Course Name: Practical Python Programming

Week 4: Python Programing – Modules, Classes and Objects

a. Modules, Classes, and Objects.

Python is something called an “object- oriented programming language.” What this means is there’s a construct in Python called a class that lets us structure our software in a particular way. Using classes, we can add consistency to our programs so that they can be used in a cleaner way, or at least that’s the theory.

Modules Are Like Dictionaries

You know how a dictionary is created and used and that it is a way to map one thing to another. That means if you have a dictionary with a key 'apple' and you want to get it, then you do this:

```
mystuff = {'apple': "I AM APPLES!"}
print mystuff['apple']
```

Keep this idea of “get X from Y” in your head, and now think about modules. You’ve made a few so far and used them, in accordance with the following process:

1. You know that a module is a Python fi le with some functions or variables in it.
2. You then import that file.
3. And then you can access the functions or variables in that module with the '.' (dot) operator.

Imagine if I have a module that I decide to name mystuff.py, and I put a function in it called apple. Here’s the module `mystuff.py`:

```
# this goes in mystuff.py
def apple():
print "I AM APPLES!"
```

Once I have that, I can use that module with import and then access the apple function:

```
import mystuff

mystuff.apple()
```

I could also put a variable in it named tangerine, like this:

```
def apple():
      print "I AM APPLES!"

# this is just a variable
tangerine = "Living reflection of a dream"
```

Then again I can access that the same way:

```
import mystuff

mystuff.apple()
print mystuff.tangerine
```

Refer back to the dictionary, and you should start to see how this is similar to using a dictionary, but the syntax is different. Let's compare:

```
mystuff['apple'] # get apple from dict
mystuff.apple() # get apple from the module
mystuff.tangerine # same thing, it's just a variable
```

This means we have a very common pattern in Python:

1. Take a key=value style container.
2. Get something out of it by the key's name.

In the case of the dictionary, the key is a string and the syntax is [key]. In the case of the module, the key is an identifier, and the syntax is .key. Other than that, they are nearly the same thing.

Classes Are Like Modules

A way to think about a module is that it is a specialized dictionary that can store Python code so you can get to it with the '.' operator. Python also has another construct that serves a similar purpose called a class. A class is a way to take a grouping of functions and data and place them inside a container so you can access them with the '.' (dot) operator. If I were to create a class just like the mystuff module, I'd do something like this:

If I were to create a class just like the mystuff module, I'd do something like this:

```
class MyStuff(object):

      def __init__(self):
            self.tangerine = "And now a thousand years between"

      def apple(self):
            print "I AM CLASSY APPLES!"
```

That looks complicated compared to modules, and there is defi nitely a lot going on by comparison, but you should be able to make out how this is like a "mini- module" with MyStuff having an `apple()` function in it. What is probably confusing with this is the `__init__()` function and use of `self.tangerine` for setting the tangerine variable.

Here's why classes are used instead of modules: You can take the above class and use it to craft many of them, millions at a time if you want, and they won't interfere with each other. With modules, when you import there is only one for the entire program, unless you do some monster hacks.

Before you can understand this though, you need to know what an "object" is and how to work with MyStuff just like you do with the `mystuff.py` module.

Objects Are Like Mini- Imports

If a class is like a "mini- module," then there has to be a similar concept to import but for classes. That concept is called "instantiate," which is just a fancy, obnoxious, overly smart way to say "create." When you instantiate a class, what you get is called an object.

The way you do this is you call the class like it's a function, like this:

```
thing = MyStuff()
thing.apple()
print thing.tangerine
```

The first line is the "instantiate" operation, and it's a lot like calling a function. However, when you call this, there's a sequence of events that Python coordinates for you. I'll go through them using the above code for MyStuff:

1. Python looks for MyStuff() and sees that it is a class you've defined.
2. Python crafts an empty object with all the functions you've specified in the class using def.
3. Python then looks to see if you made a "magic" `__init__` function, and if you have, it calls that function to initialize your newly created empty object.
4. In the MyStuff function `__init__` I then get this extra variable self, which is that empty object Python made for me, and I can set variables on it just like you would with a module, dict, or other object.
5. In this case, I set `self.tangerine` to a song lyric and then I've initialized this object.
6. Now Python can take this newly minted object and assign it to the thing variable for me to work with.

That's the basics of how Python does this "mini- import" when you call a class like a function. Remember that this is not giving you the class, but instead it is using the class as a blueprint for how to build a copy of that type of thing.

Keep in mind that I'm giving you a slightly inaccurate idea for how these work so that you can start to build up an understanding of classes based on what you know of modules. The truth is, classes and objects suddenly diverge from modules at this point. If I were being totally honest, I'd say something more like this:

- Classes are like blueprints or definitions for creating new mini- modules.
- Instantiation is how you make one of these mini- modules and import it at the same time.
- The resulting created mini- module is called an object and you then assign it to a variable to work with it.

After this, though, classes and objects become very different from modules, and this should only serve as a way for you to bridge over to understanding classes.

Getting Things from Things

I now have three ways to get things from things:

```
# dict style
mystuff['apples']

# module style
mystuff.apples()
print mystuff.tangerine

# class style
thing = MyStuff()
thing.apples()
print thing.tangerine
```

A First-Class Example

You should start seeing the similarities in these three key=value container types and probably have a bunch of questions. Hang on with the questions, as the next exercise is going to hammer home your “object- oriented vocabulary.” In this exercise, I just want you to type in this code and get it working so that you have some experience before moving on.

```
class Song(object):

      def __init__(self, lyrics):
            self.lyrics = lyrics

      def sing_me_a_song(self):
            for line in self.lyrics:
                  print line

happy_bday = Song(["Happy birthday to you",
      "I don't want to get sued",
      "So I'll stop right there"])

bulls_on_parade = Song(["They rally around the family",
      "With pockets full of shells"])

happy_bday.sing_me_a_song()
bulls_on_parade.sing_me_a_song()
```

NOTE 1. Write some more songs using this, and make sure you understand that you’re passing a list of strings as the lyrics.

NOTE 2. Put the lyrics in a separate variable, then pass that variable to the class to use instead.

NOTE 3. See if you can hack on this and make it do more things. Don’t worry if you have no idea how, just give it a try, see what happens. Break it, trash it, thrash it, you can’t hurt it.

NOTE 4. Search on-line for “object- oriented programming” and try to overflow your brain with what you read. Don’t worry if it makes absolutely no sense to you. Half of that stuff makes no sense to me either.

QUESTION 1: **Why do I need self when I make __init__ or other functions for classes?**
If you don't have self, then code like cheese = 'Frank' is ambiguous. That code isn't clear about whether you mean the instance's cheese attribute or a local variable named cheese. With self.cheese = 'Frank' it's very clear you mean the instance attribute self.cheese.

b. Learning to Speak Object Oriented.

In this exercise, we are going to learn how to speak "object oriented." What we will do is to list a small set of words with definitions that we need to know. Then we will see a set of sentences with holes in them that we'll have to understand. Finally, we are going to give have set of exercises that we have to complete to make these sentences solid in our Python programming vocabulary.

Word Drills

- class—Tell Python to make a new kind of thing.
- object—Two meanings: the most basic kind of thing, and any instance of some thing.
- instance—What you get when you tell Python to create a class.
- def—How you define a function inside a class.
- self—Inside the functions in a class, self is a variable for the instance/object being accessed.
- inheritance—The concept that one class can inherit traits from another class, much like you and your parents.
- composition—The concept that a class can be composed of other classes as parts, much like how a car has wheels.
- attribute—A property classes have that are from composition and are usually variables.
- is- a—A phrase to say that something inherits from another, as in a "salmon" is- a "fi sh."
- has- a—A phrase to say that something is composed of other things or has a trait, as in "a salmon has- a mouth."

Phrase Drills

Next we have a list of Python code snippets on the left and the English sentences for them:

-
  ```
  class X(Y)
  ```
  "Make a class named X that is- a Y."

-
  ```
  class X(object):
     def __init__(self, J)
  ```
  "class X has- a __init__ that takes self and J parameters."

-
  ```
  class X(object):
     def M(self, J)
  ```
  "class X has- a function named M that takes self and J parameters."

-
  ```
  foo = X()
  ```
  "Set foo to an instance of class X."

-
  ```
  foo.M(J)
  ```
  "From foo get the M function, and call it with parameters self, J."

-
  ```
  foo.K = Q
  ```
  "From foo get the K attribute, and set it to Q."

In each of these where we see X, Y, M, J, K, Q, and foo, we can treat those like blank spots. For example, I can also write these sentences as follows:

1. "Make a class named ??? that is- a Y."
2. "class ??? has- a __init__ that takes self and ??? parameters."
3. "class ??? has- a function named ??? that takes self and ??? parameters."
4. "Set foo to an instance of class ???."
5. "From foo get the ??? function, and call it with self=??? and parameters ???."
6. "From foo get the ??? attribute and set it to ???."

Again, practice with similar sentence to get use to the concepts. You have to be able to say the sentence exactly the same every time whenever you see that form. Not sort of the same, but exactly the same.

c. Is- A, Has- A, Objects, and Classes.

An important concept that you have to understand is the difference between a class and an object. The problem is, there is no real "difference" between a class and an object. They are actually the same thing at different points in time. Let us ask a Zen like question:

```
What is the difference between a Fish and a Salmon?
```

Did that question sort of confuse you? Really sit down and think about it for a minute. I mean, a Fish and a Salmon are different but, wait, they are the same thing, right? A Salmon is a kind of Fish, so I mean it's not different. But at the same time, because a Salmon is a particular type of Fish, it's actually different from all other Fish. That's what makes it a Salmon and not a Halibut. So a Salmon and a Fish are the same but different. Weird.

This question is confusing because most people do not think about real things this way, but they intuitively understand them. You do not need to think about the difference between a Fish and a Salmon because you know how they are related. You know a Salmon is a kind of Fish and that there are other kinds of Fish without having to understand that.

Let's take it one step further: say you have a bucket full of three Salmon and, because you are a nice person, you have decided to name them Frank, Joe, and Mary. Now, think about this question:

```
What is the difference between Mary and a Salmon?
```

Again, this is a weird question, but it's a bit easier than the Fish versus Salmon question. You know that Mary is a Salmon, and so she's not really different. She's just a specific "instance" of a Salmon. Joe and Frank are also instances of Salmon. What do I mean when I say "instance"? I mean they were created from some other Salmon and now represent a real thing that has Salmon- like attributes.

Now for the mind- bending idea: Fish is a class, and Salmon is a class, and Mary is an object. Think about that for a second. Alright let's break it down real slow and see if you get it.

A Fish is a class, meaning it's not a real thing, but rather a word we attach to instances of things with similar attributes. Got fins? Got gills? Lives in water? Alright it's probably a Fish.

Someone with a PhD then comes along and says, "No, my young friend, this Fish is actually Salmosalar, affectionately known as a Salmon." This professor has just clarified the Fish further and made a new class called "Salmon" that has more specific attributes. Longer nose, reddish flesh, big, lives in the ocean or fresh water, tasty? OK, probably a Salmon.

Finally, a cook comes along and tells the PhD, "No, you see this Salmon right here, I'll call her Mary and

I'm going to make a tasty fillet out of her with a nice sauce." Now you have this instance of a Salmon (which also is an instance of a Fish) named Mary turned into something real that is filling your belly. It has become an object.

There you have it: Mary is a kind of Salmon that is a kind of Fish. object is a class is a class.

How This Looks in Code

This is a weird concept, but to be very honest, you only have to worry about it when you make new classes and when you use a class. I will show you two tricks to help you figure out whether something is a class or object.

First, you need to learn two catch phrases: "is- a" and "has- a." You use the phrase is- a when you talk about objects and classes being related to each other by a class relationship. You use has- a when you talk about objects and classes that are related only because they reference each other.

Now, go through this piece of code and replace each ## ?? comment with a replacement comment that says whether the next line represents an is- a or a has- a relationship and what that relationship is. In the beginning of the code, I've laid out a few examples, so you just have to write the remaining ones.

Remember, is- a is the relationship between Fish and Salmon, while has- a is the relationship between Salmon and Gills.

```
## Animal is- a object (yes, sort of confusing) look at the extra credit
class Animal(object):
      pass

## ??
class Dog(Animal):
      def __init__(self, name):
      ## ??
      self.name = name

## ??
class Cat(Animal):
      def __init__(self, name):
      ## ??
      self.name = name

## ??
class Person(object):
      def __init__(self, name):
      ## ??
      self.name = name
      ## Person has- a pet of some kind
      self.pet = None

## ??
class Employee(Person):
      def __init__(self, name, salary):
            ## ?? hmm what is this strange magic?
            super(Employee, self).__init__(name)
            ## ??
            self.salary = salary
```

```
## ??
class Fish(object):
      pass

## ??
class Salmon(Fish):
      pass

## ??
class Halibut(Fish):
      pass

## rover is- a Dog
rover = Dog("Rover")

## ??
satan = Cat("Satan")

## ??
mary = Person("Mary")

## ??
mary.pet = satan

## ??
frank = Employee("Frank", 120000)

## ??
frank.pet = rover

## ??
flipper = Fish()

## ??
crouse = Salmon()

## ??
harry = Halibut()
```

About class Name(object)

What is this `class Animal(object)`? Are we saying: “class is- a object”?. Python creator decided that to create a new class, we would use the word “object,” lowercased, to be the “class 0” that you inherit from to make a class. Confusing, right? A class inherits from the class named object to make a class, but it’s not an object. (Really it’s a class, but do not forget to inherit from object.)

What happened is Python’s original rendition of class was broken in many serious ways. By the time they admitted the fault, it was too late, and they had to support it. In order to fix the problem, they needed some “new class” style so that the “old classes” would keep working but you could use the new, more correct version. In Python 3, this confusion has being corrected. So to create a new (non Inherited) class, you would do:

```
class MyClass:
    """A simple example class"""
    i = 12345
```

```
        def f(self):
            return 'hello world'
```

As you can see, the “(object)” is no longer needed.

NOTE 1. Is it possible to use a class like it’s an object?
NOTE 3. Fill out the animals, fish, and people in this exercise with functions that make them do things. See what happens when functions are in a “base class” like Animal versus Dog.
NOTE 4. Find other people’s code and work out all the is- a and has- a relationships.
NOTE 5. Make some new relationships that are lists and dicts so you can also have “has- many” relationships.
NOTE 6. Do you think there’s a such thing as an “is- many” relationship? Read about “multiple inheritance,” then avoid it if you can.

QUESTION 1: **What are these ## ?? comments for?**
Those are “fill- in- the- blank” comments that you are supposed to fi ll in with the right “is- a,” “has- a” concepts. Re- read this exercise and look at the other comments to see what I mean.

QUESTION 2: **What is the point of self.pet = None?**
That makes sure that the self.pet attribute of that class is set to a default of None.

QUESTION 3: **What does super(Employee, self).__init__(name) do?**
That’s how you can run the __init__ method of a parent class reliably. Go search for “python super” and read the various advice on it being evil and good for you.

d. Inheritance vs. Composition.

In object- oriented programming, inheritance is the evil forest. Experienced programmers know to avoid this evil because they know that deep inside the dark forest of inheritance is the evil queen, multiple inheritance. She likes to eat software and programmers with her massive complexity teeth, chewing on the flesh of the fallen. But the forest is so powerful and so tempting that nearly every programmer has to go into it and try to make it out alive with the evil queen’s head before they can call themselves real programmers. One just can’t resist the inheritance forest’s pull, so you go in. After the adventure then we learn to just stay out of that stupid forest and bring an army if we are ever forced to go in again.

This is basically a funny way to say that we are learning something we should avoid, called inheritance. Programmers who are currently in the forest battling the queen will probably tell you that you have to go in. They say this because they need your help, since what they’ve created is probably too much for them to handle. But you should always remember this:

Most of the uses of inheritance can be simplified or replaced with composition, and multiple inheritance should be avoided at all costs.

What Is Inheritance?

Inheritance is used to indicate that one class will get most or all of its features from a parent class. This happens implicitly whenever you write class Foo(Bar), which says “Make a class Foo that inherits from Bar.” When you do this, the language makes any action that you do on instances of Foo also work as if they were done to an instance of Bar. Doing this lets you put common functionality in the Bar class, then specialize that functionality in the Foo class as needed.

When we are doing this kind of specialization, there are three ways that the parent and child classes can interact:

1. Actions on the child imply an action on the parent.
2. Actions on the child override the action on the parent.
3. Actions on the child alter the action on the parent.

We will now see each of these in order and the code for them.

I. Implicit Inheritance

First we will see the implicit actions that happen when we define a function in the parent, but not in the child.

```
class Parent(object):
      def implicit(self):
      print "PARENT implicit()"

class Child(Parent):
      pass

dad = Parent()
son = Child()

dad.implicit()
son.implicit()  #1
```

The use of pass under the class Child: is how you tell Python that you want an empty block. This creates a class named Child but says that there's nothing new to define in it. Instead, it will inherit all its behavior from Parent. When you run this code you get the following:

```
$ python ex44a.py
PARENT implicit()
PARENT implicit()
```

Notice how even though we are calling son.implicit() on line #1, and even though Child does not have an implicit function defined, it still works and it calls the one defined in Parent. This shows you that, if you put functions in a base class (i.e., Parent), then all subclasses (i.e., Child) will automatically get those features. Very handy for repetitive code you need in many classes.

II. Override Explicitly

The problem with implicitly having functions called is sometimes you want the child to behave differently. In this case, you want to override the function in the child, effectively replacing the functionality. To do this, just define a function with the same name in Child. Here's an example:

```
class Parent(object):

      def override(self):
            print "PARENT override()"

class Child(Parent):
        def override(self):
            print "CHILD override()"

dad = Parent()
son = Child()

dad.override()
son.override() #2
```

In this example, we have a function named override in both classes, so let's see what happens when we run it.

```
$ python ex44b.py
PARENT override()
CHILD override()
```

As you can see, when line #2 runs, it runs the `Parent.override` function because that variable (dad) is a Parent. But when line 15 runs, it prints out the `Child.override` messages because son is an instance of Child and Child overrides that function by defining its own version.

III. Alter Before or After

The third way to use inheritance is a special case of overriding where you want to alter the behavior before or after the Parent class's version runs. You first override the function just like in the last example, but then you use a Python built- in function named super to get the Parent version to call. Here's the example of doing that so you can make sense of this description:

```
class Parent(object):
    def altered(self):
        print "PARENT altered()"

class Child(Parent):
    def altered(self):
        print "CHILD, BEFORE PARENT altered()"    #1
        super(Child, self).altered()              #2
        print "CHILD, AFTER PARENT altered()"     #3


dad = Parent()
son = Child()

dad.altered()
son.altered()
```

The important lines here are #1–#3, where in the Child I do the following when `son.altered()` is called:

1. Because I've overridden `Parent.altered` the `Child.altered` version runs, and line #1 executes like you'd expect.
2. In this case, I want to do a before and after, so after line #1, I want to use super to get the `Parent.altered` version.
3. On line #2, I call `super(Child, self).altered()`, which is a lot like the getattr function you've used in the past, but it's aware of inheritance and will get the Parent class for you. You should be able to read this as "call super with arguments Child and self, then call the function

altered on whatever it returns."

4. At this point, the `Parent.altered` version of the function runs, and that prints out the Parent message.
5. Finally, this returns from the `Parent.altered`, and the `Child.altered` function continues to print out the after message.

If we then run this, we should see the following:

```
$ python ex44c.py
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

IV. All Three Combined

To demonstrate all these, I have a final version that shows each kind of interaction from inheritance in one file:

```
class Parent(object):

    def override(self):
        print "PARENT override()"

    def implicit(self):
        print "PARENT implicit()"

    def altered(self):
        print "PARENT altered()"

class Child(Parent):

    def override(self):
        print "CHILD override()"

    def altered(self):
        print "CHILD, BEFORE PARENT altered()"
        super(Child, self).altered()
        print "CHILD, AFTER PARENT altered()"


dad = Parent()
son = Child()

dad.implicit()
son.implicit()

dad.override()
son.override()

dad.altered()
son.altered()
```

Go through each line of this code, and write a comment explaining what that line does and whether it's an override or not. Then run it and see that we get what we expected:

```
$ python ex44d.py
PARENT implicit()
PARENT implicit()
PARENT override()
CHILD override()
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

The Reason for super()

This should seem like common sense, but then we get into trouble with a thing called multiple inheritance. Multiple inheritance is when you defi ne a class that inherits from one or more classes, like this:

```
class SuperFun(Child, BadStuff):
pass
```

This is like saying, "Make a class named SuperFun that inherits from the classes Child and BadStuff at the same time."

In this case, whenever you have implicit actions on any SuperFun instance, Python has to look up the possible function in the class hierarchy for both Child and BadStuff, but it needs to do this in a consistent order. To do this, Python uses something called "method resolution order" (MRO) and an algorithm called C3 to get it straight.

Because the MRO is complex and a well- defined algorithm is used, Python can't leave it to you to get it right. That'd be annoying, wouldn't it? Instead, Python gives you the super() function, which handles all this for you in the places that you need the altering type of actions demonstrated in `Child.altered` above. With super(), you don't have to worry about getting this right, and Python will find the right function for you.

Using super() with __init__

The most common use of `super()` is actually in `__init__` functions in base classes. This is usually the only place where you need to do some things in a child, then complete the initialization in the parent. Here's a quick example of doing that in the Child from these examples:

```
class Child(Parent):
      def __init__(self, stuff):
            self.stuff = stuff
            super(Child, self).__init__()
```

This is pretty much the same as the `Child.altered` example above, except I'm setting some vari-ables in the `__init__` before having the Parent initialize with its `Parent.__init__`.

Composition

Inheritance is useful, but another way to do the exact same thing is just to use other classes and modules, rather than rely on implicit inheritance. If you look at the three ways to exploit inheritance, two of the three involve writing new code to replace or alter functionality. This can easily be replicated by just calling functions in another class. Here's an example of doing this:

```
class Other(object):
      def override(self):
            print "OTHER override()"
      def implicit(self):
            print "OTHER implicit()"
      def altered(self):
            print "OTHER altered()"

class Child(object):
      def __init__(self):
            self.other = Other()
      def implicit(self):
            self.other.implicit()
      def override(self):
            print "CHILD override()"
      def altered(self):
            print "CHILD, BEFORE OTHER altered()"
            self.other.altered()

print "CHILD, AFTER OTHER altered()"
son = Child()
son.implicit()
son.override()
son.altered()
```

In this code we are not using the name Parent, since there is not a parent- child is- a relationship. This is a has- a relationship, where Child has- a Other that it uses to get its work done. When we run this, we get the following output:

```
$ python ex44e.py
OTHER implicit()
CHILD override()
CHILD, BEFORE OTHER altered()
```

You can see that most of the code in Child and Other is the same to accomplish the same thing. The only difference is that I had to define a `Child.implicit` function to do that one action. I could then ask myself if I need this Other to be a class, and could I just make it into a module named other.py?

When to Use Inheritance or Composition

The question of inheritance versus composition comes down to an attempt to solve the problem of reusable code. You don't want to have duplicated code all over your software, since that's not clean and efficient. Inheritance solves this problem by creating a mechanism for you to have implied features in base classes. Composition solves this by giving you modules and the ability to simply call functions in other classes.

If both solutions solve the problem of reuse, then which one is appropriate in which situations? The answer is incredibly subjective, but I'll give you my three guidelines for when to do which:

1. Avoid multiple inheritance at all costs, as it's too complex to be useful reliably. If you're stuck with it, then be prepared to know the class hierarchy and spend time finding where everything

is coming from.
2. Use composition to package up code into modules that are used in many different unrelated places and situations.
3. Use inheritance only when there are clearly related reusable pieces of code that fit under a single common concept or if you have to because of something you're using.

However, do not be a slave to these rules. The thing to remember about object- oriented programming is that it is entirely a social convention programmers have created to package and share code. Because it's a social convention, but one that's codified in Python, you may be forced to avoid these rules because of the people you work with. In that case, find out how they use things and then just adapt to the situation.

QUESTION 1: **How do I get better at solving problems that I haven't seen before?**
The only way to get better at solving problems is to solve as many problems as you can by yourself. Typically people hit a difficult problem and then rush out to find an answer. This is fine when you have to get things done, but if you have the time to solve it yourself, then take that time. Stop and bang your head against the problem for as long as possible, trying every possible thing, until you solve it or give up. After that, the answers you find will be more satisfying and you'll eventually get better at solving problems.

QUESTION 2: **Aren't objects just copies of classes?**
In some languages (like JavaScript), that is true. These are called prototype languages and there are not many differences between objects and classes other than usage. In Python, however, classes act as templates that "mint" new objects, similar to how coins were minted using a die(template).

e. Python Exercises I.

This exercise section is is aimed at for the folks who have just gone through an introductory Python course. He or she should be able to solve some problems with 1 or 2 Python classes or functions.

Question 1.1

Question:
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.

Hints:
Consider use range(#begin, #end) method

Question 1.2

Question:
Write a program which can compute the factorial of a given numbers. The results should be printed in a comma-separated sequence on a single line.

Suppose the following input is supplied to the program: `8`,
Then, the output should be: `40320`

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input.

Question 1.3

Question:
With a given integral number n, write a program to generate a dictionary that contains (i, i*i) such that is an integral number between 1 and n (both included). and then the program should print the dictionary.
Suppose the following input is supplied to the program:
```
8
```
Then, the output should be:
```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64}
```

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input. Consider use dict().

Question 1.4

Question:
Write a program which accepts a sequence of comma-separated numbers from console and generate a list and a tuple which contains every number.
Suppose the following input is supplied to the program:
```
34,67,55,33,12,98
```
Then, the output should be:
```
['34', '67', '55', '33', '12', '98']
('34', '67', '55', '33', '12', '98')
```

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input. tuple() method can convert list to tuple

Question 1.5

Question:
Define a class which has at least two methods: getString: to get a string from console input printString: to print the string in upper case. Also please include simple test function to test the class methods.

Hints:
Use __init__ method to construct some parameters

Question 1.6

Question:
Write a method which can calculate square value of number

Hints:
Using the ** operator

Question 1.7

Question:
Python has many built-in functions, and if you do not know how to use it, you can read document on-line or find some books. But Python has a built-in document function for every built-in functions. Please write a program to print some Python built-in functions documents, such as abs(), int(), raw_input(). And add document for your own function.

Hints:
The built-in document method is __doc__

Question 1.8

Question:
Define a class, which have a class parameter and have a same instance parameter.

Hints:
Define a instance parameter, need add it in __init__ method. You can init a object with construct parameter or set the value later

Question 1.9

Question:
Write a program to generate and print another tuple whose values are even numbers in the given tuple (1,2,3,4,5,6,7,8,9,10).

Hints:
Use "for" to iterate the tuple. Use tuple() to generate a tuple from a list.

Question 1.10

Question:
Write a function to compute 5/0 and use try/except to catch the exceptions.

Hints:
Use try/except to catch exceptions.

Question 1.11

Question:
Define a class named American and its subclass NewYorker.

Hints:
Use class Subclass(ParentClass) to define a subclass.

f. Python Exercises II.

This exercise section is at a intermediate level, it meant for someone who has just learned Python, but already familiar with basic computer programing knowledge. He or she should be able to solve problems which may involve 3 or 3 Python classes or functions.

Question 2.1

Question:
Write a program that calculates and prints the value according to the given formula:
Q = Square root of [(2 * C * D)/H]
Following are the fixed values of C and H:
C is 50. H is 30.
D is the variable whose values should be input to your program in a comma-separated sequence.

Example
Let us assume the following comma separated input sequence is given to the program:
```
100,150,180
```
The output of the program should be:
```
18,22,24
```

Hints:
If the output received is in decimal form, it should be rounded off to its nearest value (for example, if the output received is 26.0, it should be printed as 26). In case of input data being supplied to the question, it should be assumed to be a console input.

Question 2.2

Question:
Write a program which takes 2 digits, X,Y as input and generates a 2-dimensional array. The element value in the i-th row and j-th column of the array should be i*j.

Note: i=0,1.., X-1; j=0,1.., Y-1.

Example
Suppose the following inputs are given to the program:
```
3,5
```
Then, the output of the program should be:
```
[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8]]
```

Hints:
Note: In case of input data being supplied to the question, it should be assumed to be a console input in a comma-separated form.

Question 2.3

Question:
Write a program that accepts sequence of lines as input and prints the lines after making all characters in the sentence capitalized.

Suppose the following input is supplied to the program:
```
Hello world
Practice makes perfect
```
Then, the output should be:
```
HELLO WORLD
PRACTICE MAKES PERFECT
```

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input.

Question 2.4

Question:
Write a program that accepts a sequence of whitespace separated words as input and prints the words after removing all duplicate words and sorting them alphanumerically.

Suppose the following input is supplied to the program:
```
hello world and practice makes perfect and hello world again
```
Then, the output should be:
```
again and hello makes perfect practice world
```

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input. We use set container to remove duplicated data automatically and then use sorted() to sort the data.

Question 2.5

Question:
Write a program which accepts a sequence of comma separated 4 digit binary numbers as its input and then check whether they are divisible by 5 or not. The numbers that are divisible by 5 are to be printed in a comma separated sequence.

Example:
```
0100,0011,1010,1001
```
Then the output should be:
```
1010
```

Notes: Assume the data is input by console.

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input.

Question 2.6

Question:
Write a program, which will find all such numbers between 1000 and 3000 (both included) such that each digit of the number is an even number. The numbers obtained should be printed in a comma-separated sequence on a single line.

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input.

Question 2.7

Question:
Write a program that accepts a sentence and calculate the number of letters and digits.

Suppose the following input is supplied to the program:
```
hello world! 123
```
Then, the output should be:
```
LETTERS 10
DIGITS 3
```

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input.

Question 2.8

Question:
Write a program that accepts a sentence and calculate the number of upper case letters and lower case letters.
Suppose the following input is supplied to the program:
```
Hello world!
```
Then, the output should be:
```
UPPER CASE 1
LOWER CASE 9
```

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input.

Question 2.9

Question:
Write a program that computes the value of a+aa+aaa+aaaa with a given digit as the value of a.
Suppose the following input is supplied to the program:
```
9
```
Then, the output should be:
```
11106
```

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input.

Question 2.10

Question:
Use a list comprehension to square each odd number in a list. The list is input by a sequence of comma-separated numbers.

Suppose the following input is supplied to the program:
```
1,2,3,4,5,6,7,8,9
```
Then, the output should be:
```
1,3,5,7,9
```

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input.

Question 2.11

Question:
Write a program that computes the net amount of a bank account based a transaction log from console input. The transaction log format is shown as following:
```
D 100
W 200
```
where D means deposit while W means withdrawal.

Suppose the following input is supplied to the program:
```
D 300
D 300
W 200
D 100
```
Then, the output should be:
```
500
```

Hints:
In case of input data being supplied to the question, it should be assumed to be a console input.