Problem Set #2

Successive Approximation and a Wordgame!

Handed out: Lecture 4
Due date: 11:59pm, Lecture 6

Introduction

Successive approximation is a problem-solving method where you try to guess the right answer to a problem and then check your guess. If the guess is good enough, you're done. Otherwise, you keep improving your guess in small increments and checking it, getting closer and closer to the right answer, until you determine that the guess is good enough. For the first 3 problems of this problem set, we will look at Newton's method, which uses successive approximation to find the roots of a function.

Secondly, we will have some fun with Python, and get some practice using strings and string operations. We would like you to implement the word game hangman as a 1-player game against the computer.

Getting Started

Download and save

1. Problem Set 2: A zip file of all the skeleton code you'll be filling in.

Polynomials

For this problem set, we will be representing polynomials as tuples. The index of a number in the tuple represents the power, and the value at that index represents the coefficient for that term. So for example, the polynomial $x^4 + 3x^3 + 17.5x^2 – 13.39$ would be represented by the tuple (-13.39, 0.0, 17.5, 3.0, 1.0). This is because the tuple represents $-13.39x^0 + 0.0x^1 + 17.5x^2 + 3.0x^3 + 1.0x^4$, which is the same as $x^4 + 3.0x^3 + 17.5x^2 – 13.39$.

Problem #1

Implement the *evaluate_poly* function. This function evaluates a polynomial function for the given x value. It takes in a tuple of numbers poly and a number x. By number, we mean that x and each element of poly is a float. *evaluate_poly* takes the polynomial represented by poly and computes its value at x. It returns this value as a float.

```
def evaluate_poly(poly, x):
```

```
"""
Computes the polynomial function for a given value x. Returns that value.
Example:
>>> poly = (0.0, 0.0, 5.0, 9.3, 7.0)     # f(x) = 7.0x^4 + 9.3x^3 + 5.0x^2
>>> x = -13
>>> print evaluate_poly(poly, x)  # f(-13) = 7.0(-13)^4 + 9.3(-13)^3 + 5.0(-13)^2
180339.9
poly: tuple of numbers, length > 0
x: number
returns: float
"""
# TO DO ...
```

Derivatives

As stated before, we will need to find $f'(x_n)$, where $f'(x)$ is the derivative of $f(x)$. Recall that the derivative of a polynomial $f(x) = ax^b$ is $f'(x) = abx^{b-1}$, unless b=0, in which case $f'(x) = 0$. To compute the derivative of a polynomial function with many terms, you just do the same thing to every term individually. For example, if $f(x) = x^4 + 3x^3 + 17.5x^2 - 13.39$, then $f'(x) = 4x^3 + 9x^2 + 35x$.

Problem #2

Implement the *compute_deriv* function. This function computes the derivative of a polynomial function. It takes in a tuple of numbers *poly* and returns the derivative, which is also a polynomial represented by a tuple.

```
def compute_deriv(poly):

"""
Computes and returns the derivative of a polynomial function. If the
derivative is 0, returns (0.0,).
Example:
>>> poly = (-13.39, 0.0, 17.5, 3.0, 1.0)    # x4 + 3.0x3 + 17.5x2 - 13.39
>>> print compute_deriv(poly)        # 4.0x3 + 9.0x2 + 35.0x
(0.0, 35.0, 9.0, 4.0)
poly: tuple of numbers, length > 0
returns: tuple of numbers
"""
# TO DO ...
```

Newton's Method

Newton's method (also known as the Newton-Raphson method) is a successive approximation method for finding the roots of a function. Recall that the roots of a function $f(x)$ are the values of x such that $f(x) = 0$. You can read more about Newton's method here.

Here is how Newton's method works:

1. We guess some x_0.
2. We check to see if it's a root or close enough to a root by calculating $f(x_0)$. If $f(x_0)$ is within some small value epsilon of 0, we say that's good enough and call x_0 a root.
3. If $f(x_0)$ is not good enough, we need to come up with a better guess, x_1. x_1 is calculated by the equation: $x_1 = x_0 - f(x_0)/f'(x_0)$.
4. We check to see if x_1 is close enough to a root. If it is not, we make a better guess x_2 and check that. And so on and so on. For every x_n that is not close enough to a root, we replace it with $x_{n+1} = x_n - f(x_n)/f'(x_n)$ and check if that's close enough to a root. We repeat until we finally find a value close to a root.

For simplicity, we will only be using polynomial functions in this problem set.

Implementing Newton's Method

Problem #3

Implement the compute_root function. This function applies Newton's method of successive approximation as described above to find a root of the polynomial function. It takes in a tuple of numbers poly, an initial guess x_0, and an error bound epsilon. It returns a tuple. The first element is the root of the polynomial represented by poly; the second element is the number of iterations it took to get to that root.

The function starts at x_0. It then applies Newton's method. It ends when it finds a root x such that the absolute value of f(x) is less than epsilon, i.e. f(x) is close enough to zero. It returns the root it found as a float.

```
def compute_root(poly, x_0, epsilon):

"""
Uses Newton's method to find and return a root of a polynomial function.
Returns a tuple containing the root and the number of iterations required to
get to the root.
Example:
>>> poly = (-13.39, 0.0, 17.5, 3.0, 1.0)     #x4 + 3.0x3 + 17.5x2 - 13.39
>>> x_0 = 0.1
>>> epsilon = .0001
>>> print compute_root(poly, x_0, epsilon)
(0.80679075379635201, 8)
poly: tuple of numbers, length > 1.
Represents a polynomial function containing at least one real root.
The derivative of this polynomial function at x_0 is not 0.
x_0: float
epsilon: float > 0
returns: tuple (float, int)
"""
# TO DO ...
```

A Wordgame: Hangman

For this problem, you will implement a variation of the classic wordgame Hangman. For those of you who are unfamiliar with the rules, you may read all about it here. In this problem, the second player will always be the computer, who will be picking a word at random.

Problem #4

Implement a function, `hangman()`, that will start up and carry out an interactive Hangman game between a player and the computer.

For this problem, you will need the code files `ps2_hangman.py` and `words.txt`, which were included in the zip file from the top of this homework. Make sure your file runs properly before editing. You should get the following output when running the unmodified version of `ps2_hangman.py`.

```
Loading word list from file...
55900 words loaded.
```

You will want to do all of your coding for this problem within this file as well because you will be writing a program that depends on each function you write.

Requirements

Here are the requirements for your game:

1. The computer must select a word at random from the list of available words that was provided in words.txt. The functions for loading the word list and selecting a random word have already been provided for you in ps2_hangman.py.
2. The game must be interactive: it should let a player know how many letters the word the computer has picked contains and ask the user to supply guesses. The user should receive feedback immediately after each guess. You should also display to the user the partially guessed word so far, as well as either the letters that the player has already guessed or letters that the player has yet to guess.
3. A player is allowed some number of guesses. Once you understand how the game works, pick a number that seems reasonable to you. Make sure to remind the player of how many guesses s/he has left after each turn.
4. A player loses a guess only when s/he guesses incorrectly.
5. The game should end when the player constructs the full word or runs out of guesses. If the player runs out of guesses (s/he "loses"), reveal the word to the player when the game ends.

The output of an example game may look like this:

```
>>>
Welcome to the game, Hangman!
I am thinking of a word that is 4 letters long.
-------------
You have 8 guesses left.
Available letters: abcdefghijklmnopqrstuvwxyz
```

```
Please guess a letter: a
Good guess: _a _ _
------------
You have 8 guesses left.
Available letters: bcdefghijklmnopqrstuvwxyz
Please guess a letter: s
Oops! That letter is not in my word: _a _ _
------------
You have 7 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: t
Good guess: ta _t
------------
You have 7 guesses left.
Available letters: bcdefghijklmnopqruvwxyz
Please guess a letter: r
Oops! That letter is not in my word: ta _t
------------
You have 6 guesses left.
Available letters: bcdefghijklmnopquvwxyz
Please guess a letter: m
Oops! That letter is not in my word: ta _t
------------
You have 5 guesses left.
Available letters: bdefghijklmnopquvwxyz
Please guess a letter: c
Good guess: tact
------------
Congratulations, you won!
```

Do not be intimidated by this problem! It's actually easier than it looks. Make sure you break down the problem into logical subtasks. What functions will you need to have in order for this game to work?

Hints:

- You should start by using the provided functions to load the words and pick a random one.
- Consider using `string.lowercase`.
- Consider writing helper functions. For instance, we found that creating functions to fill in guessed letters (generating strings like "ta_t") and to display unused letters made partitioning the problem easier.

This completes the problem set!

Handin Procedure

1. Save
Save your solutions as they were provided: `ps2_newton.py` and `ps2_hangman.py`.
Do not ignore this step or save your file(s) with a different name!
2. Time and collaboration info

At the start of the file, in a comment, write down the number of hours (roughly) you spent on this problem set, and the names of whomever you collaborated with. For example:

```
# Problem Set 2
# Name: Jane Lee
# Collaborators (Discussion): John Doe
# Collaborators (Identical Solution): Jane Smith
# Time: 1:30
#
.... your code goes here ...
```

3. Submit

Anything uploaded after the deadline time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.