Version 2 – includes more class exercises and the questions from the sample midterms.

**General Notes**:

* This document is being extended constantly. Look for the version number at the top.
* Many questions have several possible valid solutions. Here we propose just one.
* The questions are divided into subjects for your convenience, though some of the questions combine several subjects.
* It is advised to try and answer the questions yourself before reading the solution.

**Python Basics, Lists, Strings and Functions**:

**Question 1:**

1. Shortly explain the difference between a compiler and an interpreter.
2. A Python program consists of 20 lines, all in the \_\_main\_\_ scope. There are print() calls on lines 8, 13 and 20. Line 18 consists of an error that will raise an exception. Will the print() commands be executed? If so, which of them?

**Solution 1:**

1. A compiler takes an entire program written in a high-level programming language and translates it into machine code. An interpreter translates a program written in a high-level programming language to machine code line-by-line. Compilers can find (some) errors in advance, while an interpreter will only find errors during runtime.
2. Since an interpreter runs the program line-by-line, the print commands at lines 8 and 13 will be executed, and the one on line 20 will not because the exception prevented the interpreter to reach it**.**

**Question 2:**

Define a function overlapping() that takes two lists and returns a list of all the members they have in common. Do the exercise twice:

* With two nested for loops.
* With a single loop using the ‘in’ operator inside the loop

**Solution 2:**

**def** overlapping(lst1,lst2):  
 common = []  
 **for** element1 **in** lst1:  
 **for** element2 **in** lst2:  
 **if** element1 == element2:  
 common.append(element1)  
 **return** common  
  
**def** overlappingSecondVersion(lst1,lst2):  
 common = []  
 **for** element **in** lst1:  
 **if** element **in** lst2:  
 common.append(element)  
 **return** common  
  
lst1 = [1,2,3,4]  
lst2 = [3,4,5,6,7]  
lst3 = overlapping(lst1,lst2)  
print(lst3)  
lst3 = overlappingSecondVersion(lst1,lst2)  
print(lst3)

**Question 3:**

Define a function IsPalindrome() that recognizes palindromes (i.e. words that look the same written backwards). For example, IsPalindrome("radar") should return True. Use only one index variable in your loop.

**Solution 3:**

**def** IsPalindrome(text):  
 **for** i **in** range(len(text)//2):  
 **if** text[i] != text[-(i+1)]:  
 **return False  
 return True**print(IsPalindrome(**"radar"**))  
print(IsPalindrome(**"raard"**))

**Question 4:**

Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 5, between 2000 and 3200 (both included). The numbers obtained should be printed in a comma-separated sequence on a single line (hint: use the join function).

**Solution 4:**

l=[]  
**for** i **in** range(2000, 3201):  
 **if** (i%7 == 0) **and** (i%5 != 0):  
 l.append(str(i))  
  
outputString = **','**.join(l)  
print(outputString)

**Question 5:**

Write a recursive function that takes a number and prints the “99 bottles” song with the following format:

99 bottles of beer on the wall, 99 bottles of beer.  
Take one down, pass it around, 98 bottles of beer on the wall.

**Solution 5:**

**def** songGenerator(n):  
 **if** n == 0:  
 **return  
 if** n == 1:  
 print (n, **" bottle of beer on the wall,"**,n,**"bottle of beer"**)  
 print (**"Take it down, pass it around,"**,n-1,**"bottles of beer on the wall\n"**)  
 **else**:  
 print (n, **" bottles of beer on the wall,"**,n,**"bottles of beer"**)  
 print (**"Take one down, pass it around,"**,n-1,**"bottles of beer on the wall\n"**)  
 songGenerator(n-1)  
  
songGenerator(3)

**Question 6:**

Write a function that gets a list of strings and return the longest string. The list can also contain other lists of strings, so you need to look inside those lists as well. **Hint**: Use the *isinstance* function.

**Solution 6:**

**def** longest(l):  
 longestWord = **""  
 for** element **in** l:  
 **if** isinstance(element,list):  
 longest\_in\_inner\_list = longest(element)  
 **if** len(longest\_in\_inner\_list) > len(longestWord):  
 longestWord = longest\_in\_inner\_list  
 **else**:  
 **if** len(element) > len(longestWord):  
 longestWord = element  
 **return** longestWord  
  
lst = [**'one'**,**'two'**,[**'three'**,[**'four'**,**'five'**]]]

print(longest(lst))

**Question 7**:

What would be printed here?

x = 1  
y = 2  
z = 3  
x, y, z = z, x, y  
print(x, y, z)

**Solution 7**: 3 1 2

**Question 8**:

Write a function Rearrange() that gets a list and rearranges the list as follows:

The elements of the list are names of colors. Any color name is possible as an element on the list. The function should rearrange the list in-place (i.e., change the given list) so that all “RED” elements would appear before all of the “GREEN” elements.

For instance, if the list is:

["GREEN","BLUE","RED","BLUE","YELLOW","GREEN","BLUE","RED","GREEN","RED"]

Then possible acceptable outputs are:

['BLUE', 'RED', ‘BLUE', 'RED', 'YELLOW', 'RED', 'GREEN', ‘BLUE’, ’GREEN', 'GREEN']

['RED', 'BLUE', 'RED', 'BLUE', 'YELLOW', 'RED', 'BLUE', 'GREEN', 'GREEN', 'GREEN']

Any arrangement in which all the “RED”s appear before all the “GREEN”s is legal.

Time limitation: O(n), where n is the length of the list.

Space limitation: O(1).

Partial points will be granted for:

1. A correct implementation using more than O(1) space.

2. An algorithm written in plain words or pseudo-code.

**Solution 8**:

In this proposed solution (others are possible), we traverse the list once counting each color's number of appearances and saving them in a dictionary. Since there is a finite number of colors, the dictionary's space is O(1). Then, we insert all the REDs to the beginning of the list, then all the GREENs, then everything else.

**def** Rearrange(lst):  
 d = {}  
 **for** color **in** lst:  
 **if** color **not in** d:  
 d[color] = 1  
 **else**:  
 d[color] += 1  
 location = 0  
 **for** i **in** range(d[**"RED"**]):  
 lst[location] = **"RED"** location += 1  
 **for** i **in** range(d[**"GREEN"**]):  
 lst[location] = **"GREEN"** location += 1  
  
 **for** color, appearances **in** d.items():  
 **if** color != **"RED" and** color != **"GREEN"**:  
 **for** i **in** range(appearances):  
 lst[location] = color  
 location += 1  
  
lst = [**"GREEN"**, **"BLUE"**, **"RED"**, **"BLUE"**, **"YELLOW"**, **"GREEN"**, **"BLUE"**, **"RED"**, **"GREEN"**, **"RED"**]  
Rearrange(lst)  
print(lst)

**Question 9**: Given a list lst, write two ways to add an element at the end of the list.

**Solution 9**: lst.append(element), lst.insert(len(lst),element) , lst += [element], lst.extend([element])

**Question 10**: What is the output of the following code?

>>> x = [3,19,1,10,33,2].sort()

>>> print(x)

Think closely on the return value.

**Solution 10**: None

**Question 11**: How does the else clause of for loops work? Explain and provide a code sample.

**Solution 11**:

The else clause is called when the loop is traversed fully and we did not encounter a break. When the loop's condition is not valid, only then we enter the else clause.

Here is a code sample:

**def** CheckSingleDigit(lst):  
 **for** num **in** lst:  
 **if** num < 10:  
 print(num, **"is single digit"**)  
 **else**:  
 print(num, **"has more then one digit!!!"**)  
 **break  
 else**:  
 print(**"no single digit numbers were found"**)  
  
CheckSingleDigit(range(10))

**Question 12**:

Write a function SortBinaryList() that takes a list of only zeros and ones and sorts it.

For instance, if the list given is [0,1,1,0,0,1,0] then it should be changed to [0,0,0,0,1,1,1].

You are not allowed to count the number of zeros or ones in the list. The execution limitations are O(n) time and O(1) space.

Hint: use two indices starting from the left and right edges, and loop both of them towards the center.

**Solution 12**:

**def** SortBinaryList(lst):  
 leftIndex = 0  
 rightIndex = len(lst)-1  
 **while** leftIndex < rightIndex:  
 **if** lst[leftIndex] > lst[rightIndex]:  
 lst[leftIndex], lst[rightIndex] = lst[rightIndex], lst[leftIndex]  
 **if**(lst[leftIndex]==0):  
 leftIndex += 1  
 **if**(lst[rightIndex]==1):  
 rightIndex -=1  
  
lst = [1, 0, 1, 0, 0, 1]  
SortBinaryList(lst)  
print(lst)

**Question 13**: What will be printed here?

string = **"solomon"**print(string[:3] + string[-4:])

**Solution 13**: Solomon

**Question 14**: What will be printed here?

x = [1, 2, 3]  
y = x  
y.append(4)  
z = y  
z.extend([5])  
y = 7  
print(x, y, z)

**Solution 14**:

[1, 2, 3, 4, 5] 7 [1, 2, 3, 4, 5]

**Question 15**:

Write two different for loops that iterate and print the characters of a string called *string*.

**Solution 15**:

**for** char **in** string:  
 print(char)  
**for** i **in** range(len(string)):  
 print(string[i])

**Question 16**:

What will be printed here?

l1 = [9, 3, 17]  
l2 = [15, 6, 2]  
**for** x, y **in** zip(l1, l2):  
 print(x+y)

**Solution 16**:

24

9

19

**List Comprehensions:**

**Question 1:**

Write a single-line command that maps a list of words into a list of integers representing the lengths of the corresponding words.

**Solution 1:**

list = [**'tiny'**,**',medium'**,**'gigantic'**]  
lengths = [len(word) **for** word **in** list]  
print(lengths)

**Question 2:**

Now take another list of words and create a list by concatenating each element in the first list with each element in the second list.

**Solution 2:**

lst = [**'tiny'**,**',medium'**,**'gigantic'**]  
lst2 = [**'mouse'**,**'dog'**,**'elephant'**]  
concatenated = [x+**'\_'**+y **for** x **in** lst **for** y **in** lst2]  
print(concatenated)

**Question 3:**

Write a single-line function *twoLetterSubs* that gets a list of words and returns a list of all two-letter substrings that appear in those words. For example: twoLetterSubs(['hello','world']) should return ['he', 'el', 'll', 'lo', 'wo', 'or', 'rl', 'ld'].

**Solution 3:**

**def** twoLetterSubs(lst):  
 **return** [word[i:i+2] **for** word **in** lst **for** i **in** range(len(word)-1)]

**Question 4:**

Write a function Nahman() that gets a list lst, and by using list comprehensions returns the following list: [lst[0], lst[0:1], lst[0:2],....., lst[0:(n-1)], where n is the length of lst.

For instance, if lst = ['n', 'a', 'h', 'm', 'a', 'n', "meuman"] then the output should be:

['n', 'n', 'a', 'n', 'a', 'h', 'n', 'a', 'h', 'm', 'n', 'a', 'h', 'm', 'a', 'n', 'a', 'h', 'm', 'a', 'n']

**Solution 4:**

**def** Nahman(lst):  
 **return** [x **for** i **in** range(len(lst)) **for** x **in** lst[0:i+1]]

print(Nahman([**'n'**, **'a'**, **'h'**, **'m'**, **'a'**, **'n'**, **"meuman"**]))

**Question 5:**

Write a single-line function that gets a list and returns a new list with the values from the uneven positions in the given list. Hint: You can use list comprehensions and enumerate.

**Solution 5**:

**def** GetUnevenElements(lst):  
 **return** [x **for** (i, x) **in** enumerate(lst) **if** i % 2 != 0]  
print(GetUnevenElements([12, 24, 35, 70, 88, 120, 155]))

**Question 6:**

Write a single line function CreateTuples(A,B,C) that gets 3 lists of integers and returns a list of all possible tuples of the form (a, b, c), where a in A, b in B, c in C and c<a<b.

**Solution 6**:

**def** CreateTuples(A, B, C):  
 **return** [(a, b, c) **for** a **in** A **for** b **in** B **for** c **in** C **if** c < a < b]  
  
A = list(range(1, 5))  
B = list(range(2, 6))  
C = list(range(-2 ,2))  
print(CreateTuples(A, B, C))

**Question 7**:

Write a single-line function called ReplaceKeysValues() that gets a dictionary and returns a dictionary with replaced keys and values of the given dictionary.

**Solution 7**:

**def** ReplaceKeysValues(d):  
 **return** {value: key **for** key, value **in** d.items()}  
print(ReplaceKeysValues({1: **"a"**, 2: **"b"**, 3: **"c"**}))

**Question 8**:

Write a function AvgSoFar(lst) that gets a list of numbers, and by using list comprehensions produces a list whose ith entry is equal to the average of the numbers lst[0],...,lst[i]. For instance, for lst = [1,2,3,4,5] the output should be [1.0, 1.5, 2.0, 2.5, 3.0].

Hint: first use list comprehensions to create a list of the sublists [[1], [1,2], … , [1,2,3,4,5]]. Then use list comprehensions on that list along with the sum() and len() functions to produce another list of the averages.

**Solution 8**:

**def** AvgSoFar(lst):  
 **return** [sum(sub)/len(sub) **for** sub **in** [lst[0:i+1] **for** i **in** range(len(lst))]]

print(AvgSoFar([1, 2, 3, 4, 5]))

**Tuples, Sets and Dictionaries:**

**Question 1:**

Create a program that reads triplets of inputs from the user until the value 0 is read as the third input. The program will create a list of tuples representing cubes and will print that list. Then it will create a list of the cube’s volumes and print that list as well. Try to write the shortest code when creating tuples and reading their values.

**Solution 1:**

x,y,z = (int(input(**"-->"**)),int(input(**"-->"**)),int(input(**"-->"**)))  
cubes = []  
**while** (z != 0):  
 tuple = (x,y,z)  
 cubes.append(tuple)  
 x,y,z = (int(input(**"-->"**)),int(input(**"-->"**)),int(input(**"-->"**)))  
print(cubes)  
  
volumes = []  
**for** cube **in** cubes:  
 x,y,z = cube  
 volumes.append(x\*y\*z)  
print(volumes)

**Question 2:**

Write a program which accepts a sequence of comma-separated numbers from the console and generates a list and a tuple which contains every number.

**Solution 2:**

values=input(**"-->"**)  
l=values.split(**","**)  
t=tuple(l)  
print (l)  
print (t)

**Question 3:**

Create a program that reads 5 **different** words from the user. Then the program will print a few empty lines and ask the user to provide the same words again from memory, not necessarily in the same order. The program will print out the words that the user remembered, the words that he forgot, and the excessive words.

**Solution 3:**

print(**"Please enter 5 different words"**)  
wordsSet = set()  
**while** len(wordsSet) < 5:  
 wordsSet.add(input(**"-->"**))  
print (**"\n\n\n\n\n\n\n"**)  
  
print(**"Now, please enter the same 5 words from memory, not necessarily in the same order"**)  
wordsSet2 = set()  
**while** len(wordsSet2) < 5:  
 wordsSet2.add(input(**"-->"**))  
  
print(**"You remembered the words:"**,wordsSet & wordsSet2)  
print(**"You forgot the words:"**,wordsSet - wordsSet2)  
print(**"The following words:"**,wordsSet2 - wordsSet,**"are excessive"**)

**Question 4:**

Write a function numberToDict() that gets a number and returns a dictionary mapping its digits location to their value, going from right to left. Then write the opposite function dictToNumber().

**Solution 4:**

**def** numberToDict(num):  
 d = {}  
 location = 0  
 **while**(num > 0):  
 d[location] = num % 10  
 num = num // 10  
 location += 1  
 **return** d  
  
**def** dictToNumber(d):  
 number = 0  
 **for** key,value **in** d.items():  
 number += 10 \*\* key \* value  
 **return** number  
  
d = numberToDict(85432)  
print(d)  
number = dictToNumber(d)  
print(number)

**Question 5:**

What are the similarities and the differences between a list and a tuple?

**Solution 5:**

Tuples and lists are both ordered sequences that can hold any type of elements. They can be accessed by an index and looped in a certain order. Tuples, unlike lists are immutable and we cannot change a tuple's element once it is created. Lists are mutable and have several methods and operators for adding, removing and changing the list's elements.

**Question 6:**

Write a function that gets two lists and returns a new list containing the elements that appear in both lists without duplications. Do not use any loops.

**Solution 6**:

**def** GetIntersection(lst1,lst2):  
 intersectionSet = set(lst1) & set(lst2)  
 **return** list(intersectionSet)  
  
print(GetIntersection([3, 6, 78, 35, 55], [12 ,24, 35, 24, 78, 120, 155]))

**Question 7**:

What are the similarities and the differences between a list and a set?

**Solution 7**:

Lists and sets are both mutable objects. They can hold elements of any type, and elements can be added/removed from lists and sets after they are created (unlike tuples which are immutable).

Lists are ordered sequences; elements can be accessed by an index and looped in a certain order. Sets on the other hand, are non-ordered; they cannot be accessed by an index looping the element does not guarantee the same order of insertion.

Unlike lists, sets do not store duplicates of an element, and they also support group

**Question 8**:

Complete the following code as to attain the indicated output:

d1 = {2: **"a"**, 1: **"b"**}  
d2 = {4: **"c"**, 5: **"d"**, 3: **"e"**}  
**for** k1, k2 **in** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_:  
 print(k1, k2)

output:

1 3

2 4

**Solution 8**:

zip(sorted(d1.keys()), sorted(d2.keys()))

**Question 9**:

Write two different for loops that iterate and print the keys and values of a dictionary *d*.

**Solution 9**:

**for** key **in** d.keys():  
 print(key, **":"**, d[key])  
**for** key,value **in** d.items():  
 print(key, **":"**, value)

**Lambda Expressions, Higher Order Functions and Functions Args**:

**Question 1:**

Write a function filter\_long\_words() that takes a list of words and an integer n and returns the list of words that are longer than n.

**Solution 1**:

**def** filter\_long\_words(lst,n):  
 **return** list(filter(**lambda** x: len(x)>n, lst))  
  
print(filter\_long\_words([**'hello'**, **'everyone'**],5))

**Question 2:**

Write a function find\_longest\_word() that takes a list of words and returns the length of the longest one. Use only higher order functions and lambda expressions. You can use max() to compare 2 numbers, but do not use max() on a list.

**Solution 2**:

**from** functools **import** reduce  
**def** find\_longest\_word(lst):  
 **return** reduce(**lambda** x,y: max(x,y),map(**lambda** x:len(x),lst))  
  
print(find\_longest\_word([**'a'**, **'bbbbb'**, **'cc'**, **'ddd'**]))

**Question 3:**

Write a function *mysum* that gets an arbitrary number of arguments and returns the sum of its integer arguments (it should ignore non-integer argument). For example, mysum(10, 20, 30) should return 60 and mysum(10, **'foo'**, **'bar'**, 20) should return 30.

**Solution 3**:

**def** mysum(\*args):  
 sum = 0  
 **for** arg **in** args:  
 **if** isinstance(arg, int):  
 sum += arg  
 **return** sum

**Question 4:**

Write a function *CalcArea* that returns the area of different shapes depending on the args it received. The first arg is called *edge* and its default value should be 10. The function also accepts any number of keyword args. If the function received an arg called *anotherEdge*, it means that the shape is a rectangle with two different edges. If there is no *anotherEdge* arg but there is a *pi* arg, it means that the shape is a circle where *edge* is its radius. If there aren't any args besides *edge*, then the shape is a square. For example, CalcArea() should return 100, CalcArea(30,anotherEdge=20) should return 600 and CalcArea(pi=3.14) should return 314.

**Solution 4**:

**def** CalcArea(edge=10, \*\*keywords):  
 **if 'anotherEdge' in** keywords:  
 **return** edge \* keywords[**'anotherEdge'**]  
 **if 'pi' in** keywords:  
 **return** keywords[**'pi'**] \* edge \*\* 2  
 **else**:  
 **return** edge \*\* 2

**Modules & Packages:**

**Question 1**:

A file called mymod.py includes the following code:

**import** sys  
x = 5  
**def** f():  
 print(x)  
f()

In another file called test.py file we have the following code:

**import** mymod  
x = 4  
**def** f():  
 print(x)  
print(dir()[-3:]) *#last 3 names*print(dir(mymod)[-3:]) *#last 3 names*f()

What will be the output of test.py?

**Solution 1**:

5

['f', mymod, 'x'] #the order doesn't matter for the solution

['f', 'sys', 'x'] #the order doesn't matter for the solution

4

**Namespaces & Scopes:**

**Question 1**:

We've learned that a namespace is a mapping from names to objects and that each scope has its own namespace. What are the namespaces in the following code and what names are listed in them? For each namespace describe its scope in the code, the names in it and the type of their object. For each namespace, address to latest time it existed.

x = 5  
**def** f():  
 i = 3  
**class** C:  
 lst = []  
 counter = 0  
 **def** f(self):  
 self.lst.append(self.counter)  
 total = sum(self.lst)  
 self.counter += 1

f()

c = C()  
c.f()

**Solution 1**:

* f()'s definition has a namespace that includes a name called 'i' of type int.
* Class C's definition has a namespace which includes the names 'lst' of type list, 'counter' of type int, and 'f' of type function.
* f()'s definition inside C has a namespace that includes a name called total of type int and a name 'self' of type 'C'.
* The global scope has a namespace that includes the names 'x' of type int, 'f' of type function, 'C' of type class, and 'c' of type C.

**Object Oriented:**

**Question 1:**

Define a class which has the following methods: readString() to read a string from the console input, and printUpperString() to print the string in upper case.

**Solution 1**:

**class** InputOutString:  
 **def** getString(self):  
 self.s = input(**"-->"**)  
  
 **def** printUpperString(self):  
 print(self.s.upper())  
  
ios = InputOutString()  
ios.getString()  
ios.printUpperString()

**Question 2:**

Define a class called Shape that gets a length in its constructor. The class will have the following methods: \_\_*str*\_\_(), \_\_*lt*\_\_() and *CalcArea*(). \_\_*str*\_\_() should return a string with all the shape’s properties. This allows us to call *print*() on a *Shape* object.

Define 3 derived classes to Shape: *Square*, *Rectangle* and *Circle*. Decide whether or not they need to receive more arguments in their constructors. You should also decide which methods are implemented at Shape, which at its sons, and which should be overridden.

Each shape should print its unique properties including its area in a readable way.

The comparison between shapes should be based on the area.

*Shape* should also include a class variable called *shapes*, which is a list that holds all the existing shapes, and a static function called PrintShapes() that prints all the shapes.

**Solution 2**:

**class** Shape:  
 shapes = []  
 **def** PrintShapes():  
 **for** shape **in** Shape.shapes:  
 print(shape)  
  
 **def** \_\_init\_\_(self,length):  
 self.length = length  
 Shape.shapes.append(self)  
  
 **def** \_\_str\_\_(self):  
 **pass  
  
 def** CalcArea(self):  
 **pass  
  
 def** \_\_lt\_\_(self, other):  
 **if** self.CalcArea() < other.CalcArea():  
 **return True  
 else**:  
 **return False  
  
class** Square(Shape):  
 **def** \_\_str\_\_(self):  
 **return "Square. length: "** + str(self.length) + **", area: "** + str(self.CalcArea())  
  
 **def** CalcArea(self):  
 **return** self.length \*\* 2  
  
**class** Rectangle(Shape):  
 **def** \_\_init\_\_(self, length, width):  
 Shape.\_\_init\_\_(self, length)  
 self.width = width  
  
 **def** \_\_str\_\_(self):  
 **return "Rectangle. length: "** + str(self.length) + **", width: "** + str(self.width) + **", area: "** + str(self.CalcArea())  
  
 **def** CalcArea(self):  
 **return** self.length \* self.width  
  
**class** Circle(Shape):  
 pi = 3.14  
  
 **def** \_\_str\_\_(self):  
 **return "Circle. radius: "** + str(self.length) + **", area: "** + str(self.CalcArea())  
  
 **def** CalcArea(self):  
 **return** Circle.pi \* self.length \*\* 2  
  
Shape.PrintShapes()  
circle = Circle(3)  
square = Square(3)  
rectangle = Rectangle(3,4)  
Shape.PrintShapes()  
Shape.shapes.sort()  
print(**"After sort:"**)  
Shape.PrintShapes()

**Question 3**: What is the output of the following code?

**class** X:  
 y = 15  
  
 **def** f(self):  
 X.y += 1  
  
 **def** g(self):  
 self.y += 1  
  
z = X()  
print(z.y)  
z.f()  
print(z.y)  
z.g()  
print(X.y)

**Solution 3**:

15

16

16

**Question 4**:

What does *self* mean in Python and when is it used?

Do all methods in a class definition have to include self as their first argument?

**Solution 4**:

Self is a reference to the current instance of an object. It is the first argument of all the methods related to an instance of the class, and access to data attributes and other methods is done via the self prefix followed by a dot. Only methods that are not bound to an instance of the class, but to the class itself do not include *self* as their first argument. They cannot be

**Question 5**: Is the following code legal in Python? Shortly explain your answer.

**class** X:  
 **pass  
def** f(x):  
 **return** x+1  
  
x = X()  
x.g = f  
x.g(5)

**Solution 5**:

Yes. In x.g = f we assign x with a new method called g, which is actually the f() function. Then there is no problem to call g() via x.

**Question 6**:

What is the output of the following code?

**class** X:  
 lst = []  
 **def** f(self,e):  
 self.lst.append(e)  
 print(X.lst)  
  
x = X()  
x.f(1)  
x.f(2)  
x.f(3)

**Solution 6**:

[1]

[1, 2]

[1, 2, 3]

**Question 7**:

The following code:

**class** X:  
 **def** f(self):  
 **pass**x = X()  
print(X.f)  
print(x.f)

Produces the following output:

<function X.f at 0x01F84420>

<bound method X.f of <\_\_main\_\_.X object at 0x00815870>>

Why don't X.f and x.f point to the same object?

**Solution 7**:

X.f is a function inside the namespace which is the class definition, whereas x.f is a method bound to a specific instance of class X (the instance x).