**10.009 The Digital World**

Term 3. 2019

Problem Set 3 (for Week 3)

Last update: January 17, 2019

Due dates:

- **Problems: Cohort sessions**: Following week: Tuesday 11:59pm.
- **Problems: Homework**: Same as for the cohort session problems.
- **Problems: Exercises**: These are practice problems and will not be graded. You are encouraged to solve these to enhance your programming skills. Being able to solve these problems will likely help you prepare for the midterm examination.

**Objectives:**

1. Learn the definitions and literals for list.
2. Learn the `while` statements.
3. Learn basic techniques for program debugging.
4. Learn to write simple loops.

**Note**: Solve the programming problems listed using your favorite text editor. Make sure you save your programs in files with suitably chosen names, **and try as much as possible to write your code with good style (see the style guide for python code)**. In each problem find out a way to test the correctness of your program. After writing each program, test it, debug it if the program is incorrect, correct it, and repeat this process until you have a fully working program. Show your working program to one of the cohort instructors.

**Problems: Cohort sessions**

1. *Loops, summation*: Write a function named `list_sum` that takes a list of values as input and returns the sum of all the values. For example, if the input is $[3, 5.0, 9]$ the function must return 17.0. Return 0.0 if the list is empty. You are not allowed to use the built-in function `sum`. You can assume that the input list contains only either float or int objects.

   ```
   print("Test case 1: [4.25,5.0,10.75]")
   ans=list_sum([4.25,5.0,10.75])
   print(ans)

   print("Test case 2: [5.0]")
   ans=list_sum([5.0])
   print(ans)

   print("Test case 3: []")
   ans=list_sum([])
   print(ans)
   ```

   The output should be:

   ```
   Test case 1: [4.25,5.0,10.75]
   20.0
   Test case 2: [5.0]
   5.0
   Test case 3: []
   0.0
   ```

2. *Loops: maximum:* Write a function named `minmax_in_list` that takes a list of integers as an input and returns the minimum and maximum values in the list. Return `(None, None)` if the list is empty. Note that the maximum and minimum integers in Python are given by, respectively, the constants `sys.maxsize` and `-sys.maxsize-1`. You must use loops rather than any built-in function.

   ```
   print("Test case 1: [1,2,3,4,5]")
   ans=minmax_in_list([1,2,3,4,5])
   print(ans)

   print("Test case 2: [1,1,3,0]")
   ans=minmax_in_list([1,1,3,0])
   print(ans)

   print("Test case 3: [3,2,1]")
   ans=minmax_in_list([3,2,1])
   print(ans)

   print("Test case 4: [0,10]")
   ans=minmax_in_list([0,10])
   print(ans)

   print("Test case 5: [1]")
   ans=minmax_in_list([1])
   print(ans)

   print("Test case 6: []")
   ```

```
ans=minmax_in_list([])
print(ans)

print("Test case 7: [1,1,1,1,1]")
ans=minmax_in_list([1,1,1,1,1])
print(ans)
```

The output should be:

```
Test case 1: [1,2,3,4,5]
(1, 5)
Test case 2: [1,1,3,0]
(0, 3)
Test case 3: [3,2,1]
(1, 3)
Test case 4: [0,10]
(0, 10)
Test case 5: [1]
(1, 1)
Test case 6: []
(None, None)
Test case 7: [1,1,1,1,1]
(1, 1)
```

3. *Functions: palindrome:* A number is a *palindrome number* if it reads the same from left to right as from right to left. For example, 1, 22, 12321, 441232144 are palindrome numbers. Write a function named `is_palindrome` that takes an input integer and returns a boolean value that indicates whether the integer is a palindrome number. Use iteration with while loop for this exercise.

```
print("Test case 1: 1")
ans=is_palindrome(1)
print(ans)

print("Test case 2: 22")
ans=is_palindrome(22)
print(ans)

print("Test case 3: 12321")
ans=is_palindrome(12321)
print(ans)

print("Test case 4: 441232144")
ans=is_palindrome(441232144)
print(ans)

print("Test case 5: 441231144")
ans=is_palindrome(441231144)
print(ans)

print("Test case 6: 144")
ans=is_palindrome(144)
print(ans)

print("Test case 7: 12")
ans=is_palindrome(12)
print(ans)
```

The output should be:

```
Test case 1: 1
True
Test case 2: 22
True
Test case 3: 12321
True
Test case 4: 441232144
True
Test case 5: 441231144
False
Test case 6: 144
False
Test case 7: 12
False
```

**Problems: Homework**

1. *Functions:* Write a function named `temp_convert` that takes two arguments. The first argument is a string ‘C” or ’F’ and the second argument is a number (an integer or a float). If the first argument is a ‘C’ then return the centigrade equivalent of the input number. If the first argument is an ‘F’ then return the fahrenheit equivalent of the input number. If the input string is neither ‘C’ nor ‘F” then return `None`. To do the requested conversion, use the `fahrenheit_to_celsius` and `celsius_to_fahrenheit` functions that have been written by you in the Problem Set 2 Homeworks.

```
print("Test case 1: C = 32")
ans=temp_convert("F", 32)
print(ans)

print("Test case 2: C = -40")
ans=temp_convert("F", -40)
print(ans)

print("Test case 3: C= 212")
ans=temp_convert("F", 212)
print(ans)

print("Test case 4: F = 0")
ans=temp_convert("C", 0)
print(ans)

print("Test case 5: F = -40")
ans=temp_convert("C", -40)
print(ans)

print("Test case 6: F = 100")
ans=temp_convert("C", 100)
print(ans)

print("Test case 7: Neither "C’ nor "F")
ans=temp_convert("", 0)
print(ans)

print("Test case 8: Neither "C’ nor "F")
```

```
ans=temp_convert("A", 0)
print(ans)
```

The output should be:

```
Test case 1: C = 32
89.6
Test case 2: C = -40
-40.0
Test case 3: C= 212
413.6
Test case 4: F = 0
-17.7777777778
Test case 5: F = -40
-40.0
Test case 6: F = 100
37.7777777778
Test case 7: Neither "C' nor "F
None
Test case 8: Neither "C' nor "F
None
```

2. *Functions: even:* Write a function `get_even_list` that takes a list and returns a new list containing all the even numbers in the original list. The returned list should follow the order of the input list. You can assume that input list only contains integers.

```
print("get_even_list([1,2,3,4,5])")
ans=get_even_list([1,2,3,4,5])
print(ans)

print("get_even_list([11,22,33,44,55])")
ans=get_even_list([11,22,33,44,55])
print(ans)

print("get_even_list([10,20,30,40,50])")
ans=get_even_list([10,20,30,40,50])
print(ans)

print("get_even_list([11,21,31,41,51])")
ans=get_even_list([11,21,31,41,51])
print(ans)
```

The output should be:

```
get_even_list([1,2,3,4,5])
[2, 4]
get_even_list([11,22,33,44,55])
[22, 44]
get_even_list([10,20,30,40,50])
[10, 20, 30, 40, 50]
get_even_list([11,21,31,41,51])
[]
```

3. *Functions: Prime:* Write a function named `is_prime` that takes an integer argument and returns a boolean value that indicates whether the number is a prime number. (Hint: you can try to find whether there exists a number that divides the number under query. If the answer is yes, then it is not a prime number. Otherwise it is a prime number.)

```
print("is_prime(2)")
ans=is_prime(2)
print(ans)

print("is_prime(3)")
ans=is_prime(3)
print(ans)

print("is_prime(7)")
ans=is_prime(7)
print(ans)

print("is_prime(9)")
ans=is_prime(9)
print(ans)

print("is_prime(21)")
ans=is_prime(21)
print(ans)
```

The output should be:

```
is_prime(2)
True
is_prime(3)
True
is_prime(7)
True
is_prime(9)
False
is_prime(21)
False
```

4. *Euler's Method:* Numerical integration is very important for solving ordinary differential equations which cannot be solved analytically. In such cases, an approximation will have to do, and there are many different algorithms that achieve different levels of accuracy. One of the simplest methods to understand and implement is Euler's method, which is essentially a first order approximation. In Euler's method, if we know the solution to the differential equation at time $t_n$ $(y(t_n))$, we may estimate the solution at time $t_{n+1}$ $(y(t_{n+1}))$ using the following relationship.

   $$y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n))$$

   where $h = t_{n+1} - t_n$ is the step size and $\frac{dy}{dt} = f(t, y)$. Now, write a function `approx_ode` by implementing the Euler's method with step size, $h = 0.1$, to find the approximate values of $y(t)$ up to 3 decimal places for the following initial value problem (IVP):

   $$\frac{dy}{dt} = 3 + e^{-t} - \frac{1}{2}y, \quad y(0) = 1$$

   from $t = 0$ to $t = 5$ at a time interval of $h$. Note that the above IVP can also be solved exactly by the integrating factor method.

The arguments to the functions are: `h, t0, y0, tn`, which stands for the step size, initial time, initial value, and the ending time.

To test:

```
print('approx_ode(0.1, 0, 1, 5)')
ans = approx_ode(0.1, 0, 1, 5)
print('Output: ', ans)

print('approx_ode(0.1, 0, 1, 2.5)')
ans = approx_ode(0.1, 0, 1, 2.5)
print('Output: ', ans)

print('approx_ode(0.1, 0, 1, 3) ')
ans = approx_ode(0.1, 0, 1, 3)
print('Output: ', ans)

print('approx_ode(0.1, 0, 1, 1) ')
ans = approx_ode(0.1, 0, 1, 1)
print('Output: ', ans)

print('approx_ode(0.1, 0, 1, 0) ')
ans = approx_ode(0.1, 0, 1, 0)
print('Output: ', ans)
```

The output should be:

```
approx_ode(0.1,0,1,5)
Output:  5.770729097292093

approx_ode(0.1,0,1,2.5)
Output:  5.045499895356848

approx_ode(0.1,0,1,3)
Output:  5.291824495018364

approx_ode(0.1,0,1,1)
Output:  3.51748514403281

approx_ode(0.1,0,1,0)
Output:  1
```

More details of Eulers method can be found here: `http://tutorial.math.lamar.edu/Classes/DE/EulersMethod.aspx`.

**Problems: Exercises**

1. *Functions: types:* Write a function named `may_ignore`. This function takes one argument that could be an integer, a float, a string, a complex, or of any other valid type in Python. The function adds 1 to the input and returns the sum, only if the input is an integer. If the input is not an integer, the function returns the value `None`. Use the function you have written to print the values of `may_ignore(1)`, `may_ignore(1.0)`, `may_ignore(1+2j)`, and `may_ignore(‘‘Hello")`.

2. *Functions: reverse:* The `reverse` method of a list reverses the list. Write a function `my_reverse`, which takes a list as an input and returns the reverse of the list, without changing the original list. For example, if the input list is [5, -2, 15, 4] then the function must return the list [4, 15, -2, 5]. Use loops and do not use any built-in function to reverse a list, or `[::-1]`.

3. *Functions and loops: approximation of $\pi$* Write a function named `approx_pi` that takes an input $n$ and returns the value of $\pi$ via:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\sum_{k=0}^{n}\frac{(4k)!\ (1103 + 26390k)}{(k!)^4\ 396^{4k}}$$

4. *Loops: GCD:* Write a function named `gcd` that takes two positive integers and returns their greatest common divisor. (Hint: one naive solution is to try all possible divisors and find the largest one. But there are better methods to do it — `http://en.wikipedia.org/wiki/Greatest_common_divisor`).

5. Simpsons's rule for numerical integration: One of the methods for numerical integration is to use Simpson's rule. Sometimes this method produces more accurate answers than the trapezoidal rule. Given a single valued and continuous function $f(x)$, the following formula corresponds to the Simpson"s rule and works well for small values of h even when the function is not smooth over the interval of integration.

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right]$$

Here $h = (b-a)/n$, $x_0 = a$, $x_n = b$, and $x_j = a + jh$, for $j = 0, 1, 2...n-1, n$.

Write a function named `simpsons_rule`, that takes a function f(), integer $n > 1$, limits $a$ and $b$ as inputs and returns an approximation to $\int_a^b f(x)dx$. Use your function to approximate each of the following integrals. To understand the accuracy of the numerical method used in this problem, manually compute the error in the approximation obtained numerically. For each integral, compare the errors obtained when using the Simpson's rule with those obtained using the n-point trapezoidal rule.

$\int_1^3 x^2 dx$ $\qquad\qquad$ $\int_0^\pi sin(x)dx$ $\qquad\qquad$ $\int_{-1}^1 e^{-x} dx$

Note: For those of you who want to learn more about numerical integration read the following article on Simpson's rule `http://en.wikipedia.org/wiki/Simpson%27s_rule`.

**Test Cases:**

Note: All outputs below are rounded to two decimal digits.

**Test case 1**

Input: $f = x^2$, $n = 1000, a = 1, b = 3$

Output: 8.67

**Test case 2**

Input: $f = sin(x)$, $n = 1000, a = 0$, $b{=}\pi$

Output: 2.0

**Test case 3**

Input: $f = e^{-x}$, $n = 1000, a = -1, b = 1$

Output: 2.35

*End of Problem Set 3.*