Review of OOP and Python

This example includes a review of object-oriented programming basics, and how to apply them in Python. Some of the Python syntax may be new.

Starter Code on Github Classroom

Please clone this assignment from Github Classroom – the URL given in class.

1. Visit the URL and accept the assignment.

   you may be asked to associate your real name with your Github ID. Please do.

2. Wait a few seconds (really!) and then click refresh. The page will refresh to show a link to your repository (repo) on Github.
3. Click the link to visit your repository.
4. Clone the repository to your local computer.

Ref: If you want to use git inside of PyCharm, see Connecting PyCharm to Github Classroom instructions by Aj. Chaiporn. (This was done in Programming 1.)

Money

Money is a unit of exchange having a value and a currency.

Money is *fungible*, meaning that any two objects of same value and currency are equivalent.

In Python, the starter code looks like:

```
class Money:
    """Money represents a unit of value with a currency."""

    def __init__(self, value, currency):
        self.value = value
        self.currency = currency
```

We will add behavior to satisfy some requirements.

When is Money Equal?

Money should be fungible. That means all money with the same value and same currency is equal. Is it?

Try this:

```
>>> m1 = Money(10, "Baht")
>>> m2 = Money(10, "Baht")
>>> m1 == m2
False
```

How to fix this?

Magic Methods

Python classes have several “magic” methods whose names have the form __methodname__.

To see them, try this (output is easier to read if you use ipython interpreter):

```
>>> dir(str)
['__add__',
 ...
 '__eq__',
 '__doc__',
 '__gt__',
 ...
]
```

They are called “magic” because Python invisibly calls them when you write certain expressions:

What You Write	What Python Invokes
`x == y`	`x.__eq__(y)`
`x > y`	`x.__gt__(y)`
`print(x), str(x)`	`x.__str__()`
`z = x + y`	`z = x.__add__(y)`

`z = x * y`	`z = x.__mul__(y)`
	`if that is not implemented, then`
	`z = y.__rmul__(x)`

Exercise: Write __eq__ for Money

Write the code for __eq__:

1. Two Money objects are equal if they have the same value and currency.
2. Money is never equal to non-Money. So m1 == 10 should be False.

A Template for __eq__

There is a standard “template” for an __eq__ method. It has two steps

```
def __eq__(self, other):
    test if other is the same type as this (self).
        if not, return False
    compare the attributes of self and other
        according to whatever makes sense for this class
```

Writing Good Code

You should write descriptive ***Docstring comments*** in methods, especially public methods.

```
    def __eq__(self, other):
        c
```

There are multiple standards for how to format docstring comments. This is unfortunate -- a single standard (as in Java) would be better. [Ref: my summary of docstrings]

Test Your Code

After writing __eq__ verify that it works correctly in all cases.

Test Case Compare Money that:	Expected Result
same value, same currency	True
same value, different currency	False
different value, same currency	False
different value and currency	False
compare to non-Money	False

Then Commit and Push your code to Github.

Displaying Objects as a String

```
>>> m1 = Money(10, "Baht")
>>> print(m1)
<Money object at 0x7fcf9ac13978>

>>> str(m1)
'<Money object at 0x7fcf9ac13978>'
```

(the number is the object's address in memory)

Where did this ugly string come from?

Exercise: write a **`__str__`** method to print the money nicely, such as:

```
10 Baht
1,000 Baht
0.50 Baht
```

Try to write simple, clean code without a lot of "if - else".

Hint: Try this `x = 1234.5678, print(f"{x:,.2f}")`

Characteristics of Object-Oriented Programming

The ***Three Pillars* of OOP** are:

1. **Encapsulation** - a class encapsulates both data (attributes) and the methods that apply to that data. Money *encapsulates* the value & currency, along with methods that use those.
2. **Inheritance** - one class can *extend* another class, called the superclass (parent). The subclass inherits the attributes and behavior (methods) of the superclass, but it can also *override* methods or add new methods. Money inherits the object methods from the object class, so if we invoke `money.__str__` it inherits from the object class (until we override the method).
3. **Polymorphism** - Many classes (kinds of objects) can implement the same method in different ways and we can invoke the method using an object reference without knowing which class will perform the method.
   In Money, we can write str(money) or money.__str__() without knowing whether it will be performed by the Money class or another class (object).
   from datetime import date

```
for x in ["hello", 123, date.today()]:
    print(str(x))    # invokes x.__str__()
```

Exercise: Describe each of these, and explain how the Money class illustrates them.

Better Encapsulation using Properties

In O-O programming, we usually want to **encapsulate and hide details** of how a class is implemented. Only the **interface** (methods) should be visible to other parts of the program.

This preserves data integrity (avoids unwanted changes to attributes) and gives the programmer freedom to change how a class is implemented.

But, Python does not have truly "private" attributes.

Other parts of a code can modify an object's attributes:

```
m1 = Money(10.0, "Baht")
print(m1)
'10 Baht'
# change the value to 10,000 Baht
m1.value = 10000
print(m1)
'10,000 Baht'
```

We want to **protect** the value from change outside of the Money class.

You can make the "value" be read-only (immutable) in two steps:

1. change the attribute name to **`_value`**
2. write a **`@property`** named "value" that returns the _value

```
    def __init__(self, value, currency):
        self._value = value
```

```
        self.currency = currency

    @property
    def value(self):
        """Get the value of this money object."""
        return self._value
```

Try it!

```
>>> m = Money(10, "Baht")
>>> m.value              # a property looks like an attribute!
10
>>> m.value = 1000
AttributeError: can't set attribute
```

Exercise: Make the currency a read-only property

Then, money objects will be ***immutable***. (That's a good thing.)

Python Convention: *Don't Touch Anything Whose Name Starts with Underscore*

Python relies on ***convention*** instead of enforcing rules for public/private members.

The Python convention is: you should not call or access any members of another class if the member name starts with underscore.

WRONG	CORRECT
`money._value`	`money.value`
`money.__str__()`	`str(money)`

Throw Exceptions When Something is Wrong

You studied exceptions in Programming 1.

Exercise: What are the 2 most common exception types in Python?

Complete this table:

Exception Name	Meaning
ValueError	Value of an argument is invalid or not allowed.
TypeError	Argument has the wrong type (e.g. string instead of float).

If you don't know the exception names, try this:

```
s = "hello"
s + 3
```

and this:

```
import math
math.sqrt(-1)
# create a date:  date(year, month, day)
import datetime
day = datetime.date(2022, 1, 36)       # 36 January 2022
```

How to Raise or “Throw” an Exception

Raise (*aka* "throw") an exception using the **`raise`** keyword, followed by an exception **object**.
The Exception constructors allow you to specify a string message, which you should do.

```
def foo(x):
    """x must be an integer."""
    if not isinstance(x, int):
        raise TypeError("Argument must be type 'int'")
        print("this statement is NEVER printed")   <--- never reached
```

when you raise an exception, the program flow *immediately* exits the function or code block, so the `print` statement is never executed.

Exercise: Money Constructor Should Raise Exceptions

In the constructor:

- if the currency is not a string raise a TypeError
- if the currency is an empty string, raise a ValueError
- if the value is not an int or float, raise a TypeError: isinstance(value, (int,float))

No Magic Numbers (or Strings): Avoid String Literals for Special Values

In Money, the currency is a string. This can result in inconsistencies:

```
m1 = Money(10, "Baht")
m2 = Money(10, "baht")
m3 = Money(10, "THB")
```

You should avoid using string literals in code for things that have special meanings.

There are 2 solutions for this.

1. If you need only one currency, define a string constant. In Python, names of constants should be in UPPERCASE.

   ```
   CURRENCY = "Baht"
   m1 = Money(10, CURRENCY)
   ```

2. If you need to use several currencies, define an **Enum** for currencies. An Enum is a class with a fixed set of static values. We’ll study Enum later.

Operator Overloading

We would *like* to be able to add money, like this:

```
m1 = Money(10, "Baht")
m2 = Money(20, "Baht")
```

```
total = m1 + m2

print(total)
30 Baht
```

When you write m1 + m2, Python invokes m1.__add__(m2).

This is called **operator overloading**.

We "overload" the "+" operator by defining it to work on a new data type (Money).

But, you should only add money of the same currency:

```
m1 = Money(10, "Baht")
m2 = Money(20, "Ringgit")
total = m1 + m2

***ValueError: currencies must be the same
```

Exercise: implement "+" for Money

Write an **__add__** method that does what this Docstring comment says

```
def __add__(self, other):
    """Add two money objects having the same currency.

    Parameters:
        other:  another money object to add to this one
    Returns:
        the sum of this money and other as a new Money object
    Raises:
        ValueError if the objects have different currencies.
     """
```

Test Your Code

The file **test_money.py** contains unit tests for the methods in this assignment. It's a good practice to test your code frequently. Some tests will fail for code you haven't written yet. That's normal.

Overload Multiplication

How can we compute **interest** or **VAT**?

```
total = Money(198, "Baht")
vat = total * 0.07
```

In this case, we want to compute Money x float. (It does not make sense to write Money x Money.)

- What method do you need to implement for this?
- What is the datatype of the parameter?

Exercise: Implement Money * float (overload the * operator)

Write a magic method and a *good docstring comment* (similar to __add__), so we can write:

```
total = Money(200, "Baht")
vat = total * 0.07
print(vat)
14 Baht
```

Static versus Instance Methods

Methods that are "bound" to an object and require an object reference are called ***instance methods***. (An object is an *instance* of a class.) Examples of instance methods are:

```
str(x)              # calls x.__str__(), an instance method
"hello".upper()     # upper is an instance method of string objects
file = open("README.md")
file.readline()     # readline is an instance method of File objects
```

In Python, the first parameter to an instance method is always **`self`**, which refers to the object.

Class methods or **Static methods** are methods performed by the class, and not "bound" (tied) to a specific object.

You invoke a class method using: `ClassName.methodname()`

Example: The Python **`date`** class has a class method **`today`**.

```
from datetime import date     # 'date' is a class in datetime module
d = date.today()              # today is class method, it returns a date
type(d)
<class 'datetime.date'>
print(d)
2022-01-12
```

To define a class method inside a class, write:

```
@classmethod
def methodname(cls):
    # cls parameter refers to this class, not an instance of the class
```

Conceptual View of Instance & Class Methods:

Instance methods are behavior performed by objects

Class methods are services provided by a class

Exercise: We would like to create a money object from a String. This would be useful, for example, if we read strings from a file and need to convert the strings to Money.

```
m = Money.fromstring("1500 Baht")
```

Write a **class method** named **`fromstring`** that splits a string into a value and currency and returns a new Money object with the value & currency. Write a *docstring comment* for this method.

Hint: how to split a string into substrings at whitespace? Try `"10 Baht".split()`

What To Submit

Push your Money code to Github Classroom. Your code should include:

- properties for value and currency
- __eq__ and __str__ methods
- __add__ and __mul__ methods to overload + and *
- class `fromstring` method to create money objects from strings
- Money constructor and __add__ method raise exceptions
- docstring comments for all methods
- well-formatted code, including a blank line between each method. If your code looks like *junk*, your score will be *junk*, too.

Summary

1. Classes encapsulate data (attributes) and behavior (methods) related to a single “thing” or abstraction, such as Money, Person, Course.
2. Class “members” with names beginning in underscore (_) should be treated as private. Don’t directly access them, except inside the class or a subclass.
3. Protect attributes from change (when desired) using properties.
4. Write docstring comments for classes and public methods to provide documentation. Include
    - first line is a sentence that briefly describes the method or class.
    - then leave a blank line
    - describe the Arguments (Parameters)
    - describe the Return value, if any
    - describe any exceptions Raised, and when they are raised
    - It is OK to omit details for simple, obvious methods like __str__.
5. Raise an exception when something is wrong. Don’t silently ignore errors.
6. Use named constants for special values, not string or numeric literals.
7. You can *overload* operators by implementing certain magic methods, such as __eq__ for == (equality comparison) or __add__ for “+”.
    - If you don’t implement a magic method, then the superclass’s method is invoked instead.
8. In a class, *instance methods* are behavior bound to an object while *class methods* are performed by the class (not bound to an object). A common use of class methods is to create new objects (also called "factory methods").
9. To define a class method, use the `@classmethod` annotation. Python also has *static methods* with annotation `@staticmethod` which do not have a "cls" parameter. Static methods aren't used much in the Python API since a top-level function can be used instead.