Logging in Python

Logging is a means of tracking events that happen when some software runs. Logging is important for software developing, debugging, and running. If you don't have any logging record and your program crashes, there are very few chances that you detect the cause of the problem. And if you detect the cause, it will consume a lot of time. With logging, you can leave a trail of breadcrumbs so that if something goes wrong, we can determine the cause of the problem.

Python has a built-in module **logging** which allows writing status messages to a file or any other output streams. The file can contain the information on which part of the code is executed and what problems have been arisen.

Levels of Log Message

There are five built-in levels of the log message.

- **Debug :** These are used to give Detailed information, typically of interest only when diagnosing problems.
- **Info :** These are used to confirm that things are working as expected
- **Warning :** These are used an indication that something unexpected happened, or is indicative of some problem in the near future
- **Error :** This tells that due to a more serious problem, the software has not been able to perform some function
- **Critical :** This tells serious error, indicating that the program itself may be unable to continue running

Each built-in level has been assigned its numeric value.

Level	Numeric Value
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

Example 1:

```
# importing module
# log report in a file with time-stamp

import logging

# Create and configure logger
logging.basicConfig(filename="newfile.log",
                format='%(asctime)s %(message)s',
                filemode='w')

# Creating an object
logger = logging.getLogger()

# Setting the threshold of logger to DEBUG
logger.setLevel(logging.DEBUG)

# Test messages
logger.debug("Harmless debug Message")
logger.info("Just an information")
logger.warning("Its a Warning")
logger.error("Did you try to divide by zero")
logger.critical("Internet is down")
```

Example 2:

```
# class with out logging

class Pizza():
    def __init__(self, name, price):
        self.name = name
        self.price = price
        print("Pizza created: {} for ${}".format(self.name, self.price))

    def make(self, quantity=1):
        print("Made {} {} pizza".format(quantity, self.name))

    def eat(self, quantity=1):
        print("Ate {} pizza".format(quantity, self.name))

p1 = Pizza("Sicilian ", 15)
p1.make()
p1.eat()
```

```
p2 = Pizza("Margherita", 12)
p2.make(2)
p2.eat()
```

Example 3:

```
# class with logging

import logging

logging.basicConfig(level=logging.DEBUG)
#logging.basicConfig(filename="test.log", level=logging.DEBUG)

class Pizza():
    def __init__(self, name, price):
        self.name = name
        self.price = price
        logging.debug("Pizza created: {} (${})".format(self.name, self.price))

    def make(self, quantity=1):
       logging.debug("Made {} {} pizza(s)".format(quantity, self.name))

    def eat(self, quantity=1):
        logging.debug("Ate {} pizza(s)".format(quantity, self.name))

p1 = Pizza("Sicilian ", 15)
p1.make()
p1.eat()

p2 = Pizza("Margherita", 12)
p2.make(2)
p2.eat()
```

Python object serialization

Python pickle module is used for serializing and de-serializing python object structures. The process to converts any kind of python objects (list, dict, etc.) into byte streams (0s and 1s) is called pickling or serialization or flattening or marshalling. We can convert the byte stream (generated through pickling) back into python objects by a process called as unpickling.

In real world scenario, the use pickling and unpickling are widespread as they allow us to easily transfer data from one server/system to another and then store it in a file or database.

```
# Program for pickling

import pickle

class Employee:
    def __init__(self, eno, name, esal, address):
        self.eno=eno;
        self.name = name;
        self.esal=esal;
        self.address= address;

    def show(self):
        print(self.eno,  self.name,  self.esal,  self.address)

#pickling
with open("D://emp.dat", "wb") as f:
    e =Employee(111,'chandu',25000,'vizag')
    pickle.dump(e,f)

print("Pickling of employee object completed...")


#unpickling
with open("D://emp.dat", "rb") as f:
    obj=pickle.load(f)

print("unpickling finished")
obj.show()
```

Python zip():

The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and returns it.

```
# zip function
user_name = ['user1', 'user2', 'user3']
values = ['ramu', 'somu', 'gomu']

# zipping the values
print(zip(user_name, values))

# zipping and converting to list
print(list(zip(user_name, values)))

# zipping and converting to dictionary
print(dict(zip(user_name, values)))



# converting zip values to dictionary
sample = [('a',1), ('b',2), ('c',3)]
print(dict(sample))


l = [(1,2),(3,4),(5,6),(7,8)]
print(list(zip(*l)))


l1, l2 = list(zip(*l))
print(l1)
print(l2)
print(list(l1))
print(list(l2))


# get max no's of two lists
l1 = [1,9,11,7]
l2 = [2,4,6,8]
new_list = []

for pair in zip(l1,l2):
```

```
    new_list.append(max(pair))
print(new_list)
```

More Examples:

Ex: 1

```
languages = ['Java', 'Python', 'JavaScript']
versions = [14, 3, 6]

result = zip(languages, versions)
print(list(result))

# Output: [('Java', 14), ('Python', 3), ('JavaScript', 6)]
```

Ex: 2

```
numbersList = [1, 2, 3]
str_list = ['one', 'two']
numbers_tuple = ('ONE', 'TWO', 'THREE', 'FOUR')

# Notice, the size of numbersList and numbers_tuple is different
result = zip(numbersList, numbers_tuple)

# Converting to set
result_set = set(result)
print(result_set)

result = zip(numbersList, str_list, numbers_tuple)

# Converting to set
result_set = set(result)
print(result_set)
```

Ex: 3

```
# Unzipping the Value Using zip()
coordinate = ['x', 'y', 'z']
value = [3, 4, 5]

result = zip(coordinate, value)
result_list = list(result)
```

```
print(result_list)

c, v =  zip(*result_list)
print('c =', c)
print('v =', v)
```

Ex: 4

```
# Python code to demonstrate the application of
# zip()

# initializing list of players.
players = ["Sachin", "Sehwag", "Gambhir", "Dravid", "Raina"]

# initializing their scores
scores = [100, 15, 17, 28, 43]

# printing players and scores.
for pl, sc in zip(players, scores):
   print("Player : %s    Score : %d" % (pl, sc))
```