Read and write configuration files using python

There are several file formats you can use for your configuration file, the most commonly used format are .ini, .json and .yaml.

Read .ini file

Below is a example of the ini file, you can define the sections (e.g. [LOGIN]) as much as you want to separate the different configuration info.

```
[LOGIN]
user = admin
#Please change to your real password
password = admin
[SERVER]
host = 192.168.0.1
port = 8088
```

In python, there is already a module configparser to read and parse the information from the ini file in to dictionary objects. Assume you have saved above as config.ini file into your current folder, you can use the below lines of code to read.

```
import configparser
config = configparser.ConfigParser()
config.read("config.ini")
login = config['LOGIN']
server = config['SERVER']
```

You can assign each of the sections into a separate dictionary for easier accessing the values. The output should be same as below:

```
login["user"]
```

```
'admin'
```

```
server["host"]
```

```
'192.168.0.1'
```

Note that the line starting with # symbol (or ;) will be taken as comment line and omitted when parsing the keys and values.

Also all the values are taken as string, so you will need to do your own data type conversion after you read it.

Write to .ini file

Now let's see how we can write to an ini file.

You will still need this configparser library, and the idea is that you need to set the keys and values into the configparser object and then save it into a file.

```
config = configparser.ConfigParser()
if not config.has_section("INFO"):
```

```
config.add_section("INFO")
config.set("INFO", "link", "www.codeforests.com")
config.set("INFO", "name", "ken") with open("example.ini", 'w') as configfile:
config.write(configfile)
```

And this would create the example.ini file with below content:

```
[INFO]
link = www.codeforests.com
name = ken
```

Reading and Writing to text files in Python

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language: 0s and 1s).

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

File Access Modes

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the **File Handle** in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

1. **Read Only ('r') :** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
2. **Read and Write ('r+') :** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
3. **Write Only ('w') :** Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists.
4. **Write and Read ('w+')** : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
5. **Append Only ('a')** : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. **Append and Read ('a+') :** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Opening a File

It is done using the open() function. No module is required to be imported for this function.

File_object = open(r"File_Name","Access_Mode")

The file should exist in the same directory as the python program file else, full address of the file should be written on place of filename.

Note: The **r** is placed before filename to prevent the characters in filename string to be treated as special character. For example, if there is \temp in the file address, then \t is treated as the tab character and error is raised of invalid address. The r makes the string raw, that is, it tells that the string is without any special characters. The r can be ignored if the file is in same directory and address is not being placed.

```
file1 = open("MyFile.txt","a")
file2 = open(r"D:\Text\MyFile2.txt","w+")
```

Here, file1 is created as object for MyFile1 and file2 as object for MyFile2

Closing a file

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

File_object.close()

```
file1 = open("MyFile.txt","a")
file1.close()
```

Writing to a file

There are two ways to write in a file.

1. **write() :** Inserts the string str1 in a single line in the text file.
   File_object.write(str1)
2. **writelines() :** For a list of string elements, each string is inserted in the text file.Used to insert multiple strings at a single time.
   File_object.writelines(L) for L = [str1, str2, str3]

Reading from a file

There are three ways to read data from a text file.

1. **read() :** Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.
   File_object.read([n])
2. **readline() :** Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.
   File_object.readline([n])
3. **readlines() :** Reads all the lines and return them as each line a string element in a list.
   File_object.readlines()

Note: '\n' is treated as a special character of two bytes

Examples:

```
file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]
file1.write("Hello \n")
file1.writelines(L)
file1.close()
file1 = open("myfile.txt","r+")
```

```
print("Output of Read function is ")
print(file1.read())
# move to start of file again
file1.seek(0)
print("Output of Readline function is ")
print(file1.readline())
# move to start of file again
file1.seek(0)
print("Output of Read(9) function is ")
print(file1.read(9))
# move to start of file again
file1.seek(0)
print("Output of Readline(9) function is ")
print(file1.readline(9))
# move to start of file again
file1.seek(0)
print("Output of Readlines function is ")
print(file1.readlines())
file1.close()
```

Output:

```
Output of Read function is
Hello
This is Delhi
This is Paris
This is London

Output of Readline function is
Hello

Output of Read(9) function is
Hello
Th

Output of Readline(9) function is
Hello

Output of Readlines function is
['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']
```

Appending to a file

```
file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]
file1.close()
```

```
# Append-adds at last
file1 = open("myfile.txt","a")#append mode
file1.write("Today \n")
file1.close()

file1 = open("myfile.txt","r")
print("Output of Readlines after appending")
print(file1.readlines())
file1.close()

# Write-Overwrites
file1 = open("myfile.txt","w")#write mode
file1.write("Tomorrow \n")
file1.close()

file1 = open("myfile.txt","r")
print("Output of Readlines after writing")
print(file1.readlines())
file1.close()
```

Output:
Output of Readlines after appending
['This is Delhi \n', 'This is Paris \n', 'This is London \n', 'Today \n']

Output of Readlines after writing
['Tomorrow \n']

Using write along with with() function
We can also use write function along with with() function:

```
# Python code to illustrate with() alongwith write()
with open("file.txt", "w") as f:
    f.write("Hello World!!!")
```

Object Oriented Programming
The programs designed around functions i.e. blocks of statements which manipulate data are called the procedure-oriented way of programming. There is another way of organizing the program that combine data and functionality and wrap it inside something called an object. This is called the object oriented programming paradigm. Most of the time we can use procedural programming, but when writing large programs or have a problem that is better suited to this method, we can use object oriented programming techniques.
Classes and **objects** are the two main aspects of object oriented programming. A class creates a new type where objects are instances of the class.
Objects can store data using ordinary variables that belong to the object. Variables that belong to an object or class are referred to as fields. Objects can also have functionality by using functions that belong to a class. Such functions are called methods of the class. Fields are of two types -

they can belong to each instance/object of the class or they can belong to the class itself. They are called instance variables and class variables respectively.
A class is created using the ***class*** keyword. The fields and methods of the class are listed in an indented block.

The self
Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you do not give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object itself, and by convention, it is given the name self.
Although, you can give any name for this parameter, it is strongly recommended that you use the name self.
The self in Python is equivalent to the ***this*** pointer in C++ and the ***this*** reference in Java and C#.
If you have a method which takes no arguments, then you still have to have one argument - the self.

Classes
The simplest class possible is shown in the following example (save as oop_simplestclass.py).

```
class Person:
   pass  # An empty block

p = Person()
print(p)
```

Output:

```
$ python oop_simplestclass.py
<__main__.Person instance at 0x10171f518>
```

Notice that the address of the computer memory where your object is stored is also printed. The address will have a different value on your computer since Python can store the object wherever it finds space.

Methods
Classes/objects can have methods just like functions except that we have an extra self variable. We will now see an example (save as oop_method.py).

```
class Person:
   def say_hi(self):
      print('Hello, how are you?')

p = Person()
p.say_hi()
```

Output:

```
$ python oop_method.py
Hello, how are you?
```

The __init__ method

The __init__ method is run as soon as an object of a class is instantiated (i.e. created). The method is useful to do any initialization (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores both at the beginning and at the end of the name.
Example (save as oop_init.py):

```
class Person:
    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Swaroop')
p.say_hi()
```

Output:

```
$ python oop_init.py
Hello, my name is Swaroop
```

Class variables And Instance Variables

The data part, i.e. fields, are nothing but ordinary variables that are bound to the namespaces of the classes and objects. This means that these names are valid within the context of these classes and objects only. That's why they are called name spaces.
There are two types of fields - ***class variables*** and ***object variables(instance variables)*** which are classified depending on whether the class or the object owns the variables respectively.
Class variables are shared - they can be accessed by all instances of that class. There is only one copy of the class variable and when any one object makes a change to a class variable, that change will be seen by all the other instances.
Object variables are owned by each individual object/instance of the class. In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance. An example will make this easy to understand (save as oop_objvar.py):

```
class Robot:
    # A class variable, counting the number of robots
    population = 0

    def __init__(self, name):
        self.name = name
        Robot.population += 1

    def die(self):
      print("{} is being destroyed!".format(self.name))
      Robot.population -= 1

        if Robot.population == 0:
            print("{} was the last one.".format(self.name))
        else:
            print("There are still {:d} robots working.".format(Robot.population))
```

```
    def say_hi(self):
        print("Greetings, my masters call me {}.".format(self.name))

    @classmethod
    def how_many(cls):
        """Prints the current population."""
        print("We have {:d} robots.".format(cls.population))


droid1 = Robot("R2-D2")
droid1.say_hi()
Robot.how_many()

droid2 = Robot("C-3PO")
droid2.say_hi()
Robot.how_many()

print("\nRobots can do some work here.\n")

print("Robots have finished their work. So let's destroy them.")
droid1.die()
droid2.die()

Robot.how_many()
```

Output:

```
$ python oop_objvar.py
Greetings, my masters call me R2-D2.
We have 1 robots.
Greetings, my masters call me C-3PO.
We have 2 robots.

Robots can do some work here.

Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3PO is being destroyed!
C-3PO was the last one.
We have 0 robots.
```

Inheritance

One of the major benefits of object oriented programming is reuse of code and one of the ways this is achieved is through the inheritance mechanism. Inheritance can be best imagined as implementing a type and subtype relationship between classes.

To make a class inherit from another, we apply the name of the base class in parentheses to the derived class' definition.

```
class Person:
        pass
class Student(Person):
        pass
```

here, student is a subclass of person.

We will now see this example as a program (save as oop_subclass.py):

```
class SchoolMember:
    def __init__(self, name, age):
      self.name = name
      self.age = age
    def tell(self):
     print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")

class Teacher(SchoolMember):
    def __init__(self, name, age, salary):
      SchoolMember.__init__(self, name, age)
      self.salary = salary

   def tell(self):
      SchoolMember.tell(self)
      print('Salary: "{:d}"'.format(self.salary))


class Student(SchoolMember):
   '''Represents a student.'''
   def __init__(self, name, age, marks):
      SchoolMember.__init__(self, name, age)
      self.marks = marks

   def tell(self):
      SchoolMember.tell(self)
      print('Marks: "{:d}"'.format(self.marks))

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)

# prints a blank line
print()

members = [t, s]
for member in members:
   # Works for both Teachers and Students
   member.tell()
```

Output:
$ python oop_subclass.py
Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"25" Marks: "75"

Types of Inheritance in Python

There are five types of inheritance in python.

a. Single Inheritance

A single inheritance is when a single class inherits from a class.

```
class fruit:
        def __init__(self):
                print("I'm a fruit")
class citrus(fruit):
        def __init__(self):
                super().__init__()
                print("I'm citrus")

lime=citrus()
```

I'm a fruit
I'm citrus

b. Multiple Inheritance

Multiple inheritance are when a class inherits from multiple base classes.

```
class Color:
        pass
class Fruit:
        pass
class Orange(Color,Fruit):
        pass
```

c. Multilevel Inheritance

When one class inherits from another, which in turn inherits from another, it is multilevel python inheritance.

```
class A:
        x=1
class B(A):
        pass
class C(B):
        pass
cobj=C()
cobj.x
```

1

d. Hierarchical Inheritance

When more than one class inherits from a class, it is hierarchical Python inheritance.

```
class A:
        pass
class B(A):
        pass
```

```
class C(A):
        pass
```

e. Hybrid Inheritance

Hybrid Python inheritance is a combination of any two or more kinds of inheritance.

```
class A:
        x=1
class B(A):
        pass
class C(A):
        pass
class D(B,C):
        pass
dobj=D()
dobj.x
1
```

Method Overriding

A subclass may change the functionality of a Python method in the superclass. It does so by redefining it. This is termed python method overriding. Lets see this Python Method Overriding Example.

```
        class A:
                def sayhi(self):
                        print("I'm in A")
        class B(A):
                def sayhi(self):
                        print("I'm in B")
        bobj=B()
        bobj.sayhi()
```

output:

I’m in B

Data Hiding

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. By default all members of a class can be accessed outside of class.

You can prevent this by making class members private or protected.

In Python, we use double underscore (__) before the attributes name to make those attributes ***private***.

We can use single underscore (_) before the attributes name to make those attributes ***protected***.

```
class MyClass:
   __hiddenVariable = 0     # Private member of MyClass
   _protectedVar = 0        # Protected member of MyClass

   # A member method that changes __hiddenVariable
```

```
    def add(self, increment):
        self.__hiddenVariable += increment
        print (self.__hiddenVariable)

myObject = MyClass()
myObject.add(2)
myObject.add(5)

# This will causes error
print (MyClass.__hiddenVariable)
print (MyClass._protectedVar)
```

In the above program, we tried to access hidden variable outside the class using object and it threw an exception.

We can access the value of hidden attribute by a tricky syntax as **object._className__attrName.**

```
# A Python program to demonstrate that hidden members can be accessed outside a class
class MyClass:
    __hiddenVariable = 10    # Hidden member of MyClass

myObject = MyClass()
print(myObject._MyClass__hiddenVariable)
```

Private methods are accessible outside their class, just not easily accessible. Nothing in Python is truly private.

Difference between public, private and protected:

Mode	Description
Public	A public member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function. By default all the members of a class would be public
Private	A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class members can access private members. Practically, we make data private and related functions public so that they can be called from outside of the class
Protected	A protected member is very similar to a private member but it provided one additional benefit that they can be accessed in sub classes which are called derived/child classes.

Python Operator Overloading

You can change the meaning of an operator in Python depending upon the operands used. Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.
So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y


p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

Output

```
Traceback (most recent call last):
  File "<string>", line 9, in <module>
    print(p1+p2)
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Here, we can see that a TypeError was raised, since Python didn't know how to add two Point objects together.

Python Special Functions
Suppose we want the print() function to print the coordinates of the Point object instead of what we got. We can define a __str__() method in our class that controls how the object gets printed. Let's look at how we can achieve this:

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
```

Now let's try the print() function again.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0}, {1})".format(self.x, self.y)
```

```
p1 = Point(2, 3)
print(p1)
```

Output

```
(2, 3)
```

Overloading the + Operator

To overload the + operator, we will need to implement __add__() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is more sensible to return a Point object of the coordinate sum.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
```

Now let's try the addition operation again:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)
```

```
print(p1+p2)
```

Output

```
(3,5)
```

What actually happens is that, when you use p1 + p2, Python calls p1.__add__(p2) which in turn is Point.__add__(p1,p2). After this, the addition operation is carried out the way we specified. Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Overloading Comparison Operators

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.
Suppose we wanted to implement the less than symbol < symbol in our Point class.
Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

```
# overloading the less than operator
```

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# use less than
print(p1<p2)
print(p2<p3)
print(p1<p3)
```

Output

```
True
False
False
```

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)