10.009 The Digital World

Term 3. 2020

Problem Set 5 (for Week 5)

Last update: February 12, 2020

- **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 modularity.
2. Learn how to divide complex problems into smaller modules.
3. Learn recursion.
4. Learn how to create custom modules.

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. *Game: Craps:* Craps is a popular dice game played in casinos. Write a program allowing you to play a variation of the game. This is how the game would be played:

   (a) For the first round, roll two dice. Each die has six faces, and each face reflects a value from 1 to 6. Check the sum of the two dice.

      i. If the sum is 2, 3 or 12 (called craps), you lose

      ii. If the sum is 7 or 11 (called natural), you win

      iii. If the sum is another value (i.e. 4,5,6,8,9, or 10), you earn points equal to the sum obtained.

   (b) Continue to roll the dice until the sum of both dice is either a 7 or the points obtained in the first round.

      i. If 7 is rolled, you lose

      ii. If the sum obtained is equal to the points you obtained in the first round, you win

      iii. For other sums, continue to roll the dice

   .

   Your program acts as a single player, and should print the output you see below when the various conditions are met. The function `play_craps()` should return 0 if you lose and 1 if you win. The main program is given here, together with the sub functions you will need to define. Hint: if you are unsure how to start, take a look at the main function `play_craps()` and see how the sub functions are being called by the main function, to get a better idea of what is required of each sub function. For the print methods, print lose and print win, you should *print a string* and not return it.

```python
import random

craps=set([2,3,12])
naturals=set([7,11])

def roll_two_dices():
    #Write here

def print_lose():
    # Write here

def print_win():
    # Write here

def print_point(p):
    # Write here

def is_craps(n):
    # Write here


def is_naturals(n):
    # Write here


def play_craps():
    point=-1
    while True:
        n1,n2=roll_two_dices()
        sumn=n1+n2
        print('You rolled {:d} + {:d} = {:d}'.format(n1,n2,sumn))
        if point==-1:
            if is_craps(sumn):
                print_lose()
                return 0
            elif is_naturals(sumn):
                print_win()
                return 1
            point=sumn
            print_point(point)
        else:
            if sumn==7:
                print_lose()
                return 0
            elif sumn==point:
                print_win()
                return 1
```

Here are some sample runs:

```
You rolled 5 + 6 = 11

You win

You rolled 1 + 2 = 3

You lose

You rolled 4 + 4 = 8

Your points are 8

You rolled 6 + 2 = 8

You win

You rolled 3 + 2 = 5

Your points are 5


You rolled 2 + 5 = 7

You lose
```

2. *Calendar year:* The goal of top-down design is for each module to provide clearly defined functionality, which collectively provides all of the required functionality of the program.

The three overall steps of the calendar year program are getting the requested year from the user, creating the calendar year structure, and displaying the year. The functionality of displaying the calendar year is not too complex and can be contained in a single function. However, constructing the calendar year takes significantly more steps. Part of a well-designed program is to break those steps up, along their logical boundaries into their own functions. Implement the following functions to produce a program that can construct and display a calendar year:

(a) `def leap_year(year)`: Returns True if the input argument `year` is a leap year. Otherwise, returns False. Check `http://en.wikipedia.org/wiki/Leap_year#Algorithm` for how to do this.

(b) `def day_of_week_jan1(year)`: Returns the day of the week for January 1 of the input argument `year`. `year` must be between 1800 and 2099. The returned value must be in the range 0-6 (where 0-Sun, 1-Mon, ..., 6-Sat). Check `http://en.wikipedia.org/wiki/Determination_of_the_day_of_the_week#Gauss.27_algorithm` for how to do this. The weekday of the first of January in year A is given by:

$$d = R(1 + 5R(A - 1, 4) + 4R(A - 1, 100) + 6R(A - 1, 400), 7)$$

where $R(y, x)$ is a function that returns the remainder when y is divided by x. In Python, it is similar to executing `y % x`.

(c) `def num_days_in_month(month_num, leap_year)`: Returns the number of days in a given month. `month_num` must be in the range 1-12, inclusive. `leap_year` must be True if the month occurs in a leap year. Otherwise, it should be False.

(d) `def construct_cal_month(month_num, first_day_of_month, num_days_in_month)`: Returns a formatted calendar month for display on the screen. `month_num` must be in the range 1-12, inclusive. `first_day_of_month` must be in the range 0-6 (where 0-Sun, 1-Mon, ..., 6-Sat). Return a list of strings of the form,

```
[month_name, week1, week2, ...,]
```

For example, the first two weeks of January 2015 will be

```
['January','              1  2  3','  4  5  6  7  8  9 10']
```

as the first two weeks for January 2015 will be displayed as

```
              1  2  3
  4  5  6  7  8  9 10
```

If the number of days of the last week is less than seven, no spaces are added after the last date. For example, the last week of December 2015 will be

```
' 27 28 29 30 31'
```

Notice that the number of days is five days, and that there are no spaces added after the characters 31. Note also that there are three spaces reserved for each day. In this way, there are two spaces before 4 and two spaces between 4 and 5, but only one space before 27 and between 27 and 28. To test, define the following function:

```
def print_cal_month(list_of_str):
    ret_val=''
    for l in list_of_str:
        ret_val+= l.replace(' ','*')
        ret_val+='\n'
    return ret_val
```

The above function replaces the spaces with `*` and display each item in the list in a new line. If you run the following code:

```
ans=construct_cal_month(1,5,31)
print(print_cal_month(ans))
```

it should output:

```
January
****************1**2
**3**4**5**6**7**8**9
*10*11*12*13*14*15*16
*17*18*19*20*21*22*23
*24*25*26*27*28*29*30
*31
```

(e) `def construct_cal_year(year)`: Return a formatted calendar year for display on the screen. `year` must be in the range 1800-2099, inclusive. Return a list of the form,

```
[year, month1, month2, month3, ..., month12]
```

in which year is an int and each month is a sublist (i.e. month1, month2, ...) is of the form

```
[month_name, week_1_dates, week_2_dates, ...,]
```

The main function is:

(f) `def display_calendar(calendar_year)`: Returns a formatted calendar display as a string, based on the provided `calendar_year`.

You should test the individual functions separately. Ensure that your code does not add a new line after the month of December. Once each function is working properly, do the integration testing. Calling `display_calendar` will display the calendar from January to December for that particular year. An example for two of the months are shown below.

```
March
```

```
  S  M  T  W  T  F  S
  1  2  3  4  5  6  7
  8  9 10 11 12 13 14
 15 16 17 18 19 20 21
 22 23 24 25 26 27 28
 29 30 31
```

```
April
  S  M  T  W  T  F  S
           1  2  3  4
  5  6  7  8  9 10 11
 12 13 14 15 16 17 18
 19 20 21 22 23 24 25
 26 27 28 29 30
```

To test, define the following function:

```python
def print_space_display_calendar(calendar):
    temp=calendar.replace(' ','*')
    return temp.replace('\n','+\n')
```

The above function will replace all spaces with `*` and every new line with `+` plus a new line. Then run the following Python script:

```python
ans=display_calendar(2015)
print('There are {} characters in ans.'.format(len(ans)))
print('START')
print(print_space_display_calendar(ans))
print('END')
```

The output should be as shown below. For 2015, there are 1606 characters in the output. **N**otice that between every month is a new line. There is a new line at the end of each week, except the last week of December.

```
There are 1606 characters in ans.
START
January+
**S**M**T**W**T**F**S+
**************1**2**3+
**4**5**6**7**8**9*10+
*11*12*13*14*15*16*17+
*18*19*20*21*22*23*24+
*25*26*27*28*29*30*31+
+
February+
**S**M**T**W**T**F**S+
**1**2**3**4**5**6**7+
  .
  .
```

.

```
December+
**S**M**T**W**T**F**S+
********1**2**3**4**5+
**6**7**8**9*10*11*12+
*13*14*15*16*17*18*19+
*20*21*22*23*24*25*26+
*27*28*29*30*31
END
```

3. *Recursion:* Write a function named `factorial` that takes in an integer n, and returns its factorial. Solve this problem using recursion. Note that $0! = 1$, and $n! = n \times (n-1)!$ for $n > 0$.

Problems: Homework

1. *Modular Design:* Implement a set of functions called `get_data`, `extract_values`, and `calc_ratios`. Function `get_data` should prompt the user to enter a pair of integers in the same line and read the pair as a single string. For example,

   ```
   Enter integer pair (hit Enter to quit):
   134 289
   ```

   This is read as `'134 289'`.

   This string should be passed into function `extract_values`, which is designed to return the string as a tuple of two integer values, i.e.

   ```
   extract_values('134 289')
   ```

   should return

   ```
   (134, 289)
   ```

   Finally, this tuple is passed to function `calc_ratios` which returns the ratio of the two values. For example,

   ```
   calc_ratios((134,289))
   ```

   returns

   ```
   0.46366782006920415
   ```

   When the second value of the tuple is zero, the ratio is not defined. In this case, the function should return `None`.

   Vocareum Submission: Submit only the function definitions for `extract_value` and `calc_ratios`.

2. *Modular: Calendar* Add another function, `display_calendar_modified()`, to the Calendar Year Program. This function is a modification of `display_calendar()` in the Cohort Sessions problem 2(f). The function `display_calendar_modified()` takes in two inputs. The first input is the calendar year. If the second input is `None`, the entire calendar is returned, identical to what `display_calendar()` does. If the second input is an integer between 1 and 12 inclusive, only the calendar for that particular month is returned. In this case, the output does not contain a newline character at the end of the final week. A file `hw2_others.pyc` is provided. This file contains the code for the following functions. By importing this file, you may use these following functions directly.

```
# Function returns true if the year is a leap year,
# and False otherwise
leap_year(year)

# Function returns the first day of January for
# the given year as an integer
day_of_week_jan1(year)

# Function returns the number of days in the months
# as an integer
num_days_in_month(month_num, leap_year)

# Function constructs the calendar month and
# returns it as a list
construct_cal_month(month_num, first_day_of_month, num_days_in_month)

# Function constructs the calendar year and
# returns it as a list
construct_cal_year(year)

# Function returns a string containing
# a formatted calendar for the year
display_calendar(year)
```

Vocareum Submission: Submit only the function definitions for `display_calendar_modified()`

Problems: Exercises

1. *Recursion:* Write a function called `move_disks(n, fromTower, toTower, auxTower,sol)` to solve the Towers of Hanoi (Check `http://en.wikipedia.org/wiki/Tower_of_Hanoi`) problem recursively. `n` is the number of disks, the next three parameters are strings that contain labels given to each tower:

   (a) `from_tower`: the tower from which you are supposed to move the disks from

   (b) `to_tower`: the tower to which you are supposed to move the disks to

   (c) `aux_tower`: an auxiliary tower to aid you in moving the disks to complete the task

The last argument is a list to contain the solutions. The function should return a list of strings. For example,

```
>>> sol = []
>>> move_disks(1,‘A’,‘B’,‘C’,sol)
>>> print(sol)
[’Move disk 1 from A to B’]

>>> sol = []
>>> move_disks(3,‘A’,‘B’,‘C’,sol)
>>> print(sol)
[‘Move disk 1 from A to B’, ‘Move disk 2 from A to C’,
‘Move disk 1 from B to C’, ‘Move disk 3 from A to B’,
‘Move disk 1 from C to A’, ‘Move disk 2 from C to B’,
‘Move disk 1 from A to B’]
```

End of Problem Set 5.