# CS 112 – Fall 2020 – Programming Assignment 12

**Classes and Objects**

**Due Date: Tuesday, November 24th, 11:59pm**

## Background

Classes allow us to define new types for Python. We can first think of a class as defining a new container – instead of a list, tuple, set, or dictionary, we can have our own collection of values, each with a chosen name rather than an index/key. We can then read out a value or update a value, much like reading or replacing the values in a list or dictionary. But we can also put methods in a class definition, giving us a way to specify the exact ways we should interact with values of this new type. Once we have created a class definition, we can create as many objects of the new type as we want and use them in our programs. We can create entirely new types during the design phase of writing a program. This enables us to think in terms of types (and instances of types) that better model what we see in the real world, instead of endlessly relying on lists or dictionaries (and having to remember exactly how we intended to use those things as proxies for the values we actually had in mind).

## Guidelines

- You are **not** allowed to **`import`** anything.
- You are **not** allowed to use anything that hasn't been covered in class, including the *dictionary/list comprehension construct, lambda functions, etc.*
- **Adding definitions**: The provided specification is just the bare minimum; you may add additional methods and functions in support of your solution.
- Hard coding is not allowed.
- You should be skimming Piazza regularly in the event that clarifications get made.

## Testing

You will be provided with a tester to help you check your code, but keep in mind that the grader we use to grade your work will be different from this tester. This means that there will be cases that the tester doesn't check and, therefore, your code might have logic errors that the tester doesn't detect. You must do additional checks, on your own, to make sure that you haven't missed anything and your code is correct. You do ***not*** need to modify the tester we provide, just test on your own any way you like. The goal is to help you put more focus on writing logically correct programs instead of trying to pass certain tests only. Thus, the tester will be made available on **Nov. 20th** and the grader (a.k.a. autograder) will be made available on Gradescope **after** the deadline.

**Focused testing:** you can narrow down the focus of the tester by feeding it the name of a class (which only runs the **`init/str/eq`** kinds of tests), or the method name of things that are needed in a particular class; this is the `Plan` class. Note that one class may rely on other classes to work properly. We had to manually create this listing, and it's how the test cases were named. Here is the full list of test batches you can select:

- o classes (only basic init/str/eq tests are performed for `Plan`): `Phone, Call, Features, Plan`
- o methods: `add_phone, remove_phone, add_call, mins_by_phone, make_call, remove_call, billing`
- o for example:
  `demo$ python3 tester12.py yourcode.py Phone Plan add_phone mins_by_phone`

## Grading Rubric

| | | |
|---|---|---|
| Submitted correctly: | 2 | # see assignment basics file for file requirements! |
| Code is well commented: | 8 | # see assignment basics file for how to comment! |
| Autograder: | 90 | |
| TOTAL: | 100 | |

**Note:** If your code does not run and crashes due to errors, it will receive zero points. Turning in running code is essential.

## Task

You will implement some basic classes and methods to manage telephone plans. A `Plan` can have multiple `Phone` lines and include a record of multiple `Call`s. By calling the methods, someone could manage the lines of a plan and calculate the bill. Working through the classes in the given order is the simplest path to completion.

**class Phone:**

- **def __init__(self, name, area_code, number, is_active):** constructor of a telephone line: create/initialize instance variables for **name**, **area_code**, **number** and **is_active** (default value is **True**). Assume **area_code** is a three-digit integer with no leading 0/1; assume **number** is a seven-digit integer with no leading 0/1.
- **__str__:** create/return a string as in this example: "703-993-1530␣(GMU)"
- **__eq__:** determine if this object is equivalent to another. Two lines are considered equal if they have the same area code and the same number.
- **activate:** set **is_active** of this object to be **True**.
- **deactivate:** set **is_active** of this object to be **False**.

**class Call:**

- **__init__(self, caller, callee, length):** create/initialize instance variables. Assume **caller** and **callee** are Phone lines and that **length** is an integer. In case of the following errors, do not create any instance variables:

    - If either **caller** or **callee** is inactive
    - If **caller** and **callee** are equivalent
    - If **length** is negative
- **__str__:** create/return a string as in this example:
  "Call:␣703-993-1530␣(GMU)␣->␣703-993-1000␣(George)␣|␣20"
  **Hint:** when obtaining strings for the Phone lines, how can you rely upon the __str__ definition of Phone to make this a short/trivial method to write?
- **is_local:** return **True** if both **caller** and **callee** have the same **area_code**; **False** otherwise.

**class Features:**

- **__init__(self, basic_rate, default_mins, rate_per_min, has_rollover):** create/initialize instance variables for all four non-self parameters. You can assume all arguments are valid and there is no need to perform any checking.
- **__str__:** create/return a string as in this example:
  "Basic␣rate:␣25.50,␣Default␣mins:␣200,␣Rate/min:␣0.50,␣Rollover:␣True"

**class Plan:**

- **__init__(self, features, phones):** create/initialize instance variables for **features** and the list of **phones** (optional) in the plan. When **phones** is not provided, use an empty list as its initial value. Create an instance variable **calls** to keep a list of calls made/received by any line of this plan, initialize it to be an empty list. Also create instance variables for **balance**, **rollover_mins**, and **mins_to_pay**, all starting at zero.
- **__str__:** create/return a string similar to:
  "Plan␣(Basic␣rate:␣25.50,␣Default␣mins:␣200,␣Rate/min:␣0.50,␣Rollover:␣True,␣[],␣[])"
  See test cases for more examples. **Hint:** when obtaining strings for the two lists of lines and calls, how can you rely upon other __str__ definitions to make this a short/trivial method to write?
- **add_call(call):** add the given **call** to the end of the list of calls. Either caller or callee of the given **call** should be a phone of this plan, otherwise do not append and return **False**. If **call** can be added, check whether the call is billable based on the following rules:
    - A local call is free and not billable; **Hint:** *use your Call class's is_local method to help.*
    - A call made between two lines of this plan is free and not billable

  For a billable call, increment instance variable **mins_to_pay** based on the length of the call. All updates made in this method change instance variables directly and the method returns **None**.
- **remove_call(call):** remove the given **call** from the list of calls and return **None**. If **call** is not present in the list, return **False**.
- **make_call(caller, callee, length):** create a call based on **caller**, **callee**, and **length**, then add it to the end of the list of calls and update **mins_to_pay** accordingly. Return **True** if the call is created and added successfully; return **False** otherwise (e.g. the call cannot be created or the call cannot be added).
- **mins_by_phone(phone):** calculate and return the *number of call minutes* associated with the given **phone** line in this plan based on the current list of **calls**. A call is associated with a

phone line if the line is either its caller or callee. Return zero if the given **phone** is not included in this plan. **Hint:** *Make sure your Phone class's __eq__ method works, first.*

- **add_phone(phone):** Add the given **phone** line to the end of the list of phone lines. If there is already a phone in the plan by the same area code and number, do not append and return **False**. All updates made in this method change instance variables directly and the method returns **None**. **Hint:** *Your Phone class's __eq__ method can help.*
- **remove_phone(phone):** Remove the given **phone** line from the list of phone lines; remove calls that are **only** associated with the given **phone** (not any other phone of this plan) from the list of calls. A call is associated with a phone line if the line is either its caller or callee. If **phone** is not present in the list of phone lines, perform no removal and return **False**. All updates made in this method change instance variables directly and the method returns **None**. **Hint:** *Your Phone class's __eq__ method can help.*
- **billing:** perform billing activity as if it is at the end of a pay period:
    - Calculate the amount to pay based on the billable minutes and plan features. Assume the following rules:
        - If the number of minutes to pay (**mins_to_pay**) is no greater than **default_min** of the plan, no additional charge beyond the **basic_rate**;
        - Otherwise, check if the remaining billable minutes can be covered by **rollover_mins** from the previous pay period;
        - Any remaining minutes will result in an additional charge beyond **basic_rate** using **rate_per_min**;
    - Update balance and perform maintenance of the plan account:
        - Update the **rollover_mins** based on the phone plan and actual usage. Any unused default minutes gets accumulated into **rollover_mins** if rollover is allowed. You can assume rollover minutes never expire.
        - Clear the list of calls and reset the number of minutes to pay (**mins_to_pay**) to zero.
        - Update balance. If the unpaid balance is too high (strictly greater than five times of the **basic_rate**), deactivate all lines of the plan.
    - All updates made in this method change instance variables directly and the method returns **None**.