ECE326 – Fall 2019: Week 6 Exercise Questions

1. True or False [1 mark each]

Circle T is true, otherwise circle F for false.

1. In Python, types are reified at runtime. **(T)** **F**
2. The decorator function is called every time the decorated function is called. **T** **(F)**

   Decorator is called once – after the decorated function is defined.
3. C++11 supports static reflection. **T** **(F)**
4. In multiple inheritance, TypeError is raised when there is a shared base metaclass. **T** **(F)**

   Sharing base is OK, as long as a single metaclass can be resolved unambiguously.
5. `type` is to classes as `object` is to instances. **(T)** **F**

2. Multiple Answers **[2 marks each]**

Pick all answer(s) that are correct. You will lose 1 mark per wrong choice, down to 0 marks.

1. Which of the following statements about descriptors or properties are true?

   (a) A descriptor can manage multiple attributes at once, a property can only manage one.
   (b) A descriptor with only __get__ can be overwritten or deleted, a property with only getter cannot.
   (c) A descriptor can manage a method, a property cannot (data attribute only).
   (d) A descriptor keeps data within its own instance, a property uses that of the parent instance.
   (e) A descriptor with only __set__ has the same behaviour as a property with only setter.

   (c) a property can return a method too.

   (d) a descriptor can still access an instance's attributes, but usually you'd keep data within the descriptor.

   (e) a property with only setter is not readable. A descriptor with no __get__ will allow its attributes to be accessed

2. Which of the following statements about class decorators or metaclasses are true?

(a) Both class decorators and metaclasses require only a callable object to work.
(b) Both class decorators and metaclasses must return a class (new or existing).
(c) Class decorators do not support inheritance, metaclasses do.
(d) Class decorators modifies a class only after it is created, metaclasses modifies a class only before and during class creation.
(e) Metaclass is the only way to overload operators for classes.

(b) class decorator only needs to return a callable that will create instances of the class.

(c) Technically speaking, if the class decorator is a functor, then the functor supports inheritance. So both yes or no is accepted for this choice.

3. Decorator **[10 marks]**

Create a decorator, memoize, that will cache results of the decorated function. You may assume:

1. The decorated function will be a pure function.
2. Only positional arguments are used by the decorated function, no keyword arguments.
3. All arguments are hashable types

```
# Note that this will work for multiple functions because of
# how closure works
def memoize(func):
    cache = {}
    def memoizer(*args):
        if args in cache:
            return cache[args]
        else:
            result = func(*args)
            cache[args] = result
            return result
    return memoizer
```

4. Metaclass **[10 marks]**

Write a metaclass, MethodCounter, that will add the functionality to a class such that it counts how many times a method is called by any instance of the class. For example, if *a* called foo twice and *b* called foo once, then the count for foo should be three (both *a* and *b* are instances of the class that inherits your metaclass). Remember that it should keep a separate count for each method. Hint: use the built-in function `callable` to check if an attribute is callable.

```
def getattribute(self, name):
    # NOTE: cannot use super() here because
    # this function is defined outside of the
    # parent class
    val = object.__getattribute__(self, name)
    counter = object.__getattribute__(self, '_counter')
    if name in counter:
        counter[name] += 1
    return val

class MethodCounter(type):
    def __new__(mcs, name, base, attrs):
        assert('_counter' not in attrs)
        counter = {}
        for a_name in attrs:
            if callable(attrs[a_name]):
                counter[a_name] = 0
        attrs['_counter'] = counter
        attrs['__getattribute__'] = getattribute
        return super().__new__(mcs, name, base, attrs)

    def get_count(cls, name):
        return cls._counter.get(name, 0)
```

5. Descriptor **[10 marks]**

Write a descriptor that will check the type and the range of the managed attribute before allowing it to be stored. The descriptor should also disallow deletion of the managed attribute. Here is how your descriptor is used:

```
class Foo:
    likelihood = Attr(type=float, minimum=0.0, maximum=1.0)
```

Note that omitting any one of the checks results in not making that check. For example, if type is omitted, then type checking is disabled. Similarly, if maximum is omitted, then upper bound is unlimited (not checked). Raise AttributeError if the check fails.

```
class Attr:
    def __init__(self, type=None, minimum=None, maximum=None):
        self.type = type
        self.minimum = minimum
        self.maximum = maximum

    def __get__(self, instance, owner):
        return getattr(self, 'value')

    def __set__(self, instance, value):
        if self.type is not None and \
           not isinstance(value, self.type):
            raise AttributeError("incorrect type")
        if self.minimum is not None and value < self.minimum:
            raise AttributeError("value less than minimum")
        if self.maximum is not None and value > self.maximum:
            raise AttributeError("value greater than maximum")
        self.value = value

    def __delete__(self, instance):
        raise AttributeError("cannot delete this descriptor")
```