Working Up A Thirst

Much of the power of object-oriented programming derives from the notion of *inheritance*. A name that immediately follows a ‘dot’ is called a qualified name (it is qualified by the value of the expression to the left of the dot). When you use a qualified name — e.g., the name `attr` in the expression `obj.attr` — Python searches a tree of namespaces to find the value associate with the name. This name-space tree is defined using the **class hierarchy**. This exercise will deepen your understanding of inheritance and the *scope rules* that Python uses with qualified names.

**Part (a): [Class hierarchy]** The header of a class definition must contain a type name in parenthesis after the class name. This type is called the *parent* of the new class, and the new class is called a *child* of the parent type.

Answer the following questions about the program `drinks.py`:

**Q1:** For each type, what is its parent?

Parent of `Drink`: __________________________  Parent of `CocaCola`: __________________________

Parent of `Milk`: __________________________  Parent of `ChocolateMilk`: __________________________

**Q2:** For each type, what are its children (if any)?

Children of `Drink`: __________________________  Children of `CocaCola`: __________________________

Children of `Milk`: __________________________  Children of `ChocolateMilk`: __________________________

The parent/child relation on the classes in a Python program defines the class hierarchy. Computer scientists show a class hierarchy as a tree and use similar terminology as for a family tree. As in a family tree, a child is drawn below its parent and a line connects them. *Ancestors* and *descendants* are defined as in a family tree. Draw a picture of the class hierarchy defined by the drinks program below.
**Part (b): [Inheritance, Scope rules for qualified names, Class hierarchy]** The idea behind inheritance is that a descendant inherits the attributes of its ancestors, but also defines its own unique attributes and can redefine inherited ones. The shell uses special scope rules for qualified names to implement inheritance: To find the value of an expression of the form `obj.attr`, the shell searches namespaces for one that contains the name `attr`. In order, it searches:

1. First, the instance’s *local namespace*—i.e., the name space referenced by `obj`
2. Then, the *class namespace*—i.e., the namespace defined by `obj`’s type (i.e., its class)
3. Then, the namespaces of this class’ ancestors, in turn, going up the class hierarchy

If no namespace in this search contains `attr`, the shell raises an `AttributeError`.

An *adorned class diagram* is useful in showing the data attributes and methods associated with instances of a class, as well as where they are defined. Beside each class name, the adorned diagram lists any data attributes (re)defined in the class constructor and lists all methods (re)defined in the class definition. Finish adorning the class diagram below and then discuss with a partner how it helps you answer the questions below the diagram.

Assume the assignment: 

```
A, C = CocaCola(), ChocolateMilk()
```

Which of the following are **not** legal?

- A.price
- C.price
- A.get_price()
- C.get_price()
- C.get_fat()
- A > C
- A.kind
- C.kind
- A.percent_fat
- C.percent_fat
- A + C
- A == C
- str(A)
- print(C)
For each block of code that follows, assume the class definitions in drinks.py.

1. Hand trace execution of the code, writing the numbers of the statements in drinks.py that will be executed in the order that they will be executed in the space provided.
2. Below the list of statements executed, show what is printed.
3. Draw a picture of the object that d references.

To illustrate, we will walk through the first one with you.

Code Segment:

```python
import drinks
d = drinks.Drink()
print(d)
print()
d.print_price_sticker()
```

Statements executed:  Picture:

Prints:

Code Segment:

```python
import drinks
d = drinks.CocaCola()
print(d)
print()
print()
d.print_price_sticker()
```

Statements executed:

Prints:

Step through Visualization 1 (link on the website) to check your understanding. After executing the class definitions, you should press ‘hide attributes’ to elide details in each class (type) object. Doing so will let you see the frames and class instances more easily as they are being created and manipulated.
import drinks
d = drinks.Milk(1)
print(d)
print()
d.print_price_sticker()

Statements executed:

Prints:

Step through Visualization 2 (link on the website) to check your understanding. After executing the class definitions, you should press ‘hide attributes’ to elide details in each class (type) object. Doing so will let you see the frames and class instances more easily as they are being created and manipulated.

For this one, don’t worry about logging the statements that are executed. But predict what the drink object will ‘look like’ and what will be printed.

import drinks
d = drinks.ChocolateMilk(2)
print(d)
print()
d.print_price_sticker()

Prints:

Step through Visualization 3 (link on the website) to check your understanding. After executing the class definitions, you should press ‘hide attributes’ to elide details in each class (type) object. Doing so will let you see the frames and class instances more easily as they are being created and manipulated.
Part (c): [Inheritance and typing] A descendant class is called a *subclass* or a *subtype* of all of its ancestor classes. Similarly, an ancestor class is also called a *superclass* or a *supertype* of all of its descendant classes. An instance is considered to be an instance of its own type and also an instance of all of its supertypes (superclasses). Given the class definitions in *drinks.py*, predict what is printed by the code below.

```python
from drinks import Drink, CocaCola, Milk, ChocolateMilk

generic_drink = Drink()
coke = CocaCola()
milk = Milk(2)
choc_milk = ChocolateMilk()

print( type( generic_drink ) == Drink )  # Prints:______________
print( type( coke ) == Drink )  # Prints:______________
print( type( milk ) == Milk )  # Prints:______________
print( type( choc_milk ) == Milk )  # Prints:______________
print( isinstance(generic_drink, Drink) )  # Prints:________
print( isinstance( coke, Drink ) )  # Prints:________
print( isinstance( generic_drink, CocaCola ) )  # Prints:________
print( isinstance( choc_milk, Milk ) )  # Prints:________
print( isinstance( milk, ChocolateMilk ) )  # Prints:________
print( isinstance( choc_milk, CocaCola ) )  # Prints:________
print( isinstance( choc_milk, WhiteMilk ) )  # Prints:________
print( isinstance( choc_milk, object ) )  # Prints:________
print( isinstance( Milk, Drink ) )  # Prints:________
print( isinstance( Milk, type ) )  # Prints:________
```

To check your answers, download *drinks.py* and *subtypeDemo.py* (to the same directory) then run *subtypeDemo.py*.

Part (d): Add methods to the appropriate classes in *drinks.py* that can be called to do the following.

1. `d.set_price(x)`: Set the price of drink `d` to `x` (a numeric value).
2. `m.set_fat(x)`: Set the percentage of fat of milk drink `m` to `x` (a numeric value).
**Part (e):** Write a program that imports `drinks` and behaves as follows.

- It repeatedly prompts the user to select from a menu to add a drink to an order or to quit.
- For a Coca Cola, it prompts for a size, ‘L’ (large) or ‘S’ (small); uses ‘S’ if the size is ill formed, and 1.75 and 1.25 as the price for, respectively, large and small milks.
- For a Milk, it then prompts for the percent fat, 0, 1 or 2; uses 2 as the default if an illegal value is entered, and uses 0.75 as the price.
- It stores the user’s order as a list of `Drink` objects with the indicated prices and percent fat.
- When the order is finished, it prints out the order and the total cost.

For example:

To add a drink to your order or to quit, enter:
'0' to quit
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Enter your selection here: 1
Enter 'L' (large, $1.75) or 'S' (small, the default, $1.25): l

Enter your selection here: 2
Enter the % fat (0, 1, or 2 (the default)):

Enter your selection here: 3
Enter the % fat (0, 1, or 2 (the default)): 0

Enter your selection here: ?
Illegal selection: ? Try again.

Enter your selection here: 1
Enter 'L' (large, $1.75) or 'S' (small, the default, $1.25): S

Enter your selection here: 3
Enter the % fat (0, 1, or 2 (the default)):

Enter your selection here: 0

Your order:
--------------
Coca-Cola Classic @ $ 1.75
White Milk @ $ 0.75 (2 % fat)
Chocolate Milk @ $ 0.75 (0 % fat)
Coca-Cola Classic @ $ 1.25
Chocolate Milk @ $ 0.75 (2 % fat)

Total is: $5.25