Programming Fundamentals Using Python

2018

Problem Set 3

Last update: July 16, 2018

Objectives:

1. Learn to use if-elif-else statement.
2. Learn the definitions and literals for list.
3. Learn the `while` and `for-in` statements.
4. Learn basic techniques for program debugging.
5. 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. *Program tracing:* What is printed when the following Python code snippets are executed? For each problem assume three cases: Case 1: $x < y$, Case 2: $x > y$, and Case 3: $x = y$.

   (a)
   ```
   if x<y:
       print(x)
   elif x>y:
       print(y)
   else:
       print(x+y)
   ```

   (b)
   ```
   if x<y:
       print(x+y)
   elif x>y:
       print(y)
       if x==y:
           print(x-y)
   else:
       print(x)
   ```

2. *List*: Answer the following questions:

   (a) How to print the nth element from a list (starting element at 0)?

   (b) How to modify the nth element of a list?

   (c) How to get length of a list?

   (d) How to add into a list?

   (e) How to delete an element in a list?

   (f) How to get a sublist from a list?

   (g) How to get a sublist with a step size in the index?

   (h) How to get a sublist with a negative index?

3. *Lists*: The following problems test your knowledge of lists in Python. There is no need to write a program for these, but you can verify your answers by writing programs. Write 'E' if the code returns any error (or any errors for that line for parts d, e, and f).

   (a) Specify the value of `x[0]` at the end the following code snippet.
   ```
   x=[1,2,3]
   x[0]=0
   y=x
   y[0]=1
   ```

   (b) Specify the value of `x[0]` after the following code snippet.
   ```
   x=[1,2,3]
   def f(l):
     l[0]='a'
   f(x)
   ```

(c) What is the value of `a[0][0][0][0]` after executing the following code snippet? Write 'E' if there are any errors.

```
x=[1,2,3]
y=[x]
a=[y,x]
y[0][0] = (1,2)
```

(d) Specify the values of expressions (a), (b), (c) and (d) in the following code.

```
x=[1,2,3]
y1=[x,0]
y2=y1[:]
y2[0][0]=0
y2[1]=1
y1[0][0] # (a)
y1[1] # (b)
y2[0][0] # (c)
y2[1] # (d)
```

(e) Specify the values of expressions (a), (b), (c) and (d) in the following code.

```
import copy
x=[1,2,3]
y1=[x,0]
y2=copy.deepcopy(y1)
y2[0][0]=0
y2[1]=1
y1[0][0] # (a)
y1[1] # (b)
y2[0][0] # (c)
y2[1] # (d)
```

(f) What is the value of `l` after steps (a), (b), (c) and (d) below?

```
l=[1,2,3]
l[2:3]=4 # (a)
l[1:3]=[0] # (b)
l[1:1]=1 #(c)
l[2:]=[] # (d)
```

4. *Loops:* Write a function named `find_average` that takes in a list of lists as an input. Each sublist contains numbers. The function returns a list of the averages of each sublist, and the overall average. If the sublist is empty, take the average to 0.0.

For example, if the input list is $[[3,4],[5,6,7],[-1,2,3]]$, the program returns the list $[3.5, 6.0, 1.333]$, and the overall average 3.625, calculated by summing all the numbers in all the sublists and dividing this total sum by the total count of all the numbers.

```
>>> ans=find_average([[3,4],[5,6,7],[-1,2,8]])
>>> print(ans)
([3.5, 6.0, 3.0], 4.25)

>>> ans=find_average([[13.13,1.1,1.1],[],[1,1,0.67]])
>>> print(ans)
([5.11, 0.0, 0.89], 3.0)
```

```
>>> ans=find_average([[3.6],[1,2,3],[1,1,1]])
>>> print(ans)
([3.6, 2.0, 1.0], 1.8)

>>> ans=find_average([[2,3,4],[2,6,7],[10,5,15]])
>>> print(ans)
([3.0, 5.0, 10.0], 6.0)
```

5. *Lists and nested loops*: Use of a nested list in Python allows you to implement a data structure as a 2-dimensional matrix along with various matrix operations. Write a function named `transpose_matrix`, which takes a n x m integer matrix (i.e. a list with n items each of which is a list of m integer items) as an argument, and returns its transposed matrix. For example:

```
>>> a = [[1,2,3], [4,5,6], [7,8,9]]
>>> transpose_matrix(a)
[[1,4,7], [2,5,8], [3,6,9]]
>>>
```

   Use a nested for-loop (i.e. a for loop inside a for loop) appropriately to implement this. You are not allowed to use any built-in transpose function from any libraries.

6. *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_palidrome` that takes an input integer and returns a boolean value that indicates whether the integer is a palindrome number.

```
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
```

7. *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
```

8. *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 Eulers method, which is essentially a first order approximation. One step using Eulers method from t_n to t_{n+1} is given by

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

where h 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
```

9. *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`).

10. *List comprehension* Using a list comprehension, create a new list called "newlist" out of the list "numbers", which contains only the positive numbers from the list, as integers.

```
>>> numbers = [34.6, -203.4, 44.9, 68.3, -12.2, 44.6, 12.7]
...
>>> print(newlist)
[34.6, 44.9, 68.3, 44.6, 12.7]
```

11. *List comprehension* Find all of the numbers from 1-1000 that have a 3 in them.

```
# your output of list comprehension should be
>>> print(numbers_with_3)
[3, 13, 23, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 43, 53, 63, 73,
    83, 93, 103, 113, 123, 130, 131, 132, 133, 134, 135, 136, 137,
    138, 139, 143, 153, 163, 173, 183, 193, 203, 213, 223, 230, 231,
    232, 233, 234, 235, 236, 237, 238, 239, 243, 253, 263, 273, 283,
    293, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311,
    312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324,
    325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337,
    338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350,
    351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363,
    364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376,
    377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389,
    390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 403, 413, 423,
    430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 443, 453, 463,
    473, 483, 493, 503, 513, 523, 530, 531, 532, 533, 534, 535, 536,
    537, 538, 539, 543, 553, 563, 573, 583, 593, 603, 613, 623, 630,
    631, 632, 633, 634, 635, 636, 637, 638, 639, 643, 653, 663, 673,
    683, 693, 703, 713, 723, 730, 731, 732, 733, 734, 735, 736, 737,
    738, 739, 743, 753, 763, 773, 783, 793, 803, 813, 823, 830, 831,
    832, 833, 834, 835, 836, 837, 838, 839, 843, 853, 863, 873, 883,
    893, 903, 913, 923, 930, 931, 932, 933, 934, 935, 936, 937, 938,
    939, 943, 953, 963, 973, 983, 993]
```

12. *Generators* Use Generator to create an iterator for all of the numbers from 1-1000 that have a 3 in them.

```
>>> type(numbers_with_3)
<class 'generator'>
>>> print(numbers_with_3)
<generator object <genexpr> at 0x1012374c0>
>>> for num in numbers_with_3:
...     print(num, end=', ')
...
```

```
3, 13, 23, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 43, 53, 63, 73,
    83, 93, 103, 113, 123, 130, 131, 132, 133, 134, 135, 136, 137,
    138, 139, 143, 153, 163, 173, 183, 193, 203, 213, 223, 230, 231,
    232, 233, 234, 235, 236, 237, 238, 239, 243, 253, 263, 273, 283,
    293, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311,
    312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324,
    325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337,
    338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350,
    351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363,
    364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376,
    377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389,
    390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 403, 413, 423,
    430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 443, 453, 463,
    473, 483, 493, 503, 513, 523, 530, 531, 532, 533, 534, 535, 536,
    537, 538, 539, 543, 553, 563, 573, 583, 593, 603, 613, 623, 630,
    631, 632, 633, 634, 635, 636, 637, 638, 639, 643, 653, 663, 673,
    683, 693, 703, 713, 723, 730, 731, 732, 733, 734, 735, 736, 737,
    738, 739, 743, 753, 763, 773, 783, 793, 803, 813, 823, 830, 831,
    832, 833, 834, 835, 836, 837, 838, 839, 843, 853, 863, 873, 883,
    893, 903, 913, 923, 930, 931, 932, 933, 934, 935, 936, 937, 938,
    939, 943, 953, 963, 973, 983, 993,
```

End of Problem Set 3.