A Drink for Inheritance!

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 \texttt{attr} in the expression \texttt{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 heirarchy. 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 \texttt{drinks.py} in the handout:

Q1: For each type, what is its parent?

Parent of \texttt{Drink}: ___________________  Parent of \texttt{CocaCola}: ___________________

Parent of \texttt{Milk}: ___________________  Parent of \texttt{ChocolateMilk}: _______________

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

Children of \texttt{Drink}: ___________________  Children of \texttt{CocaCola}: ___________________

Children of \texttt{Milk}: ___________________  Children of \texttt{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 diagram 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 scope rules for qualified names are used 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`.

To see how these rules work, we add the following information to the class hierarchy diagram: For each class, the names of all data attributes (re)defined in the class constructor and the line number where they are (re)defined, and the names of all methods (re)defined in the class definition and the line number at which they start executing. This produces the following diagram for the class hierarchy in `drinks.py`.

![Class Hierarchy Diagram](image)

Discuss with a partner how the information in the boxes in this diagram was produced from the code in `drinks.py`. Then discuss how this diagram helps you answer the following question.

Assume the assignments: `A = CocaCola()` and `C = ChocolateMilk()`

Which of the following are legal and which are not?

- `A.price`
- `A.kind`
- `A.percent_fat`
- `C.price`
- `C.kind`
- `C.percent_fat`
- `A.get_price()``A.get_fat()`
- `C.get_price()``C.get_fat()`
- `A + C`
- `A == C`

Put up your green tent when you are done, or your pink tent if you have any questions.
For each block of code that follows, assume the class definitions in `drinks.py`. Hand trace execution of the code and, in the spaces provided, show:

1. The statements in `drinks.py` that are executed in the order that they are executed (use the numbers given as comments to represent statements).
2. What is printed.
3. A picture of the object that `d` references.

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

-----------------------------------------------------------------------------------------------------------------------------

Code Segment: 

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

Statements executed:

Prints:

-----------------------------------------------------------------------------------------------------------------------------

When you are done, put up your green tent; one of us will come check your answer.
from drinks import *
d = Milk( 1 )
print( d )
print()
d.print_price_sticker()

Statements executed:

Prints:

When you are done, put up your green tent; one of us will come check your answer.

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.

Code Segment:

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

Prints:

When you are done, put up your green tent; one of us will come check your answer.
Part (c): [Inheritance and typing] A descendant class is also 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 object is considered to be an instance of its own type and also an instance of all of its supertypes (superclasses, ancestors). Given the class definitions in drinks.py, predict what is printed by the code below.

```python
from drinks import *

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. \texttt{d.set_price(x)}: Set the price of drink \texttt{d} to \texttt{x} (a numeric value).
2. \texttt{m.set_fat(x)}: Set the percentage of fat of milk drink \texttt{m} to \texttt{x} (a numeric value).
Part (e): Download the program `drinksProgram.py` to the folder that contains `drinks.py`. Modify it so that it imports `drinks.py` and behaves as follows.

- It repeatedly prompts the user to select a number from a drinks menu or a '0' to end their order.
  - For a Coca Cola (‘1’), it prompts for a size, ‘L’ (large) or ‘S’ (small), and creates a Coca Cola drink with price $1.75, for large, or $1.25, for small.
  - For a Milk (‘2’ or ‘3’), it prompts for the percent fat, ‘0’, ‘1’ or ‘2’, and creates the appropriate Milk drink (ChocolateMilk or WhiteMilk) with the given percent fat and the price $0.75.
  - Prints a message indicating the drink ordered and adds the drink to the user’s order.

- When the order is finished, it prints out the drinks in the user’s order and the total cost.
- For any illegal input, it just prints an error message and re-prompts for a drink or to end the order.

For example:

Select a drink from the menu below, or enter '0' to end your order:
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Make your selection: 1

Enter 'L' (large, $1.75) or 'S' (small, $1.25): L
You ordered: Coca-Cola Classic @ $ 1.75

Select a drink from the menu below, or enter '0' to end your order:
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Make your selection: 1

Enter 'L' (large, $1.75) or 'S' (small, $1.25): Bad value for size:

Select a drink from the menu below, or enter '0' to end your order:
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Make your selection: 2

Enter '0', '1', or '2' for the % fat: 1
You ordered: White Milk @ $ 0.75 (1 % fat)
Select a drink from the menu below, or enter '0' to end your order:
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Make your selection: 3

Enter '0', '1', or '2' for the % fat: 0
You ordered: Chocolate Milk @ $0.75 (0 % fat)

Select a drink from the menu below, or enter '0' to end your order:
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Make your selection: 4
Bad menu selection: 4

Select a drink from the menu below, or enter '0' to end your order:
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Make your selection: 3
Enter '0', '1', or '2' for the % fat: 1
You ordered: Chocolate Milk @ $0.75 (1 % fat)

Select a drink from the menu below, or enter '0' to end your order:
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Make your selection: 2
Enter '0', '1', or '2' for the % fat: 5
Bad value input for fat: 5

Select a drink from the menu below, or enter '0' to end your order:
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Make your selection: 2
Enter '0', '1', or '2' for the % fat: 3
Bad value input for fat: 3

Select a drink from the menu below, or enter '0' to end your order:
'1' for a Coca Cola
'2' for a white milk
'3' for a chocolate milk

Make your selection: 0
Your order:
--------
Coca-Cola Classic @ $ 1.75
White Milk @ $ 0.75 (1 % fat)
Chocolate Milk @ $ 0.75 (0 % fat)
Chocolate Milk @ $ 0.75 (1 % fat)

Total is: $4.00