Assignment 11 Solutions

Q1. What is the concept of a metaclass?

Ans: Metaclass in Python is a class of a class that defines how a class behaves. A class is itself a instance of Metaclass, and any Instance of Class in Python is an Instance of type metaclass. E.g. Type of of `int` , `str` , `float` , `list` , `tuple` and many more is of metaclass type.

In [4]:

```
class MetaCls(type):
    """A sample metaclass without any functionality"""
    def __new__(cls, clsname, supercls, attrdict):

        return super(MetaCls, cls).__new__(cls, clsname, supercls, attrdict)

 ## a class of the type metclass
A = MetaCls('A', (object, ), {})
print('Type of class A:', type(A))

class B(object):
    pass
print('Type of class B:', type(B))

## class C inherits from both the class, A and B
class C(A, B):
    pass
print('Type of class C:', type(C))
```

```
Type of class A: <class '__main__.MetaCls'>
Type of class B: <class 'type'>
Type of class C: <class '__main__.MetaCls'>
```

Q2. What is the best way to declare a class's metaclass?

Ans: A way to declare a class' metaclass is by using **`metaclass`** keyword in class definition.

In [5]:

```
class MetaCls(type):
    """A sample metaclass without any functionality"""
    def __new__(cls, clsname, supercls, attrdict):

        return super(MetaCls, cls).__new__(cls, clsname, supercls, attrdict)

 C = MetaCls('C', (object, ), {})
## class A inherits from MetaCls
class A(C):
    pass

print(type(A))
```

```
<class '__main__.MetaCls'>
```

Q3. How do class decorators overlap with metaclasses for handling classes ?

Ans: Decorators are much, much simpler and more limited and therefore should be preferred whenever the desired effect can be achieved with either a metaclass or a class decorator.

we can do anything with a class decorator, we can of course do with a custom metaclass (just apply the functionality of the "decorator function", i.e., the one that takes a class object and modifies it, in the course of the metaclass's `new` or `init` that make the class object.

The same applies to all magic methods, i.e., to all kinds of operations as applied to the class object itself (as opposed to, ones applied to its instances, which use magic methods as defined in the class operations on the class object itself use magic methods as defined in the metaclass).

In [7]:

```
from functools import wraps

def debug(func):
    '''decorator for debugging passed function'''

    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Full name of this method:", func.__qualname__)
        return func(*args, **kwargs)
    return wrapper

def debugmethods(cls):
    '''class decorator make use of debug decorator to debug class methods '''
    # check in class dictionary for any callable(method) if exist, replace it with debugged version
    for key, val in vars(cls).items():
        if callable(val):
            setattr(cls, key, debug(val))
    return cls
# sample class
@debugmethods
class Calc:
    def add(self, x, y):
        return x+y
    def mul(self, x, y):
        return x*y
    def div(self, x, y):
        return x/y

mycal = Calc()
print(mycal.add(2, 3))
print(mycal.mul(5, 2))
print(mycal.div(5, 2))
```

```
Full name of this method: Calc.add
5
Full name of this method: Calc.mul
10
Full name of this method: Calc.div
2.5
```

Q4. How do class decorators overlap with metaclasses for handling instances?

Ans: Anything you can do with a class decorator, you can of course do with a custom metaclass (just apply the functionality of the "decorator function", i.e., the one that takes a class object and modifies it, in the course of the metaclass's **__new__** or **__init__** that make the class object!).

In [8]:

```
import time
import math
# decorator to calculate duration taken by any function.
def calculate_time(func):
# added arguments inside the inner1, if function takes any arguments, can be added like this.
    def inner1(*args, **kwargs):
        # storing time before function execution
        begin = time.time()
        func(*args, **kwargs)
        # storing time after function execution
        end = time.time()
        print("Total time taken in : ", func.__name__, end - begin)
    return inner1
# this can be added to any function present, in this case to calculate a factorial
@calculate_time
def factorial(num):
    # sleep 20 seconds because it takes very less time so that we can see the actual difference
    time.sleep(20)
    print(math.factorial(num))
factorial(5)
```

```
120
Total time taken in :  factorial 20.003939151763916
```