value, which may be True or False. Notice the colon and whitespace needed for Python to recognize the conditional statement.

You also learned how to define **functions**. You can think of functions as mappings similarly to mathematics: they take an input and produce an output. The output of the function is defined by the `return` statement. We can also write functions without this statement — these will just run through (and technically return `None`, a special value). Functions are useful for making our code readable and modular, and avoiding repetition which may cause problems when we change our code. For example, the following function takes a score and a list of gradelevels and prints out the result.

```python
gradeLevels = [80,50] # >= 80: distinction >=50: pass < 50: fail

def printGrade(score,gradeLevels):
    """Assumes score is an int or float and gradeLevels is a list of two ints
    If score is at least the first number in gradeLevels, prints 'distinction'
    Else if score is at least the second number, prints 'pass'
    Else prints 'fail'
    """
    if score >= gradeLevels[0]:
        print('distinction')
    elif score >= gradeLevels[1]:
        print('pass')
    else: print('fail')
```

To work with a function in Spyder, you'll need to execute the entire function code (such as the indented section above). This defines the function, making it available for you in a similar way that you can call built-in functions like `print`. After you have defined a function like this, you can call it simply by using its definition and giving the appropriate inputs in parenthesis:

```
printGrade(48,gradeLevels)
```

Loops are useful for repeating statements. We often use for-loops when we know how many times we will need to iterate and while-loops when we don't. What does the following loop do?

```
score = 54
scoreList = [20,54,54]
scoreCount = 0
for s in scoreList:
    if score == s:
        print('Bingo!')
        scoreCount +=1
print(scoreCount)
```

2 Iteration and square roots

Recall Heron's square root algorithm from class, which we used "manually" to find square roots. It looked like in that case it was rather quick to converge. But how quick is it actually?

```
def squareRootHeron(x,eps=0.01): # Default value eps = 0.01 if not specified
    guess = x/2
    while abs(guess**2-x) >= eps:
        guess = (guess + x/guess)/2
    return guess

y = squareRootHeron(20)
print(y)
```

Notice here the `return` statement in the function: this keyword stops the function execution, and determines the function output as the variable that follows it: in the function above, it's `guess`.

Nothing that would happen in the code after it hits a return statement will get executed by Python. So if you wanted to print the value of `guess`, you would have to do it before the `return guess` statement. When the function returns an output value, we often want to assign it to a variable for future use: here `y`. In terms of math, this is simply saying

$$y = \sqrt{20},$$

using Heron's algorithm.

Exercise: Modify the algorithm to count the number of iterations of the algorithm and return that together with the result. Use a variable similar to `scoreCount` above. You can return several variables in Python conveniently using *multiple assignment*, which works as follows:

```python
x, y = 2, 3
y, x = x, y # We can use multiple assignment swap values
# In your function, you can return multiple variables using a comma...
```

Do you need to do many more iterations for larger numbers?

In addition to Heron's ancient method, there are many other ways for finding a square root. A classic one is a general approach called *bisection search*. The idea is simple: we know that the square root of a number x is somewhere between 0 and x. We start by dividing this range in half, and make this midpoint our guess. Like above, if the guess is close enough, we stop. If the guess is too low (guess squared is lower than x), we update our range by discarding the bottom half. Otherwise, we discard the top half. We then pick the midpoint of the updated range as a new guess. We repeat this until we're close enough. The procedure is called *bisection search* because it halves the search area at each iteration.

For example, to find the square root of 10, our first guess in the search range $(0, 10)$ would be $(10 + 0)/2 = 5$. Since this guess is too high ($5^2 > 10$), we would discard everything above 5 and repeat the search in the range $(0, 5)$. Our new guess would then be 2.5...

The recipe is thus roughly as follows:

- Start with a guess g as average of search range $low = 0$ and $high = \max\{1.0, x\}$
- If $g * g$ is close to x, stop and return g as the answer
- Otherwise, if $g * g < x$, update search range: $low = g$,

- Otherwise, if $g * g >= x$, update search range: $high = g$
- Make new guess as average of updated search range
- Repeat process using new guess until close enough

Using this recipe, complete the algorithm below. Your implementation should update the search range $(low, high)$ correctly at each iteration.

```python
def squareRootBisection(x,eps=0.01):
    low = 0.0
    high = max(1.0,x) # Why are we doing this?
    guess = (low + high)/2 # first guess at midpoint using low and high
    while abs(guess**2-x) >= eps:
        if guess**2 < x:
            # update high/low?
        else:
            # update high/low?
        guess =  # update new guess?
    return guess
```

Add print statements to verify that each iteration is cutting the search area in half. Which algorithm does more iterations: Heron's method or bisection search? Why do you think that is the case?

EXTRA: If you're familiar with Newton's method for finding roots, what would that do in this case?

3 Recursion and Fibonacci numbers

The Fibonacci sequence is defined recursively as follows:

$$F(0) = 0; \quad F(1) = 1;$$
$$F(n) = F(n-1) + F(n-2) \quad \text{if } n \geq 2.$$

The study of this sequence dates back to at least 200BC India, but it is named after a medieval Italian mathematician known as Fibonacci. Fibonacci numbers appear often not only in mathematics (where they define the golden ratio) but fascinatingly also in the nature, for example in the arrangement of leaves on tree branches.

Exercise: write a recursive Python function that takes as input an integer n and returns the Fibonacci number corresponding to n.

```python
def fiboRec(n):
    """Assumes that n is an int > 0
    returns Fibonacci of n"""
    if n == 0:
        # code: return Fibonacci number in base case
    elif n == 1:
        # code: return Fibonacci number in base case
    else:
        # code: return Fibonacci number recursively by calling fiboRec itself
```

Try it with 5, 10, 20,... does it seem efficient? Do **not** try a much larger number! What is going on? Hint: try printing out something every time the recursion hits a base case.

Let's try an iterative algorithm instead. We'll store all the calculated Fibonacci numbers in a list called fibNumbers.

```python
def fiboIter(n):
    """Assumes that n is an int > 0
    returns Fibonacci of n"""
    fibNumbers = [0,1] # Initialize list with first two numbers.
    for i in range(2,n+1): # Loop through
         # code: Use Fibonacci definition and list.append()
         # note the last values of a list can be accessed by list[-1], list[-2]...
    return fibNumbers[n]
```

Try the iterative Fibonacci function. It looks like iteration is much faster than the recursion. We are using some more space to store the numbers though, which may become a problem for very large numbers. A challenge problem below will ask you to eliminate this too and write a fast algorithm without using a list.

4 String operations and palindromes

We have already seen some strings in the tutorial: we started out by printing 'Hello World'. Strings are similar to lists in that you can go through their elements (characters).

```
testWord = 'lovely'
#testWord = 'What a lovely day!'
# String slicing
testWord[1:5]
testWord[:-1]
testWord[5:1:-2]
testWord[::-1]
# Concatenation
testWord + testWord
```

We often want to manipulate strings in various ways, and could spend a whole lecture going through different string operations. For now we're happy with these.

Here's a more complex example: a loop to count the number of vowels in a word. The function goes through each letter in the input `word` and checks whether that letter is in the string `vowels`.

```
def printVowels(word):
    vowels = 'aeiouy'
    vowelCount = 0
    for letter in word:
        for vowel in vowels:
            if letter == vowel:
                vowelCount += 1
```

```python
    print(vowelCount)
```

The following exercises will let you apply these ideas.

Exercise. Write a function that takes a string and returns the same string but in lower case letters and discarding everything but the English alphabet.

```python
def toChars(s):
    """ Input: string s
    Output: returns string with only lower-case chars of the English alphabet
    """
    s = s.lower() # this makes the string lower-case
    alphabet = 'abcdefghjiklmnopqrstuvwxyz' # English alphabet
    ans = '' # Initialize the answer as an empty string
    for c in s: # Loop through characters in the string
        # code here: add character to ans if it is in the English alphabet
    return ans

exampleStr = "Never (1) odd or (2) even..."
testStr = toChars(exampleStr)
```

Exercise. A *palindrome* is a string that is the same forwards and backwards, eg. 'radar' or 'A man, a plan, a canal - Panama.' Write a function that tests whether an arbitrarily long string is a palindrome.

How should we go about this? One idea is to use recursion: think about how you would intuitively start checking whether the word 'racecar' is a palindrome. Perhaps you would start from the 'outside' of the word, comparing pairs of letters at either end? If the word passes the 'outside' test, you could then *recursively* move towards the middle of the word... What is the recursion base case here? What happens when the word is just a single character?

```python
def isPal(s):
    """ input: string s that has gone through toChars
```

```
        output: returns True if s is palindrome, False otherwise
    """
    if len(s) <= 1: # Recursion base case: string length no greater than one
        # code here -  what should the function return here?
    else:
        # code here --- how should a palindrome recursion intuitively work?
        # you need to add return statement based on whether the string is a palindrome
        # one way to do this is through two conditions:
        # 1. what should be the condition for the current iteration?
        # 2. what should be the condition for a recursive call of isPal?
        # you will need to use string slicing
```

Test your implementation with the strings provided in the file, but pass them through your `toChars`-function first.

Demetri Martin is a comedian who is fond of inventing long palindromes. Is one of the quotes provided in the file a palindrome?

5 Challenge problems

5.1 More Fibonacci

1. Why is the recursive Fibonacci slow? Modify the recursive Fibonacci so that it counts the number of recursive calls made. Hint: one way to do this is using *global variables*.
2. Write a more elegant Fibonacci function which does not store an entire list of values but as few as you can. Hint: use multiple assignment.

5.2 Anagrams

Two strings are called *anagrams* if they have the exact same letters (in the exact same numbers), eg 'earth' and 'heart'. Write a function which takes two strings, discards everything but the English alphabet, and returns True if the two words are anagrams, False if not.

There are several ways to approach this problem. You could, for example, generate all possible strings from the characters in one of the strings and see if the other string is one of these. But there are several smarter ways to do this using loops. Can you figure one out? Can you identify anagrams by finding each character of one string in the other one? Can you identify anagrams using the English alphabet? How do you know your algorithm is correct? Do you think your approach is fast compared to others?