ASSIGNMENT 1 DOCUMENTATION

The following documentation will help understand the code defined within the multiple classes defined for assisting the Mewbile Tech company application. Covered below is the classes Term Contract, Prepaid Contract and Month-To-Month Contract with a thorough description of each classes' methods and attributes with a rundown on how each method runs along with general code structure followed throughout.

Class TermContract(Contract)

This class defines a term contract for a customer's phone-line. A term contract is a sort of contract for a customer that has a defined start and ending date and necessitates commitment through the end date. An initial term deposit is added to the first month's payment when a contract is for a set period. This unique term deposit will only be reimbursed to the customer if they cancel the contract after the designated return date.

Within this class several methods are overridden using the parent Contract class definitions but set to the needs of the current Term Contract class. These methods will help in the definition of this class and to change specific attribute values used within this class.

These methods are:

- Initializer method *(__init__())* : Method to initialize the attributes stored within the class to specified values.

- New month method *(new_month())* : Method to allow customer to advance to new month in contract with the specified month and year and bill accordingly

- Bill call method *(bill_call())* :Method to bill minutes of any call made by customer to the corresponding bill. Specific to the Term contract all free minutes are used up first before billing of call minutes starts.

- Cancel contract method *(cancel_contract())* :Method to cancel and close phoneline associated with contract and returns amount owed once called. Specific to Term contract is a check to see if the term is carried to date specified within contract.

Along with these methods there are certain attributes defined to be used within the functions and assist in executing code.

These attributes include:

-start : Attribute storing start date of the current contract

-end : Attribute storing end date of the current contract as specified.

-bill : Attribute storing bill of the contract for current month of contract

Initializer method:

__init__ (self, start: datetime.date, end: datetime.date) => None

This method takes in two parameters and initializes all attributes to be used within the class. This method takes in a start and end date specific to a particular term contract in datetime format as arguments in its function call. This method returns a None value.

Implementation:

The attributes storing the contract start date and end date are initialized to the values given within function call in the datetime format. The attribute storing the current month and year of the contract is set to the value of the contract start date month and year both as integer values. The attribute bill is initialized to None so that it can be changed to an object holding data later within the class.

Private attribute:- current: Attribute used to track which month and year of the contract is the customer currently on.

New month method:

new_month(self, month: int, year: int, bill: Bill) => None

This method allows the customer to advance to a new month in the contract as per given month and year values. This method takes in the values of month and year as integers and the current bill instance as an object within its call. This method returns a None value.

Implementation:

The attribute storing the contract bill is set to the value of the bill object given within function call. This attribute is given values storing the bill type and bill minute rate stored within *TERM_MINS_COST* (which is a constant value given) using the function *set_rates()* within the *Bill* class. The bill attribute is instantiated with more values by altering the fixed cost of the bill by the term contract's monthly fee stored in *TERM_MONTHLY_FEE* (which is a constant value given) using the function *add_fixed_costs()* within the *Bill* class.

The method also checks whether the contract is in its first month of being initiated by comparing equivalence between the month and year in function call and month, year of contract start date stored within the class attribute start. If found to be in the first month of use term deposit is added to fixed costs using the function *add_fixed_costs()* within the *Bill* class with the term deposit value in *TERM_DEPOSIT* (which is a constant value given). The attribute storing the current month and year of the contract is also updated to the values of the month and year given within function call.

Bill call method:

bill_call(self, call: Call) => None

This method bills the minutes of call for any call the customer makes. Specific to term contract is a number of free minutes that the customer can use up before starting to get billed. This is done through a value of the bill attribute *free_min* which stores the value of how many free minutes the customer has used up to that point. This method takes in the current call instance for a particular call as an object within its function call. This method returns a None value.

Precondition: A bill call has already been created and it is safe to assume that a bill has already been created for that month/year when this function is called. As in the class attribute bill is already in the right month/year.

Implementation:

Call duration is added to the value *free_min* using the function *add_free_minutes()* within the *Bill* class with the value of the call-in minutes calculated by rounding up [using *ceil()* function] the quotient calculated by dividing the call duration by 60 as call duration is stored in terms of seconds. If at any point the free minutes for the bill surpasses the maximum amount of free minutes which is stored in the variable *TERM_MINS* (which is a constant value given) then the *free_min* is reset back to the maximum free minutes. The amount which is over is calculated and is billed to the customer using the function *add_billed_minutes()* within the Bill class using the difference of the free minutes and the maximum free minutes in the contract.

Cancel contract method:

cancel_contract(self) => float

This method allows the customer to cancel the contract whenever they wish too. This method takes in no values in its function call. This method returns a floating-point value which represents the amount/bill the customer must pay once the contract is cancelled.

Precondition: A bill call has already been created and it is safe to assume that a bill has already been created for that month/year when this function is called. As in the class attribute bill is already in the right month/year.

Implementation:

Since the conditions of the term contract state that the term deposit will be returned if and only if the date of the cancelation is after the end date specified within the term contract. This is done by checking if the end month is less than the current month value stored within the current attribute of the class and checking equivalence between the end year and current year value stored within the attribute *_current*.

It also checks if the end year value is less than the current year value stored within the current attribute of the class. If cancelation date is after the specified end date, then the term deposit stored within constant variable *TERM_DEPOSIT*, is removed from the fixed cost value stored within the class *Bill* within attribute *fixed_cost*, as a means of returning it to the customer. Otherwise, if the cancelation date is before the specified end date then term deposit is not returned and bill *fixed_cost* remains the same and this amount is returned using the *get_cost()* function defined within the Bill class. This method also resets the contract's start and end date by assigning it a None value.

Class MTMContract(Contract)

A contract of the month-to-month type lacks both an end date and any initial term deposit. There are no free minutes included and the call prices are greater than with a term contract, but there is no long-term commitment.

Within this class most methods are inherited from the Contract class except for one method which is the abstract new month method.

Along with these methods there are certain attributes defined to be used within the functions and assist in executing code.

These attributes include:

-start: Attribute storing start date of the current contract

-bill: Attribute storing bill of the contract for current month of contract

New Month method:

new_month(self, month: int, year: int, bill: Bill) => None:

This method allows the customer to advance to a new month in the contract as per given month and year values. This method takes in the values of month and year as integers and the current bill instance as an object within its call. This method returns a None value.

Implementation:

The attribute storing the contract bill is set to the value of the bill object given within function call. This attribute is given values storing the bill type [contract type] and bill minute rate stored in *MTM_MINS_COST* (which is a constant value given) using the function *set_rates()* within the Bill class. The bill attribute is given more values by altering the fixed cost of the bill by the month-to-month contract's monthly fee stored in *MTM_MONTHLY_FEE* (which is a constant value given) using the function *add_fixed_costs()* within the *Bill* class.

Class PrepaidContract(Contract)

Prepaid contracts come without included minutes and have start dates but no end dates. It has a balance attached to it, which represents how much money the consumer owes. The consumer has this much credit, or has paid this amount in advance, if the balance is negative. Upon signing up for the prepaid contract, the customer must pay a prepayment, which can be any amount. The balance from the previous month is carried over into the current month, and if there is less than $10 in credit remaining when the new month begins, the balance must be topped off with $25 in credit. Also note that if the contract is cancelled with any remaining credit (a negative balance), the balance is lost. If the balance is positive, the amount should be returned, and an application can inform the user that there is still a balance due (although the application does not check for this).

Within this class several methods are overridden using the parent Contract class definitions but set to the needs of the current Term Contract class. These methods will help in the definition of this class and to change specific attribute values used within this class.

- Initializer method *(__init__())* : Method to initialize the attributes stored within the class to specified values.

- New month method *(new_month())* : Method to allow customer to advance to new month in contract with the specified month and year and bill accordingly. Specific to Prepaid contract is that balance must be loaded up with 25$ if it ever goes below 10$.

- Bill call method *(bill_call())* :Method to bill minutes of any call made by customer to the corresponding bill.

- Cancel contract method *(cancel_contract())* :Method to cancel and close phoneline associated with contract and returns amount owed once called. Specific to Prepaid contract is that balance of phoneline is forfeited if there exists any balance remaining else it is returned to customer.

Along with these methods there are certain attributes defined to be used within functions and assist in executing code.

These attributes include:

-start: Attribute storing start date of the current contract

-bill: Attribute storing bill of the contract for current month of contract

Initializer method:

(__init__ (self, start: datetime.date, balance: float)) => None

This method takes in two parameters and initializes all attributes to be used within the class. This method takes in the values of starting date of specific Prepaid contract in datetime format and starting balance as set by customer as a floating-point value within class call. This method returns a None value.

Implementation:

The attributes storing the contract start date are initialized to the value given within function call in datetime format. The attribute storing the current balance of customer is also initialized to the negative value of balance given within function call. The attribute bill is initialized to None holding no value so that it can be changed to an object holding data later within the class.

Private attribute:- balance: Attribute storing current balance of the customer with this contract. [Balance amount is in negative]

New month method:

(new_month(self, month: int, year: int, bill: Bill)) => None

This method allows the customer to advance to a new month in the contract as per given month and year values. This method takes in the values of month and year as integers and the current bill instance as an object within its call. This method returns a None value.

Precondition: A bill call has already been created and it is safe to assume that a bill has already been created for that month/year when this function is called. As in the class attribute bill is already in the right month/year.

Implementation:

The attribute storing the contract bill is set to the value of the bill object given within function call. This attribute is given values storing the bill type [contract type] and bill minute rate stored within *PREPAID_MINS_COST* (which is a constant value given) using the function *set_rates()* within the Bill class. If the current balance is less than 10 dollars, then as per the contract definition the customer must top up 25 to the balance.

The method also checks whether the contract is in its first month of being initiated by comparing equivalence between the month and year in function call and month, year of contract start stored within the class attribute *start*. If found to be in first month of use the balance is added to fixed costs using the function *add_fixed_costs()* within the Bill class with the value stored within private attribute *balance*. Otherwise, the balance is topped up by 25 by subtracting 25 from the balance and fixed costs of contract stored within *fixed_costs* is updated by adding the balance plus the 25 dollar using the function *add_fixed_costs()* within the *Bill* class.

Bill call method:

bill_call(self, call: Call) => None

This method bills the minutes of call for any call the customer makes. This method takes in the current call instance for a particular call as an object within its call. This method returns a None value.

Implementation:

The call minutes are billed to the customer using the function *add_billed_minutes()* within the Bill class using the call duration in seconds and converting it into minutes using integer division and rounding up using *ceil()*. And the balance is updated by adding the cost of the call duration in minutes and the per minute cost of the prepaid contract which is a constant value given stored in the PREPAID_MINS_COST variable.

Cancel contract method:

cancel_contract(self) => float

This method allows the customer to cancel the contract whenever they wish to. This method takes in no values in its function call. This method returns a floating-point value which represents the amount/bill the customer has to pay once the contract is cancelled.

Precondition: A bill call has already been created and it is safe to assume that a bill has already been created for that month/year when this function is called. As in the class attribute bill is already in the right month/year.

Implementation:

Since the terms of the prepaid contract state that the balance will be forfeited if there is still an amount remaining for the customer else the balance is returned to the customer to pay off. This is done by checking if the value of the private attribute *balance* is a positive or negative value and accordingly return what is needed. If balance is negative, then the amount is forfeited by the customer and nothing is returned else the balance is returned as part of the customer bill cost using the *get_cost()* function within the *Bill* class. This method also resets the contract's start by assigning it a None value.

What have I changed?

According to the TA feedback I received I was able to see that the work I had written was write but was very messy and confusing to the reader. From the useful TA feedback I was able to reformat and redo my documentation in such a way that it made it a much easier and simpler read for the reader. The TA feedback often critiqued the fact that my explanation was very long and overdrawn to a point and so I was able to split it up into different, smaller and more understandable parts rather than one large single explanation of what was going on. The TA feedback was quite useful for me as it allowed me to really understand what was needed from the reader and where I was lacking in my documentation and suggested things to fix or touch upon the documentation resubmission.