Systems Development for
Computational Science

CS107/AC207 Fall 2022

Harvard John A. Paulson
School of Engineering
and Applied Sciences

F. Wermelinger
Office: Pierce 211

Homework 3
Git reset and revert, Regression, Object Oriented Programming

Issued:	September 27th, 2022
Due:	October 11th, 2022

Note this is a 2 week exercise. *Do not procrastinate the work.*

Problem 1: Homework Submission Requirements (*10 points*)

Requirements:

1. Complete the homework on the designated branch (*5 points*)

2. Create a pull request to merge the designated homework branch into your default branch, e. g., main or master (*2 points*)

3. Merge the open pull request of the *previous homework* into your default branch **after** you have received feedback from the teaching staff and regrade requests are resolved (*3 points*).

Deliverables: your solutions to the problems below must be submitted in a directory called submission.

See https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-hw.

- *5 points* if homework has been solved on the required branch.

- *2 points* if a pull request has been opened to merge this homework branch into the default branch. No points if the pull request has been merged prematurely (before receiving feedback and regrade requests have been resolved).

- *3 points* if the previous branch of the previous homework has been merged *after* homework feedback has been received and all regrade request are resolved.

Problem 2: Git Reset and Revert (*10 points*)

Deliverables:

1. P2a_reset.md
2. P2a_reset.png
3. P2b_revert.png
4. P2b_revert.md

In the previous homework you already experimented with git revert. Recall that commit objects in Git are *immutable* once they are written to the history. This can cause trouble when mistakes are discovered later as they can not just be fixed in that commit object, but a new commit must be added to the history that fixes the issue. The same is true if a commit in the history must be undone (recall git revert from an earlier homework).

The above *is true* if you have pushed your history to a remote where other developers also push and pull their code. If you have not pushed your changes yet, that is, *your current history is local on your laptop*, then you still can *rewrite your local history* without affecting other people. Commits itself are still immutable, rewriting history therefore means you create new commits *that replace the old commit(s)*. Commands like git reset and git rebase can rewrite history. Because of this power, they can create destruction when not careful. Since your name is associated with commits, you want to avoid causing havoc.

In this problem we practice these destructive commands in a dummy repository. This problem will follow the same directory structure suggested in the previous homework. You are free to implement your own choice. Be sure that the directories local and bare are not inside any Git repository. This problem will follow this structure:

```
classes/
|-- CS107
|   |-- git
|   |   |-- myrepo      <- your private Git repository
|   |   |-- sandbox     <- forked sandbox from previous homework
|   |   |-- hw2         <- from last homework
|   |   |-- hw3
|   |   |   |-- remote  <- remote (bare) repository for this problem
|   |   |   |-- devA    <- developer A local repository for this problem
|   |   |   \-- devB    <- developer B local repository for this problem
|   |   \-- main
|   \-- CS107_other_data
\-- CS205 (some other class in this directory)
```

All of this will be local on your laptop, no need to create a remote online. Here remote serves as our remote repository (this is a bare Git repository, see last homework), devA and devB are two local repositories. You can think of them as one being you working on your laptop and the other is another hypothetical developer potentially on another computer (for this simulation all repositories are on your laptop of course).

Hint: *It is recommended that you script the following tasks in a file for replay value.*

a) **5 points**

Following the introductions above, create three directories `remote`, `devA` and `devB` which are not inside a Git repository. Initialize the `remote` as a *bare* Git repository. Initialize `devA` as a regular (local) Git repository and set the `remote` repository as the `origin` remote. This will be the initial setup (as if you would have started a new project with one developer and another will clone the project shortly).

Now setup a history by performing the following steps inside the `devA` repository:

```
# inside devA repository
cat <<EOF >func.py
def A(a):
    return a

def B(b):
    return b
EOF
git add func.py
git commit -m 'Initial'
sed -i 's/def A/def function/' func.py
git commit -am 'Change function name (devA)'
sed -i 's/function/CommonAncestor/' func.py
git commit -am 'Change function name again (devA)'
git push -u origin main
```

Change into `devB`, clone the repository and apply some changes:

```
1   # inside devB repository
2   git clone ../remote .
3   sed -i 's/CommonAncestor/devB/' func.py
4   git commit -am 'Change function name (devB)'
```

(*Note that MacOSX users need to pass an empty string the to the* `-i` *option like this "*`sed -i`
`""` *instead of only "*`sed -i`*" as in the commands shown above. This difference is due to BSD Linux on which MacOSX is based on.*) Back to developer A. You *do not intend to push* these changes because it is lunch time and you want to review something first when you are back.

In the meantime, `devA` realized a mistake. Change back into `devA` and *reset* your previous commit. Stop here and spend a few minutes reading `git help reset`. Focus on the `--soft`, `--mixed` and `--hard` options. A *soft* reset will remove a commit object and leave the files that were in that commit *staged*, exactly as they were *before* you created the commit above. The default is a *mixed* reset which will remove the commit object and leave the changes in the *working tree* instead of the staging area. Existing changes in the staging area will be unstaged. Finally, a *hard* reset is the *most destructive* variant. It will remove the referenced commit object (of course all of the three variants will),

3

clear the staging area and set the working tree to the state of the referenced commit *overwriting unstaged changes. This means any changes you had staged already as well as modified files in the working tree will be lost without recovery options.*

You should be inside the `devA` repository. Developer A now decides to *reset* its last commit. Perform a *soft* reset to the parent of the current commit with:

```
# inside devA repository
git reset --soft HEAD^
```

(See `git help rev-parse` for the meaning of "ˆ" and "~".) Inspect the status using `git status`. Issue the command

```
git commit -m 'Changed my mind (devA)'
```

and perform the `git reset` above again, this time without the `--soft` option. Inspect the status once more using `git status`. *Make sure you see the difference.* Commit the changes *again*, this time using (note the difference of the options used here!)

```
git commit -am 'Changed my mind again (devA)'
```

One odd thing though is that when you create the same commit again, you get back a commit with the exact same content, but the SHA1 identifier *for this new commit is different than from the previous one.* Explain the reason for this observation in the file P2a_reset.md (you can verify this with `git log` for example).

Finally developer A is struck by lightning and decides to get rid of the last *two* commits entirely by

```
git reset --hard HEAD~2
```

Check the status of your repository once more, it is again different from the other two resets above. Developer A then adds a new flashing idea:

```
sed -i 's/def A/def devA/' func.py
git commit -am 'Change function name (reset by devA)'
```

Check the status using `git status`. Git tells you that your actions caused your local branch to *diverge* from the remote (actually, *the divergence started with the very first* `git reset`). If you try a `git push`, Git will reject your changes because of this divergence. Blindly ignore the warning of Git and forcefully push the changes to the remote

```
# inside devA repository
git push --force
```

In general, pushing a diverged history to a shared remote is a very bad idea and you must avoid it. (If it was only you working with this remote it would not matter.)

Now developer B is back from lunch. Recall that this developer had a local commit not pushed to the remote yet. Change to `devB` and try

```
# inside devB repository
git push origin main
```

Git tells you that the remote has changed. This is fine and the reason why usually a `git pull` is required before you can push in a shared project (since new commits are added arbitrarily). So we pull first

```
git pull --no-rebase # the option specifies a method to handle conflicts
```

The pull will cause a merge conflict on developer B's side. Open the file that is in conflict with an editor and take a screenshot. Save it in the file `P2a_reset.png`. Your file should look similar to

```
1  <<<<<<< HEAD
2  def devB(a):
3  ||||||| 0a3a1fa
4  def A(a):
5  =======
6  def devA(a):
7  >>>>>>> a7c9dfa2b8f399bc9b062a30f767e69b114318a6
8      return a
9
10 def B(b):
11     return b
```

except that your commit references will be different (see `P2a_reset.md` for an explanation why). Recall from last homework what the section between lines 3–5 means. Please refresh it if needed. Developer B is confused about line 4 since it does not correspond to her/his understanding what the common ancestor for this conflict should be. This developer has never changed the code "`def A(a):`". The only code changes by developer B have been made in line 3 on page 3. The forced reset broke the history and it is very difficult to connect the dots and resolve this conflict. For developer B, the common ancestor for this conflict should be code that involves "`CommonAncestor`" since this is where code has been changed by developer B. The current conflict looks like that *there is work missing*. Keep in mind that for this example, the modified code is small and the conflict resolution is not hard. In practice this will not be the case.
You can run this example with

```
$ ./P2.sh reset
```

using the script `solution/code/p2/P2.sh`.

- ***3 points*** for explanation why commits with same content differ. The reason is that commit objects also take into account the date and time when a commit has been created. It is therefore not possible that an earlier commit can be identical to a later commit, even if the content is identical. Commit objects are *unique* and *immutable*. Solution should include keywords like "time", "date" or "author" (may differ as well). Any meta data that is different for commit will affect the hash.

- **2 points** for the screenshot `P2a_reset.png`. Full points if the common ancestor is not visible in the image. By default Git does not enable the diff-3 style for merge conflict resolution.

b) **5 points**

Repeat the steps in task 2a until the first `git reset` (do not execute it). In this task you should use `git revert` to revert the last *two* commits in the repository devA. You want to revert the two commits all at once and create one new commit with the changes applied after developer A performed the reset in task 2a.[1] If you continue with the steps as in task 2a, developer B will run into a merge conflict. Open the conflicting file and take a screenshot. Save it as `P2b_revert.png`. The file should look similar to

```
1   <<<<<<< HEAD
2   def devB(a):
3   ||||||| cb3b34c
4   def CommonAncestor(a):
5   =======
6   def devA(a):
7   >>>>>>> 5b090c88cacd982de350398a865ccd38dd75f28b
8       return a
9
10  def B(b):
11      return b
```

This makes a lot more sense to developer B as she/he can now reason about this conflict. Both developers made changes *in the exact same place* and they have to consolidate what the name of the function should be.

When Git compares files for changes, it does this per *hunk*.[2] If the same hunk has been modified on either side and Git can not resolve the conflict automatically, a conflict occurs. Please open a new file called `P2b_revert.md` and answer the following questions with either "yes" or "no":

1. The merge conflict that still occurs above is because `git revert` has been used?
2. Assume developer B performs the following modification in line 3 on page 3 (instead of the one shown there):

```
sed -i 's/def B/def devB/' func.py
```

   (a) A merge conflict would occur if developer A used `git revert`?
   (b) A merge conflict would occur if developer A used `git reset`?

Optional: inspect the history with `git log` for these different cases after conflicts are resolved. Resolve conflicts by choosing any changes you like.

You can run this example with

---

[1]See `https://stackoverflow.com/questions/4991594/revert-a-range-of-commits-in-git` for some hints.

[2]See `https://stackoverflow.com/questions/37620729/in-the-context-of-git-and-diff-what-is-a-hunk`

```
$ ./P2.sh revert
```

using the script `solution/code/p2/P2.sh`.

You can change the variable `devB_change_same_hunk` to let developer B change a different hunk.

- **2 points** for the screenshot `P2b_revert.png`. Full points if the common ancestor is not visible in the image. By default Git does not enable the diff-3 style for merge conflict resolution.
- **1 points** for question 1 in `P2b_revert.md`. Answer is *no*.
- **1 points** for question 2 in `P2b_revert.md`. Answer is *no*.
- **1 points** for question 3 in `P2b_revert.md`. Answer is *yes*.

Optional: this is a good read to checkout after you worked through this problem: `https://www.atlassian.com/git/tutorials/undoing-changes/git-reset`.

Problem 3: Linear Regression in Python (*40 points*)

Deliverables:

1. `P3a.py`
2. `P3b.py`
3. `P3c.py`
4. `P3d.py`
5. `P3e.py`
6. `P3f.py`
7. `P3f.png`

So far, we have only been writing short Python scripts. However, when your code base starts to get bigger, you might want to organize your functions and class definitions. The idea behind *modules* is to split your function and class definitions into multiple, *logical units*. When you want to use a function or class you simply *import* it from the module. In essence, a module is a file containing Python definitions and statements.

In this problem, you will create a Python module with custom classes for two related types of linear regression:

1. Ordinary least squares (OLS) linear regression
2. Ridge regression

You are prohibited from using standard regression libraries in Python such as `sklearn`. The code must be your own based on the handout. However, you are permitted to check your answers against the standard libraries. You may use `numpy` to perform simple computations such as computing averages. **You are allowed** to use tools from `sklearn` to pre-process your data (for example scaling or centering).

Consider the multivariate linear model

$$y = X\beta + \beta_0, \tag{1}$$

where $y \in \mathbb{R}^n$ is a vector of length n, $X \in \mathbb{R}^{n \times p}$ is a $n \times p$ matrix, $\beta \in \mathbb{R}^p$ is a coefficient vector of length p and $\beta_0 \in \mathbb{R}$ is the intercept term. Note that for *centered* data $\beta_0 = 0$.

The goal is to find the coefficients β such that the linear model fits the data best. There are many approaches to this, but in this problem you will only consider the two mentioned above. OLS linear regression[3] seeks to minimize the following cost function

$$\mathcal{C} = |y - X\beta|^2. \tag{2}$$

The best fit coefficients are given by

$$\hat{\beta} = (X^\intercal X)^{-1} X^\intercal y, \tag{3}$$

---

[3]https://en.wikipedia.org/wiki/Ordinary_least_squares

where $X^\intercal$ is the transpose of the matrix X and X^{-1} is its inverse. Note that equation 3 is called the normal equation. The coefficients $\hat{\beta}$ minimize the cost function in equation 2 given the data.

Ridge regression[4] introduces an L_2 regularization so the new cost function is

$$C_\Gamma = |y - X\beta|^2 + |\Gamma\beta|^2, \tag{4}$$

where $\Gamma = \alpha I$ for some constant $\alpha \in [0, 1]$ and I is the identity matrix. The best fit coefficients for this case are given by

$$\hat{\beta} = \left(X^\intercal X + \Gamma^\intercal \Gamma\right)^{-1} X^\intercal y. \tag{5}$$

Note that it is important to *scale* the data before equation 5 is applied because the units in X and Γ are not the same.

You will use the R^2 statistic to assess the performance of the models. The R^2 score is defined as

$$R^2 = 1 - \frac{SS_E}{SS_T}, \tag{6}$$

where $SS_T = \sum_i (y_i - \bar{y})^2$ and $SS_E = \sum_i (y_i - \hat{y}_i)^2$. The y_i are the original data values, $\bar{y}$ is the mean of the original data values, and $\hat{y}_i$ are the values predicted by the model.

a) **6 points**

Start with the provided Python module code/p3/P3a.py. This file implements a base class called Regression with the following attributes:

__init__(): initializes an empty dictionary called params. Note that params should be an instance attribute.

fit(X, y): fits a linear model to X and y. It stores best-fit parameters in the dictionary attribute called params. The first key should be the coefficients (not including the intercept) and the second key should be the intercept.

get_params(): returns the params dictionary for a fitted model.

predict(X): predict new values with the fitted model given X.

score(X, y): returns the R^2 value of the fitted model.

set_params(): manually set parameters of the linear model. The method should accept variable keyword arguments (**kwargs) containing model parameters. In this problem, it will be used to set the regularization coefficient α in the ridge regression model.

You will find that some of these methods cannot be concretely defined for this base class. Such methods are intended to be fully implemented within the derived classes. You should raise a NotImplementedError exceptions in those cases. Write a quick test to ensure each of these methods throws a NotImplementedError when called. You do not need to submit this test. Use try/except blocks in your test code.[5] If a method is

---

[4]https://en.wikipedia.org/wiki/Tikhonov_regularization
[5]See https://docs.python.org/3/library/exceptions.html

shared in derived classes, you should implement it in the base class to avoid unnecessary *code bloat* due to duplicate implementations. It also reduces the possibility of bugs.

Please see `solution/code/p3/P3a.py` for the solution code.

b) ***5 points***

Import the `Regression` base class from task 3a in the file `P3b.py` *using an alias* and print a list of every attribute that can be accessed through an instance of this class. Use a built-in Python function to achieve this. Print the list to the standard output (your terminal). Ensure you only print the attributes for the `Regression` class and possibly its bases.

Please see `solution/code/p3/P3b.py` for the solution code.

```python
from P3a import Regression as Base

if __name__ == "__main__":
    r = Base()  # create instance
    print(dir(r))  # print attributes of instance (this includes `params`)
```

c) ***10 points***

Write a class called `LinearRegression` that implements the OLS regression model described in equation 3 and inherits from the `Regression` base class. Write your code in `P3c.py`.

Hint: *Note that general data may not be centered. The intercept will take care of this in the model. However, you must account for the intercept when you compute the fit. You can either center the data and compute the intercept after or you can introduce an artificial column of ones in the matrix X to increase the parameter space by one. See* `https://numpy.org/doc/ stable/reference/generated/numpy.append.html` *for example.*

Hint: *You will need to compute an inverse. Instead of using* `numpy.linalg.inv`, *a better choice would be to use a pseudo-inverse. See* `https://numpy.org/doc/stable/reference/ generated/numpy.linalg.pinv.html`.

Please see `solution/code/p3/P3c.py` for the solution code.

d) ***10 points***

Write a class called `RidgeRegression` that implements the ridge regression model described in equation 5 and inherits from the `LinearRegression` class. The parameter α should be passed with a constructor argument. Write your code in `P3d.py`.

Hint: *Since this model has a regularization term, it is important that the data is correctly scaled. If you choose not to center the data, you must be careful not to penalize the intercept. The best practice is to standardize the data.*

Hint: *You can use tools from the* `sklearn` *library to pre-process the data.*

Hint: *Consult any standard text for regression analysis for information about data normalization. For example Draper & Smith, Wiley 1998.*

Please see `solution/code/p3/P3d.py` for the solution code.

e) *3 points*

In this task you will evaluate the R^2 statistic for your two implemented models. You will use the California housing data set[6] for this. Split this data set into a training set and a test set. Please use an 80%–20% training-test split with a `random_state=42` using code similar to

```python
from sklearn.model_selection import train_test_split

dataset = # sklearn data set
X_train, X_test, y_train, y_test = train_test_split(
    dataset['data'], dataset['target'], test_size=0.2, random_state=42
)
```

Write code in the file `P3e.py` that imports your regression classes and instantiate a `LinearRegression` object and a `RidgeRegression` object with $\alpha = 0.1$. Compute the best fit coefficients using the appropriate data set and evaluate the R^2 statistic for each model, again using the appropriate data set, and print them to the standard output.

Hint: If you are not familiar with a train-test split of a data set, all this means is that you take your data and split it into two parts, a training set and test set. You perform the analysis with the training set in order to determine the parameters in your model. Then, you use that model to make a prediction with data from the test set.

Please see `solution/code/p3/P3e.py` for the solution code.

```
$ python P3e.py
R^2 = 0.5757877060269814 LinearRegression
R^2 = 0.5783920704401443 RidgeRegression
```

f) *6 points*

Compare the performance of the two models by varying values of α. Write a code to loop over at least 10 consecutive values for $\alpha \in [0,1]$, where the first and last values should be 0 and 1, respectively, and compute the R^2 statistic for each parameter set. Use the same data set and train/test split as in task 3e. You can use `set_params` to change the parameters for a model instance. Write your code in `P3f.py`.

Create one plot with the α values on the abscissa and the corresponding R^2 scores on the ordinate and plot a curve for each of the two models. Do not forget to label the axes and add a legend to distinguish the data in the plot. Save your plot as `P3f.png`.

Please see `solution/code/p3/P3f.py` for the solution code.

The model performance should look like the data shown in figure 1. Note that if $SS_E = 0$ in equation 6, such that $R^2 = 1$, means that the model predictions are identical to the true observation. Larger R^2 values indicate therefore a better model performance. The ridge regression model improves performance compared to ordinary least squares if $\alpha \approx 0.2$, but degrades performance if α is chosen too large. This characteristic applies

---

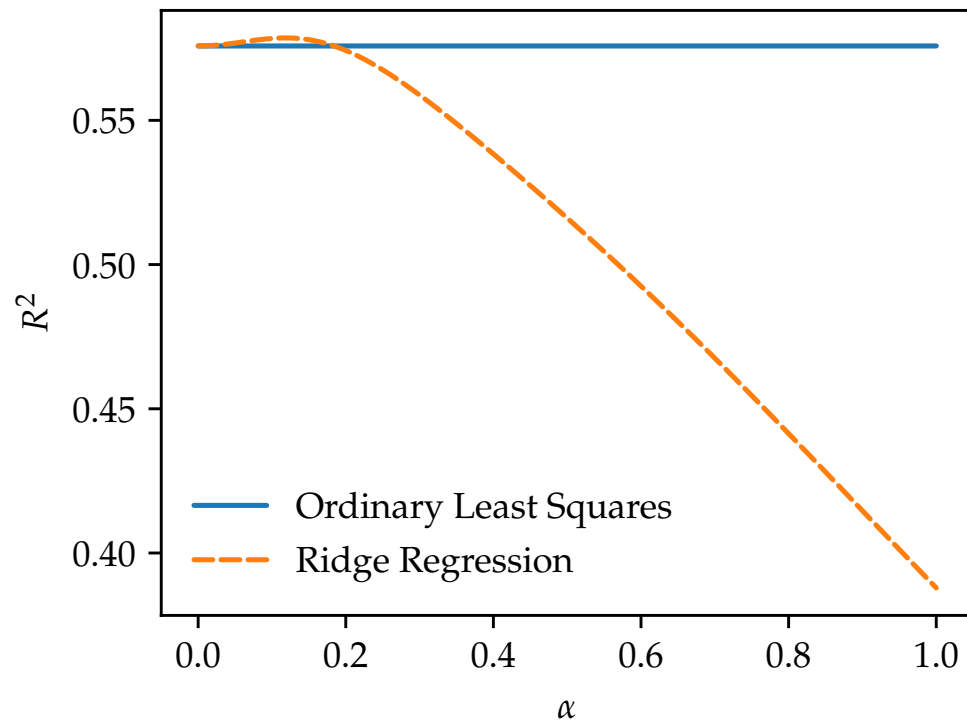[6]`https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_california_housing.html`

FIGURE 1: R^2 statistic of varying α for ordinary least squares and ridge regression models.

to ridge regression in general but also depends on the data. For $\alpha = 0$, ridge regression is identical to ordinary least squares.

Problem 4: Bank Account Revisited (*40 points*)

> Deliverables:
>
>   1. P4.py
>
> **Important:** this problem will be *auto-graded*. You are expected to work with the skeleton code provided in code/p4/P4.py. Do not change existing code in that skeleton code. Only add your solution code in appropriate places or update values in existing variables. Otherwise you will risk that the auto-grader fails and you lose points. Read carefully the problem statement and ensure your implementation follows the specifications given.
>
> The auto-grader works by running a set of tests on your implementation. It is therefore important that you are throwing exceptions at places in your code where values do not correspond to what is expected. If a method is supposed to return a number, then the return value should have a numeric type (e. g., float or int depending what the value represents) and not, for example, a string. The values 1 and '1' are not the same. You already saw how this mechanism works in the previous homework. Below you can find some more information.

In this problem you are going to revisit the bank account closure problem from the previous homework, only this time developing a formal class for a bank user and bank account to use in our closure. Recall that previously we just used a nonlocal variable amount that we changed.

a) Open the provided skeleton file code/p4/P4.py and study the function test(). You can run this Python file with

```
$ python P4.py
```

to examine the output of the code. The test() function tests some basic behavior related to the enumeration type

```python
class Account(Enum):
    '''Simple enumeration'''
    SAVINGS = 1
    CHECKING = 2
```

You can read the link provided in the docstring to learn a bit more about enumerations in Python. Be sure you understand the test code that has been implemented for you.

b) *10 points*

In this task you implement the BankAccount class in the file code/p4/P4.py. A bank account in this assignment has the following properties:

   1. An account type encoded by an enum
   2. An account balance encoded as a float

When a new bank account is created the balance is initialized to zero.

The account interface implements the following methods:

get_type(self): return the type of the account. This should be implemented using the enumeration types.

get_balance(self): return the current balance of the account.

withdraw(self, amount): withdraw amount from the current balance and return the balance after the transaction.

deposit(self, amount): deposit amount to the current balance and return the balance after the transaction.

__str__(self): returns an informative string that includes the type of the account and the current balance.

__init__(self, type): creates a new bank account of type. New bank accounts have a zero balance.

Two special scenarios must be taken care of:

1. The withdrawal amount cannot be larger than the current balance
2. The withdrawal or deposit amount can not be negative (zero is valid)

If any of these are violated, your implementation is expected to raise an exception with an appropriate description of what went wrong (see task 4a above).

Hint: You can add test code in the test() *function and check whether your implementation follows the specification above.*

Please see solution/code/p4/P4.py for the solution code.

c) *15 points*

In this task you implement the Customer class in the file code/p4/P4.py. A bank customer in this assignment has the following properties

1. A name encoded as a string
2. At most *two* accounts of type BankAccount
3. At most one savings account
4. At most one checking account

When a new bank customer is created only the name is initialized based on the constructor argument given. The other values must have a meaningful state.

The customer interface implements the following methods:

add_account(self, type): adds a new account of type for the customer. An exception with appropriate message must be raised if an account of the given type already exists.

get_balance(self, type): return the current balance for the account of given type. An exception with appropriate message must be raised if the account of the given type does not exist.

withdraw(self, type, amount): withdraw amount from the current balance for the account of type and return the balance after the transaction. An exception with appropriate message must be raised if the account of the given type does not exist.

deposit(self, type, amount): deposit amount to the current balance of account with type and return the balance after the transaction. An exception with appropriate message must be raised if the account of the given type does not exist.

__str__(self): returns an informative string that includes the name of the customer, the number of valid accounts and information *for each valid account* as specified in task 4b. *Ensure to reuse the code you have implemented in* BankAccount.

__len__(self): returns the number of valid accounts.

__init__(self, name): creates a new bank customer for name.

Hint: You can add test code in the test() *function and check whether your implementation follows the specification above.*

Please see solution/code/p4/P4.py for the solution code.

d) *15 points*

In this last part you are going to implement an interactive ATM session using a closure and the input() built-in function.[7] Implement the closure ATMSession(user) in the handout file code/p4/P4.py. The user argument is an instance of the Customer class implemented in task 4c. The ATMSession function must throw an exception if user is not an instance of Customer. The inner function returned by the closure will start the user session when called. The main screen should print the following options in this order:

```
>>> user = Customer('Test')
>>> session = ATMSession(user)
>>> session()
1) Exit
2) Create Account
3) Check Balance
4) Deposit
5) Withdraw
Enter Option:
```

If the user inputs "1" the session will exit. Any other option will lead to a second set of options, for example:

```
>>> session()
1) Exit
2) Create Account
3) Check Balance
4) Deposit
5) Withdraw
Enter Option: 2
1) Checking
```

---

[7]https://docs.python.org/3/library/functions.html#input

```
2) Savings
Choose Account:
```

Similarly, an example for a deposit:

```
>>> session()
1) Exit
2) Create Account
3) Check Balance
4) Deposit
5) Withdraw
Enter Option: 4
1) Checking
2) Savings
Choose Account: 1
Enter Deposit Amount:
```

For any sub-menu, checking is the first option and savings is the second option. After creating accounts, finishing a transaction or viewing the balance, the session should return to the main screen as shown above and await new input. If there is invalid user input, your implementation must raise exceptions according to the specifications in tasks 4b and 4c. When the user exits the session, the function session() above should return 0. You are free to design the appearance yourself but the *option correspondence* must follow the examples above. Any invalid option entered, e. g. "6" in the main screen, must raise an exception with appropriate error message.

Please see solution/code/p4/P4.png for the solution code.