15-112, Lab 8, Object-Oriented Programming Practice

See how many of these problems you and a partner can solve during today's lab.

Part 1: Concept Review

Answer these on paper. Be ready to explain *why*, not just state the answer.

Use a pencil when solving these problems.

1. Classes and instances

What is the difference between a class and an instance? Write a small amount of code that creates a class, and code that creates an instance of it.

2. Multiple choice

For each question, circle the most correct answer.

a. What is a class in Python?

- A template
- A specific item
- A specialized function
- An inheritance

b. What is an instance in Python?

- A template
- A specific item
- A specialized function
- An inheritance

c. What does it mean for a class to override a method?

- It gets the method from its superclass
- It gets the method from its subclass
- It calls a method from the superclass using `super()`
- It changes how the method works from the original version

d. Which of the following can be a superclass of `Student`?

- `University`
- `ComputerScienceStudent`
- `Course`
- `Person`

e. What is the difference between `__repr__(self)` and `__str__(self)`?

- `__str__` is used when the object is converted to a string; `__repr__` is used when the object is written to a file
- `__str__` is used when the object is directly converted to a string; `__repr__` is used when the object is indirectly converted to a string
- `__str__` is used when the object is converted to a string; `__repr__` is used when the object is hashed
- There is no difference

3. Inheritance hierarchy

Consider the following four classes, implemented by the designers of a game: `Monster`, `Player`, `Bat`, `FireBat`. Assuming inheritance is used properly, fill in the blanks. If the answer to a blank is "nothing", write "nothing".

- `Monster` is a subclass of _____
- `Monster` is the superclass of _____
- `Player` is a subclass of _____
- `Player` is the superclass of _____
- `Bat` is a subclass of _____
- `Bat` is the superclass of _____
- `FireBat` is a subclass of _____
- `FireBat` is the superclass of _____

4. Circles and points

Consider the following class:

```
class circle:
    def __init__(self, r):
        self.radius = r

    def getRadius(self):
        return self.radius
```

a. Given the code `c = circle(20)`, which of the following statements are correct? (Mark *all* correct answers.)

1. `circle` is an object
2. `c` refers to an object
3. `circle(20)` creates a `circle` object
4. `c` is the class `circle`
5. None of the above

b. Assume that the list `allCircles` contains objects of type `circle`. Write code that prints the radius of each circle. You must use the `getRadius` method.

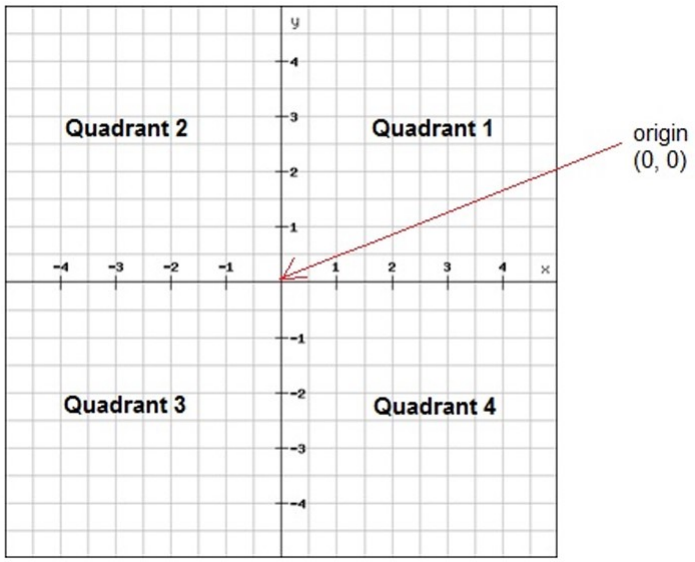
```
allCircles = [ ... ]   # assume a bunch of circle objects have been added to the list
# Your code here
```

5. Points

Consider the following class:

```
class point:
    def __init__(self, x, y, col):
        self.x = x
        self.y = y
        self.c = col    # the point's color
```

a. Add a method `getQuadrant(self)` that returns which quadrant of the Cartesian plane this point is in, as shown in the diagram below: `1` for the first quadrant, `2` for the second, `3` for the third, and `4` for the fourth. If the point lies on an axis, return `0`.



b. Add a method `sameQuadrant(self, other)` that takes another `point` and returns `True` if both points are in the same quadrant (1–4), and `False` otherwise. You may assume `getQuadrant` from part a works, and you must use it.

6. TeaPot

The following `TeaPot` class is given:

```
class TeaPot(object):
    def __init__(self, capacity, ounces):
        self.size = capacity              # max amount of tea the pot can hold
        self.ouncesRemaining = ounces     # total tea currently in the pot

    def fillUp(self):
        self.ouncesRemaining = self.size

    def getOuncesRemaining(self):
        return self.ouncesRemaining
```

Write a method `pourOut` for the `TeaPot` class that takes one integer, the number of ounces to pour, and attempts to pour out that much tea. If there isn't enough tea left, it pours out tea until the pot is completely empty. The method returns the number of ounces actually poured out. Make sure the state of the teapot is consistent afterward.

```
    def pourOut(self, amount):
        # Put your code here
```

Part 2: Code Tracing

Don't run the code — predict on paper exactly what gets printed, line by line. *Then* run it to check yourself, and try changing it slightly to see what happens.

Use a pencil when solving this problem.

Trace #1

```
class A(object):
    def __init__(self):
        self.b = "Kitten"

    def f(self):
        print("Cat")

    def g(self):
        print("Dog")

    def h(self):
        print(f"{self.b}: Horse")
        self.g()

class B(A):
    def g(self):
        print("Camel")
        super().g()

class C(B):
    def __init__(self, a):
        self.b = a
        self.f()

    def h(self, s):
        super().h()
        print(f"{s}: Lion")

def ct():
    c = C("Goat")
    c.h("Cow")

ct()
```

Think about: does `"Kitten"` *ever get printed? Which* `g` *runs when* `A.h` *calls* `self.g()`*?*

Part 3: Free Response

Do these exercises on a computer. You can find a digital version of this handout on the course website.

For each problem, write the requested classes so that the given test function passes. Use the OOP constructs we learned this week as appropriate. Don't hardcode against the values used in the test cases.

---

1. Books

Write the `Book` class so that it passes `testBookClass`.

```
def testBookClass():
    print("Testing Book class...", end="")
    # A Book has a title, and author, and a number of pages.
    # It also has a current page, which always starts at 1. There is no page 0!
    book1 = Book("Harry Potter and the Sorcerer's Stone", "J. K. Rowling", 309)
    assert(str(book1) == "Book")
    book2 = Book("Carnegie Mellon Motto", "Andrew Carnegie", 1)
    assert(str(book2) == "Book")

    # You can turn pages in a book. Turning a positive number of pages moves
    # forward; turning a negative number moves backwards. You can't move past
    # the first page going backwards or the last page going forwards
    book1.turnPage(4) # turning pages does not return
    assert(book1.getCurrentPage() == 5)
    book1.turnPage(-1)
    assert(book1.getCurrentPage() == 4)
    book1.turnPage(400)
    assert(book1.getCurrentPage() == 309)
    assert(str(book1) == "Book")
    book2.turnPage(-1)
    assert(book2.getCurrentPage() == 1)
    book2.turnPage(1)
    assert(book2.getCurrentPage() == 1)

    # You can also put a bookmark on the current page. This lets you turn
    # back to it easily. The book starts out without a bookmark.
    book3 = Book("The Name of the Wind", "Patrick Rothfuss", 662)
    assert(str(book3) == "Book")
    assert(book3.getBookmarkedPage() == None)
    book3.turnPage(9)
    book3.placeBookmark() # does not return
    assert(book3.getBookmarkedPage() == 10)
    book3.turnPage(7)
    assert(book3.getBookmarkedPage() == 10)
    assert(book3.getCurrentPage() == 17)
    assert(str(book3) == "Book")
    book3.turnToBookmark()
    assert(book3.getCurrentPage() == 10)
    book3.removeBookmark()
    assert(book3.getBookmarkedPage() == None)
    book3.turnPage(25)
    assert(book3.getCurrentPage() == 35)
    book3.turnToBookmark() # if there's no bookmark, don't turn to a page
    assert(book3.getCurrentPage() == 35)
    assert(str(book3) == "Book")

    # Finally, you should be able to compare two books directly
```

```
    book5 = Book("A Game of Thrones", "George R.R. Martin", 807)
    book6 = Book("A Game of Thrones", "George R.R. Martin", 807)
    book7 = Book("A Natural History of Dragons", "Marie Brennan", 334)
    book8 = Book("A Game of Spoofs", "George R.R. Martin", 807)
    assert(book5 == book6)
    assert(book5 != book7)
    assert(book5 != book8)
    book5.turnPage(1)
    assert(book5 != book6)
    book5.turnPage(-1)
    assert(book5 == book6)
    book6.placeBookmark()
    assert(book5 != book6)
    print("Done!")
```

---

2. Stapler

Write the classes `Stapler` and `ElectricStapler` so that `testStaplerClass` runs without errors. You can assume the test cases cover the needed functionality. You must use proper object-oriented design, including good inheritance.

```
def testStaplerClass():
    print("Testing Stapler classes...", end="")
    # A stapler starts out full of the specified number of staples
    s = Stapler(50)
    assert(str(s) == "Stapler(50/50)")

    # You can staple something if the stapler has staples
    s.staple()
    assert(str(s) == "Stapler(49/50)")

    # You can refill the stapler to get it back to full. Full is the number
    # of staples it started with, which could be different from stapler to stapler.
    s.refill()
    assert(str(s) == "Stapler(50/50)")
    for i in range(50):
        s.staple()
    assert(str(s) == "Stapler(0/50)")

    # If you try to staple while empty, nothing changes
    s.staple()
    assert(str(s) == "Stapler(0/50)")

    # An electric stapler always starts with 500 staples, and is unplugged
    e = ElectricStapler()
    assert(str(e) == "DeadStapler(500/500)")

    # You can plug in an electric stapler
    e.plugIn()
    assert(str(e) == "ElectricStapler(500/500)")

    # An electric stapler can also staple and refill
    e.staple()
    assert(str(e) == "ElectricStapler(499/500)")
    e.refill()
    assert(str(e) == "ElectricStapler(500/500)")
```

```
# But an unplugged electric stapler can't staple: you get a "No Power" exception
e.unPlug()
ok = False
try:
    e.staple()
except:
    ok = True
assert(ok)

# Like a normal stapler, an electric stapler also doesn't change
# if you staple while empty
e.plugIn()
for i in range(500):
    e.staple()
assert(str(e) == "ElectricStapler(0/500)")
e.staple()
assert(str(e) == "ElectricStapler(0/500)")

# Checking inheritance
assert(isinstance(s, Stapler) == True)
assert(isinstance(s, ElectricStapler) == False)
assert(isinstance(e, Stapler) == True)
assert(isinstance(e, ElectricStapler) == True)
print("Done!")
```

Hint: `type(self).__name__` *gives you the name of an object's class as a string.*