ECE326 – TUTORIAL 7

PREPARED BY MARTIYA ZARE JAHROMI

UNIVERSITY OF TORONTO

BOUNDLESS

AGENDA

- Exercise 6
- Exercise 7

EXERCISE 6 – 1. TRUE OR FALSE

1. In Python, types are reified at runtime. [✓ T] F

EXERCISE 6 – 1. TRUE OR FALSE

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.

EXERCISE 6 – 1. TRUE OR FALSE

3. C++11 supports static reflection. T [✓ F]

EXERCISE 6 – 1. TRUE OR FALSE

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.

EXERCISE 6 – 1. TRUE OR FALSE

5. type is to classes as object is to instances. [✓ T] F

EXERCISE 6 – 2. MULTIPLE ANSWERS

1. Which of the following 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.

EXERCISE 6 – 2. MULTIPLE ANSWERS

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.

EXERCISE 6 – 3 DECORATOR

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

EXERCISE 6 – 3 DECORATOR

```
# 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
```

UNIVERSITY OF TORONTO

BOUNDLESS

EXERCISE 6 – 4 METACLASS

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.

EXERCISE 6 – 4 METACLASS

```
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)
```

EXERCISE 6 – 4 DESCRIPTOR

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.

EXERCISE 6 – 4 DESCRIPTOR

```
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")
```

EXERCISE 7 – 1. TRUE OF FALSE

1. A covariant tuple parameter is always type-safe. [✓ T] F

E.g. Tuple<Apple> can always replace Tuple<Fruit> (tuples are immutable).

EXERCISE 7 – 1. TRUE OF FALSE

2. Widening conversion guarantees you can get back the original data if needed. T [✓ F]

Loss of precision can occur.

EXERCISE 7 – 1. TRUE OF FALSE

3. In C++, type inference for variable declaration (using the auto keyword) cannot fail. [✓ T] F

EXERCISE 7 – 1. TRUE OF FALSE

3. Macro systems do not have knowledge of the underlying programming language. [✓ T] F

EXERCISE 7 – 1. TRUE OF FALSE

3. explicit keyword is used to prevent implicit conversion of class objects when the class overloads the cast operator. T [✓ F]

It’s used to prevent implicit construction of an object

EXERCISE 7 – 2. SHORT ANSWERS

1. Give an example to show that contravariant function return type is not type safe.

EXERCISE 7 – 2. SHORT ANSWERS

-
  ```
  class Animal {}
  class Cat : public Animal {}

  struct CatShelter {
  	virtual Cat * adopt();
  };

  struct AnimalShelter : public CatShelter {
  	virtual Animal * adopt();
  };

  CatShelter * cs = new AnimalShelter();
  Cat * cat = cs->adopt();   // type-unsafe, Animal may not be a Cat
  ```

EXERCISE 7 – 2. SHORT ANSWERS

2. What is the value of the following expression? What is the name of the behaviour? Assume the integer is 32-bit

```
int a = 1 << 31;
printf("%d", -a);
```

```
-2147483648 (a == -a)
Overflow
```

a = 1000 0000 0000 0000 0000 0000 0000 0000
2's complement = 0111 1111 1111 1111 1111 1111 1111 1111 + 1 = 1000 0000 …

.

EXERCISE 7 – 2. SHORT ANSWERS

3. Write a safe C macro, CIRCLE_AREA, that takes one parameter and will calculate the radius of a circle. You also need to define the constant PI.

```
#define PI 3.14159
#define CIRCLE_AREA(r) PI*(r)*(r)
* better yet (GNU C extension) */
#define CIRCLE_AREA(r) ({ typeof(r) _r = (r); PI*_r*_r; })
```

EXERCISE 7 – 2. SHORT ANSWERS

4. Write the precondition and postcondition for the square root function, sqrt() that takes in a double and returns a double.

Let n be the parameter to sqrt, i.e. sqrt(double n), and r be the return value

Precondition: n >= 0

Postcondition: n >= 0 (if only the principle square root is returned)

EXERCISE 7 – 2. SHORT ANSWERS

4. …Suppose Complex is a subclass of double, what’s the variance relationship between double sqrt(double) and Complex sqrt(Complex)?

Covariant parameter, covariant return type.

Note that both parameter and return type are problematic because covariant parameter is not type safe, and the postcondition of sqrt() is weakened, which violates Liskov’s substitution principle.

EXERCISE 7 – 3. TAGGED UNION

Complete the implementation of the tagged union example from class by adding a copy constructor, overload the equality operator, and type-safe getter/setter function.

EXERCISE 7 – 3. TAGGED UNION

```
struct Tagged {
        enum { INT, STR } tag;
        union {                         // anonymous union (members
                int * i;                // can enter parent scope)
                string * s;
        };
        Tagged(int i) : tag(INT), i(new int(i)) {}
        Tagged(const char * s) : tag(STR), s(new string(s)) {}
        ~Tagged() {
                if (tag == INT) delete i; else delete s;
        }
```

EXERCISE 7 – 3. TAGGED UNION

```
Tagged(const Tagged & rhs) {
        tag = rhs.tag;
        if (rhs.tag == INT)
            i = new int(*rhs.i);
        else
            s = new string(*rhs.s);
    }
```

EXERCISE 7 – 3. TAGGED UNION

```
bool operator==( const Tagged & rhs) {
                if (rhs.tag == INT)
                        return *i == *rhs.i;
                else
                        return *s == *rhs.s;
        }
```

EXERCISE 7 – 3. TAGGED UNION

```
 const string * get_str() const {
                if (tag == INT)
                        return nullptr;
                else
                        return s;
        }

void set_str(const string & s) {
                destroy();
                this->s = new string(s);
                this->tag = STR;
        }
```

UNIVERSITY OF TORONTO

BOUNDLESS

Questions?