NATIONAL UNIVERSITY OF SINGAPORE
Department of Computer Science, School of Computing
IT5001—Software Development Fundamentals
Academic Year 2023/2024, Semester 2
PROBLEM SET 6

OBJECT-ORIENTED PROGRAMMING

These exercises are optional and ungraded, and for your own practice only. Contact `yongqi@nus.edu.sg` for queries.

EXERCISES

Question 1. We are going to create some classes that support the use of bank accounts.

Question 1 (i). Create the `BankAccount` class with the following methods:

1. A constructor that takes in three fields:
   - the name of the account holder
   - the initial amount of money in the account in **cents**
   - the interest rate of the account
2. A `deposit` method that accepts a positive integer as the amount of money in **cents** to deposit into the account
3. A `withdraw` method that accepts
   - a positive integer as the amount of money in **cents** to withdraw
   - the name of the account holder

   If the amount to withdraw exceeds the amount of money in the account, or if the name supplied does not match the name of the account holder, nothing is withdrawn. The method returns the amount of money withdrawn in cents.
4. A `show_balance` method that prints the account balance in **dollars**
5. A `compound` method that compounds the account balance based on the interest supplied in the constructor
6. A `transfer` method that transfers money from the target account into another input recipient. Withdrawal rules still apply, i.e. the transferrer must supply the correct name of the account holder and must have enough money to make the transfer.

An example use of the class follows.

```
>>> a = BankAccount('Alice', 100000, 0.04)
>>> a.show_balance()
Hi Alice, you have $1000.00.
>>> a.deposit(10000)
```

© 2024 National University of Singapore. All rights reserved.

```
>>> a.show_balance()
Hi Alice, you have $1100.00.
>>> a.withdraw(110100, 'Alice')
0
>>> a.withdraw(110000, 'Bob')
0
>>> a.show_balance()
Hi Alice, you have $1100.00.
>>> a.compound()
>>> a.show_balance()
Hi Alice, you have $1144.00.
>>> a.withdraw(114400, 'Alice')
114400
>>> a.show_balance()
Hi Alice, you have $0.00.
>>> b = BankAccount('Bob', 100000, 0.05)
>>> b.transfer(a, 50000, 'Bob')
>>> b.show_balance()
Hi Bob, you have $500.00.
>>> a.show_balance()
Hi Alice, you have $500.00.
```

Question 1 (ii). Create a `MinimalAccount` class that behaves the same as normal bank accounts except that whenever we compound the balance, if the amount in the account is less than $1000, a $20 administration fee is deducted from the account before the balance is compounded. The account balance will never be below zero. An example use of the class follows.

```
>>> a = MinimalAccount('Alice', 100000, 0.04)
>>> a.show_balance()
Hi Alice, you have $1000.00.
>>> a.compound()
>>> a.show_balance()
Hi Alice, you have $1040.00.
>>> a.withdraw(101900, 'Alice')
101900
>>> a.show_balance()
Hi Alice, you have $21.00.
>>> a.compound()
>>> a.show_balance()
Hi Alice, you have $1.04.
>>> a.compound()
>>> a.show_balance()
Hi Alice, you have $0.00.
```

Question 1 (iii). Create a `JointAccount` class that behaves the same as normal bank accounts except that there are two account holders, and either account holder can withdraw from the account. An example use of the class follows.

© 2024 National University of Singapore. All rights reserved.

```
>>> a = JointAccount('Alice', 'Bob', 100000, 0.04)
>>> a.show_balance()
Hi Alice and Bob, you have $1000.00.
>>> a.withdraw(10000, 'Alice')
10000
>>> a.show_balance()
Hi Alice and Bob, you have $900.00.
>>> a.withdraw(10000, 'Bob')
10000
>>> a.show_balance()
Hi Alice and Bob, you have $800.00.
```

Question 2. Smart GPS watches have become indispensible tools used by athletes. These watches have capabilities of monitoring several key statistics used by endurance athletes, such as heart rate, and pace. In this question, our task is to simulate some of the main business logic used by these watches.

Question 2 (i). Create a class `Monitor`. All monitors are able to begin monitoring, and show the results of the monitoring. The initializer of all monitors only have one parameter, which is the number of data points to be collected by the monitor during one analysis.

Question 2 (ii). Create a class `HeartRateMonitor` that monitors heart rate. When the heart rate monitor begins monitoring (i.e. the `begin_monitoring` method is called), the monitor will start an internal timer and wait for the user to press enter (you can use the `time` function from the `time` library). When the user feels their heartbeat, they press enter in the console, where the monitor will keep track of the time the user presses enter. This process repeats until all data points have been collected. When the monitoring process has concluded, when viewing the results of the monitoring, the average heart rate is printed. All heart rates are in beats per minute (bpm). An example run follows, assuming that the user presses enter at each of their heartbeats.

```
>>> a = HeartRateMonitor(10)
>>> a.begin_monitoring()
Press enter to begin heart rate monitoring...
Press enter at every heartbeat...
Instantaneous heart rate: 65bpm
Instantaneous heart rate: 59bpm
Instantaneous heart rate: 58bpm
Instantaneous heart rate: 54bpm
Instantaneous heart rate: 53bpm
Instantaneous heart rate: 55bpm
Instantaneous heart rate: 58bpm
Instantaneous heart rate: 59bpm
Instantaneous heart rate: 63bpm
Instantaneous heart rate: 63bpm
Analysis complete.
>>> a.show_results()
Average heart rate: 65.07bpm
```

Question 2 (iii). Create a class `PaceMonitor` that monitors the pace of an athlete while running.

© 2024 National University of Singapore. All rights reserved.

During the monitoring process, the user of the class will simply enter to indicate that they have just ran 10m. When the monitoring process has concluded, when viewing the results of the monitoring, the average pace is printed. All paces are in minutes per kilometre (min/km). An example run follows, assuming that the user presses Enter each time they ran 10m.

```
>>> a = PaceMonitor(10)
>>> a.begin_monitoring()
Press enter to begin pace monitoring...
Press enter at every 10m covered...
Instantaneous pace: 4:47min/km
Instantaneous pace: 5:23min/km
Instantaneous pace: 4:49min/km
Instantaneous pace: 4:42min/km
Instantaneous pace: 4:31min/km
Instantaneous pace: 4:47min/km
Instantaneous pace: 4:38min/km
Instantaneous pace: 4:38min/km
Instantaneous pace: 4:31min/km
Instantaneous pace: 4:36min/km
Analysis complete.
>>> a.show_results()
Average pace: 4:44min/km
```

Question 3. The Kent Ridge Cruise Centre has just opened and you are required to design a program to decide how many loaders to buy based on a single-day cruise schedule.

Loaders:

- Loaders have to be purchased to serve cruises. Each loader will serve a cruise as soon as it arrives, and continues to do so until the service time has elapsed.
- While the loaders are in the midst of serving a cruise, they cannot serve another cruise.
- For example, if an incoming cruise arrives at 12PM, requires two loaders, and 60 min for it to be fully served, then, at 12PM, there must be two vacant loaders. These two loaders will serve the cruise from 12PM - 1PM. They can only serve another cruise from 1PM onwards.

Cruises:

- All cruises have four attributes: a unique identifier, and a time of arrival, the time needed to serve the cruise (in minutes) and the number of loaders needed to serve the cruise.
- The unique identifier is a string, for example, `'S1234'`. The time of arrival comes in HHMM format, such as `2359`, denoting that the cruise has arrived at 11:59PM on that day.
- Every cruise must be served by loaders immediately upon arrival. There are two types of cruises:
  - `Cruise`:
    - Cruises have an identifier that starts with a character `'S'` (case sensitive).
    - Takes a fixed 30min for a loader to fully load.
    - Requires only one loader for it to be fully served.

© 2024 National University of Singapore. All rights reserved.

    - A schedule entry is read as `S1234 0000` denoting that cruise `S1234` arrives at 12AM, requires 1 loader and 30min to be fully served
  - `BigCruise`:
    - BigCruises have an identifier that starts with the character 'B' (case-sensitive). Furthermore, they have a variable number of loaders and time required to serve it.
    - Takes one minute to serve every 50 passengers.
    - Requires one loader per every 40 meters in length of the cruise (or part thereof) to fully load,
    - A schedule entry is read as `B1234 0000 200 3720` denoting that cruise `B1234` arrives at 12AM, has a length of 200m and has 3720 passengers. This cruise requires $\lceil \frac{200}{40} \rceil = 5$ loaders and $\lceil \frac{3720}{50} \rceil = 75$ minutes to be fully served.

The program determines the number of loaders and the allocation schedule using the following steps:

- For each cruise, check through the inventory of loaders, starting from the loader first purchased and so on.
- The first (or first few) loaders available will be used to serve the cruise.
- If there are not enough loaders, purchase a new one(s), and that loader(s) will be used to serve the cruise.

Your task is to write a program that reads in a file of the cruise schedule (i.e. the cruises that will arrive for that day).

The program will print the loader allocation schedule. Take note of the following assumptions:

- Input cruises are presented chronologically by arrival time.
- There can be up to 30 cruises in one day.
- The number of loaders servicing a cruise will not exceed 9.
- There are no duplicates in the input cruises.
- All cruises will arrive and be completely served within a single day.
- Input validation is not required and all inputs are assumed to be correct.

Question 3 (i). We are first going to create the `Cruise` class. Design an immutable (frozen) dataclass `Cruise` that represents a Cruise, with a unique identifier string and the time of arrival as an integer.

Note that the time of arrival is in HHMM format. Specifically, `0` refers to 00:00 (12AM), `30` refers to 00:30 (12:30AM), and `130` refers to 01:30 (1:30AM).

Also implement the following get-only properties:

- `service_completion_time` which returns the time (in minutes) the service is completed since the start of time 0,
- `arrival_time` which returns the arrival time in minutes since the start of the time 0.
- `num_loaders_required` which returns the number of loaders required to load the cruise

© 2024 National University of Singapore. All rights reserved.

For example, if the Cruise arrives at 12PM, the arrival time is $(12 \times 60) = 720$; the service completion time is 12:30PM, which is 750 minutes from 00:00. $((12 \times 60) + 30 = 750)$.

The string representation (`__str__`) of a cruise is in the form: `cruiseID@HHMM`

The `%0Xd` format specifier might be of use to you, where the integer will be represented by an X-digit zero-padded number.

Example uses follow.

```
>>> Cruise('S1234', 130)
Cruise(_id='S1234', _arrival_time=130)
>>> str(Cruise('S1234', 130))
'S1234@0130'
>>> Cruise('S1234', 130).arrival_time
90
>>> Cruise('S1234', 130).num_loaders_required
1
>>> Cruise('S1234', 130).service_completion_time
120
```

Question 3 (ii). Design an immutable dataclass `Loader` that is able to serve any cruise using the `serve` method:

- the `can_serve` method receives any cruise and returns **`True`** if it can serve the cruise, **`False`** otherwise
- the `serve` method receives any cruise and returns a new loader (think of it as a new state of the loader) with the same ID and that loader is serving this input cruise. If the loader cannot actually serve that cruise, a `CannotServeError` should be raised (define this exception yourself!)

The string representation of loaders is in the form
`Loader id serving cruiseID@cruisearrivaltime`, or if the loader is not currently serving any cruise, it is just `Loader id`.

Example runs follow:

```
>>> Loader(1)
Loader(_id=1, _currently_serving=None)
>>> str(Loader(1))
'Loader 1'
>>> Loader(1).can_serve(Cruise('S1234', 0))
True
>>> Loader(1).serve(Cruise('S1234', 0))
Loader(_id=1, _currently_serving=Cruise(_id='S1234', _arrival_time=0))
>>> str(Loader(1).serve(Cruise('S1234', 0)))
'Loader 1 serving S1234@0000'
>>> Loader(1).serve(Cruise('S1234', 0)).can_serve(Cruise('S2345', 30))
```

© 2024 National University of Singapore. All rights reserved.

```
True
>>> Loader(1).serve(Cruise('S1234', 0)).serve(Cruise('S2345', 30))
Loader(_id=1, _currently_serving=Cruise(_id='S2345', _arrival_time=30))
>>> Loader(1).serve(Cruise('S1234', 0)).can_serve(Cruise('S2345', 29))
False
>>> Loader(1).serve(Cruise('S1234', 0)).serve(Cruise('S2345', 29))
Traceback (most recent call last):
__main__.CannotServeError: Loader 1 serving S1234@0000 cannot serve S2345@0029!
```

Question 3 (iii). Now design the `BigCruise` class where its constructor/initializer receives its ID, time of arrival, length and number of passengers. Example runs follow:

```
>>> b = BigCruise('B0001', 0, 70, 3000)
>>> b.arrival_time
0
>>> b.service_completion_time
60
>>> b.num_loaders_required
2
>>> Loader(1).serve(b)
Loader(_id=1, _currently_serving=BigCruise(_id='B0001', _arrival_time=0,
_length=70, _num_passengers=3000))
>>> Loader(1).serve(b).serve(b)
Traceback (most recent call last):
__main__.CannotServeError: Loader 1 serving B0001@0000 cannot serve B0001@0000!
>>> Loader(4).serve(BigCruise('B2345', 0, 30, 1450)).serve(Cruise('S0000', 29))
Loader(_id=4, _currently_serving=Cruise(_id='S0000', _arrival_time=29))
>>> Loader(4).serve(BigCruise('B2345', 0, 75, 1510)).serve(Cruise('S0000', 30))
Traceback (most recent call last):
__main__.CannotServeError: Loader 4 serving B2345@0000 cannot serve S0000@0030!
```

Question 3 (iv). Now its time to complete the program. Your program will read in the schedule from a file called `schedule.txt` where each schedule item is on its own line. Your program will then print out the loader schedule. Example runs follow.

Test case 1
Suppose `schedule.txt` only contains `S1111 1300`. The output of your program should be
`Loader 1 serving S1111@1300`

Test case 2
Suppose `schedule.txt` contains:

```
B1111 1300 80 3000
S1111 1359
S1112 1400
S1113 1429
```

© 2024 National University of Singapore. All rights reserved.

The output of your program should be

```
Loader 1 serving B1111@1300
Loader 2 serving B1111@1300
Loader 3 serving S1111@1359
Loader 1 serving S1112@1400
Loader 2 serving S1113@1429
```

Test case 3
Suppose `schedule.txt` contains:

```
S1111 0900
B1112 0901 100 1
B1113 0902 20 4500
S2030 1031
B0001 1100 30 1500
S0001 1130
```

The output of your program should be

```
Loader 1 serving S1111@0900
Loader 2 serving B1112@0901
Loader 3 serving B1112@0901
Loader 4 serving B1112@0901
Loader 2 serving B1113@0902
Loader 1 serving S2030@1031
Loader 2 serving B0001@1100
Loader 1 serving S0001@1130
```

© 2024 National University of Singapore. All rights reserved.

SOLUTIONS

Question 1. Solutions are self-explanatory. Some key points:

- in `MinimalAccount`, we use `super().compound()` to simply compound the bank account just like normal bank accounts do.
- in `JointAccount`, we 'cheated' in the withdraw method by leveraging the existing superclass definition of withdraw.

```
class BankAccount:
    def __init__(self, name, balance, interest_rate):
        self._name = name
        self._balance = balance
        self._interest_rate = interest_rate

    def show_balance(self):
        print(f'Hi {self._name}, you have ${self._balance // 100}.'
            f'{self._balance % 100:02d}.')

    def compound(self):
        self._balance += int(self._interest_rate * self._balance)

    def withdraw(self, amount, name):
        if amount <= 0 or \
                self._balance < amount or \
                self._name != name:
            return 0
        self._balance -= amount
        return amount

    def deposit(self, amount):
        if amount <= 0:
            return
        self._balance += amount

    def transfer(self, account, amount, name):
        account.deposit(self.withdraw(amount, name))

class MinimalAccount(BankAccount):
    def compound(self):
        if self._balance < 100000:
            self._balance = max(0, self._balance - 2000)
        super().compound()

class JointAccount(BankAccount):
    def __init__(self, name1, name2, balance, interest_rate):
        self._name = name1
```

© 2024 National University of Singapore. All rights reserved.

```
        self._name2 = name2
        self._balance = balance
        self._interest_rate = interest_rate

    def withdraw(self, amount, name):
        if name == self._name2:
            return super().withdraw(amount, self._name)
        return super().withdraw(amount, name)

    def show_balance(self):
        print(f'Hi {self._name} and {self._name2}, you have '
            f'${self._balance // 100}.{self._balance % 100:02d}.')
```

Question 2. Solutions are self-explanatory. It might be odd to define a `Monitor` class this way, but the unique thing about this class design is that we can create monitors that monitor statistics in completely different ways while abiding by good design principles.

```
from time import time

class Monitor:
    def __init__(self, num_points):
        self._results = []
        self._num_points = num_points

    def begin_monitoring(self):
        pass

    def show_results(self):
        pass

class HeartRateMonitor(Monitor):
    def begin_monitoring(self):
        input('Press enter to begin heart rate monitoring...')
        t = time()
        print('Press enter at every heartbeat...')
        for _ in range(self._num_points):
            input()
            s = time()
            self._results.append(s - t)
            t = s
            print(f'Instantaneous heart rate: {int(1 / self._results[-1] * 60)}bpm')
        print('Analysis complete.')

    def show_results(self):
        print(f'Average heart rate: {(self._num_points + 1) / sum(self._results) * 60:.2f}b

class PaceMonitor(Monitor):
    def begin_monitoring(self):
```

© 2024 National University of Singapore. All rights reserved.

```
        input('Press enter to begin pace monitoring...')
        t = time()
        print('Press enter at every 10m covered...')
        for _ in range(self._num_points):
            input()
            s = time()
            self._results.append(s - t)
            t = s
            inst_pace = self._sec_to_min(self._results[-1] * 100)
            print(f'Instantaneous pace: {inst_pace[0]}:{inst_pace[1]:02d}min/km')
        print('Analysis complete.')

    def _sec_to_min(self, sec):
        return int(sec // 60), int(sec % 60)

    def show_results(self):
        avg_pace = self._sec_to_min(sum(self._results) * 100 / self._num_points)
        print(f'Average pace: {avg_pace[0]}:{avg_pace[1]:02d}min/km')
```

Question 3. Solutions are self-explanatory. It is a good idea to have each class defined in its own file.

```
# Cruise.py
from dataclasses import dataclass

def _to_minutes(t):
    return t // 100 * 60 + t % 100

@dataclass(frozen=True, eq=True)
class Cruise:
    _id: str
    _arrival_time: int
    @property
    def service_completion_time(self):
        return self.arrival_time + 30
    @property
    def arrival_time(self):
        return _to_minutes(self._arrival_time)
    @property
    def num_loaders_required(self):
        return 1
    def __str__(self):
        return f'{self._id}@{self._arrival_time:04d}'
```

In this question, it is extremely important to make use of the abstractions provided by your own Cruise objects and not to break the abstraction barrier.

© 2024 National University of Singapore. All rights reserved.

```python
# Loader.py
from dataclasses import dataclass
from CannotServeError import CannotServeError

@dataclass(frozen=True, eq=True)
class Loader:
    _id: int
    _currently_serving: 'Cruise' = None
    def can_serve(self, cruise):
        return self._currently_serving is None or \
            self._currently_serving.service_completion_time <= cruise.arrival_time
    def serve(self, cruise):
        if not self.can_serve(cruise):
            raise CannotServeError(f'{self} cannot serve {cruise}!')
        return Loader(self._id, cruise)
    def __str__(self):
        if self._currently_serving is None:
            return f'Loader {self._id}'
        return f'Loader {self._id} serving {self._currently_serving}'
```

```python
# CannotServeError.py
class CannotServeError(Exception):
    pass
```

Since big cruises are cruises, we should let `BigCruise` extend `Cruise`. By this extension, we now only need to add the attributes and re-define properties that are specific to big cruises only, making our solution much shorter.

```python
# BigCruise.py
from dataclasses import dataclass
from math import ceil
from Cruise import Cruise

@dataclass(frozen=True, eq=True)
class BigCruise(Cruise):
    _length: int
    _num_passengers: int
    @property
    def service_completion_time(self):
        return self.arrival_time + ceil(self._num_passengers / 50)
    @property
    def num_loaders_required(self):
        return ceil(self._length / 40)
```

© 2024 National University of Singapore. All rights reserved.

The brilliance of OOP stems from only needing to understand the abstractions of our objects. In fact, with polymorphism, we do not even need to care if the cruise we are dealing with is a normal cruise or big cruise, since a big cruise is-a cruise.

```python
# App.py
from Cruise import Cruise
from BigCruise import BigCruise
from Loader import Loader

# this function creates the correct cruise given
# the string in the schedule
def create_cruise(s):
    s = s.split()
    s[1] = int(s[1])
    if len(s) == 2:
        # cruise
        return Cruise(*s)
    s[2], s[3] = map(int, (s[2], s[3]))
    return BigCruise(*s)

def main():
    # read in the schedule from file
    with open('schedule.txt', 'r') as f:
        schedule = f.read().strip().split('\n')
    # keep track of the list of loaders
    loaders = []
    # iterate over every cruise in the schedule
    for i in schedule:
        # create the cruise; we do not need to care what the
        # actual cruise object is
        cruise = create_cruise(i)
        # get the number of loaders needed
        num_loaders = cruise.num_loaders_required
        # for iterating over the loaders
        loader_index = 0
        # repeat until cruise fully served
        while num_loaders:
            # create a new loader if we ran out of usable
            # loaders
            if loader_index >= len(loaders):
                loaders.append(Loader(loader_index + 1))
            # get the loader
            loader = loaders[loader_index]
            if loader.can_serve(cruise):
                # proceed to serve the cruise
                new_loader = loader.serve(cruise)
                print(new_loader)
```

© 2024 National University of Singapore. All rights reserved.

```
                # save the new state of the loader in the
                # list
                loaders[loader_index] = new_loader
                # one less loader required
                num_loaders -= 1
            # increase the index to look at the next loader
            loader_index += 1

if __name__ == '__main__':
    main()
```

– End of Problem Set 6 –

© 2024 National University of Singapore. All rights reserved.