```
# Iterator Example

# iterate over something in Python
my_numbers=(1,2,3,4,5)

# "under the hood" there's a design pattern at work
for number in my_numbers:
   print(number)

# doing the same as before, but now
# explicitly calling the iterator
it = iter(my_numbers)
print(next(it))
print(next(it))
print(next(it))

# or with loop
# experiment with this. For example, run it to a
certain number and then start a loop again, how does it
behave compared to doing the same in a for loop
while True:
   try:
       number = next(it)
   except StopIteration:
       break
   else:
       print(number)

# iterating over a file preserves state
def parse_email(f):
   envelope = ''
   for line in f:
       envelope.join(line)
       break
   headers = {}
   for line in f:
```

```
        if line == '\n':
            break
        name, value = line.split(':', 1)
        headers[name.strip()] =
value.lstrip().rstrip('\n')
    body = []
    for line in f:
        if line.startswith('From'):
            break
        body.append(line)
    return envelope, headers, body

with open("email.txt") as f:
    envelope, header, body = parse_email(f)

print(body)

# make your own iterator with your own objects:

# 1. an iterable class: create iterator object
class OddIterator():
    def __init__(self, container):
        self.container = container
        self.n = -1

    def __next__(self):
        self.n += 2
        if self.n > self.container.maximum:
            raise StopIteration

        return self.n

    def __iter__(self):
        return self
```

```
# 2. an iterator class: traverse the container
class OddNumbers:
    def __init__(self, maximum):
        self.maximum = maximum

    def __iter__(self):
        return OddIterator(self)

numbers = OddNumbers(7)
#
for number in numbers:
    print(number)

# First: prep for Singleton: What is __new__
class A():
    def __new__(cls):
        print("Creating instance")
        return super(A, cls).__new__(cls)

    def __init__(self):
        print("Init is called")

a = A()


class Singleton:
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton,
cls).__new__(cls)
        return cls.instance

s = Singleton()
```

```
print("Object created:", s)
s1 = Singleton()
print("Object created:", s1)
# Singleton solved with decorator
#
https://medium.com/better-programming/singleton-in-pyth
on-5eaa66618e3d
class Singleton:
    def __init__(self, cls):
        self._cls = cls

    def Instance(self):
        try:
            return self._instance
        except AttributeError:
            self._instance = self._cls()
            return self._instance

    def call(self):
        raise TypeError

    def __instancecheck__(self, instance):
        return isinstance(instance, self._cls)

@Singleton
class DBConnection:
    def __init__(self):
        print("established db connection")

    def __str__(self):
        return "Database connection object"

db_con = DBConnection()    #causes the TypeError

db_connection1 = DBConnection.Instance()
db_connection2 = DBConnection.Instance()
```

```
print(f"ID of connection 1: {id(db_connection1)}")
print(f"ID of connection 2: {id(db_connection1)}")
# decorators
# function revision:
#
https://medium.com/better-programming/decorators-in-pyt
hon-72a1d578eac4

def say_name():
    print("Bianca")


def say_nationality():
    print("German")


def say(function):
    return function


say(say_name)()
say(say_nationality)()

# now with an inner function
# output will be the same

def say():

    def say_name():
        print("Bianca")

    def say_nationality():
        print("German")

    say_name()
```

```
    say_nationality()

say()

# now make it a decorator
def say(func):
    def say_name():
        print("Bianca")

    def say_nationality():
        print("German")

    def wrapper():
        say_name()
        say_nationality()
        func()

    return wrapper

@say
def start_example():
    print("In the example")

start_example()


# an implementation of a facade pattern
#
https://aaravtech.medium.com/design-patterns-in-python-
facade-65b8a393ff68

class Cutter:

    def cut_vegetables(self):
        print("Cutting veggies!")
```

```
class Boiler:

    def boil_vegetables(self):
        print("Boiling veggies!")


class Frier:

    def fry_vegetables(self):
        print("Frying veggies!")


class Cook:

    def prepare_dish(self):
        self.my_cutter = Cutter()
        self.my_cutter.cut_vegetables()


        self.my_boiler = Boiler()
        self.my_boiler.boil_vegetables()


        self.my_frier = Frier()
        self.my_frier.fry_vegetables()



# my_cook = Cook()
# my_cook.prepare_dish()
```