Contents

Tips, Tricks, and Pitfalls 1

Level 2 1

I: Special Homework Instructions 1

II: Outputting Objects 2

III: Comparing Objects 3

IV: Attributes 4

V: Class Metadata 5

VI: Inheritance & Mixin Classes 7

VII: Recursive Functions 9

Tips, Tricks, and Pitfalls

Level 2

I: Special Homework Instructions

1. Previous levels were very explicit about what your exercise code should do. When writing classes, the focus changes somewhat: The exercises will generally specify what your classes should look like and their internal functionality. However, the exercises will be somewhat more vague about what your program should actually do when run (i.e., what's in your **main** function). This is because the primary focus of Object Oriented Programming is writing robust and well-designed classes. Your **main** function should always contain code that tests each and every aspect/method of your classes; you should test everything using simple cases, and also test everything using 'edge' cases. 'Edge' cases are situations that may occur rarely, but are possible to occur (for example, an unlikely function input that causes the program to crash due to a **ZeroDivisionError**). It is up to you to figure out what needs to be tested and to ensure that you have properly tested everything.

   Strong testing cannot be stressed enough; if you have robust test code in your **main** function, then you are much more likely to have issue-free classes (and a better grade).

II: Outputting Objects

1. In a previous point (in Level 1) we discussed the difference between outputting a variable by using the variable name only versus using a **print** statement. It's worth understanding how this works for classes. The following example illustrates the default behavior for classes:

```
class Loan(object):
    def __init__(self, face, rate, term):
        self._face = face
        self._rate = rate
        self._term = term

    # Additional Loan class code here
```

```
>>> from loan import Loan
>>> l = Loan(100000, .02, term=180)
>>> l
<loan.Loan object at 0x0457F580>
>>> print(l)
<loan.Loan object at 0x0457F580>
```

As you can see, the output in both cases are the same, and not very descriptive. It tells you that you have a Loan object located at a certain memory address. To make this more descriptive, we can customize our classes with the **__str__** and **__repr__** functions. The former is what will be displayed when the object is converted to string (either when using **str** or **print**) and the latter is what will be displayed when using the variable name only (the Python ***representation***). See example below:

```
class Loan(object):
    def __init__(self, face, rate, term):
        self._face = face
        self._rate = rate
        self._term = term

    # Additional Loan class code here

    def __str__(self):
        # Returns the str value of the Loan object.
        # I've put (str) into the string to make it obvious which function is called,
        # for demonstrative purposes.
        return 'Loan (str): ' + str(self._face) + ',' + str(self._rate) + ", " + str(self._term)

    def __repr__(self):
        # Returns the repr value of the Loan object.
        # I've put (repr) into the string to make it obvious which function is called,
        # for demonstrative purposes.
        return 'Loan (repr): ' + str(self._face) + ',' + str(self._rate) + ", " + str(self._term)
```

```
>>> from loan import Loan
>>> l = Loan(100000, .02, term=180)
>>> print(l)
Loan (str): 100000,0.02, 180
>>> str(l)
'Loan (str): 100000,0.02, 180'
>>> l
Loan (repr): 100000,0.02, 180
```

III: Comparing Objects

1. We take for granted that comparing two values works in Python. For example, 5 == 5 (True), 5 != 6 (True), or 5 != 5 (False). What happens if you try to check equivalence of two user-defined object types? Let's see…

```python
# Stripped-down Point class
class Point(object):
    def __init__(self, x, y):
        self._x = x
        self._y = y

p1 = Point(1,2)
p2 = Point(1,2)
print(p1 == p2) # False, but should be True!
```

As you can see, checking equivalence does not work! The reason for this is that Python does not know what is considered equivalent for a user-defined class; it is up to the programmer to define this. To enable proper equivalence checking in your classes, you need to implement the **__eq__** function in your class. The logic of this function should be customized to determine whether or not the objects are equivalent (it returns a Boolean, either True or False). For example:

```python
# Stripped-down Point class
class Point(object):
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def __eq__(self, other):
        # This gets called when doing p1 == p2.
        # 'self' is this object on the left-hand side (p1).
        # 'other' is the object on the right-hand-side (p2).
        # Return True if x and y are the same for self and other.
        return other._x == self._x and other._y == self._y

p1 = Point(1,2)
p2 = Point(1,2)
print(p1 == p2) # True!
```

There are other, similar functions for the other comparison operators: **__ne__** (!=), **__lt__** (<), **__gt__** (>), **__le__** (<=), **__ge__** (>=).

Note that there is no need to implement any of these unless you wish for your objects to be comparable, for whatever reason.

There is a lot more info on this topic, especially in regards to inheritance, multiple inheritance, and hashes, but this is the subject of an advanced course.

IV: Attributes

1. Attributes in Python refer to member data of a class. We have seen setting and reading member data directly from an object (or class-level data). It is also possible to indirectly create or set member data using the **hasattr**, **setattr**, and **getattr** built-in Python functions. A few examples below:

```
class Point(object):
    # The below are both 'attributes' stored at both the object and class level.
    _x = 0.0
    _y = 0.0

    def __init__(self, x=None, y=None):
        # Directly setting the object-level attribute from within the class.
        # This does not change the class-level attribute.
        # However, if x or y parameters are None,
        # it will default to the value of the Point-level attribute.
        self._x = x if x is not None else Point._x
        self._y = y if y is not None else Point._y

    def checkFunction(self):
        self._checkFunctionCalled = True

pt = Point()
# Directly setting the object-level attribute from outside the class.
pt._x = 5
pt._y = 10

# Indirectly getting the object-level attribute from outside the class.
print(getattr(pt, '_x'))
print(getattr(pt, '_y'))

print(pt._checkFunctionCalled) # Will cause an error, since checkFunction was never called.

# Checks if _checkFunctionCalled exists on the object.
# It will only exist if checkFunction has been called.
if hasattr(pt, '_checkFunctionCalled'):
    print(pt._checkFunctionCalled)

print(hasattr(pt, 'myNewAttr')) # Will print False
setattr(pt, 'myNewAttr', 'Test') # Same as pt.myNewAttr = 'Test'
print(pt.myNewAttr) # Will print 'Test'
```

The usefulness of **hasattr** has already been demonstrated in the lecture. The **getattr** and **setattr** are useful to be able to dynamically access a member variable of an object. Since it uses a string representation of the attribute names, one does not need to know the attribute name when writing the code (can ask the user or have it saved down somewhere). Exactly when and why this is useful in practice is beyond the scope of this course, but it is important to understand how these functions work.

V: Class Metadata

1. Say you are debugging your code and encounter an object of a class type that you are not familiar with (and do not necessarily have easy access to the class' source code). There is a simple way to get a list of all functions and attributes on the object: The **dir** function. See example below:

```
class MyClass(object):
    value = 5
    def Func(self):
        print('DoSomething')

obj = MyClass()
# This displays a list of all the functions and variables in the object.
# As expected, value and MyFunc appear in the list.
# However, there are many additional double-underscored ones that appear as well.
# These double-underscored ones are either metadata (see next point)
# or built-in functions that come with every class.
print(dir(obj))
```

2. Classes in Python contain *metadata.* The metadata is stored in variables that begin/end with double-underscore. A few of them are:
    - **__class__**: Contains he class instance of the object. It's essentially an alias for the actual class:
        - **__class__** itself contains an object, with its own metadata. One 'sub' metadata worth noting is **__name__**, which gives the string representation of the class name. For example:

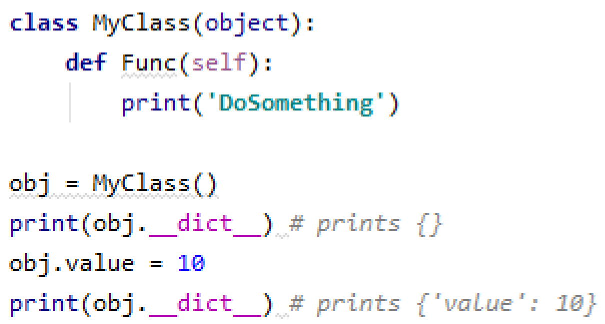
```
class MyClass(object):
    def Func(self):
        print('DoSomething')

obj = MyClass()
print(obj.__class__) # prints <class '__main__.MyClass'>
print(obj.__class__.__name__) # prints MyClass
```

        - As the object stored in **__class__** is essentially an alias for the actual class, it can be stored and used in lieu of the class name to instantiate new objects. For example:

```
class MyClass(object):
    def Func(self):
        print('DoSomething')

obj = MyClass()
print(MyClass.__name__) # prints MyClass
classInst = obj.__class__ # classInst is not an alias for MyClass
print(classInst.__name__) # prints MyClass

# ClassInst is not a synonym for MyClass.
# Can instantiate new objects with it!
newObj = classInst()
print(newObj.Func())
print(newObj.__class__) # prints <class '__main__.MyClass'>
print(newObj.__class__.__name__) # prints MyClass

print(obj.__class__) # prints <class '__main__.MyClass'>
print(obj.__class__.__name__) # prints MyClass
```

This can be very useful for indirectly instantiating new objects of the same type as existing objects.

- **__dict__**: Contains a dictionary ‘snapshot’ of all the attribute values in the object. For example:



```python
class MyClass(object):
    def Func(self):
        print('DoSomething')

obj = MyClass()
print(obj.__dict__) # prints {}
obj.value = 10
print(obj.__dict__) # prints {'value': 10}
```

- **__module__**: Contains the full module path where the object’s class resides. This can be useful for debugging and investigatory purposes.

VI: Inheritance & Mixin Classes

1. As discussed in the lectures, a benefit to using inheritance is that one can avoid duplicating common functionality between classes. For example, if every shape (i.e., Point, Line, Circle) has a **Draw** function, it's simpler to define a Shape base class that contains this common **Draw** function, then to redefine **Draw** in each individual shape class.

Another example would be a loan class hierarchy. There are many different types of loans: Mortgage, Auto, Student, etc. Every loan, no matter what type, has a face-value, rate, and term. Therefore, by defining a **Loan** base class (and deriving the specific loan type classes), there is no need to explicitly define the face, rate, and term variables in each of the derived loan classes. Additionally, the base class can implement common functions such as **montlhyPmt**, **loanBalance**, etc. – there is no need to duplicate the complex logic of these functions in each derived loan class. The derived loan classes can then *solely* focus on implementing functionality that is specific to its type (for example, **MortgageLoan** can implement Private Mortgage Insurance (PMI)).

In certain situations, one may wish to customize functionality in a derived class that may already exist in the base class. For example, a **MortgageLoan** has a different formula for calculating **monthlyPmt** (since a mortgage may require an extra monthly payment amount on account of PMI). The simplest way to account for this would be to *override* the **monthlyPmt** method that exists in the base **Loan** class, by explicitly defining the same method (with the custom formula/logic) in the derived, **MortgageLoan** class.

The above approach to derived-class customization has a major shortcoming: If one has multiple loan classes that are mortgages (such as a **VariableRateMortgage** and a **FixedRateMortgage**), the custom **monthlyPmt** logic will need to be defined for each mortgage class – which is redundant. To this end, we use something called a *mixin class*. A *mixin class* is a class that is used to 'mix in' with other classes; it provides additional functionality to the class it's getting mixed into.

In the above loans example, we can simply create a **MortgageMixin** class that defines the mortgage-specific functionality. This includes a **PMI** function (which returns the extra monthly payment amount) in addition to a custom **monthlyPmt** method that is meant to override the standard Loan **monthlyPmt** method. Then, **VariableRateMortgage** and a **FixedRateMortgage** derive-from both **MortgageMixin** *and* **Loan** (making sure to derive-from the mixin class first, to ensure its version of **monthlyPmt** takes precedence).

Note that **MortgageMixin** is a standalone class; it *should not* derive from **Loan** -- it's meant to be *mixed-in* to existing Loan classes, to turn them into mortgages. From a technical standpoint, deriving **MortgageMixin** from **Loan** would cause a diamond-inheritance problem when creating the **VariableRateMortgage** and **FixedRateMortgage** classes since the **Loan** base class be inherited multiple times, from multiple parent classes (this is beyond the scope of this course, but see https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem for more info).

Notice in the above example how mixin classes may define functions that override functionality from another base class. **MortgageMixin** is not a **Loan** in of itself, but it's expected to be mixed-into a **Loan** derived class. To this end, mixin classes are *never* meant to be instantiated as standalone objects directly; in Python, mixin classes are usually used as the first base class in the inheritance list, to ensure its functionality takes precedence over the actual base class' functionality. It's also important to clearly comment in the mixin class' docstring what its intended use is.

One more thing to note is that it’s possible for a class to derive-from multiple mixin classes. For example, a **Dog** can be derived from **TailedAnimalMixin** (which provides tail-specific functionality), **PetMixin** (which provides pet-specific functionality), and of course the base **Animal** class (which provides the common functionality for all animals). See the provided sample code, to get a better sense of how these mixin classes are structured.

For additional info on mixin classes, see https://en.wikipedia.org/wiki/Mixin#In_Python.

VII: Recursive Functions

1. The lecture on designing the **Loan** class mentioned that the exercises will require you to implement a regular and *recursive* version of the **balance**, **principalDue**, and **interestDue** functions of your **Loan** class. We also demonstrated how the recursive version works, conceptually, in Excel. However, for those who have not programmed before, we did not discuss how to actually implement recursive code in general.

   Conceptually, a recursive function is one that calls itself successively to get the final result. This is in contrast with an iterative function, that uses a loop (or a closed formula, such as for the simple **balance** function for **Loan**). Not every function is a candidate for a recursive version; only functions that are conceptually recursive. For example, a factorial is conceptually recursive since factorial(5) is the same as 5*factorial(4), factorial(4) is the same as 4*factorial(3), and so on. See below for the iterative and recursive solutions for factorial to compare:

```python
# Iterative version
def factorial(N):
    if N < 0:
        print('N must be at least 0')
        return
    res = 1
    # There are better ways,
    # but using the obvious approach for illustration.
    for i in range(1, N+1):
        res *= i

    return res

# Recursive version.
def factorialRecursive(N):
    if N < 0:
        print('N must be at least 0')
        return
    # This is called the 'base case'.
    # Factorial(0) == 1.
    # Without this base case, the recursion would go on forever.
    elif N == 0:
        return 1
    else:
        return N*factorialRecursive(N-1)
```

   The recursive version of a function is generally more intuitive than the iterative or formulaic version of the same function; however, recursive functions generally have much poorer performance so should be avoided when possible.