Question 1. True or False

Circle **T** if the statement is true, otherwise circle **F** if the statement is false.

1. `type` is to classes as `object` is to instances. **(T)** **F**
2. In multiple inheritance, TypeError is raised when there is a shared base metaclass. **T** **(F)**
3. `vars(self)` returns `self.__dict__`. **(T)** **F**
4. The `__delete__` special method is also known as the destructor. **T** **(F)**
5. A metaclass does not need to be a class; it can also be a function. **(T)** **F**

Question 2. Multiple Choices

Pick all answer(s) that are correct.

a) Which of the following statements about descriptors or properties are true?

i. A descriptor can manage multiple attributes at once, a property can only manage one.

(ii.) A descriptor with only __get__ can be overwritten or deleted, a property with only getter cannot.

iii. A descriptor can manage a method, a property cannot (data attribute only).

iv. A descriptor keeps data within its own instance, a property uses that of the parent instance.

v. A descriptor with only __set__ has the same behaviour as a property with only setter.

Note: there was an error in the 2019 exercise solution of the same question for choice iv.

b) Which of the following about Python metaclass are true?

i. The __call__ method of a metaclass initiates the process of creating a new class.

ii. The __new__ method of a metaclass instantiates new objects for its classes.

iii. To avoid infinite recursion, a metaclass cannot have its own metaclass.

iv. Like regular classes, multiple inheritance is supported for metaclasses.

v. During name resolution, a class's metaclass is looked up before its super classes are searched.

Note: none of the above is true.

Question 3. Short Answer

Describe the differences between the following three built-in methods: `__set__`, `__setitem__`, and `__setattr__`.

`__set__` is used by a descriptor to manage the assignment of one attribute on an instance of a different class.

`__setitem__` is used to overload assignment to an index. e.g. foo[2] = "hello".

`__setattr__` is used to manage **all** attribute assignment to instances of a class.

Question 4. Programming Questions

a) 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)
```

b) You are developing a package for adding type safety checks to the end users' custom-defined classes. The interface you provide requires that their classes inherit from `Base`, and that the fields they want to type check be specified as a class member attribute of type `Field`, such as:

```
class User(Base):
    name = Field(type=str)
    age = Field(type=int)
    height = Field(type=float)

fred = User(name="Fred", age=23, height=6.2)
anne = User(name="Anne", age="19") # error: age is not an integer
```

Complete the class `Base`, with metaclass named `Meta`, such that the `__init__` method of `Base` takes in a variable number of keyword arguments and stores each key value pair as a field in `Base`, with the field name of each pair being its key. You may not make use of the instance's `__dict__` attribute for this question.

```
class Base(metaclass=Meta):
    def __init__(self, **kwargs):
        for col, val in kwargs.items():
            setattr(self, col, val)
```

Write a descriptor class named `Field` such that upon intercepting attribute assignment, it checks that the type of the value matches what is specified in the constructor. If a mismatch occurs, raise an `AttributeError` with the message "wrong type". Next, write a metaclass, `Meta`, to work with the `Field` class such that user-defined classes support multiple instances correctly. Note that field order does not matter. Hint: how can you use the metaclass so that each `Field` instance knows its attribute name?

```
class Field:
    def __init__(self, type):
        self.type = type

    # complete class Field here, and add metaclass Meta


    def setname(self, name):
        self.name = "_" + name

    def __set__(self, inst, value):
        if not isinstance(value, self.type):
            raise AttributeError("wrong type")
        setattr(inst, self.name, value)

    def __get__(self, inst, cls):
        return getattr(inst, self.name)



class Meta(type):
    def __init__(cls, name, bases, attrs):
        for col, val in attrs.items():
            if isinstance(val, Field):
                val.setname(col)
```

You are helping a classmate with their assignment and the issue is that their user-defined class only supports exactly one instance. The symptom looks like this:

```
joe = User(name="Joe", age=12, height=5.4)
fred = User(name="Fred", age=23, height=6.2)
print(joe)

Fred: Age 23, Height 6.200000
```

Write down one line of code in `Field.__set__` that would produce this output (ignore the type checking part of `__set__`).

```
self.value = value
```

You are helping another classmate with their assignment and the issue is that all the fields have the same value, but multiple instances seem to be supported. The symptom looks like this:

```
Joe: Age Joe, Height Joe
Fred: Age Fred, Height Fred
```

Write down one line of code in `Field.__set__` that would produce this output.

```
inst.value = value
```

You are helping your friends with the same assignment again and the issue looks like this:

```
In Base.__init__:
   RecursionError: maximum recursion depth exceeded
```

Describe (do not write code) what the problem may be and why it happened.

In `setname`, the same name is used for the value and the descriptor, causing Field.__set__ to be called infinitely. (must mangle the name to fix it, e.g. add underscore to name).