Review of OOP and Python

This example includes a review of object-oriented programming basics, and how to apply them in Python.

## 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 repo to your local computer.

## 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 interpretter):

>>> 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 | *complete this* |

## 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.

Push your code to Github.

## Lesson Learned

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

def \_\_eq\_\_(self, other):

*test that other is the same type as this (self).*

*if not, return False*

*compare the attributes of self and other*

*according to whatever makes sense*

## Writing Good Code

You should write *Docstring* comments in methods, esp. public methods.

**def** **\_\_eq\_\_**(self, other):

"""Two money objects are equal if they have same value and currency.

Parameters:

other - a Money object to compare to this

Returns:

True if they have same value and currency, otherwise False

"""

There are other formats for writing docstrings.

## Test Your Code

After writing \_\_eq\_\_ verify that it works correctly in all cases.

| Test Case | 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 |

## Exercise: Display Money 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 -
2. Inheritance -
3. Polymorphism -

**Describe each of these**, and explain how the simple 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 ensures data integrity and gives the programmer freedom to change the way 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 the value*

**print**(m1.value)

*# change the value to 10,000 Baht*

m1.value **=** 10000

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 @propert named “value” that returns the \_value

**def** **\_\_init\_\_**(self, value, currency):

self.\_value **=** value

self.currency **=** currency

**@**property

**def** **value**(self):

**return** self.\_value

Try it!

**>>>** m **=** Money**(**10, "Baht"**)**

**>>>** m.value

10

**>>>** m.value **=** 1000

AttributeError: can't set attribute

**Exercise**: Make the currency a property

## Python Convention: *Don’t Touch Anything Starting 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.

| BAD | CORRECT |
| --- | --- |
| money.\_value | money.value |
| money.\_\_str\_\_() | str(money) |

## Throw Exceptions When Something is Wrong

You studied exceptions in Programming 1.

What are the 2 most common exception types in Python?

Complete this table:

| Exception | Meaning |
| --- | --- |
|  | Value of an argument is not allowed. |
|  | Type of an argument is incorrect or incompatible. |

If you don’t know, try this:

s **=** "hello"

s **+** 3

and:

import math

math.sqrt(**-**1)

*# create a date: date(year, month, day)*

import datetime

day **=** datetime.date(2022, 1, 36)

## How to Raise or “Throw” an Exception

The Exception classes allow you to specify a string message:

**def** **foo**(x):

"""x must be an int."""

**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 method or function.

## Exercise: Money Should Raise Exceptions

In the constructor:

* if the currency is not a str 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

## 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 constants 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)

1. 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 redefining it to work on a new datatype (Money).

But, you can 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 the Docstring comment says

**def** **\_\_add\_\_**(self, other):

"""Add two money objects having the same currency.

Arguments:

other - another money object to add to this one

Returns:

the sum of this money and other other as a **new** Money object

Raises:

ValueError if the object have different currencies.

"""

## Test Your Code

explained in class

## 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 double. (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 x double

Write the method and a docstring comment.

## 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.