

ASSIGNMENT -3 BANKING SYSTEM

Task 1: Conditional Statements

In a bank, you have been given the task is to create a program that checks if a customer is eligible for a loan based on their credit score and income. The eligibility criteria are as follows:

- Credit Score must be above 700.
- Annual Income must be at least \$50,000.

Tasks: 1. Write a program that takes the customer's credit score and annual income as input.

2. Use conditional statements (if-else) to determine if the customer is eligible for a loan.

3. Display an appropriate message based on eligibility

```
#1
def check_loan_eligibility(credit_score, annual_income):
    if credit_score > 700 and annual_income >= 50000:
        return True
    else:
        return False

#2
def main():
    credit_score = int(input("Enter your credit score: "))
    annual_income = float(input("Enter your annual income: $"))

    #3
    eligibility = check_loan_eligibility(credit_score, annual_income)

    if eligibility:
        print("Congratulations! You are eligible for a loan.")
    else:
        print("Sorry, you are not eligible for a loan at this time.")

#4
if __name__ == "__main__":
    main()
```

OUTPUT:-

```
(1) Enter your credit score: 720
(2) Enter your annual income: $50000
(3) Congratulations! You are eligible for a loan.
(4) Process finished with exit code 0
```

Activate MySandbox

Task 2: Nested Conditional Statement

Create a program that simulates an ATM transaction. Display options such as "Check Balance," "Withdraw," "Deposit,".

```
#1
def check_balance(balance):
    print(f"Your current balance: ${balance}")

#2
def withdraw(balance, amount):
    if amount > balance:
        print("Insufficient funds. Withdrawal failed.")
    elif amount % 100 != 0 or amount < 0:
        print("Invalid withdrawal amount. Please enter a multiple of 100 and greater than 0.")
    else:
        balance -= amount
        print(f"Withdrawal successful. Remaining balance: ${balance}")
    return balance

#3
def deposit(balance, amount):
    if amount <= 0:
        print("Invalid deposit amount. Please enter an amount greater than 0.")
    else:
        balance += amount
        print(f"Deposit successful. New balance: ${balance}")
    return balance
```

Ask the user to enter their current balance and the amount they want to withdraw or deposit. Implement checks to ensure that the withdrawal amount is not greater than the available balance and that the withdrawal amount is in multiples of 100 or 500.

```
*** 11     def main():
12
13         current_balance = float(input("Enter your current balance: $"))
14
15         print("ATM Options:")
16         print("1. Check Balance")
17         print("2. Withdraw")
18         print("3. Deposit")
19
20         # Get user choice
21         choice = int(input("Enter your choice (1, 2, or 3): "))
22
23         if choice == 1:
24             check_balance(current_balance)
25         elif choice == 2:
26             withdrawal_amount = float(input("Enter the amount to withdraw: $"))
27             current_balance = withdrawal(current_balance, withdrawal_amount)
28         elif choice == 3:
29             deposit_amount = float(input("Enter the amount to deposit: $"))
30             current_balance = deposit(current_balance, deposit_amount)
31         else:
32             print("Invalid choice. Please enter 1, 2, or 3.")
```

Display appropriate messages for success or failure

```
Enter your current balance: $2000
ATM Options:
 1. Check Balance
 2. Withdraw
 3. Deposit
Enter your choice (1, 2, or 3): 2
Enter the amount to withdraw: $1000
Withdrawal successful.. Remaining balance: $10000.0

Process finished with exit code 0
```

Task 3: Loop Structures

You are responsible for calculating compound interest on savings accounts. You need to calculate the future balance for each customer's savings account after a certain number of years.

- Tasks:
1. Create a program that calculates the future balance of a savings account.
 2. Use a loop structure (e.g., for loop) to calculate the balance for multiple customers
 - . 3. Prompt the user to enter the initial balance, annual interest rate, and the number of years.
 4. Calculate the future balance using the formula: $\text{future_balance} = \text{initial_balance} * (1 + \text{annual_interest_rate}/100)^{\text{years}}$.

```
*** 11     def calculate_future_balance(initial_balance, annual_interest_rate, years):
12
13         future_balance = initial_balance * (1 + annual_interest_rate / 100) ** years
14         return future_balance
15
16
17     #Main
18     def main():
19         num_customers = int(input("Enter the number of customers: "))
20
21         for customer in range(1, num_customers + 1):
22             print(f"\nCustomer {customer}:")
23
24             initial_balance = float(input("Enter the initial balance: $"))
25             annual_interest_rate = float(input("Enter the annual interest rate (%): "))
26             years = int(input("Enter the number of years: "))
27
28             future_balance = calculate_future_balance(initial_balance, annual_interest_rate, years)
29
30             print(f"\nFuture Balance for Customer {customer}: ${future_balance:.2f}")
```

5. Display the future balance for each customer.

```
Enter the number of customers: 3
Customer 1:
Enter the initial balance: $1000
Enter the annual interest rate (%) : 5
Enter the number of years: 3
Future balance for Customer 1: $1076.25

Customer 2:
Enter the initial balance: $2000
Enter the annual interest rate (%) : 3
Enter the number of years: 2
Future balance for Customer 2: $2059.56

Customer 3:
Enter the initial balance: $3000
Enter the annual interest rate (%) : 5
Enter the number of years: 1
Future balance for Customer 3: $3081.41

Process finished with exit code 0
```

Task 4: Looping, Array and Data Validation

You are tasked with creating a program that allows bank customers to check their account balances. The program should handle multiple customer accounts, and the customer should be able to enter their account number, balance to check the balance.

Tasks:

1. Create a Python program that simulates a bank with multiple customer accounts.

2. Use a loop (e.g., while loop) to repeatedly ask the user for their account number and balance until they enter a valid account number.

3. Validate the account number entered by the user.

```
def get_account_balance(accounts, account_number):
    if account_number in accounts:
        return accounts[account_number]
    else:
        return None

def main():
    bank_accounts = [
        "1234567": 1000.50,
        "4567891": 200.25,
        "5432101": 300.75
    ]
    while True:
        account_number = input("Enter your account number: ")
        account_balance = get_account_balance(bank_accounts, account_number)

        if account_balance is not None:
            print(f"Account Balance for Account {account_number}: ${account_balance:.2f}")
            break
        else:
            print("Invalid account number. Please try again.")

main()
```

4. If the account number is valid, display the account balance. If not, ask the user to try again

```
C:\Users\welcome_\Desktop\Assignment-python1\venv\scripts\python.exe C:\Users\welcome_\Desktop\Assignment-python\Assignment2-python\controlstructure.py
Enter your account number: 123457
Invalid account number. Please try again.
Enter your account number: 123456
Invalid account number. Please try again.
Enter your account number: 123456
Account balance for Account 123456: $3000.50

Process finished with exit code 0
```

Task 5: Password Validation

Write a program that prompts the user to create a password for their bank account. Implement if conditions to validate the password according to these rules:

- The password must be at least 8 characters long.
 - It must contain at least one uppercase letter.
 - It must contain at least one digit.

```
 80 # TASK_2
 81
 82 # usage
 83
 84 def is_valid_password(password):
 85     if len(password) < 8:
 86         return False
 87     if not (char.isupper() for char in password):
 88         return False
 89
 90     if not any(char.isdigit() for char in password):
 91         return False
 92
 93     return True
 94
 95
 96
 97
 98
 99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1188
1189
1190
1191
1192
1193
1194
1195
1196
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1288
1289
1290
1291
1292
1293
1294
1295
1296
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1388
1389
1390
1391
1392
1393
1394
1395
1396
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1488
1489
1490
1491
1492
1493
1494
1495
1496
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1588
1589
1590
1591
1592
1593
1594
1595
1596
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1688
1689
1690
1691
1692
1693
1694
1695
1696
1696
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1790
1791
1792
1793
1794
1795
1796
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1890
1891
1892
1893
1894
1895
1896
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1990
1991
1992
1993
1994
1995
1996
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2090
2091
2092
2093
2094
2095
2096
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2190
2191
2192
2193
2194
2195
2196
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2288
2289
2290
2291
2292
2293
2294
2295
2296
2296
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2388
2389
2390
2391
2392
2393
2394
2395
2396
2396
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2488
2489
2490
2491
2492
2493
2494
2495
2496
2496
2497
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2
```

- Display appropriate messages to indicate whether their password is valid or not.

```
Run controlstructure >

C:\Users\welcome_1\Desktop\Assignment-Python\view\scripts\python.exe C:\Users\welcome_1\Desktop\Assignment-Python\Assignment3-Python\controlstructure.py
Create your bank account password: Anshu12
Password is valid. Account created successfully!
Process finished with exit code 0
```

Task 6: Password Validation

Create a program that maintains a list of bank transactions (deposits and withdrawals) for a customer. Use a while loop to allow the user to keep adding transactions until they choose to exit.

```

130     def display_transaction_history(transaction_history):
131         print("Transaction history:")
132         for transaction in transaction_history:
133             print(transaction)
134
135     if __name__ == "__main__":
136         transaction_history = []
137         while True:
138             print("\nOptions:")
139             print("1. Add Deposit")
140             print("2. Add Withdrawal")
141             print("3. Exit")
142             choice = input("Enter your choice (1, 2, or 3): ")
143
144             if choice == "1":
145                 deposit_amount = float(input("Enter the deposit amount: "))
146                 transaction_history.append(f"Deposit: +${deposit_amount:.2f}")
147             elif choice == "2":
148                 withdrawal_amount = float(input("Enter the withdrawal amount: "))
149                 transaction_history.append(f"Withdrawal: -${withdrawal_amount:.2f}")
150             elif choice == "3":
151                 display_transaction_history(transaction_history)
152                 print("Exiting program. Thank you!")
153             else:
154                 break

```

Display the transaction history upon exit using looping statements.



```

Options:
1. Add Deposit
2. Add Withdrawal
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the deposit amount: $50000
Options:
1. Add Deposit
2. Add Withdrawal
3. Exit
Enter your choice (1, 2, or 3): 20000
Invalid choice. Please enter 1, 2, or 3.
Options:
1. Add Deposit
2. Add Withdrawal
3. Exit
Enter your choice (1, 2, or 3): 3
Transaction History:
Deposit: +$50000.00
Withdrewal: -$20000.00
Exiting program. Thank you!

```

Activate Windows
Go to [Get 35% off](#) to activate Windows.

OOPS, Collections and Exception Handling

Task 7: Class & Object

1. Create a 'Customer' class with the following confidential attributes:
 - Attributes
 - o Customer ID
 - o First Name
 - o Last Name
 - o Email Address
 - o Phone Number
 - o Address

```
 86     class Customer:
 87         def __init__(self, CustomerID, FirstName, LastName, Email, PhoneNumber, Address):
 88             self._CustomerID = CustomerID
 89             self._FirstName = FirstName
 90             self._LastName = LastName
 91             self._Email = Email
 92             self._PhoneNumber = PhoneNumber
 93             self._Address = Address
 94
 95             #usage
 96             #property
 97             @property
 98             def CustomerID(self):
 99                 return self._CustomerID
100
101             @CustomerID.setter
102             def CustomerID(self, new_CustomerID):
103                 if isinstance(new_CustomerID, str) and new_CustomerID:
104                     self._CustomerID = new_CustomerID
105                 else:
106                     raise ValueError("Customer ID must be a non-empty string.")
107
108             #usage @dynamic
109             #property
110             @property
111             def FirstName(self):
112                 return self._FirstName
113
114             @FirstName.setter
115             def FirstName(self, new_FirstName):
116                 if isinstance(new_FirstName, str) and new_FirstName:
117                     self._FirstName = new_FirstName
118                 else:
119                     raise ValueError("First Name must be a non-empty string.")
120
121             # Setter and getter methods for Last Name
122             #usage @dynamic
123             #property
124             @property
125             def LastName(self):
126                 return self._LastName
127
128             @LastName.setter
129             def LastName(self, new_LastName):
130                 if isinstance(new_LastName, str) and new_LastName:
131                     self._LastName = new_LastName
132                 else:
133                     raise ValueError("Last Name must be a non-empty string.")
134
135             #usage
136             #property
137             @property
138             def Email(self):
139                 return self._Email
```

Activate Windows

- Constructor and Methods

```
24     def FirstName(self, new_FirstName):
25         if isinstance(new_FirstName, str) and new_FirstName:
26             self._FirstName = new_FirstName
27         else:
28             raise ValueError("First Name must be a non-empty string.")
29
30         # Setter and getter methods for Last Name
31         #usage @dynamic
32         #property
33         @property
34         def LastName(self):
35             return self._LastName
36
37         @LastName.setter
38         def LastName(self, new_LastName):
39             if isinstance(new_LastName, str) and new_LastName:
40                 self._LastName = new_LastName
41             else:
42                 raise ValueError("Last Name must be a non-empty string.")
43
44         #usage
45         #property
46         @property
47         def Email(self):
48             return self._Email
```

- o Implement default constructors and overload the constructor with Customer attributes, generate getter and setter, (print all information of attribute) methods for the attributes.

```
    """  
    @property  
    def _PhoneNumber(self):  
        return self._PhoneNumber  
  
    @_PhoneNumber.setter  
    def _PhoneNumber(self, new_PhoneNumber):  
        if isinstance(new_PhoneNumber, str) and new_PhoneNumber.isdigit():  
            self._PhoneNumber = new_PhoneNumber  
        else:  
            raise ValueError('Invalid phone number format.')  
  
    @property  
    def _Address(self):  
        return self._Address  
  
    @_Address.setter  
    def _Address(self, new_Address):  
        if isinstance(new_Address, str) and new_Address:  
            self._Address = new_Address  
        else:  
            raise ValueError('Address must be a non-empty string.')  
        #print(f'Address updated to {self._Address}')
```

2. Create an 'Account' class with the following confidential attributes:

- Attributes

- Account Number

- Account Type (e.g., Savings, Current)

- Account Balance

```
    """  
    class Account:  
        def __init__(self, AccountNumber, AccountType, AccountBalance):  
            self._AccountNumber = AccountNumber  
            self._AccountType = AccountType  
            self._AccountBalance = AccountBalance  
  
        @property  
        def AccountNumber(self):  
            return self._AccountNumber  
  
        @AccountNumber.setter  
        def AccountNumber(self, new_AccountNumber):  
            if isinstance(new_AccountNumber, str) and new_AccountNumber:  
                self._AccountNumber = new_AccountNumber  
            else:  
                raise ValueError('Account Number must be a non-empty string.')  
        #print(f'Account Number updated to {self._AccountNumber}')  
  
        @property  
        def AccountType(self):  
            return self._AccountType  
  
        @AccountType.setter  
        def AccountType(self, new_AccountType):  
            if isinstance(new_AccountType, str) and new_AccountType:  
                self._AccountType = new_AccountType  
            else:  
                raise ValueError('Account Type must be a non-empty string.')  
  
        @property  
        def AccountBalance(self):  
            return self._AccountBalance  
  
        @AccountBalance.setter  
        def AccountBalance(self, new_AccountBalance):  
            if isinstance(new_AccountBalance, (int, float)) and new_AccountBalance >= 0:  
                self._AccountBalance = new_AccountBalance  
            else:  
                raise ValueError('Account Balance must be a non-negative number.')  
        #print(f'Account Balance updated to {self._AccountBalance}')
```

- Constructor and Methods

- Implement default constructors and overload the constructor with Account attributes,

- Generate getter and setter, (print all information of attribute) methods for the attributes.

```
    """  
    class Account:  
        def __init__(self, AccountNumber, AccountType, AccountBalance):  
            self._AccountNumber = AccountNumber  
            self._AccountType = AccountType  
            self._AccountBalance = AccountBalance  
  
        @property  
        def AccountNumber(self):  
            return self._AccountNumber  
  
        @AccountNumber.setter  
        def AccountNumber(self, new_AccountNumber):  
            if isinstance(new_AccountNumber, str) and new_AccountNumber:  
                self._AccountNumber = new_AccountNumber  
            else:  
                raise ValueError('Account Number must be a non-empty string.')  
  
        @property  
        def AccountType(self):  
            return self._AccountType  
  
        @AccountType.setter  
        def AccountType(self, new_AccountType):  
            if isinstance(new_AccountType, str) and new_AccountType:  
                self._AccountType = new_AccountType  
            else:  
                raise ValueError('Account Type must be a non-empty string.')  
  
        @property  
        def AccountBalance(self):  
            return self._AccountBalance  
  
        @AccountBalance.setter  
        def AccountBalance(self, new_AccountBalance):  
            if isinstance(new_AccountBalance, (int, float)) and new_AccountBalance >= 0:  
                self._AccountBalance = new_AccountBalance  
            else:  
                raise ValueError('Account Balance must be a non-negative number.')  
        #print(f'Account Balance updated to {self._AccountBalance}')
```

- Add methods to the 'Account' class to allow deposits and withdrawals.

- deposit(amount: float): Deposit the specified amount into the account.

- withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance

`. - calculate_interest():` method for calculating interest amount for the available balance.
interest rate is fixed to 4.5%

- Create a Bank class to represent the banking system. Perform the following operation in main method: o create object for account class by calling parameter constructor.

`o deposit(amount: float): Deposit the specified amount into the account.`

o withdraw(amount: float): Withdraw the specified amount from the account.

o calculateInterest(): Calculate and add interest to the account balance for savings accounts.



The screenshot shows the PyCharm IDE interface with the following details:

- Project Structure:** The left sidebar shows a project named "Assignment-python" with several files: `__init__.py`, `Bank.py`, `Customer.py`, and `Account.py`.
- Code Editor:** The main window displays Python code. The cursor is at the end of the line `def main():`. A tooltip from the code completion feature is visible, showing suggestions for the `Bank` class.
- Toolbars and Status Bar:** The top bar has tabs for "Assignment-python" and "Untitled". The status bar at the bottom right says "Activate Windows" and "Go to Settings to activate Windows".

OUTPUT:-

```
C:\Users\welcome_\Desktop\Assignment_Python\Assignment_Python\Assignment_Python\Assignment_2-pyCharm\main.py
Account created: Savings Account (AC001) with initial balance $1000.00
Deposited $100.00. New balance: $1100.00
Withdraw $200.00. New balance: $900.00
Interest calculated: $8.50
Process finished with exit code 0
```

Task 8: Inheritance and polymorphism

Overload the deposit and withdraw methods in Account class as mentioned below.

- deposit(amount: float): Deposit the specified amount into the account.
- withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
- deposit(amount: int): Deposit the specified amount into the account
- withdraw(amount: int): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance
- deposit(amount: double): Deposit the specified amount into the account.
- withdraw(amount: double): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance

```
class Account:  
    def __init__(self, amount):  
        self.account_balance = amount  
  
    def deposit(self, amount):  
        if isinstance(amount, (float, int)) and amount > 0:  
            self.account_balance += amount  
            print(f"Deposited ${amount:.2f}. New balance: ${self.account_balance:.2f}")  
        else:  
            raise ValueError("Deposit amount must be a positive number.")  
  
    def withdraw(self, amount):  
        if isinstance(amount, (float, int)) and amount > 0:  
            if amount <= self.account_balance:  
                self.account_balance -= amount  
                print(f"Withdraw ${amount:.2f}. New balance: ${self.account_balance:.2f}")  
            else:  
                print("Insufficient balance. Withdrawal failed.")  
        else:  
            raise ValueError("Withdrawal amount must be a positive number.")
```

Create Subclasses for Specific Account Types

- Create subclasses for specific account types (e.g., 'SavingsAccount', 'CurrentAccount') that inherit from the 'Account' class.

o SavingsAccount: A savings account that includes an additional attribute for interest rate. override the calculate_interest() from Account class method to calculate interest based on the balance and interest rate.

o CurrentAccount: A current account that includes an additional attribute overdraftLimit. A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).

```

class SavingsAccount(Account):
    def __init__(self, account_number, account_balance, interest_rate):
        super().__init__(account_number, account_balance)
        self.interest_rate = interest_rate

    def withdraw(self, amount):
        if amount > 0:
            available_balance = self.account_balance + self.overdraft_limit
            if amount <= available_balance:
                self.account_balance -= amount
                print(f"Withdraw $ {amount:.2f}. New balance: ${self.account_balance:.2f}")
            else:
                print("Withdrawal limit exceeded. Withdrawal failed.")
        else:
            raise ValueError("Withdrawal amount must be greater than 0.")


class CurrentAccount(Account):
    OVERDRAFT_LIMIT = 1000
    def __init__(self, account_number, account_balance):
        super().__init__(account_number, account_balance)

    def withdraw(self, amount):
        if amount > 0:
            available_balance = self.account_balance + self.overdraft_limit
            if amount <= available_balance:
                self.account_balance -= amount
                print(f"Withdraw $ {amount:.2f}. New balance: ${self.account_balance:.2f}")
            else:
                print("Withdrawal limit exceeded. Withdrawal failed.")
        else:
            raise ValueError("Withdrawal amount must be greater than 0.")

```

Create a Bank class to represent the banking system.

Perform the following operation in main method:

- Display menu for user to create object for account class by calling parameter constructor.
Menu should display options 'SavingsAccount' and 'CurrentAccount'. user can choose any one option to create account. use switch case for implementation.
- deposit(amount: float): Deposit the specified amount into the account.

```

Assignment python C:\Users\welcome\Desktop\Assignment
  |- Assignment1-py
  |- Assignment2-py
  |- Assignment3-py
  |   |- __init__.py
  |   |- Account.py
  |   |- Bank_Task.py
  |   |- Bank_Task1.py
  |   |- controlStructure.py
  |   |- Customer.py
  |- Coding2-CareerHub
  |- External Libraries
  |- Scratches and Doodles
  |- Terminal

```

```

from Account import Account, SavingsAccount, CurrentAccount

class Bank:
    def create_account(self):
        print("Select account type:")
        print("1. Savings Account")
        print("2. Current Account")
        choice = int(input("Enter your choice (1 or 2): "))
        account_number = input("Enter account number: ")
        initial_balance = float(input("Enter initial balance: "))

        if choice == 1:
            interest_rate = float(input("Enter interest rate for savings account: "))
            return SavingsAccount(account_number, initial_balance, interest_rate)
        elif choice == 2:
            return CurrentAccount(account_number, initial_balance)
        else:
            print("Invalid choice. Creating a generic account.")
            return Account(account_number, AccountType.GENERIC, initial_balance)

    def main(self):
        print("Customer Create Account First follow below steps")
        account = self.create_account()

```

- withdraw(amount: float): Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance. For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.

- `calculate_interest()`: Calculate and add interest to the account balance for savings accounts.

```
 10  print("Welcome to the Bank Management System!")
 11  print("1. Deposit")
 12  print("2. Withdraw")
 13  print("3. Calculate Interest (for Savings Account)")
 14  print("4. Exit")
 15
 16  choice = int(input("Enter your choice (1-4): "))
 17
 18  if choice == 1:
 19      amount = float(input("Enter deposit amount: "))
 20      account.deposit(amount)
 21
 22  elif choice == 2:
 23      amount = float(input("Enter withdrawal amount: "))
 24      account.withdraw(amount)
 25
 26  elif choice == 3:
 27      if isinstance(account, SavingsAccount):
 28          account.calculate_interest()
 29      else:
 30          print("Interest calculation not applicable for the current account.")
 31
 32  elif choice == 4:
 33      print("Exiting the program.")
 34      break
 35
 36  else:
 37      print("Invalid choice. Please choose a valid option.")
 38
 39
 40  bank = Bank()
 41  bank.main()
```

OUTPUTS:-

```
2. Withdraw
3. Calculate Interest (For Savings Account)
4. Exit

Enter your choice (1-4): 2
Enter withdrawal amount: 200
Withdraw $200.00. New balance: $1000.00

Select operation:
1. Deposit
2. Withdraw
3. Calculate Interest (For Savings Account)
4. Exit

Enter your choice (1-4): 3
Interest calculated and added: $50.00. New balance: $1050.00

Select operation:
1. Deposit
2. Withdraw
3. Calculate Interest (For Savings Account)
4. Exit

Enter your choice (1-4): 4
Exiting the program.
```

```
C:\Users\welcome\Desktop\Assignment\python_new\scripts\python.exe C:\Users\welcome\Desktop\Assignment\python\AssignmentC\pythonBank_Task3.py
Customer Create Account First follow below steps
Select account type:
 1. Savings Account
 2. Current Account
Enter your choice (1 or 2): 1
Enter account number: 1001
Enter initial balance: 1000
Enter interest rate for savings account: 3

Select operation:
 1. Deposit
 2. Withdraw
 3. Calculate Interest (For Savings Account)
 4. Exit
Enter your choice (1-4): 1
Enter deposit amount: 200
Deposited $200.00. New Balance: $1200.00

Select operation:
 1. Deposit
 2. Withdraw
 3. Calculate Interest (For Savings Account)
 4. Exit
```

Task 9: Abstraction

1. Create an abstract class BankAccount that represents a generic bank account. It should include the following attributes and methods:

- Attributes: o Account number. o Customer name. o Balance

```
from abc import ABC, abstractmethod
class BankAccount(ABC):
    def __init__(self, AccountNumber, CustomerName, Balance):
        self._AccountNumber = AccountNumber
        self._CustomerName = CustomerName
        self._Balance = Balance
```

- Constructors:

o Implement default constructors and overload the constructor with Account attributes, generate getter and setter, print all information of attribute methods for the attributes.

```
class BankAccount:
    @property
    def AccountNumber(self):
        return self._AccountNumber

    @AccountNumber.setter
    def AccountNumber(self, new_AccountNumber):
        if isinstance(new_AccountNumber, str) and new_AccountNumber:
            self._AccountNumber = new_AccountNumber
        else:
            raise ValueError("Account number must be a non-empty string.")

    @property
    def CustomerName(self):
        return self._CustomerName

    @CustomerName.setter
    def CustomerName(self, new_CustomerName):
        if isinstance(new_CustomerName, str) and new_CustomerName:
            self._CustomerName = new_CustomerName
        else:
            raise ValueError("Customer name must be a non-empty string.")

    @property
    def Balance(self):
        return self._Balance
```

- Abstract methods:

o deposit(amount: float): Deposit the specified amount into the account.

withdraw(amount: float): Withdraw the specified amount from the account (implement error handling for insufficient funds).

calculate_interest(): Abstract method for calculating interest.

```
class BankAccount:
    @abstractmethod
    def deposit(self, amount):
        pass

    @abstractmethod
    def withdraw(self, amount):
        pass

    @abstractmethod
    def calculate_interest(self):
        pass
```

Create two concrete classes that inherit from BankAccount:

- SavingsAccount: A savings account that includes an additional attribute for interest rate. Implement the calculate_interest() method to calculate interest based on the balance and interest rate

```
class SavingsAccount(BankAccount):  
    def __init__(self, account_number='', customer_name='', balance=0.0, interest_rate=0.0):  
        super().__init__(account_number, customer_name, balance)  
        self._interest_rate = interest_rate  
  
    usage  
    @property  
    def interest_rate(self):  
        return self._interest_rate  
  
    @interest_rate.setter  
    def interest_rate(self, new_interest_rate):  
        if isinstance(new_interest_rate, float) and 0 <= new_interest_rate <= 100:  
            self._interest_rate = new_interest_rate  
        else:  
            raise ValueError("Interest rate must be a number between 0 and 100.")  
  
    usage(B domain)  
    def deposit(self, amount):  
        if isinstance(amount, (float, int)) and amount > 0:  
            self._balance += amount  
            print(f"Deposited {amount:.2f}. New balance: ${self._balance:.2f}")  
        else:  
            raise ValueError("Deposit amount must be a positive number.")  
  
    usage(B domain)  
    def withdraw(self, amount):  
        if isinstance(amount, (float, int)) and amount > 0:  
            if amount < self._balance:  
                self._balance -= amount  
                print(f"Withdraw {amount:.2f}. New balance: ${self._balance:.2f}")  
            else:  
                print("Insufficient balance. Withdraw failed.")  
        else:  
            raise ValueError("Withdraw amount must be a positive number.")  
  
    usage(B domain)  
    def calculate_interest(self):  
        interest_amount = (self._interest_rate / 100) * self._balance  
        self._balance += interest_amount  
        print(f"Interest calculated and added. ${interest_amount:.2f}. New balance: ${self._balance:.2f}")
```

- CurrentAccount: A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).

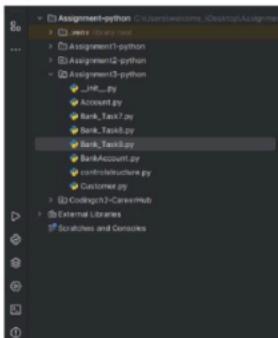
```
class CurrentAccount(BankAccount):  
    OVERDRAFT_LIMIT = 1000  
  
    def __init__(self, account_number='', customer_name='', balance=0.0):  
        super().__init__(account_number, customer_name, balance)  
  
    usage(B domain)  
    def deposit(self, amount):  
        if isinstance(amount, (float, int)) and amount > 0:  
            self._balance += amount  
            print(f"Deposited {amount:.2f}. New balance: ${self._balance:.2f}")  
        else:  
            raise ValueError("Deposit amount must be a positive number.")  
  
    usage(B domain)  
    def withdraw(self, amount):  
        if isinstance(amount, (float, int)) and amount > 0:  
            available_balance = self._balance + self.OVERDRAFT_LIMIT  
            if amount < available_balance:  
                self._balance -= amount  
                print(f"Withdraw {amount:.2f}. New balance: ${self._balance:.2f}")  
            else:  
                print("Withdraw limit exceeded. Withdraw failed.")  
        else:  
            raise ValueError("Withdraw amount must be a positive number.")
```

Activate Windows

Create a Bank class to represent the banking system.

Perform the following operation in main method

- Display menu for user to create object for account class by calling parameter constructor.
Menu should display options 'SavingsAccount' and 'CurrentAccount'. user can choose any one option to create account. use switch case for implementation. create_account should display sub menu to choose type of accounts. o Hint: Account acc = new SavingsAccount(); or Account acc = new CurrentAccount();



```
from BankAccount import SavingsAccount, CurrentAccount

class Bank:
    def __init__(self):
        self.accounts = []

    def create_account(self, account_type):
        while True:
            print("1. Create Savings Account")
            print("2. Create Current Account")
            print("3. Exit")

            choice = input("Enter your choice (1, 2, or 3): ")

            if choice == '1':
                account_type = "Savings"
            elif choice == '2':
                account_type = "Current"
            elif choice == '3':
                print("Exiting the program.")
                break
            else:
                print("Invalid choice. Please enter 1, 2, or 3.")

        account = self.create_account(account_type)
        self.perform_operations(account)

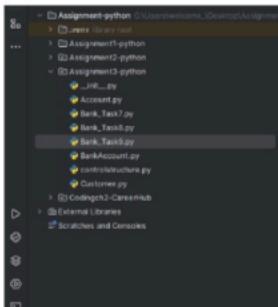
    def create_account(self, account_type):
        account_number = input("Enter account number: ")
        customer_name = input("Enter customer name: ")
        initial_balance = float(input("Enter initial balance: "))

        if account_type == "Savings":
            interest_rate = float(input("Enter interest rate for savings account: "))
            return SavingsAccount(account_number, customer_name, initial_balance, interest_rate)
        elif account_type == "Current":
            return CurrentAccount(account_number, customer_name, initial_balance)
        else:
            print("Invalid account type.")
            return None

    def perform_operations(self, account):
        while True:
            print("1. Deposit")
            print("2. Withdraw")
            print("3. Calculate Interest (for Savings Account)")
            print("4. Exit")

            choice = input("Enter your choice (1, 2, 3, or 4): ")
```

- deposit(amount: float): Deposit the specified amount into the account.
- withdraw(amount: float): Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance. For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.
- calculate_interest(): Calculate and add interest to the account balance for savings accounts



```
def create_account(self, account_type):
    account_number = input("Enter account number: ")
    customer_name = input("Enter customer name: ")
    initial_balance = float(input("Enter initial balance: "))

    if account_type == "Savings":
        interest_rate = float(input("Enter interest rate for savings account: "))
        return SavingsAccount(account_number, customer_name, initial_balance, interest_rate)
    elif account_type == "Current":
        return CurrentAccount(account_number, customer_name, initial_balance)
    else:
        print("Invalid account type.")
        return None

    def perform_operations(self, account):
        while True:
            print("1. Deposit")
            print("2. Withdraw")
            print("3. Calculate Interest (for Savings Account)")
            print("4. Exit")

            choice = input("Enter your choice (1, 2, 3, or 4): ")
```

```
Bank_Terminal
BankAccount.py
CurrentAccount.py
Customer.py
CustomerDB.py
> CodingWithOmkarHub
> (b) External Libraries
> Scratches and Consoles
D
E
S
C
X
I
O
I
Activate Windows
```

```
if choice == '1':
    amount = float(input("Enter deposit amount: "))
    account.deposit(amount)
elif choice == '2':
    amount = float(input("Enter withdrawal amount: "))
    account.withdraw(amount)
elif choice == '3' and isinstance(account, SavingsAccount):
    account.calculate_interest()
elif choice == '4':
    print("Exiting account operations.")
    break
else:
    print("Invalid choice. Please enter 1, 2, 3, or 4.")
    continue

bank = Bank()
bank.main()
```

OUTPUT FOR SAVING ACCOUNT

```
1. Create Savings Account
2. Create Current Account
3. Exit
Enter your choice (1, 2, or 3): 1
Enter account number: 101
Enter customer name: Aniket
Enter initial balance: 10000
Enter interest rate for savings account: 4

1. Deposit
2. Withdraw
3. Calculate Interest (For Savings Account)
4. Exit
Enter your choice (1, 2, 3, or 4): 3
Enter withdrawal amount: 5000
Withdraw $5000.00. New balance: $7000.00

1. Deposit
2. Withdraw
3. Calculate Interest (For Savings Account)
4. Exit
Enter your choice (1, 2, 3, or 4): 4
```

OUTPUT FOR CURRENT ACCOUNT

```
... Enter your choice (1, 2, 3, or 4): 4
Exiting account operations.
1. Create Savings Account
2. Create Current Account
3. Exit
Enter your choice (1, 2, or 3): 2
Enter account number: 202
Enter customer name: Rigni
Enter initial balance: 200000

1. Deposit
2. Withdraw
3. Calculate Interest (For Savings Account)
4. Exit
Enter your choice (1, 2, 3, or 4): 1
Enter deposit amount: 20000
Deposited $20000.00. New balance: $180000.00

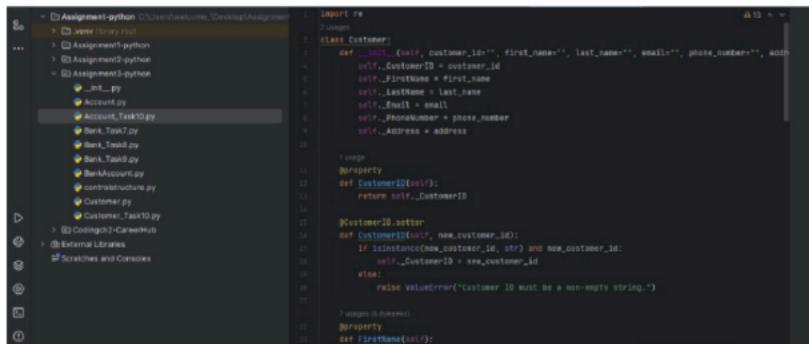
1. Deposit
2. Withdraw
3. Calculate Interest (For Savings Account)
4. Exit
Enter your choice (1, 2, 3, or 4): 4
Exiting account operations.
1. Create Savings Account
```

Activate Windows
Go to Settings to activate Windows.

Task 10: Has A Relation / Association

Create a 'Customer' class with the following attributes:

- Customer ID • First Name • Last Name • Email Address (validate with valid email address) • Phone Number (Validate 10-digit phone number) • Address

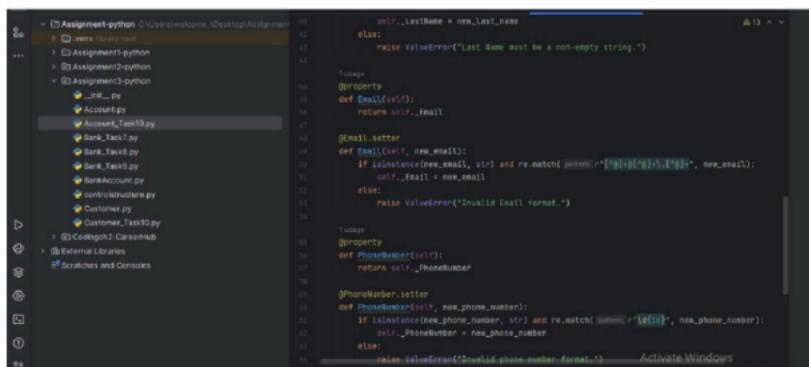


```
import re
class Customer:
    def __init__(self, customer_id='', first_name='', last_name='', email='', phone_number=''):
        self._customer_id = customer_id
        self._firstname = first_name
        self._lastname = last_name
        self._email = email
        self._phonenumber = phone_number
        self._address = address

    @property
    def CustomerID(self):
        return self._CustomerID
    @CustomerID.setter
    def CustomerID(self, new_customer_id):
        if isinstance(new_customer_id, str) and new_customer_id:
            self._CustomerID = new_customer_id
        else:
            raise ValueError("Customer ID must be a non-empty string.")

    @property
    def Firstname(self):
```

- Methods and Constructor: o Implement default constructors and overload the constructor with Account attributes, generate getter, setter, print all information of attribute) methods for the attributes



```
        self._lastname + new_last_name
    else:
        raise ValueError("Last Name must be a non-empty string.")

    @property
    def Email(self):
        return self._Email
    @Email.setter
    def Email(self, new_email):
        if isinstance(new_email, str) and re.match(r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,}$', new_email):
            self._Email = new_email
        else:
            raise ValueError("Invalid email format.")

    @property
    def Phonenumber(self):
        return self._phonenumber
    @Phonenumber.setter
    def Phonenumber(self, new_phone_number):
        if isinstance(new_phone_number, str) and re.match(r'^[0-9]{10}$', new_phone_number):
            self._phonenumber = new_phone_number
        else:
            raise ValueError("Invalid phone number format.")
```

Create an 'Account' class with the following attributes:

- Account Number (a unique identifier)
- Account Type (e.g., Savings, Current)
- Account Balance
- Customer (the customer who owns the account)

```
from Customer import Customer
class Account:
    def __init__(self, account_number='', account_type='', account_balance=0.0, customer=None):
        self._account_number = account_number
        self._account_type = account_type
        self._account_balance = account_balance
        self._customer = customer

    @property
    def accountNumber(self):
        return self._accountNumber

    @accountNumber.setter
    def accountNumber(self, new_account_number):
        if isinstance(new_account_number, str) and new_account_number:
            self._accountNumber = new_account_number
        else:
            raise ValueError("Account Number must be a non-empty string.")

    @property
    def accountType(self):
        return self._accountType

    @accountType.setter
    def accountType(self, new_account_type):
        pass
```

- Methods and Constructor:
 - o Implement default constructors and overload the constructor with Account attributes, generate getter, setter, (print all information of attribute) methods for the attributes.

```
else:
    raise ValueError("Account Type must be a non-empty string.")

    @property
    def accountBalance(self):
        return self._accountBalance

    @accountBalance.setter
    def accountBalance(self, new_account_balance):
        if isinstance(new_account_balance, (int, float)) and new_account_balance >= 0:
            self._accountBalance = new_account_balance
        else:
            raise ValueError("Account Balance must be a non-negative number.")

    @property
    def customer(self):
        return self._customer

    @customer.setter
    def customer(self, new_customer):
        if isinstance(new_customer, Customer):
            self._customer = new_customer
        else:
            raise ValueError("Customer must be a Customer object.")
```

Create a Bank Class and must have following requirements:

Create a Bank class to represent the banking system.

It should have the following methods:

- `create_account(Customer customer, long accNo, String accType, float balance)`: Create a new bank account for the given customer with the initial balance.
 - `get_account_balance(account_number: long)`: Retrieve the balance of an account given its account number. Should return the current balance of account.
 - `deposit(account_number: long, amount: float)`: Deposit the specified amount into the account. Should return the current balance of account.



The screenshot shows the PyCharm IDE interface. The left sidebar displays a project structure with several files under 'Assignment3-python'. A search bar at the top is populated with 'Bank_Task10.py'. Below the search bar, a list of search results is shown, with 'Bank_Task10.py' highlighted in blue. The main editor window on the right contains the code for 'Bank_Task10.py', which includes imports for 'Account' and 'Customer', and defines a class 'Bank' with methods for creating accounts and depositing money.

```
from Account_Task0 import Account
class Bank:
    def __init__(self):
        self.accounts = {}

    def create_account(self, customer, acc_type, balance):
        account = Account(customer, acc_type, balance)
        self.accounts[account.AccountNumber] = account
        return account

    def get_account_balance(self, account_number):
        if account_number in self.accounts:
            return self.accounts[account_number].Balance
        else:
            print("Account not found.")
            return None

    def deposit(self, account_number, amount):
        if account_number in self.accounts:
            self.accounts[account_number].deposit(amount)
        else:
            print("Account not found.")
            return None
```

- withdraw(account_number: long, amount: float): Withdraw the specified amount from the account. Should return the current balance of account.
 - transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another.
 - getAccountDetails(account_number: long): Should return the account and customer details.

```
> Assignment1-python
...> Assignment2-python
-> Assignment3-python
  _init_.py
  Account.py
  Account_Task03.py
  Bank_Task7.py
  Bank_Task8.py
  Bank_Task9.py
  BankAccount.py
  constraints.py
  Customer.py
  Customer_Task10.py
...
> Codeigniter-CourseHub
External Libraries
Scratches and Consoles

01 def withdraw(amount, account_number, amount):
02     if account_number in self.accounts:
03         self.accounts[account_number].withdraw(amount)
04     else:
05         print("Account not found.")
06         return None
07
08 usage(Dynamic)
09 def transfer(from_account_number, to_account_number, amount):
10     if from_account_number in self.accounts and to_account_number in self.accounts:
11         from_account = self.accounts[from_account_number]
12         to_account = self.accounts[to_account_number]
13         from_account.transfer_to(to_account, amount)
14     else:
15         print("One or both accounts not found.")
16
17 usage(Dynamic)
18 def get_account_details(account_number):
19     if account_number in self.accounts:
20         return self.accounts[account_number].get_account_details()
21     else:
22         print("Account not found.")
23         return None
```

3. Ensure that account numbers are automatically generated when an account is created, starting from 1001 and incrementing for each new account.



```

...
> Assignment1-python
> Assignment2-python
> Assignment3-python
  > __init__.py
  > Account.py
  > Account_Task10.py
  > Bank_Task7.py
  > Bank_Task10.py
  > Bank_Task11.py
  > Bank_Task12.py
  > BankAccount.py
  > controllers.py
  > Customer.py

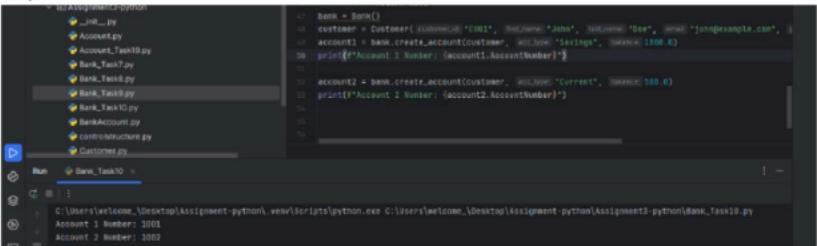
```

```

1 class Account:
2     ACCOUNT_NUMBER_COUNTER = 1001
3
4     def __init__(self, account_type='', account_balance=0.0, customer=None):
5         self._account_number = Account.generate_account_number()
6         Account.account_number_counter += 1
7         self._account_type = account_type
8         self._account_balance = account_balance
9         self._customer = customer

```

Output:-



```

> Assignment3/python
  > __init__.py
  > Account.py
  > Account_Task10.py
  > Bank_Task7.py
  > Bank_Task10.py
  > Bank_Task11.py
  > Bank_Task12.py
  > BankAccount.py
  > controllers.py
  > Customer.py

```

```

1 bank = Bank()
2 customer = Customer(firstname='Cobi', lastname='Jobs', nickname='Ste', email='jon@example.com', id=1)
3 account1 = bank.create_account(customer, account_type='Savings', balance=1000.0)
4 print(f'Account 1 Number: {account1.account_number}')
5
6 account2 = bank.create_account(customer, account_type='Current', balance=100.0)
7 print(f'Account 2 Number: {account2.account_number}')

```

```

C:\Users\welcome\Desktop\Assignment-Python\venv\scripts\python.exe C:\Users\welcome\Desktop\Assignment-Python\Assignment3-python\Bank_Task10.py
Account 1 Number: 3003
Account 2 Number: 3003

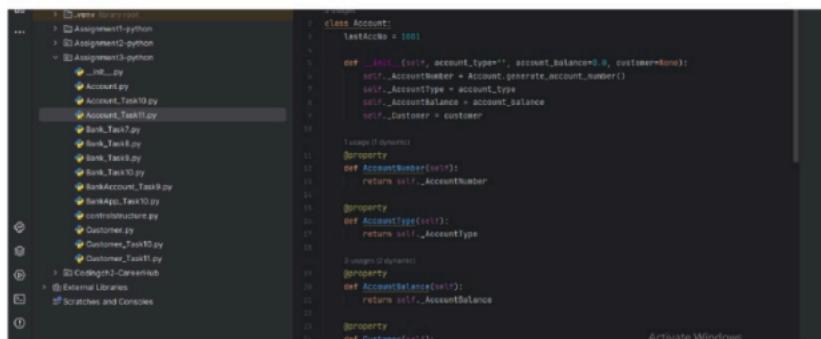
```

4. Create a BankApp class with a main method to simulate the banking system. Allow the user to interact with the system by entering commands such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails" and "exit." create_account should display sub menu to choose type of accounts and repeat this operation until user exit.

Task 11: Interface/abstract class, and Single Inheritance, static variable

1. Create a 'Customer' class as mentioned above task.

2. Create an class 'Account' that includes the following attributes. Generate account number using static variable.



```

...
> Assignment1-python
> Assignment2-python
> Assignment3-python
  > __init__.py
  > Account.py
  > Account_Task10.py
  > Bank_Task7.py
  > Bank_Task8.py
  > Bank_Task9.py
  > Bank_Task10.py
  > Bank_Task11.py
  > Bank_Task12.py
  > BankAccount_Task10.py
  > controllers.py
  > Customer.py
  > Customer_Task10.py
  > Customer_Task11.py
  > CodingHub2-CarrierHub
  > External Libraries
  > Scratches and Consoles

```

```

1 class Account:
2     LAST_ACCOUNT_NUM = 1001
3
4     def __init__(self, account_type='', account_balance=0.0, customer=None):
5         self._account_number = Account.generate_account_number()
6         self._account_type = account_type
7         self._account_balance = account_balance
8         self._customer = customer
9
10    @property
11        def AccountNumber(self):
12            return self._account_number
13
14    @property
15        def AccountType(self):
16            return self._account_type
17
18    @property
19        def AccountBalance(self):
20            return self._account_balance
21
22    @property
23        def Customer(self):
24            return self._customer

```

- Account Number (a unique identifier).
 - Account Type (e.g., Savings, Current)
 - Account Balance
 - Customer (the customer who owns the account)
 - lastAccNo

```
    def deposit(self, amount):
        if isinstance(amount, float) and amount > 0:
            self._AccountBalance += amount
            print(f"Deposited {amount}! New balance: ${self._AccountBalance:.2f}")
            return self._AccountBalance
        else:
            raise ValueError("Deposit amount must be a positive number.")

    def withdraw(self, amount):
        if amount > 0 and amount < self._AccountBalance:
            self._AccountBalance -= amount
            return self._AccountBalance
        else:
            print("Invalid withdrawal amount or insufficient funds.")
            return None

    def generateAccountNumber():
        Account.lastAcctNum += 1
        return Account.lastAcctNum
```

3. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:

- **SavingsAccount:** A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.

- **CurrentAccount**: A Current account that includes an additional attribute for `overdraftLimit(credit limit)`. `withdraw()` method to allow overdraft up to a certain limit. `withdraw` limit can exceed the available balance and should not exceed the overdraft limit.

```
Assignment python
  ↳ Assignment_1-python
  ↳ Assignment_2-python
  ↳ Assignment_3-python
    ↳ Jupyter
      ↳ Account.py
      ↳ Account_Task01.py
      ↳ Account_Task02.py
      ↳ Bank_Task7.py
      ↳ Bank_Task8.py
      ↳ Bank_Task9.py
      ↳ Bank_Task10.py
      ↳ BankAccount_Task01.py
      ↳ bankstructure.py
        ↳ Customer.py
        ↳ Customer_Task10.py
        ↳ Customer_Task11.py
  ↳ CodingH2-CareerLab
  ↳ Exercise Libraries
  ↳ Scratches and Consoles

10 class SavingsAccount(Account):
11     def __init__(self, account_balance=100.0, interest_rate=0.02, customer=None):
12         super().__init__(account_type="Savings", account_balance, customer)
13         self._interestRate = interest_rate
14
15     @property
16         def interestRate(self):
17             return self._interestRate
18
19     class CurrentAccount(Account):
20         def __init__(self, account_balance=0.0, overdraft_limit=1000.0, customer=None):
21             super().__init__(account_type="Current", account_balance, customer)
22             self._overdraftLimit = overdraft_limit
23
24         @property
25             def overdraftLimit(self):
26                 return self._overdraftLimit
27
28     def withdraw(self, amount):
29         total_withdrawn = amount + abs(min(0, self._accountBalance))
30         if total_withdrawn <= self._accountBalance + self._overdraftLimit:
31             self._accountBalance -= amount
32             print(f"Withdrawal [{amount:.2f}], New balance: [{self._accountBalance:.2f}]")
33             return self._accountBalance
34
35         print(f"Invalid withdrawal amount or exceeding overdraft limit.")
36         return None
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
59
60
61
62
63
64
65
66
67
68
69
69
70
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
99
100
101
102
103
104
105
106
107
108
109
109
110
111
112
113
114
115
116
117
118
119
119
120
121
122
123
124
125
126
127
128
129
129
130
131
132
133
134
135
136
137
138
139
139
140
141
142
143
144
145
146
147
148
149
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
188
189
189
190
191
192
193
194
195
196
197
198
199
199
200
201
202
203
204
205
206
207
208
209
209
210
211
212
213
214
215
216
217
218
219
219
220
221
222
223
224
225
226
227
228
229
229
230
231
232
233
234
235
236
237
238
239
239
240
241
242
243
244
245
246
247
248
249
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
```

- **ZeroBalanceAccount**: **ZeroBalanceAccount** can be created with **Zero** balance.

Create ICustomerServiceProvider interface/abstract class with following functions:

- `get_account_balance(account_number: long)`: Retrieve the balance of an account given its account number. Should return the current balance of account.
- `deposit(account_number: long, amount: float)`: Deposit the specified amount into the account. Should return the current balance of account.
- `withdraw(account_number: long, amount: float)`: Withdraw the specified amount from the account. Should return the current balance of account. A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
- `transfer(from_account_number: long, to_account_number: int, amount: float)`: Transfer money from one account to another.
- `getAccountDetails(account_number: long)`: Should return the account and customer details

```
from abc import ABC, abstractmethod

class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, account_number):
        pass

    @abstractmethod
    def deposit(self, account_number, amount):
        pass

    @abstractmethod
    def withdraw(self, account_number, amount):
        pass

    @abstractmethod
    def transfer(self, from_account_number, to_account_number, amount):
        pass

    @abstractmethod
    def get_account_details(self, account_number):
        pass
```

Create IBankServiceProvider interface/abstract class with following functions:

- `create_account(Customer customer, long accNo, String accType, float balance)`: Create a new bank account for the given customer with the initial balance.

```
from abc import ABC, abstractmethod

class IBankServiceProvider(ABC):
    @abstractmethod
    def create_account(self, customer, acc_type, balance):
        pass

    @abstractmethod
    def list_accounts(self):
        pass

    @abstractmethod
    def calculate_interest(self):
        pass
```

- `listAccounts():Account[] accounts`: List all accounts in the bank.
- `calculateInterest()`: the `calculate_interest()` method to calculate interest based on the balance and interest rate.

5. Create CustomerServiceProviderImpl class which implements ICustomerServiceProvider provide all implementation methods.

6.

```

 1 from ICustomerServiceProvider import ICustomerServiceProvider
 2 import ...
 3
 4 class CustomerServiceProviderImpl(ICustomerServiceProvider):
 5     def __init__(self):
 6         self.accounts = []
 7
 8     def get_account_balance(self, account_number):
 9         for account in self.accounts:
10             if account.AccountNumber == account_number:
11                 return account.AccountBalance
12         return None
13
14     def deposit(self, account_number, amount):
15         for account in self.accounts:
16             if account.AccountNumber == account_number:
17                 account.deposit(amount)
18         return None
19
20     def withdraw(self, account_number, amount):
21         for account in self.accounts:
22             if account.AccountNumber == account_number:
23                 account.withdraw(amount)
24         return None
25
26     def transfer(self, from_account_number, to_account_number, amount):
27         from_account = None
28         to_account = None
29
30         for account in self.accounts:
31             if account.AccountNumber == from_account_number:
32                 from_account = account
33             elif account.AccountNumber == to_account_number:
34                 to_account = account
35
36         if from_account and to_account:
37             from_account.transfer(to_account, amount)
38         else:
39             print('One or both accounts not found.')
40
41     def get_account_details(self, account_number):
42         for account in self.accounts:
43             if account.AccountNumber == account_number:
44                 return account.get_account_details()
45         return None

```

7.

```

 1 class BankServiceProviderImpl(BankService):
 2     def transfer(self, from_account_number, to_account_number, amount):
 3         from_account = None
 4         to_account = None
 5
 6         for account in self.accounts:
 7             if account.AccountNumber == from_account_number:
 8                 from_account = account
 9             elif account.AccountNumber == to_account_number:
10                 to_account = account
11
12         if from_account and to_account:
13             from_account.transfer(to_account, amount)
14         else:
15             print('One or both accounts not found.')
16
17     def get_account_details(self, account_number):
18         for account in self.accounts:
19             if account.AccountNumber == account_number:
20                 return account.get_account_details()
21         return None

```

7. Create BankServiceProviderImpl class which inherits from CustomerServiceProviderImpl and implements IBankServiceProvider

- Attributes o accountList: Array of Accounts to store any account objects. o branchName and branchAddress as String objects

8. Create BankApp class and perform following operation:

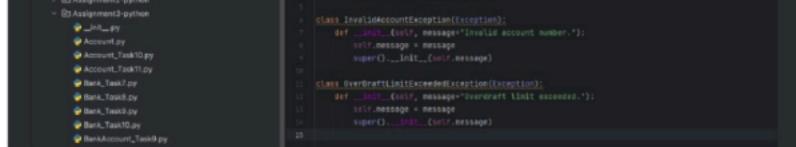
- main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts" and "exit."
- create_account should display sub menu to choose type of accounts and repeat this operation until user exit

Task 12: Exception Handling

throw the exception whenever needed and Handle in main method

1. InsufficientFundException throw this exception when user try to withdraw amount or transfer amount to another account and the account runs out of money in the account.
 2. InvalidAccountException throw this exception when user entered the invalid account number when tries to transfer amount, get account details classes.
 3. OverDraftLimitExceeded Exception throw this exception when current account customer try to withdraw amount from the current account.

```
 86 > 9  def __init__(self, message='Insufficient funds.'):
 87     10     self.message = message
 88
 89     11     super().__init__(self.message)
 90
```



Task 14: Database Connectivity.

1. Create a 'Customer' class as mentioned above task.
2. Create a class 'Account' that includes the following attributes.

Generate account number using static variable.

- Account Number (a unique identifier)
- Account Type (e.g., Savings, Current)
- Account Balance
- Customer (the customer who owns the account)
- lastAccNo

3. Create a class 'TRANSACTION' that include following attributes

- Account
- Description
- Date and Time
- TransactionType(Withdraw, Deposit, Transfer)
- TransactionAmount

```
from datetime import datetime

class Transaction:  
    def __init__(self, Account, Description, TransactionType, Amount):  
        self._Account = Account  
        self._Description = Description  
        self._dateTime = datetime.now()  
        self._transactionType = TransactionType  
        self._transactionAmount = Amount  
  
    @property  
    def Account(self):  
        return self._Account  
  
    @property  
    def Description(self):  
        return self._Description  
  
    @property  
    def dateTime(self):  
        return self._dateTime  
  
    @property  
    def TransactionType(self):  
        return self._transactionType  
  
    @property  
    def TransactionAmount(self):  
        return self._transactionAmount
```

4. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:

- SavingsAccount: A savings account that includes an additional attribute for interest rate. Savings account should be created with minimum balance 500.
- CurrentAccount: A Current account that includes an additional attribute for overdraftLimit(credit limit).
- ZeroBalanceAccount: ZeroBalanceAccount can be created with Zero balance.

```
return Account.lastAcNo
    ...
class SavingsAccount(Account):
    def __init__(self, account_type='savings', account_balance=100.0, customer=None, interest_rate=0.0):
        super().__init__(account_type, account_balance, customer)
        self._interest_rate = interest_rate

    @property
    def interest_rate(self):
        return self._interest_rate

    @interest_rate.setter
    def interest_rate(self, interest_rate):
        if isinstance(interest_rate, float) and interest_rate >= 0:
            self._interest_rate = interest_rate
        else:
            raise ValueError("Interest rate must be a non-negative number.")

class CurrentAccount(Account):
    def __init__(self, account_type='current', account_balance=0.0, customer=None, overdraft_limit=0.0):
        super().__init__(account_type, account_balance, customer)
        self._overdraft_limit = overdraft_limit

    @property
    def overdraft_limit(self):
        return self._overdraft_limit
...

```

5. Create ICustomerServiceProvider interface/abstract class with following functions:

- **get_account_balance(account_number: long):** Retrieve the balance of an account given its account number. should return the current balance of account.
 - **deposit(account_number: long, amount: float):** Deposit the specified amount into the account. Should return the current balance of account.
 - **withdraw(account_number: long, amount: float):** Withdraw the specified amount from the account. Should return the current balance of account.
- o A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
- o Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.
- **transfer(from_account_number: long, to_account_number: int, amount: float):** Transfer money from one account to another. both account number should be validate from the database use getAccountDetails method.
 - **getAccountDetails(account_number: long):** Should return the account and customer details.
 - **getTransactions(account_number: long, FromDate:Date, ToDate: Date):** Should return the list of transaction between two dates.

The screenshot shows the PyCharm IDE interface. The left sidebar displays a project structure with files like BankApp.py, BankApp_Task10.py, BankApp_Task11.py, BankApp_Task12.py, BankApp_Task13.py, BankServiceProvImpl_Task11.py, BankServiceProvImpl_Task14.py, BankServiceProvImpl_Task15.py, BankServiceProvImpl_Task16.py, BankServiceProvImpl_Task17.py, BankServiceProvImpl_Task18.py, BankServiceProvImpl_Task19.py, BankServiceProvImpl_Task20.py, BankServiceProvImpl_Task21.py, BankServiceProvImpl_Task22.py, BankServiceProvImpl_Task23.py, BankServiceProvImpl_Task24.py, DBUtil.py, Exception.py, HeadBank.py, HeadBankRepository.py, HeadBankServiceProv_Task11.py, HeadBankServiceProv_Task12.py, HeadBankServiceProv_Task13.py, HeadBankServiceProv_Task14.py, HeadBankServiceProv_Task15.py, HeadBankServiceProv_Task16.py, HeadBankServiceProv_Task17.py, HeadBankServiceProv_Task18.py, HeadBankServiceProv_Task19.py, HeadBankServiceProv_Task20.py, HeadBankServiceProv_Task21.py, HeadBankServiceProv_Task22.py, HeadBankServiceProv_Task23.py, HeadBankServiceProv_Task24.py, Transaction_Task14.py, CodingHub-DemoHub, External Libraries, and Strathmore and Cணை. The right panel shows the code editor with ICustomerServiceProvider_Task14.py selected. The status bar at the bottom indicates the file is saved, the encoding is UTF-8, and the Python version is 3.12 (Assignment-python).

```
ICustomerServiceProvider_Task14.py
```

```
from abc import ABC, abstractmethod
from datetime import datetime
from typing import List

__version__ = "1.0"

class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, account_number: int) -> float:
        pass

    @abstractmethod
    def deposit(self, account_number: int, amount: float) -> float:
        pass

    @abstractmethod
    def withdraw(self, account_number: int, amount: float) -> float:
        pass

    @abstractmethod
    def transfer(self, from_account_number: int, to_account_number: int, amount: float):
        pass

    @abstractmethod
    def get_account_details(self, account_number: int) -> dict:
        pass

    @abstractmethod
    def calculate_interest(self, account_number: int, interest_rate: float) -> float:
        pass
```

6. Create IBankServiceProvider interface/abstract class with following functions:

- `create_account(Customer customer, long accNo, String accType, float balance)`: Create a new bank account for the given customer with the initial balance.
- `listAccounts(): Array of BankAccount`: List all accounts in the bank.(`List[Account] accountsList`)
- `getAccountDetails(account_number: long)`: Should return the account and customer details.
- `calculateInterest()`: the `calculate_interest()` method to calculate interest based on the balance and interest rate.

The screenshot shows the PyCharm IDE interface. The left sidebar displays a project tree with files like BankApp, BankApp_Task10.py, BankApp_Task11.py, BankServiceProvImpl_Task11.py, BankServiceProvImpl_Task14.py, BankServiceProvider_Task11.py, BankServiceProvider_Task14.py, Customer.py, Customer_Task10.py, Customer_Task11.py, Customer_Task14.py, CustomerServiceProviderImpl_Task11.py, CustomerServiceProviderImpl_Task14.py, DBUtil.py, Exception.py, HeadBank.py, BankRepository.py, BankServiceProvider_Task11.py, and BankServiceProvider_Task14.py. The right panel shows the code for BankServiceProvider_Task14.py, which defines a class BankServiceProvider that implements the ICustomerServiceProvider interface. The code includes methods for creating accounts, listing accounts, getting account details, and calculating interest. A tooltip 'BankServiceProvider : getAccountDetails' is visible near the get_account_details method.

```
1 from abc import ABC, abstractmethod
2 from typing import List
3
4 class ICustomerServiceProvider(ABC):
5     @abstractmethod
6         def create_account(self, customer: Customer, acc_no: int, acc_type: str, balance: float):
7             pass
8
9     @abstractmethod
10        def list_accounts(self) -> List[Accounts]:
11            pass
12
13    @abstractmethod
14        def get_account_details(self, account_number: int):
15            pass
16
17    @abstractmethod
18        def calculate_interest(self):
19            pass
```

7. Create CustomerServiceProviderImpl class which implements ICustomerServiceProvider provide all implementation methods.

These methods do not interact with database directly.

```

class CustomerServiceProviderImpl(IGenericServiceProvider):
    def __init__(self):
        self.accounts_list = []

    def get_account_balance(self, account_number: int):
        for account in self.accounts_list:
            if account.AccountNumber == account_number:
                return account.AccountBalance
        return None

    def deposit(self, account_number: int, amount: float):
        for account in self.accounts_list:
            if account.AccountNumber == account_number:
                account.deposit(amount)
        return None

    def withdraw(self, account_number: int, amount: float):
        for account in self.accounts_list:
            if account.AccountNumber == account_number:
                account.withdraw(amount)
        return None

    def transfer(self, from_account_number: int, to_account_number: int, amount: float):
        from_account = None
        to_account = None

        for account in self.accounts_list:
            if account.AccountNumber == from_account_number:
                from_account = account
            elif account.AccountNumber == to_account_number:
                to_account = account

        if from_account and to_account:
            from_account.transfer(to_account, amount)
        else:
            raise InvalidAccountException("One or both accounts not found.")

    def get_account_details(self, account_number: int):
        for account in self.accounts_list:
            if account.AccountNumber == account_number:
                return account.get_account_details()
        return None

    def get_transactions(self, account_number: int, from_date: datetime, to_date: datetime):
        pass

```

8. Create BankServiceProviderImpl class which inherits from CustomerServiceProviderImpl and implements IBankServiceProvider.

- Attributes o accountList: List of Accounts to store any account objects. o transactionList: List of Transaction to store transaction objects. o branchName and branchAddress as String objects

```
BankServiceProvider_Task14.py
class BankServiceProviderApi(CustomerServiceProviderImpl, IBankServiceProvider):
    def __init__(self, branch_name: str, branch_address: str):
        super().__init__()
        self.account_list = []
        self.transaction_list = []
        self.branch_name = branch_name
        self.branch_address = branch_address

    def create_account(self, customer: Customer, acc_type: str, balance: float) -> Account:
        account = Account(acc_type, balance, customer)
        self.account_list.append(account)
        self.accounts_list.append(account) # Add to the parent class list as well.
        return account

    def list_accounts(self) -> List[Account]:
        return self.account_list

    def get_account_details(self, account_number: int):
        for account in self.account_list:
            if account.AccountNumber == account_number:
                return account.get_account_details()
        return None

    def calculate_interest(self):
        pass

    def deposit(self, account_number: int, amount: float):
        account = self.get_account_details(account_number)
        if account:
            account.Balance += amount
            self.transaction_list.append(f"Deposited {amount} into account {account_number} by {account.Customer.Name}")
        else:
            print("Account not found")

    def withdraw(self, account_number: int, amount: float):
        account = self.get_account_details(account_number)
        if account:
            if account.Balance > amount:
                account.Balance -= amount
                self.transaction_list.append(f"Withdrew {amount} from account {account_number} by {account.Customer.Name}")
            else:
                print("Insufficient balance")
        else:
            print("Account not found")
```

9. Create IBankRepository interface/abstract class which include following methods to interact with database.

- **createAccount(customer: Customer, accNo: long, accType: String, balance: float):** Create a new bank account for the given customer with the initial balance and store in database.
- **listAccounts():** List accountsList: List all accounts in the bank from database.
- **calculateInterest():** the calculate_interest() method to calculate interest based on the balance and interest rate.
- **getAccountBalance(account_number: long):** Retrieve the balance of an account given its account number. should return the current balance of account from database.
- **deposit(account_number: long, amount: float):** Deposit the specified amount into the account. Should update new balance in database and return the new balance.
- **withdraw(account_number: long, amount: float):** Withdraw amount should check the balance from account in database and new balance should updated in Database. o A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
- o Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.

- transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another. check the balance from account in database and new balance should updated in Database.
- getAccountDetails(account_number: long): Should return the account and customer details from database.
- getTransactions(account_number: long, FromDate: Date, ToDate: Date): Should return the list of transaction between two dates from database.

```

from typing import List
from customer.Task1 import Customer
from Account_Task1 import Account
from DBUtil import DBUtil
from calculateInterest import calculate_interest

class IBankRepository:
    def create_account(self, customer: Customer, acc_no: int, acc_type: str, balance: float) -> None:
        pass

    def list_accounts(self) -> List[Account]:
        pass

    def calculate_interest(self):
        pass

    def get_account_balance(self, account_number: int) -> float:
        pass

    def deposit(self, account_number: int, amount: float) -> float:
        pass

    def withdraw(self, account_number: int, amount: float) -> float:
        pass

    def transfer(self, from_account_number: int, to_account_number: int, amount: float):
        pass

```

10. Create BankRepositoryImpl class which implement the IBankRepository interface/abstract class and provide implementation of all methods and perform the database operations

```
BankRepositoryImpl.py
```

```
class BankRepositoryImpl(BankRepository):
    def __init__(self):
        self.accounts = {}
        self.transactions = []

    def create_account(self, customer: Customer, acc_no: int, acc_type: str, balance: float):
        account = Account(acc_no, acc_type, balance, customer)
        self.accounts[acc_no] = account
        return account

    def list_accounts(self) -> List[Account]:
        return list(self.accounts.values())

    def calculate_interest(self):
        pass

    def get_account_balance(self, account_number: int) -> float:
        account = self.accounts.get(account_number)
        if account:
            return account.balance
        else:
            raise InvalidAccountException("Account not found.")

    def deposit(self, account_number: int, amount: float) -> float:
        account = self.accounts.get(account_number)
        if account:
            account.balance += amount
            self.transactions.append(Transaction(account, description="Deposit", datetime.now()))
            return account.balance
        else:
            raise InvalidAccountException("Account not found.")

    def withdraw(self, account_number: int, amount: float) -> float:
        account = self.accounts.get(account_number)
        if account:
            if account.balance >= amount:
                account.balance -= amount
                self.transactions.append(Transaction(account, description="Withdraw", datetime.now()))
                return account.balance
            else:
                raise InsufficientFundsException("Insufficient Funds!")
        else:
            raise InvalidAccountException("Account not found.")

    def transfer(self, from_account_number: int, to_account_number: int, amount: float):
        self.deposit(to_account_number, amount)
        self.withdraw(from_account_number, amount)
```

. 11. Create DBUtil class and add the following method. • static getDBConn():Connection

Establish a connection to the database and return Connection reference

The screenshot shows the PyCharm IDE interface with the following details:

- Project:** Assignment-python
- File:** DBUtil.py
- Code Content:**

```
import mysql.connector
from abc import ABC, abstractmethod

class DBUtil(ABC):
    @abstractmethod
    def getDbConfig():
        pass

    def connect(self):
        db_config = {
            'host': 'localhost',
            'user': 'root',
            'password': 'Aniket@123',
            'database': 'testbase',
            'port': 3306,
        }

        connection = mysql.connector.connect(**db_config)

        if connection.is_connected():
            print("Connected to MySQL database")
            return connection
        except Exception as e:
            print(f"Error: {e}")
            return None
```

- Code Completion:** A tooltip for 'getDBConfig' is displayed, showing the method signature and the implementation.
- Toolbars and Status Bar:** Standard PyCharm toolbars and status bar at the bottom.

12. Create BankApp class and perform following operation: • main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit",

```
Project ->
  BankTask9.py
  BankTask10.py
  BankAccount_Task8.py
  BankApp.py
  BankApp_Task9.py
  BankApp_Task10.py
  BankApp_Task11.py
  BankRepositoryImpl.py
  BankServiceProviderImpl_Task11.py
  BankServiceProviderImpl_Task14.py
  controller.py
  Customer.py
  Customer_Task8.py
  Customer_Task10.py
  Customer_Task14.py
  CustomerServiceProviderImpl_Task11.py
  CustomerServiceProviderImpl_Task14.py
  DBUtil.py
  Exception.py
  FileUtil.py
  BankApp.py
  BankRepository.py
  BankRepositoryImpl.py
  DBUtil.py
  BankApp.py

def create_account(self):
    while True:
        print("1)Choose Account Type:")
        print("1. Savings Account")
        print("2. Current Account")
        print("3. Zero Balance Account")
        print("4. Back to Main Menu")

        acc_type_choice = input("Enter your choice (1-4): ")

        if acc_type_choice == "1":
            self.create_savings_account()
        elif acc_type_choice == "2":
            self.create_current_account()
        elif acc_type_choice == "3":
            self.create_zero_balance_account()
        elif acc_type_choice == "4":
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 4.")

    def create_savings_account(self):
        customer = self.get_customer_details()
        acc_id = self.generate_account_number()

        print("Customer Name: ", customer['name'])
        print("Customer Address: ", customer['address'])

        print("Saving Account Details")
        print("Account Number: ", acc_id)
        print("Balance: ", 0)
```

"withdraw", "get_balance", "transfer"

Assignment-python > Version control > BankApp > main.py

```
http://Task14.py BlankRepository.py BlankRepositoryImpl.py DBUtil.py BankApp.py
```

Project >

- Bank_Task9.py
- Bank_Task10.py
- BankAccount_Task9.py
- BankApp.py**
- BankApp_Task10.py
- BankApp_Task11.py
- BankRepositoryImpl.py
- BankServiceProviderImpl_Task11.py
- BankServiceProviderImpl_Task14.py
- Customer.py
- Customer_Task10.py
- Customer_Task11.py
- Customer_Task14.py
- CustomerServiceProviderImpl_Task11.py
- CustomerServiceProviderImpl_Task14.py
- DRUtil.py
- Exception.py
- HttpBank.py
- BlankRepository.py
- BankServiceProvider_Task11.py
- BankServiceProvider_Task14.py
- CustomerServiceProvider_Task11.py
- CustomerServiceProvider_Task14.py
- Transaction_Task14.py

Usage:

```
def create_savings_account(self):  
    customer = self.get_customer_details()  
    acc_no = self.generate_account_number()  
    acc_type = "Savings"  
    balance = float(input("Enter Initial Balance: "))  
    self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)  
    print("Savings Account created successfully. Account Number: (acc_no)")
```

Usage:

```
def create_current_account(self):  
    customer = self.get_customer_details()  
    acc_no = self.generate_account_number()  
    acc_type = "Current"  
    balance = float(input("Enter Initial Balance: "))  
    self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)  
    print("Current Account created successfully. Account Number: (acc_no)")
```

Usage:

```
def create_zero_balance_account(self):  
    customer = self.get_customer_details()  
    acc_no = self.generate_account_number()  
    acc_type = "Zero Balance"  
    balance = 0.0  
    self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)  
    print("Zero Balance Account created successfully. Account Number: (acc_no)")
```

def deposit(self):

Activate Windows
Go to Settings to activate Windows.

12:20 CRLF UTF-8 4 spaces Python 3.12 (Assignment-python) 1725 28-10-2023

, "getAccountDetails", "ListAccounts",

Assignment-python > Version control > BankApp > main.py

```
http://Task14.py BlankRepository.py BlankRepositoryImpl.py DBUtil.py BankApp.py
```

Project >

- Bank_Task9.py
- Bank_Task10.py
- BankAccount_Task9.py
- BankApp.py**
- BankApp_Task10.py
- BankApp_Task11.py
- BankRepositoryImpl.py
- BankServiceProviderImpl_Task11.py
- BankServiceProviderImpl_Task14.py
- Customer.py
- Customer_Task10.py
- Customer_Task11.py
- Customer_Task14.py
- CustomerServiceProviderImpl_Task11.py
- CustomerServiceProviderImpl_Task14.py
- DRUtil.py
- Exception.py
- HttpBank.py
- BlankRepository.py
- BankServiceProvider_Task11.py
- BankServiceProvider_Task14.py
- CustomerServiceProvider_Task11.py
- CustomerServiceProvider_Task14.py
- Transaction_Task14.py

Usage:

```
def withdraw(self):  
    account_number = int(input("Enter account number: "))  
    try:  
        balance = self.bank_service_provider.get_account_balance(account_number)  
        print(f"Current Balance: {balance}")  
        amount = float(input("Enter Withdrawal Amount: "))  
        except InvalidAmountException as e:  
            print(f"Error: {e}")
```

Usage:

```
def get_balance(self):  
    account_number = int(input("Enter account number: "))  
    try:  
        balance = self.bank_service_provider.get_account_balance(account_number)  
        print(f"Current Balance: {balance}")  
    except InvalidAccountException as e:  
        print(f"Error: {e}")
```

Usage:

```
def transfer(self):  
    from_account_number = int(input("Enter From Account Number: "))  
    to_account_number = int(input("Enter To Account Number: "))  
    amount = float(input("Enter Transfer Amount: "))  
    try:  
        self.bank_service_provider.transfer(from_account_number, to_account_number, amount)  
        print("Transfer successful.")  
    except (InvalidAccountException, InsufficientBalanceException) as e:  
        print(f"Error: {e}")
```

Usage:

```
def get_account_details(self):  
    account_number = int(input("Enter Account Number: "))  
    try:  
        ...
```

Activate Windows
Go to Settings to activate Windows.

12:20 CRLF UTF-8 4 spaces Python 3.12 (Assignment-python) 1725 28-10-2023

"getTransactions" and "exit."

```
BankApp.py
136     def list_accounts(self):
137         accounts = self.bank_service_provider.list_accounts()
138         if accounts:
139             print("\nList of Accounts:")
140             for account in accounts:
141                 print(account)
142         else:
143             print("No accounts found.")
144
145     def get_transactions(self):
146         account_number = int(input("Enter Account Number: "))
147         from_date = datetime.strptime(input("Enter From Date (YYYY-MM-DD): "), "%Y-%m-%d")
148         to_date = datetime.strptime(input("Enter To Date (YYYY-MM-DD): "), "%Y-%m-%d")
149         try:
150             transactions = self.bank_service_provider.get_transactions(account_number, from_date,
151             to_date)
152             if transactions:
153                 print("List of Transactions:")
154                 for transaction in transactions:
155                     print(transaction)
156             else:
157                 print("No transactions found.")
158         except InvalidAccountException as e:
159             print(f"Error: {e}")
160
161     def generate_account_number(self):
162         print("Generating Account Number...")
163         return "1234567890"
164
165     def get_customer_details(self):
166
167         customer_id = input("Enter Customer ID: ")
168         first_name = input("Enter First Name: ")
169         last_name = input("Enter Last Name: ")
170         email = input("Enter Email Address: ")
171         phone_number = input("Enter Phone Number: ")
172         address = input("Enter Address: ")
173
174         return Customer(customer_id, first_name, last_name, email, phone_number, address)
```

```
BankApp1.py
123     def main():
124         bank_app = BankApp()
125         bank_app.main()
```

- create_account should display sub menu to choose type of accounts and repeat this operation until user exit.

```
BankApp.py
136     def list_accounts(self):
137         accounts = self.bank_service_provider.list_accounts()
138         if accounts:
139             print("\nList of Accounts:")
140             for account in accounts:
141                 print(account)
142         else:
143             print("No accounts found.")
144
145     def get_transactions(self):
146         account_number = int(input("Enter Account Number: "))
147         from_date = datetime.strptime(input("Enter From Date (YYYY-MM-DD): "), "%Y-%m-%d")
148         to_date = datetime.strptime(input("Enter To Date (YYYY-MM-DD): "), "%Y-%m-%d")
149         try:
150             transactions = self.bank_service_provider.get_transactions(account_number, from_date,
151             to_date)
152             if transactions:
153                 print("List of Transactions:")
154                 for transaction in transactions:
155                     print(transaction)
156             else:
157                 print("No transactions found.")
158         except InvalidAccountException as e:
159             print(f"Error: {e}")
160
161     def generate_account_number(self):
162         print("Generating Account Number...")
163         return "1234567890"
164
165     def get_customer_details(self):
166
167         customer_id = input("Enter Customer ID: ")
168         first_name = input("Enter First Name: ")
169         last_name = input("Enter Last Name: ")
170         email = input("Enter Email Address: ")
171         phone_number = input("Enter Phone Number: ")
172         address = input("Enter Address: ")
173
174         return Customer(customer_id, first_name, last_name, email, phone_number, address)
```

```
BankApp1.py
123     def main():
124         bank_app = BankApp()
125         bank_app.main()
```