Namespace and Scope

`Q1:` Write `Person` Class as given below and then display it's namespace.

```
Class Name - Person

Attributes:
name - public
state - public
city - private
age - private

Methods:
address - public
It give address of the person as "<name>, <city>, <state>"
```

```
#Write your code here
class Person:
    def __init__(self, name, state):
        self.name = name
        self.state = state
        self.__city = None
        self.__age = None
    def set_city(self, city):
        self.__city = city
    def get_city(self):
        return self.__city
    def set_age(self, age):
        self.__age = age
    def get_age(self):
        return self.__age
    def address(self):
        return f'{self.name}, {self.__city}, {self.state}'

for i in Person.__dict__:
    print(i)
```

```
__module__
__init__
set_city
get_city
set_age
get_age
address
__dict__
__weakref__
__doc__
```

`Q2:` Write a program to show namespace of object/instance of above(Person) class.

```
# Write your code here
p = Person("Raj", "West Bengal")
p.set_city("Kolkata")
p.set_age(40)
print("Address: ", p.address())

for i in p.__dict__:
    print(i)
```

```
Address:  Raj, Kolkata, West Bengal
name
state
_Person__city
_Person__age
```

`Q3:` Write a recursive program to to calculate `gcd` and print no. of function calls taken to find the solution.

```
gcd(5,10) -> result in 5 as gcd and function call 3
```

```
#Write your code here
# O(log(min(a,b)))

counter = 0
def gcd(a,b):
    global counter
    counter += 1
    if b == 0:
        return a
    else:
        return gcd(b, a%b)
print(gcd(10, 5), counter)
```

```
5 2
```

Iterator And Generator

`Q4:` Create MyEnumerate class,

Create your own `MyEnumerate` class such that someone can use it instead of enumerate. It will need to return a `tuple` with each iteration, with the first element in the tuple being the `index` (starting with 0) and the second element being the `current element` from the underlying data structure. Trying to use `MyEnumerate` with a noniterable argument will result in an error.

```
for index, letter in MyEnumerate('abc'):
    print(f'{index} : {letter}')
```

Output:

```
0 : a
1 : b
2 : c
```

```
#Write your code here

class MyEnumerate:
    def __init__(self, data):
        self.data = data
        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.index >= len(self.data):
            raise StopIteration
        value = (self.index, self.data[self.index])
        self.index += 1
        return value
for index, letter in MyEnumerate('abc'):
    print(f'{index} : {letter}')
```

```
0 : a
1 : b
2 : c
```

`Q5:` Iterate in circle

Define a class, `Circle`, that takes two arguments when defined: a sequence and a number. The idea is that the object will then return elements the defined number of times. If the number is greater than the number of elements, then the sequence repeats as necessary. You can define an another class used as a helper (like I call `CircleIterator`).

```
c = Circle('abc', 5)
d = Circle('abc', 7)
print(list(c))
print(list(d))
```

Output

```
[a, b, c, a, b]
[a, b, c, a, b, c, a]
```

```
#Write your code here
class Circle:
    def __init__(self, data, max_iters):
        self.data = data
        self.index = 0
        self.max_iters = max_iters

    def __iter__(self):
        return self
    def __next__(self):
        if self.index >= self.max_iters:
            raise StopIteration
        value = self.data[self.index % len(self.data)]
        self.index += 1
        return value
c = Circle('abc', 5)
d = Circle('abc', 7)
print(list(c))
print(list(d))
```

```
['a', 'b', 'c', 'a', 'b']
['a', 'b', 'c', 'a', 'b', 'c', 'a']
```

`Q6`: Generator time elapsed

Write a generator function whose argument must be iterable. With each iteration, the generator will return a two-element tuple. The first element in the tuple will be an integer indicating how many seconds have passed since the previous iteration. The tuple's second element will be the next item from the passed argument.

Note that the timing should be relative to the previous iteration, not when the generator was first created or invoked. Thus the timing number in the first iteration will be 0

```
for t in elapsed_since('abcd'):
    print(t)
    time.sleep(2)
```

Output:

```
(0.0, 'a')
(2.005651817999933, 'b')
(2.0023095009998997, 'c')
(2.001949742000079, 'd')
```

Note: Your output may differ because of diffrent system has different processing configuration.

```
#Write yor code
import time
def elapsed_since(data):
    last_time = time.perf_counter()
    for item in data:
        current_time = time.perf_counter()
        delta = current_time - last_time
        last_time = current_time
        yield (delta, item)

for t in elapsed_since('abcd'):
    print(t)
    time.sleep(2)
```

```
(1.3999999737279722e-06, 'a')
(2.005139500000041, 'b')
(2.0140556999999717, 'c')
(2.0137271000000965, 'd')
```

Decorators

`Q7:` Write a Python program to make a chain of function decorators (bold, italic, underline etc.) on a given function which prints "hello world"

```
def hello():
    return "hello world"
```

```
bold - wrap string with <b> tag. <b>Str</b>
italic - wrap string with <i> tag. <i>Str</i>
underline- wrap string with <u> tag. <u>Str</u>
```

```
#Write your code here
def make_bold(func):
    def wrapped():
        return "<b>"+ func() + "</b>"
    return wrapped
def make_italic(func):
    def wrapped():
        return "<i>"+ func() + "</i>"
    return wrapped
def make_underline(func):
    def wrapped():
        return "<u>"+ func() + "</u>"
    return wrapped

@make_bold
@make_italic
@make_underline
def hello():
    return "hello world"

print(hello())
```

```
<b><i><u>hello world</u></i></b>
```

`Q8:` Write a decorator called `printer` which causes any decorated function to print their return values. If the return value of a given function is `None`, printer should do nothing.

```
#Write your code here
from functools import wraps
def printer(func):
    @wraps(func)
    def inner(*args, **kwargs):
        return_value = func(*args, **kwargs)
        if return_value is not None:
            print(return_value)
        return return_value
    return inner

@printer
def hello(string):
    return string

hello("hello")
```

```
hello
'hello'
```

```
help(hello)
```

```
Help on function hello in module __main__:

hello(string)
```

Q9: Make a decorator which calls a given function twice. You can assume the functions don't return anything important, but they may take arguments.

```
# Lets say given function
def hello(string):
    print(string)

# on calling after specified decorator is inplaced
hello('hello')
```

Output:

```
hello
hello
```

```
#Write your cod here
def double(func):
    @wraps(func)
    def inner(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return inner
@double
def hello(string):
    print(string)

hello("hello")
```

```
hello
hello
```

Q10: Write a decorator which doubles the return value of any function. And test that decoratos is working correctly or not using `asert`.

```
add(2,3) -> result in 10. Without decorator it should be 5.
```

```
# Write your code here
def double(func):
    @wraps(func)
    def inner(*args, **kwargs):
        return func(*args, **kwargs) * 2
    return inner
@double
def add_withDeco(a,b):
    return a+b

def add(a,b):
    return a+b

a=2
b=3
assert add(a,b)*2 == add_withDeco(a,b), "Values are not matching"
print("Values are matching")
```

```
Values are matching
```