Homework 1

MPCS 51042 - Python Programming - Fall 2021

Deadline: October 5 at 5:00 pm CDT

Write each problem's solutions in a separate source file: `problem1.py`, `problem2.py`, and `problem3.py`. Upload all 3 source files to Gradescope before the deadline.

- The source files and functions **must** be named as specified.
- You may define additional functions, if your solution requires them.
- You **must** comment each function definition with the following info. Place each function's comments immediately below its `def` statement. The comments do not need to follow a specific format.
  - The purpose of the function
  - The meaning of the function's parameters, if any
  - The meaning of the function's return value, if any

Problem 1

In `problem1.py`, write a function `isect(s1, s2)` that:

- Takes two `str`, each representing a comma-separated sequence of integers. You do not need to validate the input.
- Finds the unique integers the appear in both lists and contain the digit 2. There should be no duplicates in the results.
- Return the result as a sorted `list` of `int`.

Sample usage in console:

```
>>> isect("3,123,201,10,12,20", "20,3,201,124,0,12")
[12, 20, 201]
>>> isect("3,123,201,12,20", "20,3,201,124,0,12")
[12, 20, 201]
>>> isect("3,123,201,10,12,20", "20,201,124,0,12")
[12, 20, 201]
>>> isect("3,10,12,20,-4,20", "20,3,12,0,12")
[12, 20]
>>> isect("3,10,12", "20,3,0")
[]
>>> isect("20,12,20,201", "201,20,20,12")
[12, 20, 201]
```

Problem 2

In `problem2.py`, write a function `expand(rng)` that:

- Takes a `str` representing a comma separated list of integers or integer ranges.
- Return the result as a **sorted** `list` of `int` with the specified integers and expanded integer ranges. An integer range `"a-b"` should be expanded as $[a, b)$, which is **left-inclusive/right-non-inclusive**. There should be **no duplicate integers** in the returned list.

Sample usage in console:

```
>>> expand("1,2-5,5,6-10")
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> expand("6-10,1,2-5,5")
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> expand("1,2-6,5,6-10")
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> expand("1,2-6,5-10")
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Problem 3

In this problem, you will implement the Newton-Raphson method using Python's first-class functions.

Background

Python functions are objects and support operations available to other objects. In the terminology of programming languages, we say that Python supports **first-class functions**. From Wikipedia:

> This means the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures.

In this problem, we will use one of the most essential capabilities of first-class functions: **passing functions as arguments to other functions.** Consider this function, `do_it_more`:

```
def do_it_more(f, init_arg, n):
    res = f(init_arg)
    for i in range(n-1):
        res = f(res)
    return res
```

`do_it_more` takes a function `f` and returns the result of repeatedly calling `f`. It is assumed that `f` takes a single argument and returns a single value. The initial call to `f` uses `init_arg` as its argument. The argument `n` specifies the number of repeated calls. For example, the return value of `do_it_more(f, init_arg, 4)` is equal to `f(f(f(f(init_arg))))`.

Here are some specific examples:

```
>>> def append_bar(word):
...     return word + " bar"
...
>>> do_it_more(append_bar, "foo", 4)
'foo bar bar bar bar'

>>> def add_2(num):
...     return num + 2
...
>>> do_it_more(add_2, 3, 5)
13
```

Task

In `problem3.py`, Write a function that implements the Newton-Raphson Method, which approximates the roots of a differentiable real-valued function. That is, given a function $f(x)$, its derivate $f'(x)$, and an initial guess of a root x_0, then the following recurrence will provide increasingly better approximations of a value x where $f(x) = 0$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

Your function should be defined as `newton(f, f_pr, x_0, tol, max_iter)` where:

- `f` is a function that implements $f(x)$. Assume that f takes a single numerical argument and returns a single numerical value.

- `f_pr` is a function that implements $f'(x)$. Assume that `f_pr` takes a single numerical argument and returns a single numerical value.
- `x_0` is an initial guess at a root.
- `tol` is the absolute tolerance used to end the recurrence. That is, when $|x_{n+1} - x_n| < \text{tol}$, then x_{n+1} should be returned as the result.
- `max_iter` is the maximum number of recurrences. That is, when $n \geq \text{max\_iter}$, then x_{n+1} should be returned as the result, even if $|x_{n+1} - x_n| \geq \text{tol}$
- The return value is x_{n+1}, which results from either tol or max_iter being reached, as described above.

You do not need to perform any input validations or handle numerical errors. For example, you might get `ZeroDivisionError` if $f'(x_n)$ is close to zero, but you do **not** need to detect or handle this error.

Examples

Example usage for $f(x) = (x-1)^2$ and $f'(x) = 2(x-1)$. This finds the root $f(1) = 0$.

```
>>> def f(x):
...     return (x - 1) ** 2
...
>>> def f_pr(x):
...     return 2*(x-1)
...
>>> newton(f, f_pr, 0, 1e-16, 1)
0.5
>>> newton(f, f_pr, 0, 1e-16, 2)
0.75
>>> newton(f, f_pr, 0, 1e-16, 4)
0.9375
>>> newton(f, f_pr, 0, 1e-16, 8)
0.99609375
>>> newton(f, f_pr, 0, 1e-16, 1000000)
1.0
```

Some other good examples:

- $f(x) = x^2 - 4$ and $f'(x) = 2x$. The Newton-Raphson method could find either $f(2) = 0$ or $f(-2) = 0$, depending on your initial guess x_0.
- $f(x) = \sin(x)$ and $f'(x) = \cos(x)$. Note that $\sin(n\pi) = 0$ for any integer n. Hence, the Newton-Raphson method will find different roots, depending on x_0. Try $x_0 = -3$, $x_0 = 3$ and $x_0 = 6$.
- $f(x) = \frac{1}{x}$ and $f'(x) = -\frac{1}{x^2}$. This function has no root, so this example is an expected failure. For example, my implementation did not have any Python-related runtime errors, but it approxemated x as 2.0. Yours could fail in a different way.