CHAPTER

THREE

ROULETTE DETAILS

We'll start out by looking a the game of Roulette in *Roulette Game*. This will focus on Roulette as played in most American casinos.

We will follow this with *Available Bets in Roulette*. There are a profusion of bets available in Roulette. Most of the sophisticated betting strategies focus on just one of the even-money bets.

In *Some Betting Strategies*, we will describe some common betting strategies that we will simulate. The betting strategies are interesting and moderately complex algorithms for changing the amount that is used for each bet in an attempt to recoup losses.

We'll also include some additional topics in *Roulette Details Questions and Answers*.

3.1 Roulette Game

The game of Roulette centers around a *wheel* with thirty-eight numbered *bins*. The numbers include 0, 00 (double zero), 1 through 36. The *table* has a surface marked with spaces on which players can place *bets*. The spaces include the 38 *numbers*, plus a variety of additional bets, which will be detailed below.

After the bets are placed by the players, the wheel is spun by the house, a small ball is dropped into the spinning wheel; when the wheel stops spinning, the ball will come to rest in one of the thirty-eight numbered bins, defining the winning number. The winning number and all of the related winning bets are paid off; the losing bets are collected. Roulette bets are all paid off using *odds*, which will be detailed with each of the bets, below.

The numbers from 1 to 36 are colored red and black in an arbitrary pattern. They fit into various ranges, as well as being even or odd, which defines many of the winning bets related to a given number. The numbers 0 and 00 are colored green, they fit into none of the ranges, and are considered to be neither even nor odd. There are relatively few bets related to the zeroes. The geometry of the betting locations on the table defines the relationships between number bets: a group of numbers is given a colorful names like a street or a corner.

Note: American Rules

There are slight variations in Roulette between American and European casinos. We'll focus strictly on the American version.

3.2 Available Bets in Roulette

There are a variety of bets available on the Roulette table. Each bet has a payout, which is stated as $n : 1$ where n is the multiplier that defines the amount won based on the amount bet.

A \$5 bet at 2:1 will win \$10. After you are paid, there will be \$15 sitting on the table, your original \$5 bet, plus your \$10 additional winnings.

Note: Odds

Not all games state their odds using this convention. Some games state the odds as "2 *for* 1". This means that the total left on the table after the bets are paid will be two times the original bet. So a $5 bet will win $5, there will be $10 sitting on the table.

Here's the layout of the betting area on a Roulette table.

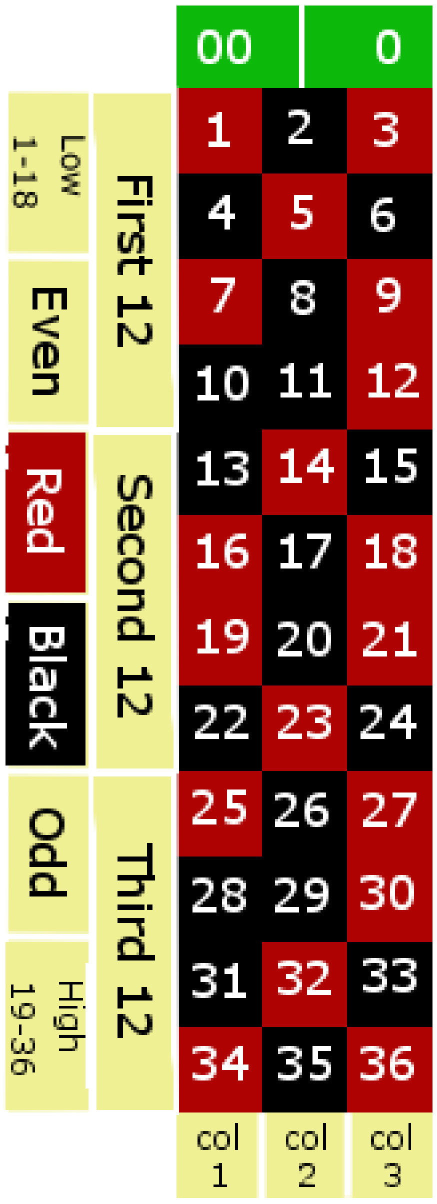

Fig. 3.1: Roulette Table Layout

The table is divided into two classes of bets. The "inside" bets are the 38 numbers and small groups of numbers; these bets all have relatively high odds. The "outside" bets are large groups of numbers, and have relatively low odds. If you are new to casino gambling, see *Odds and Payouts* for more information on odds and why they are offered.

- A "straight bet" is a bet on a single number. There are 38 possible bets, and they pay odds of 35 to 1. Each bin on the wheel pays one of the straight bets.

- A “split bet” is a bet on an adjacent pair of numbers. It pays 17:1. The table layout has the numbers arranged sequentially in three columns and twelve rows. Adjacent numbers are in the same row or column. The number 5 is adjacent to 4, 6, 2, 8; the number 1 is adjacent to 2 and 4. There are 114 of these split bet combinations. Each bin on the wheel pays from two to four of the available split bets. If the balls lands in either of two bins, the split bet is a winner.
- A “street bet” includes the three numbers in a single row, which pays 11:1. There are twelve of these bets on the table. Any of three bins make a street bet a winner.
- A square of four numbers is called a “corner bet” and pays 8:1. There are 22 of these bets available.
- At one end of the layout, it is possible to place a bet on the Five numbers 0, 00, 1, 2 and 3. This pays 6:1. It is the only combination bet that includes 0 or 00.
- A “line bet” is a six number block, which pays 5:1. It is essentially two adjacent street bets. There are 11 such combinations.

The following bets are the “outside” bets. Each of these involves a group of twelve to eighteen related numbers. None of these outside bets includes 0 or 00. The only way to bet on 0 or 00 is to place a straight bet on the number itself, or use the five-number combination bet.

- Any of the three 12-number ranges (1-12, 13-24, 25-36) pays 2:1. There are just three of these bets.
- The layout offers the three 12-number columns at 2:1 odds. All of the numbers in a given column have the same remainder when divided by three. Column 1 contains 1, 4, 7, etc., all of which have a remainder of 1 when divided by 3.
- There are two 18-number ranges: 1-18 is called *low*, 19-36 is called *high*. These are called even money bets because they pay at 1:1 odds.
- The individual numbers are colored red or black in an arbitrary pattern. Note that 0 and 00 are colored green. The bets on red or black are even money bets, which pay at 1:1 odds.
- The numbers (other than 0 and 00) are also either even or odd. These bets are also even money bets.

Odds and Payouts

Not all of the Roulette outcomes are equal probability. Let’s compare a “split bet” on 1-2 and a *even money* bet on red.

- The split bet wins if either 1 or 2 comes up on the wheel. This is 2 of the 38 outcomes, or a 1/19 probability, 5.26%.
- The red bet wins if any of the 18 red numbers come up on the wheel. The is 18 of the 38 outcomes, or a 9/19 probability, 47.4%.

Clearly, the red bet is going to win almost ten times more often than the 1-2 bet. As an inducement to place bets on rare occurences, the house offers a higher payout on those bets. Since the 1-2 split bet wins is so rarely, they will pay you 17 times what you bet. On the other hand, since the red bet wins so frequently, they will only pay back what you bet.

You’ll notice that the odds of winning the 1-2 split bet is 1 chance in 19, but they pay you 17 times your bet. Since your bet is still sitting on the table, it looks like 18 times your bet. It still isn’t 19 times your bet. This discrepency between the actual probability and the payout odds is sometimes called the *house edge*. It varies widely among the various bets in the game of Roulette. For example, the 5-way bet has 5/38 ways of winning, but pays only 6:1. There is only a 13.2% chance of winning, but they pay you as if you had a 16.7% chance, keeping the 3.5% difference. You have a 5.26% chance to win a split bet, but the house pays as if it were a 5.88% chance, a .62% discrepency in the odds.

The smallest discrepency between actual chances of winning (47.4%) and the payout odds (50%) is available on the even money bets: red, black, even, odd, high or low. All the betting systems that we will look at focus on these bets alone, since the house edge is the smallest.

3.3 Some Betting Strategies

Perhaps because Roulette is a relatively simple game, elaborate betting systems have evolved around it. Searches on the Internet turn up a many copies of the same basic descriptions for a number of betting systems. Our purpose is not to uncover the actual history of these systems, but to exploit them for simple OO design exercises. Feel free to research additional betting systems or invent your own.

Martingale. The *Martingale* system starts with a base wagering amount, w, and a count of the number of losses, c, initially 0. Each loss doubles the bet.

Any given spin will place an amount of $w \times 2^c$ on a 1:1 proposition (for example, red). When a bet wins, the loss count is reset to zero; resetting the bet to the base amount, w. This assures that a single win will recoup all losses.

Note that the casinos effectively prevent successful use of this system by imposing a table limit. At a $10 Roulette table, the limit may be as low as $1,000. A Martingale bettor who lost six times in a row would be facing a $640 bet, and after the seventh loss, their next bet would exceed the table limit. At that point, the player is unable to recoup all of their losses. Seven losses in a row is only a 1 in 128 probability; making this a relatively likely situation.

Waiting. Another system is to wait until some number of losses have elapsed. For example, wait until the wheel has spun seven reds in a row, and then bet on black. This can be combined with the Martingale system to double the bet on each loss as well as waiting for seven reds before betting on black.

This “wait for a favorable state” strategy is based on a confusion between the outcome of each individual spin and the overall odds of given collections of spins. If the wheel has spun seven reds in a row, it’s “due” to spin black.

1-3-2-6 System. Another betting system is called the *1-3-2-6* system. The idea is to avoid the doubling of the bet at each loss and running into the table limit. Rather than attempt to recoup all losses in a single win, this system looks to recoup all losses by waiting for four wins in a row.

The sequence of numbers (1, 3, 2 and 6) are the multipliers to use when placing bets after winning. At each loss, the sequence resets to the multiplier of 1. At each win, the multiplier is advanced through the sequence. After one win, the bet is now $3w$. After a second win, the bet is reduced to $2w$, and the winnings of $4w$ are “taken down” or removed from play. In the event of a third win, the bet is advanced to $6w$. Should there be a fourth win, the player has doubled their money, and the sequence resets.

Cancellation. Another method for tracking the lost bets is called the *Cancellation* system or the *Labouchere* system. The player starts with a betting budget allocated as a series of numbers. The usual example is 1, 2, 3, 4, 5, 6, 7, 8, 9.

Each bet is sum of the first and last numbers in the last. In this case 1+9 is 10. At a win, cancel the two numbers used to make the bet. In the event of all the numbers being cancelled, reset the sequence of numbers and start again. For each loss, however, add the amount of the bet to the end of the sequence as a loss to be recouped.

Here’s an example of the cancellation system using 1, 2, 3, 4, 5, 6, 7, 8, 9.

1. Bet 1+9. A win. Cancel 1 and 9, leaving 2, 3, 4, 5, 6, 7, 8.
2. Bet 2+8. A loss. Add 10, leaving 2, 3, 4, 5, 6, 7, 8, 10.
3. Bet 2+10. A loss. Add 12, leaving 2, 3, 4, 5, 6, 7, 8, 10, 12.
4. Bet 2+12. A win. Cancel 2 and 12, leaving 3, 4, 5, 6, 7, 8, 10.
5. Next bet will be 3+10.

A player could use the *Fibonacci Sequence* to structure a series of bets in a kind of cancellation system. The Fibonacci Sequence is 1, 1, 2, 3, 5, 8, 13, ...

At each loss, the sum of the previous two bets – the next numbers in the sequence – becomes the new bet amount. In the event of a win, we simply revert to the base betting amount. This allows the player to easily track our accumulated losses, with bets that could recoup those losses through a series of wins.

3.4 Roulette Details Questions and Answers

Do the house limits really have an impact on play?

Of course they do, or the house wouldn't impose them.

Many betting strategies increase bets on a loss, in an effort to break even or get slightly ahead. How many losses can occur in a row?

The answer is "indefinite." The odds of a long series of losses get less and less probable, but it never becomes zero.

For a simple $P = .5$ event – like flipping a coin – we can often see three heads in a row. One head is $P(\mathrm{h}) = .5$. Two heads is $P(\mathrm{h}, \mathrm{h}) = .5 \times .5$. If we partition a sequence of coin tosses into pairs, we expect that 1 of 4 pairs will have two heads.

Three heads is $P(\mathrm{h}, \mathrm{h}, \mathrm{h}) = .5 \times .5 \times .5$. If we partition a sequence of coin tosses into threes, we expect that 1 of 8 triples will have three heads.

A table limit that's 30 times the base bet, say $300 at a $10 table, is a way of capping play at an event has a $P = \frac{1}{30} = 0.033$. Three percent is the odds of seeing five heads in a row. If we double the bet on each loss, we'd only need to see five losses in a row to reach the table limit.

This three percent limit is in line with our ways the house maintains an edge in each game.

Why are there so many bets in Roulette?

This is a universal feature of gambling. A lot of different kinds of bets allows a player to imagine that one of those bets is better than the others.

In Blackjack, for example, there's essentially one bet. However, casinos have added a few additional bets. It doesn't reach the complexity of Roulette or Craps, but there are a number of betting opportunities even in Blackjack.

Many of the betting strategies leverage the simplest of bets: a nearly even-money proposition like red, black, even, odd, high, or low. These are easy to analyze because they're (nearly) $P = .5$ and the cost of a series of losses or a series of wins is easy to compute.

When we look at Craps, the core bet – the "pass line" – is also nearly $P = .5$, allowing the use of similar betting strategies.

The idea of the various betting strategies, then, is to avoid all of the complexity and focus on just a simple bet. For the most part, our *`Player`* implementations can follow this approach.

However, if we implement the complete game, we can write `Players` that make a number of different kinds of bets to see how the house edge breaks the various betting strategies.

CHAPTER FOUR

ROULETTE SOLUTION OVERVIEW

The first section, *Preliminary Survey of Classes*, is a survey of the classes gleaned from the general problem statement. Refer to *Problem Statement* as well as the problem details in *Roulette Details*. This survey is drawn from a quick overview of the key nouns in these sections.

We'll amplify this survery with some details of the class definitions in *Preliminary Roulette Class Structure*.

Given this preliminary of the candidate classes, *A Walkthrough of Roulette* is a walkthrough of the possible design that will refine the definitions, and give us some assurance that we have a reasonable architecture. We will make some changes to the preliminary class list, revising and expanding on our survey.

We will also include a number of questions and answers in *Roulette Solution Questions and Answers*. This should help clarify the design presentation and set the stage for the various development exercises in the chapters that follow.

4.1 Preliminary Survey of Classes

To provide a starting point for the development effort, we have to identify the objects and define their responsibilities. The central principle behind the allocation of responsibility is *encapsulation*; we do this by attempting to *isolate the information* or *isolate the processing* that must be done. Encapsulation assures that the methods of a class are the exclusive users of the fields of that class. It also makes each class very loosely coupled with other classes; this permits change without a ripple through the application.

For example, a class for each `Outcome` defines objecgs which can contain both the name and the payout odds. That way each `Outcome` instance can be used to compute a winning amount, and no other element of the simulation needs to share the odds information or the payout calculation.

In reading the background information and the problem statement, we noticed a number of nouns that seemed to be important parts of the game we are simulating.

- Wheel
- Bet
- Bin
- Table
- Red
- Black
- Green
- Number
- Odds
- Player
- House

One common development milestone is to be able to develop a class model in the Unified Modeling Language (UML) to describe the relationships among the various nouns in the problem statement. Building (and interpreting) this model takes some experience with OO programming. In this first part, we'll avoid doing extensive modeling. Instead we'll simply identify some basic design principles. We'll focus in on the most important of these nouns and describe the kinds of classes that you will build.

In a few cases, we will look forward to anticipate some future considerations. One such consideration is the house *rake*, also known as the *vigorish*, *vig*, or *commission*. In some games, the house makes a 5% deduction from some payouts. This complexity is best isolated in the `Outcome` class. Roulette doesn't have any need for a rake, since the presence of the 0 and 00 on the wheel gives the house a little over 5% edge on each bet. We'll design our class so that this can be added later when we implement Craps.

4.2 Preliminary Roulette Class Structure

We'll summarize some of the classes and responsibilities that we can identify from the problem statement. This is not the complete list of classes we need to build. As we work through the exercises, we'll discover additional classes and rework some of these preliminary classes more than once.

We'll describe each class with respect to the responsibility allocated to the class and the collaborators. Some collabotors are used by an object to get work done. We have a number of "uses-used by" collaborative relationships among our various classes.

Outcome Responsibilities.

A name for the bet and the payout odds. This isolates the calculation of the payout amount. Example: "Red", "1:1".

Collaborators.

Collected by `Wheel` into the bins that reflect the bets that win; collected by `Table` into the available bets for the `Player`; used by `Game` to compute the amount won from the amount that was bet.

Wheel Responsibilities.

Selects the `Outcome` s that win. This isolates the use of a random number generator to select `Outcome`s; and it encapsulates the set of winning `Outcome`s that are associated with each individual number on the wheel. Example: the "1" bin has the following winning `Outcome`s: "1", "Red", "Odd", "Low", "Column 1", "Dozen 1-12", "Split 1-2", "Split 1-4", "Street 1-2-3", "Corner 1-2-4-5", "Five Bet", "Line 1-2-3-4-5-6", "00-0-1-2-3", "Dozen 1", "Low" and "Column 1".

Collaborators.

Collects the `Outcome`s into bins; used by the overall `Game` to get a next set of winning `Outcome`s.

Table Responsibilities.

A collection of bets placed on `Outcome`s by a `Player`. This isolates the set of possible bets and the management of the amounts currently at risk on each bet. This also serves as the interface between the `Player` and the other elements of the game.

Collaborators.

Collects the `Outcome`s; used by `Player` to place a bet amount on a specific `Outcome`; used by `Game` to compute the amount won from the amount that was bet.

Player Responsibilities.

Places bets on `Outcome`s, updates the stake with amounts won and lost.

Collaborators.

Uses `Table` to place bets on `Outcome`s; used by `Game` to record wins and losses.

Game Responsibilities.

Runs the game: gets bets from *Player*, spins *Wheel*, collects losing bets, pays winning bets. This encapsulates the basic sequence of play into a single class.

Collaborators.

Uses *Wheel*, *Table*, *Outcome*, *Player*. The overall statistical analysis will play a finite number of games and collect the final value of the *Player*'s stake.

The class *Player* has the most important responsibility in the application, since we expect to update the algorithms this class uses to place different kinds of bets. Clearly, we need to cleanly encapsulate the *Player*, so that changes to this class have no ripple effect in other classes of the application.

4.3 A Walkthrough of Roulette

A good preliminary task is to review these responsibilities to confirm that a complete cycle of play is possible. This will help provide some design details for each class. It will also provide some insight into classes that may be missing from this overview.

A good way to structure this task is to do a Class-Reponisibility-Collaborators (CRC) *walkthrough*.

As preparation, get some 5" x 8" notecards. On each card, write down the name of a class, the responsibilities and the collaborators. Leave plenty of room around the responsibilities and collaborators to write notes. We've only identified five classes, so far, but others always show up during the walkthrough.

During the walkthrough, we identify areas of responsibility, allocate them to classes of objects and define any collaborating objects. An area of responsibility is a thing to do, a piece of information, a result. Sometimes a big piece of responsibility can be broken down into smaller pieces, and those smaller pieces assigned to other classes. There are a lot of reasons for decomposing, the purpose of this book is to explore many of them in depth. Therefore, we won't justify any of our suggestions until later in the book. For now, follow along closely to get a sense of where the exercises will be leading.

The basic processing outline is the responsibility of the *Game* class. To start, locate the *Game* card.

1. Our preliminary note was that this class "Runs the game." The responsibilities section has a summary of four steps involved in running the game.
2. The first step is "gets bets from *Player*." Find the *Player* card.
3. Does a *Player* collaborate with a *Game* to place bets? If not, update the cards as necessary to include this.
4. One of the responsibilities of a *Player* is to place bets. The step in the responsibility statement is merely "Places bets on *Outcome*s." Looking at the classes, we note that the *Table* contains the amounts placed on the Bets. Fix the collaboration information on the *Player* to name the *Table* class. Find the *Table* card.
5. Does a *Table* collaborate with a *Player* to accept the bets? If not, update the cards as necessary to include this.
6. What card has responsibility for the amount of the bet? It looks like *Table*. We note one small problem: the *Table* contains the *collection* of amounts bet on *Outcome*s.

   What class contains the individual "amount bet on an *Outcome*?" This class appears to be missing. We'll call this new class *Bet* and start a new card. We know one responsibility is to hold the amount bet on a particular *Outcome*.

   We know three collaborators: the amount is paired with an *Outcome*, all of the *Bet* s are collected by a *Table*, and the *Bet* s are created by a *Player*. We'll update all of the existing cards to name their collaboration with *Bet*.
7. What card has responsibility for keeping all of the *Bet*s? Does *Table* list that as a responsibility? We should update these cards to clarify this collaboration.

You should continue this tour, working your way through spinning the *Wheel* to get a list of winning *Outcome*s. From there, the *Game* can get all of the *Bet*s from the *Table* and see which are based on winning *Outcome*s and which are based on losing *Outcome*s. The *Game* can notify the *Player* of each losing *Bet*, and notify the *Player* of each winning *Bet*, using the *Outcome* to compute the winning amount.

This walkthrough will give you an overview of some of the interactions among the objects in the working application. You may uncover additional design ideas from this walkthrough. The most important outcome of the walkthrough is a clear sense of the responsibilities and the collaborations required to create the necessary application behavior.

4.4 Roulette Solution Questions and Answers

Why does the *Game* class run the sequence of steps? Isn't that the responsibility of some "main program?"

> **Coffee Shop Answer**. We haven't finished designing the entire application, so we need to reflect our own ignorance of how the final application will be assembled from the various parts. Rather than allocate too many responsibilities to *Game*, and possibly finding conflicts or complication, we'd rather allocate too few responsibilities until we know more.
>
> From another point of view, designing the main program is premature because we haven't finished designing the *entire* application. We anticipate a *Game* object being invoked from some statistical data gathering object to run one game. The data gathering object will then get the final stake from the player and record this. *Game*'s responsibilities are focused on playing the game itself. We'll need to add a responsibility to *Game* to collaborate with the data gathering class to run a number of games as a "session".
>
> **Technical Answer**. In procedural programming (especially in languages like COBOL), the "main program" is allocated almost all of the responsibilities. These procedural main programs usually contain a number of elements, all of which are very tightly coupled. This is a bad design, since the responsibilities aren't allocated as narrowly as possible. One small change in one place breaks the whole program.
>
> In OO languages, we can reduce the main program to a short list of object constructors, with the real work delegated to the objects. This level of coupling assures us that a small change to one class has no impact on other classes or the program as a whole.

Why is *Outcome* a separate class? Each object that is an instance of *Outcome* only has two attributes; why not use an array of Strings for the names, and a parallel array of integers for the odds?

> **Representation**. We prefer not to decompose an object into separate data elements. If we do decompose this object, we will have to ask which class would own these two arrays? If *Wheel* keeps these, then *Table* becomes very tightly coupled to these two arrays that should be *Wheel*'s responsibility. If *Table* keeps these, then *Wheel* is priviledged to know details of how *Table* is implemented. If we need to change these arrays to another storage structure, two classes would change instead of one.
>
> Having the name and odds in a single *Outcome* object allows us to change the representation of an *Outcome*. For example, we might replace the String as the identification of the outcome, with a collection of the individual numbers that comprise this outcome. This would identify a straight bet by the single winning number; an even money bet would be identified by an array of the 18 winning numbers.
>
> **Responsibility**. The principle of isolating responsibility would be broken by this "two parallel arrays" design because now the *Game* class would need to know how to compute odds. In more complex games, there would be the added complication of figuring the rake. Consider a game where the *Player*'s strategy depends on the potential payout. Now the *Game* and the *Player* both have copies of the algorithm for computing the payout. A change to one must be paired with a change to the other.
>
> The alternative we have chosen is to encapsulate the payout algorithm along with the relevant data items in a single bundle.

If *Outcome* encapsulates the function to compute the amount won, isn't it just a glorified subroutine?

If you're background is BASIC or FORTRAN, this can seem to be true. A class can be thought of as a glorified *subroutine library* that captures and isolates data elements along with their associated functions.

A class is more powerful than a simple subroutine library with private data. For example, classes introduce *inheritance* as a way to create a family of closely-related definitions in a simple way.

We discourage trying to mapping OO concepts back to other non-OO languages.

What is the distinction between an `Outcome` and a `Bet`?

We need to describe the propositions on the table on which you can place bets. The propositions are distinct from an actual amount of money wagered on a proposition. There are a lot of terms to choose from, including bet, wager, proposition, place, location, or outcome. We opted for using `Outcome` because it seemed to express the open-ended nature of a potential outcome, different from an amount bet on a potential outcome. We're considering the `Outcome` as an abstract possibility, and the `Bet` as a concrete action taken by a player.

Also, as we expand this simulation to cover other games, we will find that the randomized outcome is not something we can directly bet on. In Roulette, however, all outcomes are something we can be bet on, as well as a great many combinations of outcomes. We will revisit this design decision as we move on to other games.

Why are the classes so small?

First-time designers of OO applications are sometimes uncomfortable with the notion of *emergent behavior*. In procedural programming languages, the application's features are always embodied in a few key procedures. Sometimes a single procedure, named `main`.

A good OO design partitions responsibility. In many cases, this subdivision of the application's features means that the overall behavior is not captured in one central place. Rather, it emerges from the interactions of a number of objects.

We have found that smaller elements, with very finely divided responsibilities, are more flexible and permit change. If a change will only alter a portion of a large class, it can make that portion incompatible with other portions of the same class. A symptom of this is a bewildering nest of if-statements to sort out the various alternatives. When the design is decomposed down more finely, a change can be more easily isolated to a single class. A much simpler sequence of if-statements can be focused on selecting the proper class, which can then simply carry out the desired functions.

CHAPTER FIVE

OUTCOME CLASS

In *Outcome Analysis* we'll look at the responsibilities and collaborators of Outcome objects.

In *Design Decision – Object Identity* we'll look at how we can implement the notion of object identity and object equality. This is important because we will be matching Outcome objects based on bets and spinning the Roulette wheel.

We'll look forward to some other use cases in *Looking Forward*. Specifically, we know that players, games, and tables will all share references to single outcome objects. How do we do this properly?

In *Outcome Design* we'll detail the design for this class. In *Outcome Deliverables* we'll provide a list of modules that must be built.

We'll look at a Python programming topic in *Message Formatting*. This is a kind of appendix for beginning programmers.

5.1 Outcome Analysis

There will be several hundred instances of `Outcome` objects on a given Roulette table. The bins on the wheel, similarly, collect various `Outcome`s together. The minimum set of `Outcome` instances we will need are the 38 numbers, Red, and Black. The other instances will add details to our simulation.

In Roulette, the amount won is a simple multiplication of the amount bet and the odds. In other games, however, there may be a more complex calculation because the house keeps 5% of the winnings, called the "rake". While it is not part of Roulette, it is good to have our `Outcome` class designed to cope with these more complex payout rules.

Also, we know that other casino games, like Craps, are stateful. An `Outcome` may change the game state. We can foresee reworking this class to add in the necessary features to change the state of the game.

While we are also aware that some odds are not stated as $x : 1$, we won't include these other kinds of odds in this initial design. Since all Roulette odds are $x : 1$, we'll simply assume that the denominator is always 1. We can forsee reworking this class to handle more complex odds, but we don't need to handle the other cases yet.

The issue we have, however, is comparing outcomes. They're used in various places. The idea is to compare the outcomes from a spin of the wheel against the outcomes associated with bets.

How does all this comparison work in Python?

Hint: The default rules aren't helpful.

5.2 Design Decision – Object Identity

Our design will depend on matching `Outcome` objects. We'll be testing objects for equality.

The player will be placing bets that contain `Outcome`s; the table will be holding bets. The wheel will select the winning `Outcome`s. We need a simple test to see if two objects of the `Outcome` class are the same.

Was the *Outcome* for a bet equal to the *Outcome* contained in a spin of the wheel?

It turns out that this comparison between objects has some subtlety to it.

Here's the naïve approach to class definition that doesn't include any provision for equality tests.

Naïve Class Definition

```
>>> class Outcome:
...     def __init__(self, name, odds):
...         self.name= name
...         self.odds= odds
```

This seems elegant enough. Sadly, it doesn't work out when we need to make equality tests.

In Python, if we do nothing special, the `__eq__()` test will simply compare the internal object id values. These object id values are unique to each distinct object, irrespective of the attribute values.

This default behavior of objects is shown by the following example:

Equality Test Failure

```
>>> oc1= Outcome( "Any Craps", 8 )
>>> oc2= Outcome( "Any Craps", 8 )
>>> oc1 == oc2
False
>>> id(oc1)
4334572936
>>> id(oc2)
4334573272
```

Note: Exact ID values will vary.

This example shows that we can have two objects that appear equal, but don't compare as equal. They are distinct objects with the same attribute values. This makes them not equal according to the default methods inherited from `object`. However, we would like to have two of these objects test as equal.

Actually, we want more than that.

5.2.1 More than equal

We'll be creating collections of *Outcome* objects, and we may need to create sets or maps where hash codes are used in addition to the simple equality tests.

Hash Codes?

Every object has a hash code. The hash code is simply an integer. It can be a summary of the bits that make up the object. It may simply be based on the internal id value for the object. Python computes hash codes and uses these as a quick test for set membership and dictionary keys.

If two hash codes don't match, the objects can't possibly be equal. Further comparisons aren't necessary. If two hash codes do match, then it's worth the investment of time to do use the more detailed equality comparison.

As we look forward, the Python `set` and `dict` depend on a `__hash__()` method and an `__eq__()` method of each object in the collection.

Hash Code Failure

```
>>> hash(oc1)
270386794
>>> hash(oc2)
270392959
```

Note: Exact ID values will vary.

This shows that two objects that look the same to us can have distinct hash codes. Clearly, this is unacceptable, since we want to be able to create a set of *Outcome* objects without having things that look like repeats.

5.2.2 Layers of Meaning

The issue is that we have three distinct layers of meaning for comparing objects to see if they are "equal".

- Have the same hash code. We can call this "hash equality".

  This means the `__hash__(self)()` method for several objects that represent the same *Outcome* must also have the same hash code. When we put an object into a set or a dictionary, Python uses the `hash()` function which is implemented by the `__hash__()` method.

  Sometimes the hash codes are equal, but the object attributes aren't actually equal. This is called a hash collision, and it's rare but not unexpected.

  If we don't implement this, the default version isn't too useful for creating sets of our *Outcome* obejcts.

- Compare as Equal. We can call this "attribute equality".

  This means that the `__eq__()` method returns True. When we use the **==** operator, this is evaluated by using the `__eq__()` method. This must be overridden by a class to implement attribute equality.

  If we don't implement this, the default version isn't too useful for our *Outcome* obejcts.

- Are references to the same object. We can call this "identity".

  We can test that two objects are the same by using the **is** comparison between two objects. This uses the internal Python identifier for each object. The identifier is revealed by the `id()` function.

  When we use the **is** comparison, we're asserting that the two variables are references to the same underlying object. This is the identity comparison.

5.2.3 Basics of Equality

We note that each instance of *Outcome* has a distinct `Outcome.name` value, it seems simple enough to compare names. This is one sense of "equal" that seems to be appropriate.

We can define the `__eq__()` and `__ne__()` methods work two ways:

- When comparing *Outcome* and string, it will compares the `Outcome.name` attribute. This is easy, lazy and seems to work perfectly.
- When comparing *Outcome* and *Outcome*, it can compare both name and odds. This seems like over-engineering. The odds depend on the name. The name is the key, the odds are just an attribute.

Similarly, we can compute the value of `__hash__()` using only the string name, and not the odds. This seems elegantly simple to return the hash of the string name rather than compute a hash.

The definition for `__hash__()` in section 3.3.1 of the *Language Reference Manual* tells us to do the calculation using a modulus based on `sys.hash_info.width`. This is the number of bits, the actual value we want to use is We'd use `sys.hash_info.modulus`.

5.3 Looking Forward

We'll be looking at `Outcome` objects in several contexts.

- We'll have them in bins of a wheel as winning outcomes from each spin of the wheel.
- We'll have them in bets that have been placed on the table.
- They player will have some outcomes that they prefer to bet on.

We'll be comparing table bet outcomes and bin outcomes for equality. We have a solution to that, above.

We'll be creating outcome objects, too. This bumps into an interesting problem.

> **How do we maintain** *Don't Repeat Yourself* (DRY) **when creating** `Outcome` **objects?**

We don't want to include the odds every time we create an `Outcome`. Repeating the odds would violate the DRY principle.

What are some alternatives?

- **Global Outcome Objects**. We can declare global variables for the various outcomes and use those global objects as needed.

  Generally, globals *variables* are often undesirable because changes to those variables can have unexpected consequences in a large application.

  Global *constants* are no problem at all. The pool of `Outcome` instances are proper constant values used to create bins and bets. There would be a lot of them, and they would all be assigned to distinct variables. This sounds complicated.

- **Outcome Factory**. We can create a function which is a **Factory** for individual `Outcome` objects.

  When some part of the application needs an `Outcome` object, the factory will do one of two things. If the object doesn't yet exist, the **Factory** would create it, save it, and return a reference to it. When some part of the application asked for an `Outcome` which already exists, the **Factory** would return a reference to the existing object.

  This centralizes the pool of global objects into a single object, the **Factory**.

  Further, we can identify `Outcome` instances by their names, and avoid repeating the payout odds. The function would map a name of an `Outcome` the object with all of it's details.

  As a practical matter, the **Factory** could be seeded with all outcomes. The factory function is – in effect – a global pool of constant objects.

- **Singleton Outcome Class**. A **Singleton** class creates and maintains a single instance of itself. This requires that the class have a static `instance()` method that is a reference to the one-and-only instance of the class.

  This saves us from creating global variables. Instead, each class definition contains it's own private reference to the one-and-only object of that class.

  However, this has the profound disadvantage that each distinct outcome would need to be a distinct subclass of `Outcome`. This is an unappealing level of complexity. Further, it doens't solve the DRY problem of repeating the details of each Outcome.

A **Factory** seems like a good way to proceed. It can maintain a collection, and provide values from that collection. We can use class strings to identify `Outcome` objects. We don't have to repeat the odds.

We'll look forward to this in subsequent exercises. For now, we'll start with the basic class.

5.4 Outcome Design

class `Outcome`

`Outcome` contains a single outcome on which a bet can be placed.

In Roulette, each spin of the wheel has a number of `Outcome`s with bets that will be paid off.

For example, the "1" bin has the following winning `Outcome`s: "1", "Red", "Odd", "Low", "Column 1", "Dozen 1-12", "Split 1-2", "Split 1-4", "Street 1-2-3", "Corner 1-2-4-5", "Five Bet", "Line 1-2-3-4-5-6", "00-0-1-2-3", "Dozen 1", "Low" and "Column 1". All of these bets will payoff if the wheel spins a "1".

5.4.1 Fields

`Outcome.`**`name`**

Holds the name of the `Outcome`. Examples include `"1"`, `"Red"`.

`Outcome.`**`odds`**

Holds the payout odds for this `Outcome`. Most odds are stated as 1:1 or 17:1, we only keep the numerator (17) and assume the denominator is 1.

We can use `name` to provide hash codes and do equality tests.

5.4.2 Constructors

`Outcome.`**`__init__`**`(`*self*, *name*, *odds*`)`

Parameters

- **`name`** (*str*) – The name of this outcome
- **`odds`** (*int*) – The payout odds of this outcome.

Sets the instance name and odds from the parameter name and odds.

5.4.3 Methods

For now, we'll assume that we're going to have global instances of each `Outcome`. Later we'll introduce some kind of **Factory**.

`Outcome.`**`winAmount`**`(`*self*, *amount*`)` → amount

Multiply this `Outcome`'s odds by the given amount. The product is returned.

Parameters **`amount`** (*number*) – amount being bet

`Outcome.`**`__eq__`**`(`*self*, *other*`)` → boolean

Compare the `name` attributes of `self` and `other`.

Parameters **`other`** (Outcome) – Another `Outcome` to compare against.

Returns True if this name matches the other name.

Return type bool

`Outcome.`**`__ne__`**`(`*self*, *other*`)` → boolean

Compare the `name` attributes of `self` and `other`.

Parameters **`other`** (Outcome) – Another `Outcome` to compare against.

Returns True if this name does not match the other name.

Return type bool

`Outcome.`**`__hash__`**`(`*self*`)` → int

Hash value for this outcome.

Returns The hash value of the name, `hash(self.name)`.

Return type int

A hash calculation must include all of the attributes of an object that are essential to it's distinct identity.

In this case, we can return `hash(self.name)` because the odds aren't really part of what makes an outcome distinct. Each outcome is an abstraction and a string name is all that identifies them.

The definition for `__hash__()` in section 3.3.1 of the *Language Reference Manual* tells us to do the calculation using a modulus based on `sys.hash_info.width`. That value is the number of bits, the actual value we want to use is `sys.hash_info.modulus`, which is based on the width.

`Outcome.__str__`(*self*) → string

Easy-to-read representation of this outcome. See *Message Formatting*.

This easy-to-read String output method is essential. This should return a `String` representation of the name and the odds. A form that looks like `1-2 Split (17:1)` works nicely.

Returns String of the form `name (odds:1)`.

Return type str

`Outcome.__repr__`(*self*) → string

Detailed representation of this outcome. See *Message Formatting*.

Returns String of the form `Outcome(name, odds)`.

Return type str

5.5 Outcome Deliverables

There are two deliverables for this exercise. Both will have Python docstrings.

- The `Outcome` class.
- Unit tests of the `Outcome` class. This can be doctest strings inside the class itself, or it can be a separate `unittest.TestCase` class.

  The unit test should create a three instances of `Outcome`, two of which have the same name. It should use a number of individual tests to establish that two `Outcome` with the same name will test true for equality, have the same hash code, and establish that the `winAmount()` method works correctly.

5.6 Message Formatting

For the very-new-to-Python, there are few variations on creating a formatted string.

Generally, we simply use something like this.

```
def __str__( self ):
    return "{name:s} ({odds:d}:1)".format_map( vars(self) )
```

This uses the built-in `vars()` function to expose the attributes of an object as a simple dictionary that maps attribute names to values.

This is similar to using the `self.__dict__` internal dictionary.

The format string uses `:s` and `:d` as detailed specifications for the values to interpolate into the string. There's a lot of flexbility in how numbers are formatted.

There's another variation that can be handy.

```
def __repr__( self ):
    return "{class_:s}({name!r}, {odds!r})".format(
        class_=type(self).__name__, **vars(self) )
```

This exposes the class name as well as the attribute values.

We've used the `!r` to request the internal representation for each attribute. For a string, it means it will be explicitly quoted.

CHAPTER

SIX

BIN CLASS

This chapter will present the design for the `Bin` class. In *Bin Analysis* we'll look at the responsibilities and collaborators for a bin.

As part of designing a Bin, we need to choose what is the most appropriate kind of collection class to use. We'll show how to do this in *Design Decision – Choosing A Collection*.

In *Wrap vs. Extend a Collection* we'll look at two principle ways to embed a collection class in an application. We'll clarify a few additional issues in *Bin Questions and Answers*.

In the *Bin Design* section we'll provide the detailed design information. In *Bin Deliverables* we'll enumerate what must be built.

6.1 Bin Analysis

The Roulette wheel has 38 bins, identified with a number and a color. Each of these bins defines a number of closely related winning `Outcome`s. At this time, we won't enumerate each of the 38 `Bin`s of the wheel and all of the winning `Outcome`s (from two to fourteen in each `Bin`); we'll save that for a later exercise.

At this time, we'll define the `Bin` class, and use this class to contain a number of `Outcome` objects.

Two of the `Bin`s have relatively few `Outcome`s. Specifically, the 0 `Bin` only contains the basic "0" `Outcome` and the "00-0-1-2-3" `Outcome` . The 00 `Bin` , similarly, only contains the basic "00" `Outcome` and the "00-0-1-2-3" `Outcome` .

The other 36 `Bin`s contain the straight bet, split bets, street bet, corner bets, line bets and various outside bets (column, dozen, even or odd, red or black, high or low) that will win if this `Bin` is selected. Each number bin has from 12 to 14 individual winning `Outcome`s.

Some `Outcome`s, like "red" or "black", occur in as many as 18 individual `Bin`s. Other `Outcome`s, like the straight bet numbers, each occur in only a single `Bin`. We will have to be sure that our `Outcome` objects are shared appropriately by the `Bin`s.

Since a `Bin` is just a collection of individual `Outcome` objects, we have to select a collection class to contain the objects.

6.2 Design Decision – Choosing A Collection

There are five basic Python types that are a containers for other objects.

- **Immutable Sequence**: `tuple`. This is a good candidate for the kind of collection we need, since the elements of a `Bin` don't change. Howver, a `tuple` allows duplicates and retains things in a specific order; we can't tolerate duplicates, and order doesn't matter.

- **Mutable Sequence**: `list`. While handy for the initial construction of the bin, this isn't really useful because the contents of a bin don't change once they have been enumerated.
- **Mutable Mapping**: `dict`. We don't need the key-to-value mapping feature at all. A map does more than we need for representing a *Bin*.
- **Mutable Set**: `set`. Duplicates aren't allowed, membership tests are fast, and there's no inherent ordering. This looks close to what we need.
- **Immutable Set**: `frozenset`. Duplicates aren't allowed, membership tests are fast, and there's no inherent ordering. There's not changing it after it's been built. This seems to be precisely what we need.

Having looked at the fundamental collection varieties, we will elect to use a `frozenset`.

How will we use this collection?

6.3 Wrap vs. Extend a Collection

There are two general ways to use a collection.

- **Wrap**. Define a class which has an attribute that holds the collection. We're wrapping an existing data structure in a new class.

  Something like this:

```
class Bin:
    def __init__(self, outcomes):
        self.outcomes= frozenset(outcomes)
```

- **Extend**. Define a class which **is** the collection. We're extending an existing data structure.

  Something like this:

```
class Bin(fronzenset):
    pass
```

Both are widely-used design techniques. The tradeoff between them isn't clear at first.

Considerations include the following:

- When we **wrap**, we'll often need to write a lot of additional methods for representation, length, comparisons, inquiries, etc.

  In some cases, we will wrap a collection specifically so that these additional methods are **not** available. We want to completely conceal the underlying data structure.
- When we **extend**, we get all of the base methods of the collection. We can add any unique features.

In this case, extending the existing data structure seems to make more sense than wrapping a `frozenset`.

6.4 Bin Questions and Answers

Why wasn't *Bin* in the design overview?

> The definition of the Roulette game did mention the 38 bins of the wheel. However, when identifying the nouns, it didn't seem important. Then, as we started designing the *Wheel* class, the description of the wheel as 38 bins came more fully into focus. Rework of the preliminary design is part of detailed design. This is the first of several instances of rework.

Why introduce an entire class for the bins of the wheel? Why can't the wheel be an array of 38 individual arrays?

There are two reasons for introducing *Bin* as a separate class: to improve the fidelity of our object model of the problem, and to reduce the complexity of the *Wheel* class. The definition of the game describes the wheel as having 38 bins, each bin causes a number of individual *Outcome*s to win. Without thinking too deeply, we opted to define the *Bin* class to hold a collection of *Outcome*s. At the present time, we can't foresee a lot of processing that is the responsibility of a *Bin*. But allocating a class permits us some flexibility in assigning responsibilities there in the future.

Additionally, looking forward, it is clear that the *Wheel* class will use a random number generator and will pick a winning *Bin*. In order to keep this crisp definition of responsibilities for the *Wheel* class, it makes sense to delegate all of the remaining details to another class.

Isn't an entire class for bins a lot of overhead?

The short answer is no, class definitions are almost no overhead at all. Class definitions are part of the compiler's world; at run-time they amount to a few simple persistent objects that define the class. It's the class instances that cause run-time overhead.

In a system where were are counting individual instruction executions at the hardware level, this additional class may slow things down somewhat. In most cases, however, the extra few instructions required to delegate a method to an internal object is offset by the benefits gained from additional flexibility.

How can you introduce Set, List, Vector when these don't appear in the problem?

We have to make a distinction between the classes that are uncovered during analysis of the problem in general, and classes are that just part of the implementation of this particular solution. This emphasizes the distinction between *the problem* as described by users and *a solution* as designed by software developers.

The collections framework is part of a solution, and only hinted at by the definition of the problem. Generally, these solution-oriented classes are parts of frameworks or libraries that came with our tools, or that we can license for use in our application. The problem-oriented classes, however, are usually unique to our problem.

Why extend a built-in data structure? Why not simply use it?

There are two reasons for extending a built-in data structure:

- We can easily add methods by extending. In the case of a *Bin*, there isn't much we want to add.
- We can easily change the underlying data structure by extending. For example, we might have a different set-like collection that also inherits from `Collections.abc.Set`. We can make this change in just one place – our class extension – and the entire application benefits from the alternative set implementation.

What about hash and equality?

We are't going to be comparing bins against each other. The default rules for equality and hash computation will work out just fine.

6.5 Bin Design

class **`Bin`**(*Collections.frozenset*)

Bin contains a collection of *Outcome*s which reflect the winning bets that are paid for a particular bin on a Roulette wheel. In Roulette, each spin of the wheel has a number of *Outcome*s. Example: A spin of 1, selects the "1" bin with the following winning *Outcome*s: "1" , "Red" , "Odd" , "Low" , "Column 1" , "Dozen 1-12" , "Split 1-2" , "Split 1-4" , "Street 1-2-3" , "Corner 1-2-4-5" , "Five Bet" , "Line 1-2-3-4-5-6" , "00-0-1-2-3" , "Dozen 1" , "Low" and "Column 1" . These are collected into a single *Bin* .

6.5.1 Fields

Since this is an extension to the existing `frozenset`, we don't need to define any additional fields.

6.5.2 Constructors

We don't really need to write any more specialized constructor method.

We'd use this as follows:

Python Bin Construction

```
five= Outcome( "00-0-1-2-3", 6 )
zero= Bin( [Outcome("0",35), five] )
zerozero= Bin( [Outcome("00",35), five] )
```

1. `zero` is based on references to two objects: the "0" *`Outcome`* and the "00-0-1-2-3" *`Outcome`*.
2. `zerozero` is based on references to two objects: the "00" *`Outcome`* and the "00-0-1-2-3" *`Outcome`*.

6.5.3 Methods

We don't really **need** to write any more specialized methods.

How do we accumulate several outcomes in a single *`Bin`*?

1. Create a simple list, tuple, or set as an interim structure.
2. Create the *`Bin`* from this.

We might have something like this:

```
>>> bin1 = Bin( {outcome1, outcome2, outcome3} )
```

We created an interim set object and built the final *`Bin`* from that collection object.

6.6 Bin Deliverables

There are two deliverables for this exercise. Both should have Python docstrings.

- The *`Bin`* class.
- A class which performs a unit test of the *`Bin`* class. The unit test should create several instances of *`Outcome`*, two instances of *`Bin`* and establish that *`Bin`*s can be constructed from the *`Outcome`*s.

  Programmers who are new to OO techniques are sometimes confused when reusing individual *`Outcome`* instances. This unit test is a good place to examine the ways in which object *references* are shared. A single *`Outcome`* object can be referenced by several *`Bin`*s. We will make increasing use of this in later sections.

CHAPTER

SEVEN

WHEEL CLASS

This chapter builds on the previous two chapters, creating a more complete composite object from the `Outcome` and `Bin` classes we have already defined. In *Wheel Analysis* we'll look at the responsibilities of a wheel and it's collaboration.

In the *Wheel Design* we'll provide the detailed design information. In the *Test Setup* we'll address some considerations for testing a class which has random behavior. In *Wheel Deliverables* we'll enumerate what must be built.

7.1 Wheel Analysis

The wheel has two responsibilities: it is a container for the `Bin`s and it picks one `Bin` at random. We'll also have to look at how to initialize the various `Bin`s that comprise a standard Roulette wheel.

In *The Container Responsibility* we'll look at the container aspect in detail.

In *The Random Bin Selection Responsibility* we'll look at the random selection aspects.

Based on this, the *Constructing a Wheel* section provides a description of how we can build the Wheel instance.

7.1.1 The Container Responsibility

Since the `Wheel` is 38 `Bin`s, it is a collection. We can review our survey of available collections in *Design Decision – Choosing A Collection* for some guidance here.

In this case, the choice of the winning `Bin` will be selected by a random numeric index. We need some kind of sequential collection.

> This makes an immutable `tuple` very appealing. This is a subclass of `collections.abc.Sequence` and has the features we're looking for.

One consequence of using a sequential collection is that we have to choose an index for the various `Bin`s.

The index values of 1 to 36 are logical mappings to `Bin`s. The `Bin` at index 1 would contain `Outcome("1", 35)` among several others. The `Bin` at index 2 would contain `Outcome("2", 35)`. And so on through 36.

We have a small problem, however, with 0 and 00: we need two separate indexes. While 0 is a valid index, what do we do with 00?

Enumerate some possible solutions before reading on.

Since the index of the `Bin` doesn't have any significance at all, we can assign the `Bin` that has the 00 `Outcome` to position 37. It doesn't actually matter because we'll never really use the index for any purpose other than random selection.

7.1.2 The Random Bin Selection Responsibility

In order for the *Wheel* to select a *Bin* at random, we'll need a random number from 0 to 37 that we can use an an index.

The `random` module offers a `Random.choice()` function which picks a random value from a sequence. This is ideal for returning a randomly selected *Bin* from our list of *Bin*s.

Testability. Note that testing a class using random numbers isn't going to be easy. To do testing properly, we'll need to create a non-random random number generator that we can use in place of the built-in random number generator.

To create a non-random random-number generator, we can do something like the following.

1. When testing, we can then set a specific seed value that will generate a known sequence of values.
2. Create a mock for the random number generator that returns a known, fixed sequence of values. We can leverage the `unittest.mock` module for this.

We'll address this in detail in *Review of Testability*. For now, we'll suggest using the first technique – set a specific seed value.

7.1.3 Constructing a Wheel

Each instance of *Bin* has a list of *Outcome*s. The zero ("0") and double zero ("00") *Bin* s only have two *Outcome*s. The other numbers have anywhere from twelve to fourteen *Outcome*s.

Clearly, there's quite a bit of complexity in building some of the bins.

Rather than dwell on these algorithms, we'll apply a common OO principle of *deferred binding*. We'll build a very basic wheel first and work on the bin-building algorithms later.

It's often simplest to build a class incrementally. This is an example where the overall structure is pretty simple, but some details are rather complex.

7.2 Wheel Design

class Wheel

Wheel contains the 38 individual bins on a Roulette wheel, plus a random number generator. It can select a *Bin* at random, simulating a spin of the Roulette wheel.

7.2.1 Fields

`Wheel.bins`

Contains the individual *Bin* instances.

This is a `tuple` of 38 elements. This can be built with `tuple( Bin() for i in range(38) )`

`Wheel.rng`

A random number generator to use to select a *Bin* from the *bins* collection.

Because of the central importance of this particular source of randomness, it seems sensible to isolate it from any other processing that might need random numbers. We can then set a seed value using `os.urandom()` or specific values for testing.

7.2.2 Constructors

`Wheel.__init__`(*self*)

Creates a new wheel with 38 empty `Bin`s. It will also create a new random number generator instance.

At the present time, this does not do the full initialization of the `Bin`s. We'll rework this in a future exercise.

7.2.3 Methods

`Bin.addOutcome`(*number*, *outcome*)

Adds the given `Outcome` to the `Bin` with the given number.

Parameters

- **bin** (*int*) – bin number, in the range zero to 37 inclusive.
- **outcome** (Outcome) – The Outcome to add to this Bin

`Bin.next`() → Bin

Generates a random number between 0 and 37, and returns the randomly selected `Bin`.

The `Random.choice()` function of the `random` module will select one of the available `Bin` s from the `bins` list.

Returns A Bin selected at random from the wheel.

Return type *Bin*

`Bin.get`(*bin*) → Bin

Returns the given `Bin` from the internal collection.

Parameters **bin** (*int*) – bin number, in the range zero to 37 inclusive.

Returns The requested Bin.

Return type *Bin*

7.3 Test Setup

We need a controlled kind of random number generation for testing purposes. This is done with tests that look like the following:

Test Outline

```
class GIVEN_Wheel_WHEN_next_THEN_random_choice(unittest.TestCase):
    def setUp(self):
        self.wheel= Wheel()
        self.wheel.rng.seed(1)
    def runTest(self):
        etc.
```

The values delivered from this seeded random number generator can be seen from this experiment.

Fixed pseudo-random sequence

```
>>> x = random.Random()
>>> x.seed(1)
>>> [x.randint(0,37) for i in range(10)]
[8, 36, 4, 16, 7, 31, 28, 30, 24, 13]
```

This allows us to predict the output from the `Wheel.next()` method.

7.4 Wheel Deliverables

There are three deliverables for this exercise. The new class and the unit test will have Python docstrings.

- The *Wheel* class.
- A class which performs a unit test of building the *Wheel* class. The unit test should create several instances of *Outcome*, two instances of *Bin*, and an instance of *Wheel*. The unit test should establish that *Bin*s can be added to the *Wheel*.
- A class which tests the Wheel class by selecting "random" values from a *Wheel* object using a fixed seed value.

CHAPTER

EIGHT

BIN BUILDER CLASS

We'll look at the question of filling in the Outcomes in each Bin of the Wheel. The *Bin Builder Analysis* section will address the various outcomes in details.

In *Bin Builder Algorithms* we'll look at eight algorithms for allocating appropriate outcomes to appropriate bins of the wheel.

The *BinBuilder Design* section will present the detailed design for this class. In *Bin Builder Deliverables* we'll define the specific deliverables.

In *Internationalization and Localization* we'll identify some considerations for providing local language names for the outcomes.

8.1 Bin Builder Analysis

It is clear that enumerating each `Outcome` in the 38 `Bin`s by hand is a tedious undertaking. Most `Bin`s contain about fourteen individual `Outcome`s. We need a one-time-only algorithm to do this job.

It is often helpful to create a class that is used to build an instance of another class. This is a design pattern sometimes called a **Builder**. We'll design an object that builds the various `Bin`s and assigns them to the `Wheel`. This will fill the need left open in the *Wheel Class*.

Additionally, we note that the complex algorithms to construct the `Bin`s are only tangential to the operation of the `Wheel` object. Because these are not essential to the design of the `Wheel` class, we find it is often helpful to segregate the rather complex builder methods into a separate class.

The `BinBuilder` class will have a method that enumerates the contents of each of the 36 number `Bin`s, building the individual `Outcome` instances. We can then assign these `Outcome` objects to the `Bin`s of a `Wheel` instance. We will use a number of steps to create the various types of `Outcome`s, and depend on the `Wheel` to assign each `Outcome` object to the correct `Bin`.

8.1.1 The Roulette Outcomes

Looking at the *Available Bets in Roulette* gives us a number of geometric rules for determining the various `Outcome`s that are combinations of individual numbers. These rules apply to the numbers from one to thirty-six. A different – and much simpler – set of rules applies to 0 and 00. First, we'll survey the table geometry, then we'll develop specific algorithms for each kind of bet.

- **Split Bets**. Each number is adjacent to two, three or four other numbers. The four corners (1, 3, 34, and 36) only participate in two split bets: 1-2 and 1-4, 2-3 and 3-6, 34-35 and 31-34, 35-36 and 33-36.

  The center column of numbers (5, 8, 11, ..., 32) each participate in four split bets with the surrounding numbers.

  The remaining "edge" numbers participate in three split bets.

While this is moderately complex, the bulk of the layout (from 4 to 32) can be handled with a simple rule to distinguish the center column from the edge columns. The ends (1, 2, 3, and 34, 35, 36) are a little more complex.

- **Street Bets**. Each number is a member of one of the twelve street bets.
- **Corner Bets**. Each number is a member of one, two or four corner bets. As with split bets, the bulk of the layout can be handled with a simple rule to distinguish the column, and hence the "corners".

  A number in the center column (5, 8, 11, ..., 32) is a member of four corners.

  All of the numbers along an edge are members of two corners. For example, 4 is part of 1-2-4-5, and 4-5-7-8.

  At the ends, 1, 3, and 34, 36, we see outcomes that members of just one corner each.
- **Line Bets**. Six numbers comprise a line; each number is a member of one or two lines. The ends (1, 2, 3 and 34, 35, 36) are each part of a single line. The remaining 10 rows are each part of two lines.
- **Dozen Bets**. Each number is a member of one of the three dozens. The three ranges are from 1 to 12, 13 to 24 and 25 to 36, making it very easy to associate numbers and ranges.
- **Column Bets**. Each number is a member of one of the three columns. Each of the columns has a number numeric relationship. The values are $3c + 1$, $3c + 2$, and $3c + 3$, where $0 \leq c < 12$.
- **The Even-Money Bets**. These include Red, Black, Even, Odd, High, Low. Each number on the layout will be associated with three of the six possible even money `Outcome`s.

  One way to handle these is to create the six individual `Outcome` instances. For each number, *n*, we can write a suite of if-statements to determine which of the `Outcome` objects are associated with the given number.

The `Bin`s for zero and double zero are just special cases. Each of these `Bin`s has a straight number bet `Outcome`, plus the "Five Bet" `Outcome` (00-0-1-2-3, which pays 6:1).

One other thing we'll probably want are handy names for the various kinds of odds. We might want to define a collection of constants for this.

While can define an `Outcome` as `Outcome( "Number 1", 35 )`, this is a little opaque. A slightly nicer form is `Outcome( "Number 1", RouletteGame.StraightBet )`.

The payouts are specific to a game, not general features of all `Outcome`s. They properly belong in some kind of game related class. We haven't designed the game yet, but we can provide a simplified class which only contains these odds definitions.

8.2 Bin Builder Algorithms

This section provides the algorithms for nine kinds of bets.

Note that we're going to be accumulating sets (or lists) of individual `Outcome` objects. These are interim objects that will be used to create the final `Bin` objects which are assigned to the `Wheel`.

We'll be happiest using the core Python `set` structure to accumulate these collections.

8.2.1 Generating Straight Bets

Straight bet `Outcomes` are the easiest to generate.

```
For All Numbers. For each number, $n$, such that $1 \leq n < 37$:
    Create Outcome. Create an Outcome from the number, $n$, with odds of 35:1.
    Assign to Bin. Assign this new Outcome to Bin $n$.
```

Zero. Create an `Outcome` from the "0" with odds of 35:1. Assign this to the `Bin` at index 0 in the `Wheel`.

Double Zero. Create an `Outcome` from the "00" with odds of 35:1. Assign this to `Bin` at index 37 in the `Wheel`.

8.2.2 Generating Split Bets

Split bet `Outcomes` are more complex because of the various cases: corners, edges and down-the-middle.

We note that there are two kinds of split bets:

- **left-right pairs**. These pairs all have the form $\{n, n+1\}$.
- **up-down paris**. These paris have the form $\{n, n+3\}$.

We can look at the number 5 as being part of 4 different pairs: $\{4, 4+1\}$, $\{5, 5+1\}$, $\{2, 2+3\}$, $\{5, 5+3\}$. The corner number 1 is part of 2 split bets: $\{1, 1+1\}$, $\{1, 1+3\}$.

We can generate the "left-right" split bets by iterating through the left two columns; the numbers $1, 4, 7, ..., 34$ and $2, 5, 8, ..., 35$.

```
For All Rows. For each row, $r$, where $0 \leq r < 12$:
    First Column Number. Set $n \leftarrow 3r + 1$. This will create values $1, 4, 7, ..., 34$.
    Column 1-2 Split. Create a $\{n, n+1\}$ split Outcome with odds of 17:1.
    Assign to Bins. Associate this object with two Bins: $n$ and $n+1$.
    Second Column Number. Set $n \leftarrow 3r + 2$. This will create values $2, 5, 8, ..., 35$.
    Column 2-3 Split. Create a $\{n, n+1\}$ split Outcome.
    Assign to Bins. Associate this object to two Bins: $n$ and $n+1$.
```

A similar algorithm must be used for the numbers 1 through 33, to generate the "up-down" split bets. For each number, n, we generate a $\{n, n+3\}$ split bet. This `Outcome` belongs to two `Bin`s: n and $n+3$.

8.2.3 Generating Street Bets

Street bet `Outcomes` are very simple.

We can generate the street bets by iterating through the twelve rows of the layout.

```
For All Rows. For each row, $r$, where $0 \leq r < 12$:
    First Column Number. Set $n \leftarrow 3r + 1$. This will create values $1, 4, 7, ..., 34$.
    Street. Create a $\{n, n+1, n+2\}$ street Outcome with odds of 11:1.
    Assign to Bins. Associate this object to three Bins: $n, n+1, n+2$.
```

8.2.4 Generating Corner Bets

Corner bet `Outcomes` are as complex as split bets because of the various cases: corners, edges and down-the-middle.

Each corner has four numbers, $\{n, n+1, n+3, n+4\}$. This is two numbers in the same row, and two numbers in the next higher row.

We can generate the corner bets by iterating through the numbers $1, 4, 7, ..., 31$ and $2, 5, 8, ..., 32$. For each number, n, we generate a corner bets. This `Outcome` object belongs to four `Bin`s.

We generate corner bets by iterating through the various corners based on rows and columns. There is room for two corners within the three columns of the layout: one corner starts at column 1 and the other corner starts at column 2. There is room for 11 corners within the 12 rows of the layout.

```
For All Lines Between Rows. For each row, $r$, where $0 \leq r < 11$:
    First Column Number. Set $r \leftarrow 3r + 1$. This will create values $1, 4, 7, ..., 31$.
    Column 1-2 Corner. Create a $\{n, n + 1, n + 3, n + 4\}$ corner Outcome with odds of 8:1.
    Assign to Bins. Associate this object to four Bins: $n, n + 1, n + 3, n + 4$.
    Second Column Number. Set $n \leftarrow 3r + 2$. This will create values $2, 5, 8, ..., 32$.
    Column 2-3 Corner. Create a $\{n, n + 1, n + 3, n + 4\}$ corner Outcome with odds of 8:1.
    Assign to Bins. Associate this object to four Bins: $n, n + 1, n + 3, n + 4$.
```

8.2.5 Generating Line Bets

Line bet `Outcomes` are similar to street bets. However, these are based around the 11 lines between the 12 rows.

For lines s numbered 0 to 10, the numbers on the line bet can be computed as follows: $3s + 1, 3s + 2, 3s + 3, 3s + 4, 3s + 5, 3s + 6$. This *Outcome* object belongs to six individual *Bin*s.

```
For All Lines Between Rows. For each row, $r$, where $0 \leq r < 11$:
    First Column Number. Set $n \leftarrow 3r + 1$. This will create values $1, 4, 7, ..., 31$.
    Line. Create a $\{n, n + 1, n + 2, n + 3, n + 4, n + 5\}$ line Outcome withs odds of 5:1.
    Assign to Bins. Associate this object to six Bins: $n, n + 1, n + 2, n + 3, n + 4, n + 5$.
```

8.2.6 Generating Dozen Bets

Dozen bet `Outcomes` require enumerating all twelve numbers in each of three groups.

```
For All Dozens. For each dozen, $d$, where $0 \leq d < 3$:
    Create Dozen. Create an Outcome for dozen $d + 1$ with odds of 2:1.
    For All Numbers. For each number, $m$, such that $0 \leq m < 12$:
        Assign to Bin. Associate this object to Bin $12d + m + 1$.
```

8.2.7 Generating Column Bets

Column bet `Outcomes` require enumerating all twelve numbers in each of three groups. While the outline of the algorithm is the same as the dozen bets, the enumeration of the individual numbers in the inner loop is slightly different.

```
For All Columns. For each column, $c$, such that $0 \leq c < 3$:
    Create Column. Create an Outcome for column $c + 1$ with odds of 2:1.
    For All Rows. For each row, $r$, where $0 \leq r < 12$:
        Assign to Bin. Associate this object to Bin $3r + c + 1$.
```

8.2.8 Generating Even-Money Bets

The even money bet `Outcomes` are relatively easy to generate.

Create the Red outcome, with odds of 1:1.

Create the Black outcome, with odds of 1:1.

Create the Even outcome, with odds of 1:1.

Create the Odd outcome, with odds of 1:1.

Create the High outcome, with odds of 1:1.

Create the Low outcome, with odds of 1:1.

For All Numbers. For each number, n, such that $1 \leq n < 37$:

Low? If $1 \leq n < 19$, associate the **Low** *`Outcome`* with *`Bin`* n.

High? Otherwise, $19 \leq n < 37$, associate the **High** *`Outcome`* with *`Bin`* n.

Even? If $n \mod 2 = 0$, associate the **Even** *`Outcome`* with *`Bin`* n.

Odd? Otherwise, $n \mod 2 \neq 0$, associate the **Odd** *`Outcome`* with *`Bin`* n.

Red? If $n \in \{1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30, 32, 34, 36\}$, associate the **Red** *`Outcome`* with *`Bin`* n.

Black? Otherwise, associate the **Black** *`Outcome`* with *`Bin`* n.

8.3 BinBuilder Design

class `BinBuilder`

`BinBuilder` creates the *`Outcome`*s for all of the 38 individual *`Bin`* on a Roulette wheel.

8.3.1 Constructors

`BinBuilder.`**`__init__`**`(`*self*`)`

Initializes the *`BinBuilder`*.

8.3.2 Methods

`BinBuilder.`**`buildBins`**`(`*self*, *wheel*`)`

Creates the *`Outcome`* instances and uses the `addOutcome()` method to place each *`Outcome`* in the appropropriate *`Bin`* of *`wheel`* .

Parameters **`wheel`** (Wheel) – The Wheel with Bins that must be populated with Outcomes.

There should be separate methods to generate the straight bets, split bets, street bets, corner bets, line bets, dozen bets and column bets, even money bets and the special case of zero and double zero.

Each of the methods will be relatively simple and easy to unit test. Details are provided in *Bin Builder Algorithms*.

8.4 Bin Builder Deliverables

There are three deliverables for this exercise. The new classes should have meaningful Python docstrings.

- The *`BinBuilder`* class.

- A class which performs a unit test of the *BinBuilder* class. The unit test invoke each of the various methods that create *Outcome* instances.
- Rework the unit test of the *Wheel* class. The unit test should create and initialize a *Wheel*. It can use the `Wheel.getBin()` method to check selected *Bin*s for the correct *Outcome*s.

8.5 Internationalization and Localization

An an advanced topic, we need to avoid *hard coding* the names of the bets. Python offers extensive tools for localization (l10n) of programs. Since Python works with Unicode strings, it supports non-Latin characters, supporting internationalization (i18n), also.

Unicode has a number of characters for a variety of games. It does not have the Roulette numbers – properly color-coded – as glyphs.

CHAPTER

NINE

ROULETTE BET CLASS

In addition to the design of the `Bet` class, this chapter also presents some additional questions and answers on the nature of an object, identity and state change. This continues some of the ideas from *Design Decision – Object Identity*.

In *Roulette Bet Analysis* we'll look at the details of a Bet. This will raise a question of how to identify the Outcome associated with a Bet.

We'll look at object identiy in *Design Decision – Create or Locate an Outcome*.

We'll provide some additional details in *Roulette Bet Questions and Answers*.

The *Roulette Bet Design* section will provide detailed design for the Bet class. In *Roulette Bet Deliverables* we'll enumerate the deliverables for this chapter.

9.1 Roulette Bet Analysis

A `Bet` is an amount that the player has wagered on a specific `Outcome`. This class has the responsibilty for maintaining an association between an amount, an `Outcome`, and a specific `Player`.

The general scenario is to have the `Player` construct a number of `Bet` instances. The `Wheel` is spun to select a winning `Bin`. Then each of the `Bet` objects will be checked to see if the hoped-for `Outcome` is in the actual set of `Outcome`s in the winning `Bin`.

Each winning `Bet` has an `Outcome` that matches one in the winning `Bin`. The winning bets will return money to the `Player`. All other bets aren't in the winning `Bin`; they are losers, which removes the money from the `Player`.

We have a design decision to make. Do we create a fresh `Outcome` object with each `Bet` or do we locate an existing `Outcome` object?

9.2 Design Decision – Create or Locate an Outcome

Building a `Bet` involves two parts: an `Outcome` and an amount. The amount is just a number. The `Outcome`, however, includes two parts: a name and payout odds.

We looked at this issue in *Looking Forward*. We'll revisit this design topic in some more depth here. The bottom line is this.

We don't want to create an `Outcome` object as part of constructing a `Bet` object. Here's what it might look like to place a $25 bet on Red:

Bad Idea

```
my_bet= Bet(Outcome("red", 1), 25)
```

The `Bet` includes an `Outcome` and an amount. The `Outcome` includes a name and the payout odds.

One unfortunate feature of this is that we have repeated the odds when creating an `Outcome` object. This violates the DRY principle.

We want to get a complete `Outcome` from just the name of the outcome. The will prevent repeating the odds information.

Problem. How do we locate an existing `Outcome` object?

Do we use a collection or a global variable? Or is there some other approach?

Forces. There are several parts to this design.

- We need to pick a collection. When we look at the collections (see *Design Decision – Choosing A Collection*) we can see that a Map from name to complete `Outcome` instance is ideal. This helps us associate a `Bet` with an `Outcome` given just the name of the `Outcome`.
- We need to identify some global object that builds the collection of distinct `Outcome`s.
- We need to identify some global object that can maintain the collection of `Outcome`s for use by the Player in building Bets.

If the builder and maintainer are the same object, then things would be somewhat simpler because all the responsibilities would fall into a single place.

We have several choices for the kind of global object we would use.

- **Variable**. We can define a variable which is a global map from name to `Outcome` instance. This could be an instance of the built-in `dict` class or some other mapping object. It could be an instance of a class we've designed that maps names to `Outcome` instances.

  A truly *variable* global is a dangerous thing. An immutable global object, however, is a useful idea.

  We might have this:

  #### Global Mapping

  ```
  >>> some_map["Red"]
  Outcome('Red', 1)
  ```

- **Function**. We can define a **Factory** function which will produce an `Outcome` instance as needed.

  #### Factory Function

  ```
  >>> some_factory("Red")
  Outcome('Red', 1)
  ```

- **Class**. We can define class-level methods for emitting an instance of `Outcome` based on a name. We could, for example, add methods to the `Outcome` class which retrieved instances from a class-level mapping.

  #### Class Method

  ```
  >>> Outcome.getInstance("Red")
  Outcome('Red', 1)
  ```

After creating the *BinBuilder*, we can see that this fits the overall **Factory** design for creating *Outcome* instances.

However, the class:*BinBuilder* doesn't – currently – have a handy mapping so that we can look up an *Outcome* based on the name of the outcome. Is this the right place to do the lookup?

It would look like this:

BinBuilder as Factory

```
>>> theBinBuilder.getOutcome("Red")
Outcome('Red', 1)
```

What about the *Wheel*?

Wheel as Factory

```
>>> theWheel.getOutcome("Red")
Outcome('Red', 1)
```

Alternative Solutions. We have a number of potential ways to gather all *Outcome* objects that were created by the *BinBuilder*.

- Clearly, the *BinBuilder* can create the mapping from name to each distinct *Outcome*. To do this, we'd have to do several things.

  First, we expand the *BinBuilder* to keep a simple Map of the various *Outcome*s that are being assigned via the `Wheel.add()` method.

  Second, we would have to add specific *Outcome* getters to the *BinBuilder*. We could, for example, include a `getOutcome()` method that returns an *Outcome* based on its name.

  Here's what it might look like in Python.

```
    class BinBuilder( object ):
        ...
        def apply( self, outcome, bin, wheel ):
            self.all_outcomes.add( outcome )
            wheel.add( bin, outcome )
        def getOutcome( self, name ):
            ...
```

- Access the *Wheel*. A better choice is to get *Outcome* obects from the *Wheel*. To do this, we'd have to do several things.

  First, we expand the *Wheel* to keep a simple Map of the various *Outcome*s created by the *BinBuilder*. This Map would be maintained by the `Wheel.add()`.

  Second, we would have to add specific *Outcome* getters to the *Wheel*. We could, for example, include a `getOutcome()` method that returns an *Outcome* based on its name.

  We might write a method function like the following to Wheel.

```
    class Wheel( object ):
        ...
        def add( self, bin, outcome ):
            self.all_outcomes.add( outcome )
            ...
        def getOutcome( self, name ):
            return set( [ oc for oc in self.all_outcomes if oc.name.lower().contains( name.lower() ) 
```

Solution. The allocation of responsibility seems to be a toss-up. We can see that the amount of programming is almost identical. This means that the real question is one of clarity: which allocation more clearly states our intention?

The `Wheel` is a first-class part of the game of Roulette. It showed up in our initial noun analysis. The `BinBuilder` was an implementation convenience to separate the one-time construction of the `Bin`s from the overall work of the `Wheel`.

Since `Wheel` is a first-class part of the problem, we should augment the `Wheel` to keep track of our individual `Outcome` objects by name.

9.3 Roulette Bet Questions and Answers

Why not update each `Outcome` with the amount of the bet?

> We are isolating the static definition of the `Outcome` from the presence or absence of an amount wagered. Note that the `Outcome` is shared by the wheel's `Bin` s, and the available betting spaces on the `Table`, possibly even the `Player` class. Also, if we have multiple `Player`, then we need to distinguish bets placed by the individual players.
>
> Changing a field's value has an implication that the thing has changed state. In Roulette, there isn't any state change in the definition of an `Outcome`. However, when we look at Craps, we will see that changes in the game's state will enable and disable whole sets of `Outcome`s.

Does an individual bet really have unique identity? Isn't it just anonymous money?

> Yes, the money is anonymous. In a casino, the chips all look alike. However, the placement of the bet, really does have unique identity. A `Bet` is owned by a particular player, it lasts for a specific duration, it has a final outcome of won or lost. When we want to create summary statistics, we could do this by saving the individual `Bet` objects. We could update each `Bet` with a won or lost indicator, then we can total the wins and losses.
>
> This points up another reason why we know a `Bet` is an object in its own right: it changes state. A bet that has been placed can change to a bet that was won or a bet that was lost.

9.4 Roulette Bet Design

class `Bet`

`Bet` associates an amount and an `Outcome`. In a future round of design, we can also associate a `Bet` with a `Player`.

9.4.1 Fields

`Bet.`**`amountBet`**
> The amount of the bet.

`Bet.`**`outcome`**
> The `Outcome` on which the bet is placed.

9.4.2 Constructors

`Bet.`**`__init__`**(*self*, *amount*, *outcome*)

> **Parameters**
>
> - **`amount`** (*int*) – The amount of the bet.
> - **`outcome`** (Outcome) – The `Outcome` we're betting on.

Create a new Bet of a specific amount on a specific outcome.

For these first exercises, we'll omit the *`Player`*. We'll come back to this class when necessary, and add that capability back in to this class.

9.4.3 Methods

`Bet.`**`winAmount`**`(`*self*`)` → int

Returns amount won

Return type int

Uses the *`Outcome`*'s *`winAmount`* to compute the amount won, given the amount of this bet. Note that the amount bet must also be added in. A 1:1 outcome (e.g. a bet on Red) pays the amount bet plus the amount won.

`Bet.`**`loseAmount`**`(`*self*`)` → int

Returns amount lost

Return type int

Returns the amount bet as the amount lost. This is the cost of placing the bet.

`Bet.`**`__str__`**`(`*self*`)` → str

Returns string representation of this bet with the form "*`amount`* `on` *`outcome`*"

Return type str

Returns a string representation of this bet. Note that this method will delegate the much of the work to the *`__str__()`* method of the *`Outcome`*.

`Bet.`**`__repr__`**`(`*self*`)` → str

Returns string representation of this bet with the form "`Bet(`*`amount`*`,` *`outcome`*`)`"

Return type str

9.5 Roulette Bet Deliverables

There are four deliverables for this exercise. The new classes will have Python docstrings.

- The expanded *`Wheel`* class which creates a mapping of string name to *`Outcome`*.
- Expanded unit tests of *`Wheel`* that confirm that the mapping is being built correctly.
- The *`Bet`* class.
- A class which performs a unit test of the *`Bet`* class. The unit test should create a couple instances of *`Outcome`*, and establish that the `winAmount()` and `loseAmount()` methods work correctly.

CHAPTER

TEN

ROULETTE TABLE CLASS

This section provides the design for the `Table` to hold the bets. In the section *Roulette Table Analysis* we'll look at the table as a whole.

One of the table's resposibilities seems to be to validate bets. In *InvalidBet Exception Design* we'll look at how we can design an appropriate exception.

In *Roulette Table Design* we'll look at the details of creating the table class. Then, in *Roulette Table Deliverables* we'll enumerate the deliverables for this chapter.

10.1 Roulette Table Analysis

We'll look at several topics in detail as part of the analysis of the table.

- *Winning vs. Losing* explores how we handle the payment of a bet and the receipt of the winnings.
- In *Container Implementation* we'll look at how we store bets.
- We'll look at casino betting limits in *Table Limits*.
- The *Adding and Removing Bets* section discusses some additional details of how the bet container must work.

The `Table` has the responsibility to keep the `Bet`s created by the `Player`. Additionally, the house imposes *table limits* on the minimum amount that must be bet and the maximum that can be bet. Clearly, the `Table` has all the information required to evaluation these conditions.

Note: Betting Constraints

Casinos prevent the *Martingale* betting system from working by imposing a table limit on each game. To cover the cost of operating the table game, the casino also imposes a minimum bet. Typically, the maximum is a multiplier of the minimum bet, often in the range of 10 to 50; a table with a $5 minimum might have a $200 limit, a $10 minimum may have only a $300 limit.

It isn't clear where the responsibility lies for determining winning and losing bets. The money placed on `Bet`s on the `Table` is "at risk" of being lost. If the bet is a winner, the house pays the `Player` an amount based on the `Outcome` odds and the `Bet` amount. If the bet is a loser, the amount of the `Bet` is forfeit by the `Player`. Looking forward to stateful games like Craps, we'll place the responsibility for determining winners and losers with the game, and not with the `Table` object.

We'll wait, then, until we write the game to finalize paying winning bets and collecting losing bets.

10.1.1 Winning vs. Losing

Another open question is the timing of the payment for the bet from the player's stake. In a casino, the payment to the casino – effectively – happens when the bet is placed on the table. In our Roulette simulation, this is a subtlety that doesn't have any practical consequences. We could deduct the money as part of *Bet* creation, or we could deduct the money as part of resolving the spin of the wheel. In other games, however, there may several events and several opportunities for placing additional bets. For example, splitting a hand in blackjack, or placing additional odds bets in Craps.

Because we can't allow a player to bet more than their stake, we should deduct the payment as the *Bet* is created.

A consequence of this is a change to our definition of the *Bet* class. We don't need to compute the amount that is lost. We're not going to deduct the money when the bet resolved, we're going to deduct the money from the *Player*'s stake as part of creating the *Bet*. This will become part of the design of *Player* and *Bet*.

Looking forward a little, a stateful game like Craps will introduce a subtle distinction that may be appropriate for a future subclass of *Table*. When the game is in the **point off** state, some of the bets on the table are not allowed, and others become inactive. When the game is in the **point on** state, all bets are allowed and active. In Craps parlance, some bets are "not working" or "working" depending on the game state. This does not apply to the version of *Table* that will support Roulette.

10.1.2 Container Implementation

A *Table* is a collection of *Bet*s. We need to choose a concrete class for the collection of the bets. We can review the survey of collections in *Design Decision – Choosing A Collection* for some guidance here.

In this case, the bets are placed in no particular order, and are simply visited in an arbitrary order for resolution. Bets don't have specific names.

Since the number of bets varies, we can't use a Python `tuple`; a `list` will have to do. We could also use a set because duplicate bets don't make any sense.

10.1.3 Table Limits

Table limits can be checked by providing a public method `isValid()` that compares the total of all existing *Bet*s against the table limit. This should be used by the *Game* to confirm that bets are legal before proceeding.

In the unlikely event of the *Player* object creating an illegal *Bet*, this will raise an exception to indicate that we have a design error that was not detected via unit testing. This should be a subclass of `Exception` that has enough information to debug the problem with the *Player* that attempted to place the illegal bet.

Each individual bet must meet the table minimum. This is a separate rule that can be checked each time a bet is placed.

10.1.4 Adding and Removing Bets

A *Table* contains `Bets`. Instances of *Bet* are added by a *Player*. Later, `Bets` will be removed from the *Table* by the *Game*. When a bet is resolved, it must be deleted. Some games, like Roulette resolve all bets with each spin. Other games, like Craps, involve multiple rounds of placing and resolving some bets, and leaving other bets in play.

For bet deletion to work, we have to provide a method to remove a *Bet* instance. When we look at game and bet resolution we'll return to bet deletion. It's import not to over-design this class at this time; we will often add features as we develop designs for additional use cases.

10.2 InvalidBet Exception Design

We'll raise an exception for an invalid bet. This is, in general, better than having a method which returns True for a valid bet and False for an invalid bet.

It's better because we can simply place the bet, assuming that it is valid. The processing continues along this happy path.

If the bet is not valiid, the exception interrupts processing. The only way to get an invalid bet in Roulette is to have a badly damaged implementation of the `Player` class. We really need to have the application break in a catastrophic manner.

exception `InvalidBet`

`InvalidBet` is raised when the `Player` attempts to place a bet which exceeds the table's limit.

This class simply inherits all features of its superclass.

10.3 Roulette Table Design

class `Table`

`Table` contains all the `Bet` s created by the `Player`. A table also has a betting limit, and the sum of all of a player's bets must be less than or equal to this limit. We assume a single `Player` in the simulation.

10.3.1 Fields

`Table.`**`limit`**

This is the table limit. The sum of the bets from a `Player` must be less than or equal to this limit.

`Table.`**`minimum`**

This is the table minimum. Each individual bet from a `Player` must be greate than this limit.

`Table.`**`bets`**

This is a `list` of the `Bet`s currently active. These will result in either wins or losses to the `Player`.

10.3.2 Constructors

`Table.`**`Table`**`()`

Creates an empty `list` of bets.

10.3.3 Methods

`Table.`**`placeBet`**`(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – A Bet instance to be added to the table.

Raises InvalidBet

Adds this bet to the list of working bets.

We'll reserve the idea of raising an exception for an individual invalid bet. This is a rare circumstance, and indicates a bug in the `Player` more than anything else.

We might, for example, confirm that the Bet's `Outcome` exists in one of the `Bin`s. We might check that the bet amount is greater than or equal to the table minimum. We might also check the upper limit on betting will be honored by all existing bets plus this new bet.

It's not **necessary** to validate each bet as they're being placed. It's only necessary to validate the bets once prior to spinning the wheel. This is a function of the Game, and a separate interface is available for this.

For an interactive game – not a simulation – we would want to validate each bet prior to accepting it so that we can provide an immediate response to the player that the potential bet is invalid. In this case, we'd leave the table untouched when a bad bet is offered.

`Table.__iter__()` → iter

Returns an iterator over the available list of `Bet` instances. This simply returns the iterator over the list of `Bet` objects.

Note that we need to be able remove Bets from the table. Consequently, we have to update the list, which requires that we create a copy of the list. This is done with `bets[:]`.

Returns iterator over all bets

`Table.__str__()` → str

Return an easy-to-read string representation of all current bets.

`Table.__repr__()` → str

Return a representation of the form `Table( bet, bet, ... )`.

Note that we will want to segregate validation as a separate method, or sequence of methods. This is used by the Game just prior to spinning the wheel (or rolling the dice, or drawing a next card.)

`Table.isValid(`*self*`)`

Raises InvalidBet if the bets don't pass the table limit rules.

Applies the table-limit rules:

- The sum of all bets is less than or equal to the table limit.
- All bet amounts are greater than or equal to the table minimum.

If there's a problem an `InvalidBet` exception is raised.

10.4 Roulette Table Deliverables

There are three deliverables for this exercise. Each of these will have complete Python docstring comments.

- An `InvalidBet` exception class. This is a simple subclass of `Exception`.
- Since there's no unique programming here, the unit test for `InvalidBet` is pretty simple. Indeed, it can seem silly to be sure that this class works with the `raise` statement; however, failure to extend `Exception` would lead to a program that more-or-less worked until a faulty `Player` class caused the invalid bet situation.
- The `Table` class.
- A class which performs a unit test of the `Table` class. The unit test should create at least two instances of `Bet`, and establish that these `Bet` s are managed by the table correctly.

CHAPTER

ELEVEN

ROULETTE GAME CLASS

Between `Player` and `Game`, we have a *chicken-and-egg design problem*: it's not clear which we should do first. In this chapter, we'll describe the design for `Game` in detail. However, in order to create the deliverables, we have to create a version of `Player` that we can use just to get started.

In the long run, we'll need to create a sophisticated hierarchy of players. Rather than digress too far, we'll create a simple player, `Passenger57` (they always bet on black), which will be the basis for further design in later chapters.

The class that implements the game will be examined in *Roulette Game Analysis*.

Once we've defined the game, we can also define a simple player class to interact with the game. We'll look at this in *Passenger57 Design*.

After looking at the player in detail, we can look at the game in detail. We'll examine the class in *Roulette Game Design*.

We'll provide some additional details in *Roulette Game Questions and Answers*. The *Roulette Game Deliverables* section will enumerate the deliverables.

There are a few variations on how Roulette works. We'll look at how we can include these details in the *Appendix: Roulette Variations* section.

11.1 Roulette Game Analysis

The `RouletteGame`'s responsibility is to cycle through the various steps of a defined procedure. We'll look at the procedure in detail. We'll also look at how we match bets in *The Bet Matching Algorithm*. This will lead us to define the player interface, which we'll look at in *Player Interface*.

This is an *active* class that makes use of the classes we have built so far. The hallmark of an active class is longer or more complex methods. This is distinct from most of the classes we have considered so far, which have relatively trivial methods that are little more than getters and setters of instance variables.

The procedure for one round of the game is the following.

A Single Round of Roulette

1. **Place Bets**. Notify the `Player` to create `Bet`s. The real work of placing bets is delegated to the `Player` class. Note that the money is committed at this point; they player's stake should be reduced as part of creating a `Bet`.
2. **Spin Wheel**. Get the next spin of the `Wheel`, giving the winning `Bin`, w. This is a collection of individual `Outcome`'s which will be winners. We can say $w = o_0, o_1, o_2, ..., o_n$: the winning bin is a set of outcomes.
3. **Resolve All Bets**.

   For each `Bet`, b, placed by the `Player`:

(a) **Winner?** If `Bet` b `Outcome` is in the winning `Bin`, w, then notify the `Player` that `Bet` b was a winner and update the `Player`'s stake.

(b) **Loser?** If `Bet` b `Outcome` is not in the winning `Bin`, w, then notify the `Player` that `Bet` b was a loser. This allows the `Player` to update the betting amount for the next round.

11.1.1 The Bet Matching Algorithm

This `Game` class will have the responsibility for matching the collection of `Outcome`s in the `Bin` of the `Wheel` with the collection of `Outcome`s of the `Bet`s on the `Table`. We'll need to structure a loop to compare individual elements from these two collections.

- Driven by `Bin`. We could use a loop to visit each `Outcome` in the winning `Bin`.

  For each `Outcome` in the winning `Bin`, $o(w)$:

  > For each `Bet` contained by the `Table`, b:
  >
  > > If the `Outcome` of the `Bet` matches the `Outcome` in the winning `Bin`, this bet, b, is a winner and is paid off.

  After this examination, all `Bet`s which have not been paid off are losers.

  This is unpleasantly complex because we can't resolve a `Bet` until we've checked all outcomes in the winning `Bin`.

- Driven by `Table`. The alternative is to visit each `Bet` contained by the `Table`.

  For each `Bet` in the `Table`, b:

  > If the `Outcome` of b is in the `Outcome`s in the winning `Bin`, the bet is a winner. If the `Outcome` is not in the `Bin`, the bet is a loser.

  Since the winning `Bin` is a frozenset of `Outcome`s, we can exploit set membership methods to test for presence or absence of the `Outcome` for a `Bet` in the winning `Bin`.

11.1.2 Player Interface

The `Game` and `Player` collaboration involves mutual dependencies. This creates a "chicken and egg" problem in decomposing the relationship between these classes. The `Player` depends on `Game` features. The `Game` depends on `Player` features.

Which do we design first?

We note that the `Player` is really a complete hierarchy of subclasses, each of which provides a different betting strategy. For the purposes of making the `Game` work, we can develop our unit tests with a stub for `Player` that simply places a single kind of `Bet`. We'll call this player "Passenger57" because it always bets on Black.

Once we have a simplistic player, we can define the `Game` more completely.

After we have the `Game` finished, we can then revisit this design to make more sophisticated subclasses of `Player`. In effect, we'll bounce back and forth between `Player` and `Game`, adding features to each as needed.

For some additional design considerations, see *Appendix: Roulette Variations*. This provides some more advanced game options that our current design can be made to support. We'll leave this as an exercise for the more advanced student.

11.2 Passenger57 Design

class Passenger57

Passenger57 constructs a *Bet* based on the *Outcome* named "Black". This is a very persistent player.

We'll need a source for the Black outcome. We have several choices; we looked at these in *Roulette Bet Class*. We will query the *Wheel* for the needed *Outcome* object.

In the long run, we'll have to define a *Player* superclass, and make *Passenger57* class a proper subclass of *Player*. However, our focus now is on getting the *Game* designed and built.

11.2.1 Fields

Passenger57.**black**

This is the outcome on which this player focuses their betting.

This *Player* will get this from the *Wheel* using a well-known bet name.

Passenger57.**table**

The *Table* that is used to place individual *Bet*s.

11.2.2 Constructors

Passenger57.**__init__**(*self*, *table*, *wheel*)

Parameters

- **table** (Table) – The *Table* instance on which bets are placed.
- **wheel** (Wheel) – The *Wheel* isntance which defines all *Outcome*s.

Constructs the *Player* with a specific table for placing bets. This also creates the "black" *Outcome*. This is saved in a variable named *Passenger57.black* for use in creating bets.

11.2.3 Methods

Passenger57.**placeBets**(*self*)

Updates the *Table* with the various bets. This version creates a *Bet* instance from the *black Outcome*. It uses *Table* `placeBet()` to place that bet.

Passenger57.**win**(*self*, *bet*)

Parameters **bet** (Bet) – The bet which won.

Notification from the *Game* that the *Bet* was a winner. The amount of money won is available via a the `winAmount()` method of `theBet`.

Passenger57.**lose**(*self*, *bet*)

Parameters **bet** (Bet) – The bet which won.

Notification from the *Game* that the *Bet* was a loser.

11.3 Roulette Game Design

class Game

Game manages the sequence of actions that defines the game of Roulette. This includes notifying the *Player* to place bets, spinning the *Wheel* and resolving the *Bet*s actually present on the *Table*.

11.3.1 Fields

`wheel`
The *`Wheel`* that returns a randomly selected *`Bin`* of *`Outcome`*s.

`table`
The *`Table`* which contains the *`Bet`*s placed by the *`Player`*.

`player`
The *`Player`* which creates *`Bet`*s at the *`Table`*.

11.3.2 Constructors

We based the Roulette Game constructor on a design that allows any of the fields to be replaced. This is the **Strategy** design pattern. Each of these collaborating objects is a replaceable strategy, and can be changed by the client that uses this game.

Additionally, we specifically do not include the *`Player`* instance in the constructor. The *`Game`* exists independently of any particular *`Player`*, and we defer binding the *`Player`* and *`Game`* until we are gathering statistical samples.

`Game.`**`__init__`**`(`*self*, *wheel*, *table*`)`

Parameters

- **`wheel`** (Wheel) – The *`Wheel`* instance which produces random events
- **`table`** (Table) – The *`Table`* instance which holds bets to be resolved.

Constructs a new *`Game`*, using a given *`Wheel`* and *`Table`*.

11.3.3 Methods

`cycle``(`*self*, *player*`)`

Parameters **`player`** (Player) – the individual player that places bets, receives winnings and pays losses.

This will execute a single cycle of play with a given *`Player`*. It will

1. call `thePlayer`'s `placeBets()` method to get bets,
2. call `theWheel`'s *`next()`* method to get the next winning *`Bin`*,
3. call `theTable`'s iterator to get an `Iterator` over the *`Bet`*s. Stepping through this `Iterator` returns the individual *`Bet`* objects. If the winning *`Bin`* contains the *`Outcome`*, call the `thePlayer` `win()` method otherwise call the `thePlayer` `lose()` method.

11.4 Roulette Game Questions and Answers

Why are *`Table`* and *`Wheel`* part of the constructor while *`Player`* is given as part of the *`cycle()`* method?

We are making a subtle distinction between the casino table game (a Roulette table, wheel, plus casino staff to support it) and having a player step up to the table and play the game. The game exists without any particular player. By setting up our classes to parallel the physical entities, we give ourselves the flexibility to have multiple players without a significant rewrite. We allow ourselves to support multiple concurrent players or multiple simulations each using a different player object.

Also, as we look forward to the structure of the future simulation, we note that the game objects are largely fixed, but there will be a parade of variations on the player. We would like a main program that simplifies inserting a new player subclass with minimal disruption.

Why do we have to include the odds with the *Outcome* ? This pairing makes it difficult to create an *Outcome* from scratch.

> The odds are an essential ingredient in the *Outcome* . It turns out that we want a short-hand name for each Outcome. We have three ways to provide a short name.
>
> - A variable name. Since each variable is owned by a specific class instance, we need to allocate this to some class. The *Wheel* or the *BinBuilder* make the most sense for owning this variable.
> - A key in a mapping. In this case, we need to allocate the mapping to some class. Again, the *Wheel* or *BinBuilder* make the most sense for owning the mapping.
> - A method which returns the *Outcome* . The method can use a fixed variable or can get a value from a mapping.

11.5 Roulette Game Deliverables

There are three deliverables for this exercise. The stub does not need documentation, but the other classes do need complete Python docstrings.

- The *Passenger57* class. We will rework this design later. This class always places a bet on Black. Since this is simply used to test *Game*, it doesn't deserve a very sophisticated unit test of its own. It will be replaced in a future exercise.
- The `RouletteGame` class.
- A class which performs a demonstration of the *Game* class. This demo program creates the *Wheel*, the stub *Passenger57* and the *Table*. It creates the *Game* object and cycles a few times. Note that the *Wheel* returns random results, making a formal test rather difficult. We'll address this testability issue in the next chapter.

11.6 Appendix: Roulette Variations

In European casinos, the wheel has a single zero. In some casinos, the zero outcome has a special *en prison* rule: all losing bets are split and only half the money is lost, the other half is termed a "push" and is returned to the player. The following design notes discuss the implementation of this additional rule.

This is a payout variation that depends on a single *Outcome*. We will need an additional subclass of *Outcome* that has a more sophisticated losing amount method: it would push half of the amount back to the *Player* to be added to the stake. We'll call this subclass the the `PrisonOutcome` class.

In this case, we have a kind of hybrid resolution: it is a partial loss of the bet. In order to handle this, we'll need to have a `loss()` method in *Bet* as well as a `win()` method. Generally, the `loss()` method does nothing (since the money was removed from the *Player* stake when the bet was created.) However, for the `PrisonOutcome` class, the `loss()` method returns half the money to the *Player*.

We can also introduce a subclass of *BinBuilder* that creates only the single zero, and uses this new `PrisonOutcome` subclass of *Outcome* for that single zero. We can call this the `EuroBinBuilder` . The `EuroBinBuilder` does not create the five-way *Outcome* of 00-0-1-2-3, either; it creates a four-way for 0-1-2-3.

After introducing these two subclasses, we would then adjust *Game* to invoke the `loss()` method of each losing *Bet*, in case it resulted in a push. For an American-style casino, the `loss()` method does nothing. For a European-style casino, the `los()` method for an ordinary *Outcome* also does nothing, but the `los()` for a `PrisonOutcome` would implement the additional rule pushing half the bet back to the *Player* . The special behavior for zero then emerges from the collaboration between the various classes.

We haven't designed the *Player* yet, but we would have to bear this push rule in mind when designing the player.

The uniform interface between *Outcome* and `PrisonOutcome` is a design pattern called *polymorphism*. We will return to this principle many times.

CHAPTER

TWELVE

REVIEW OF TESTABILITY

This chapter presents some design rework and implementation rework for testability purposes. While testability is very important, new programmers can be slowed to a crawl by the mechanics of building test drivers and test cases. We prefer to emphasize the basic design considerations first, and address testability as a feature to be added to a working class.

In *Test Scaffolding* we'll look at the basic software components required to build unit tests.

One approach is to write tests first, then create software that passes the tests. We'll look at this in *Test-Driven Design*.

The application works with random numbers. This is awkward for testing purposes. We'll show one approach to solving this problem in *Capturing Pseudo-Radom Data*.

We'll touch on a few additional topics in *Testability Questions and Answers*.

In *Testability Deliverables* we'll enumerate some deliverables that will improve the overall quality of our application.

We'll look a little more deeply at random numbers in *Appendix: On Random Numbers*.

12.1 Test Scaffolding

Without pausing, we charged past the elephant standing in the saloon. It's time to pause a moment a take a quick glance back at the pachyderm we ignored.

In the *Roulette Game Class* we encouraged creating a stub `Player` and building a test that integrated Game, Table, Wheel, and the stub Player into a kind of working application. This is a kind of *integration test*. We've integrated our various classes into a working whole.

While this integration reflects our overall goals, it's not always the best way to assure that the individual classes work in isolation. We need to refine our approach somewhat.

Back in *Wheel Class* we touched on the problem of testing an application that includes a *random number generator* (RNG). There are two questions raised:

1. How can we develop formalized unit tests when we can't predict the random outcomes? This is a serious testability issue in randomized simulations. This question also arises when considering interactive applications, particularly for performance tests of web applications where requests are received at random intervals.
2. Are the numbers really random? This is a more subtle issue, and is only relevant for more serious applications. Cryptographic applications may care more deeply about the randomness of random numbers. This is a large subject, and well beyond the scope of this book. We'll just assume that our random number generator is good enough.

To address the testing issue, we need to develop some scaffolding that permits more controlled testing. We want to isolate each class so that our testing reveals problems in the class under test.

There are two approaches to replacing the random behavior with something more controlled.

- One approach is to create a mocked implementation of `random.Random` that returns specific outcomes that are appropriate for a given test.

  This is called the *Mock Objects* approach. It's very important for large applications. It allows us to test classes in isolation by creating mocks of their collaborators.

- A second approach is to record the sequence of random numbers actually generated from a particular seed value and use this to define the exected test results. We suggested forcing the seed to be 1 with `wheel.rng.seed(1)`.

12.2 Test-Driven Design

Good testability is achieved when classes are tested in isolation and there are no changes to the class being tested. We have to be careful that our design for *Wheel* works with a real random number generator as well as a mocked version of a random number generator.

An important consequence of this is that we suggested making the random number generator in *Wheel* visible. Rather than have *Wheel* use the `random` module directly, we suggesting creating an instance of `random.Random` as an attribute of *Wheel*.

This design choice reveals a tension between the encapsulation principle and the testability principle.

By *Encapsulation* we mean the design strategy where we define a class to encapsulate the details of it's implementation. It's unclear if the random number generator is an implementation detail or an explicit part of the *Wheel* implementation.

Generally, for must of the normal use cases, the random number generator inside a *Wheel* object is an invisible implementation detail. However, for testing purposes, the random number generator needs to be a configurable feature of the *Wheel* instance.

One approach to making something more visible is to provide default values in the constructor for the object.

Default Initialization

```
class Wheel:
    def __init___(self, rng=None):
        self.rng= rng if rng is not None else random.Random()
        ... rest of Wheel construction ...
```

For this particular situation, this technique is noisy. It introduces a feature that we'll never use outside writing tests.

In languages like Java, with lots of compile-time type-checking, we're forced either to provide an explicit parameter or rely on a complex framework that can inject this kind of change when testing.

Because Python type checking happens at run time, it's much easier to patch this class when writing a unit test. Since we can inject anything as the random number generator in a *Wheel*, our unit tests will look like this:

Mock Object Testing

```
from unittest.mock import MagicMock, Mock
import unittest

class GIVEN_Wheel_WHEN_spin_THEN_return_bin(unittest.TestCase):
    def setUp(self):
        self.bins = [ "bin1", "bin2" ]
        self.wheel = Wheel( self.bins )
        # Patch the random number generator
        self.wheel.rng= Mock()
        self.wheel.rng.choice = Mock( return_value="bin1" )

    def runTest(self):
        value= self.wheel.next()
        self.assertEqual( value, "bin1" )
        self.wheel.rng.choice.assert_called_with( self.bins )

unittest.main()
```

12.3 Capturing Pseudo-Radom Data

The other approach – using a fixed seed – means that we need to build and execute a program that reveals the fixed sequence of spins that are created by the non-random number generator.

We can create an instance of *Wheel*. We can set the random number generator seed to a known value, like `1`.

When can call the `Wheel.next()` method six times, and print the winning *Bin* instances. This sequence will always be the result for a seed value of `1`.

This discovery procedure will reveal results needed to create unit tests for *Wheel* and anything that uses it, for example, *Game*.

12.4 Testability Questions and Answers

Why are we making the random number generator more visible? Isn't object design about encapsulation?

> Encapsulation isn't the same thing as "information hiding" . For some people, the information hiding concept can be a useful way to begin to learn about encapsulation. However, information hiding is misleading because it is often taken to exremes. In this case, we want to encapsulate the bins of the wheel and the procedure for selecting the winning bin into a single object. However, the exact random-number generator (RNG) is a separate component, allowing us to bind any suitable RNG.
>
> Consider the situation where we are generating random numbers for a cryptographic application. In this case, the built-in random number generator may not be random enough. In this case, we may have a third-party Super-Random-Generator that should replace the built-in generator. We would prefer to minimize the changes required to introduce this new class.
>
> Our initial design has isolated the changes to the *Wheel*, but required us to change the constructor. Since we are changing the source code for a class, we must to unit test that change. Further, we are also obligated unit test all of the classes that depend on this class. Changing the source for a class deep within the application forces us to endure the consequence of retesting every class that depends on this deeply buried class. This is too much work to simply replace one object with another.
>
> We do, however, have an alternative. We can change the top-level `main()` method, altering the concrete object instances that compose the working application. By making the change at the top of the application, we don't need to change a deeply buried class and unit test all the classes that depend on the changed class. Instead, we are simply choosing among objects with the same superclass or interface.
>
> This is why we feel that constructors should be made very visible using the various design patterns for *Factories* and *Builders*. Further, we look at the main method as a kind of master *Builder* that assembles the objects that comprise the current execution of our application.
>
> See our *Roulette Solution Questions and Answers* FAQ for more on this subject.
>
> Looking ahead, we will have additional notes on this topic as we add the *SevenReds Player Class* subclass of *Player*.

If setting the seed works so well, why use a mock object?

While setting the seed is an excellent method for setting up a unit test, not everyone is comfortable with random number generators. The presence of an arbitrary but predictable sequence of values looks too much like luck for some *Quality Assurance* (QA) managers. While the algorithms for the random number generator is published and well-understood, this isn't always sufficiently clear and convincing for non-mathematicians. It's often seems simpler to build a non-random mock object than to control the existing class.

12.5 Testability Deliverables

There are two deliverables for this exercise. All of these deliverables need Python docstrings.

- Revised unit tests for `Wheel` using a proper Mock for the random number generator.
- Revised unit tests for `Game` using a proper Mock for the `Wheel`.

12.6 Appendix: On Random Numbers

Random numbers aren't actually "random". Since they are generated by an algorithm, they are sometimes called *pseudo-random*. The distinction is important. Pseudo-random numbers are generated in a fixed sequence from a given *seed value*. Computing the next value in the sequence involves a calculation that is expected to overflow the available number of bits of precision leaving apparently random bits as the next value. This leads to results which, while predictable, are arbitrary enough that they pass rigorous statistical tests and are indistinguishable from data created by random processes.

We can make an application less predictable by choosing a very hard to predict seed value. A popular choice for the seed is the system clock; this isn't ideal.

In most operating systems a special "device" is available for producing random seed values. In Linux this is typically `/dev/random`. In Python, we can access this through the `os.urandom()` function.

As a consequence, we can make an application more predictable by setting a fixed seed value and noting the sequence of numbers generated. We can write a short demonstration program to see the effect of setting a fixed seed. This will also give us a set of predictable answers for unit testing.

CHAPTER

THIRTEEN

PLAYER CLASS

The variations on *Player*, all of which reflect different betting strategies, is the heart of this application. In *Roulette Game Class*, we roughed out a stub class for *Player*. In this chapter, we will complete that design. We will also expand on it to implement the Matingale betting strategy.

We have now built enough infrastructure that we can begin to add a variety of players and see how their betting strategies work. Each player is a betting algorithm that we will evaluate by looking at the player's stake to see how much they win, and how long they play before they run out of time or go broke.

We'll look at the player problem in *Roulette Player Analysis*.

In *Player Design* we'll expand on our previous skeleton *Player* to create a more complete implementation. We'll expand on that again in *Martingale Player Design*.

In *Player Deliverables* we'll enumerate the deliverables for this chapter.

13.1 Roulette Player Analysis

The *Player* has the responsibility to create bets and manage the amount of their stake. To create bets, the player must create legal bets from known *Outcome*s and stay within table limits. To manage their stake, the player must deduct money when creating a bet, accept winnings or pushes, report on the current value of the stake, and leave the table when they are out of money.

We'll look at a number of topics:

- Our overall goal, in *Design Objectives*.
- How we manage the budget, in *Tracking the Stake*.
- How a player interacts with table limits, in *Table Limits*.
- In *Leaving the Table* we'll look at a player retiring when they're ahead. Or broke.
- In *Creating Bets from Outcomes* we'll look at a technical question of tranforming an *Outcome* instance into a *Bet* instance.

We have an interface that was roughed out as part of the design of *Game* and *Table*. In designing *Game*, we put a `placeBets()` method in *Player* to place all bets. We expected the *Player* to create *Bet*s and use the `placeBet()` method of *Table* class to save all of the individual *Bet*s.

In an earlier exercise, we built a stub version of *Player* in order to test *Game*. See *Passenger57 Design*. When we finish creating the final superclass, *Player*, we will also revise our *Passenger57* to be a subclass of *Player*, and rerun our unit tests to be sure that our more complete design still handles the basic test cases correctly.

13.1.1 Design Objectives

Our objective is to have a new abstract class, *`Player`*, with two new concrete subclasses: a revision to *`Passenger57`* and a new player that follows the Martingale betting system.

We'll defer some of the design required to collect detailed measurements for statistical analysis. In this first release, we'll simply place bets.

There are four design issues tied up in *`Player`*: tracking stake, keeping within table limits, leaving the table, and creating bets. We'll tackle them in separate subsections.

13.1.2 Tracking the Stake

One of the more important features we need to add to *`Player`* are the methods to track the player's stake. The initial value of the stake is the player's budget. There are several significant changes to the stake.

- Each bet placed will deduct the bet amount from the *`Player`*'s stake. We are stopped from placing bets when our stake is less than the table minimum.
- Each win will credit the stake. The *`Outcome`* will compute this amount for us.
- Additionally, a push will put the original bet amount back. This is a kind of win with no odds applied.

We'll have to design an interface that will create *`Bet`* s, reducing the stake. and will be used by *`Game`* to notify the *`Player`* of the amount won.

Additionally, we will need a method to reset the stake to the starting amount. This will be used as part of data collection for the overall simulation.'

13.1.3 Table Limits

Once we have our superclass, we can then define the *`Martingale`* player as a subclass. This player doubles their bet on every loss, and resets their bet to a base amount on every win. In the event of a long sequence of losses, this player will have their bets rejected as over the table limit. This raises the question of how the table limit is represented and how conformance with the table limit is assured.

We put a preliminary design in place in *Roulette Table Class*. There are several places where we could isolate this responsibility.

1. The *`Player`* stops placing bets when they are over the *`Table`* limit. In this case, we will be delegating responsibility to the *`Player`* hierarchy. In a casino, a sign is posted on the table, and both players and casino staff enforce this rule. This can be modeled by providing a method in *`Table`* that simply returns the table limit for use by the *`Player`* to keep bets within the limit.
2. The *`Table`* provides a "valid bet" method.
3. The *`Table`* throws an "illegal bet" exception when an illegal bet is placed.

The first alternative is unpleasant because the responsibility to spread around: both *`Player`* and *`Table`* must be aware of a feature of the *`Table`*. This means that a change to the *`Table`* will also require a change to the *`Player`*. This is poor object-oriented design.

The second and third choices reflect two common approaches that are summarized as:

- Ask Permission. The application has code wrapped in `if permitted:` conditional processing.
- Ask Forgiveness. The application assumes that things will work. An exception indicates something unexpected happened.

The general advice is this:

It's better to ask forgiveness than to ask permission.

Most of the time, validation should be handled by raising an exception.

Handling Game State. This raises a question about how we handle advanced games where some bets are not allowed during some game states.

There are two sources of validation for a bet.

- The *Table* may reject a bet because it's over (or under) a limit.
- The *Game* may reject a bet because it's illegal in the current state of the game.

Since these considerations are part of Craps and Blackjack, we'll set them aside for now.

13.1.4 Leaving the Table

We also need to address the issue of the *Player* leaving the game. We can identify a number of possible reasons for leaving: out of money, out of time, won enough, and unwilling to place a legal bet. Since this decision is private to the *Player* , we need a way of alerting the *Game* that the *Player* is finished placing bets.

There are three mechanisms for alerting the *Game* that the *Player* is finished placing bets.

1. Expand the responsibilities of the `placeBets()` to also indicate if the player wishes to continue or is withdrawing from the game. While most table games require bets on each round, it is possible to step up to a table and watch play before placing a bet. This is one classic strategy for winning at blackjack: one player sits at the table, placing small bets and counting cards, while a confederate places large bets only when the deck is favorable. We really have three player conditions: watching, betting and finished playing. It becomes complex trying to bundle all this extra responsibility into the `placeBets()` method.
2. Add another method to *Player* that the *Game* can use to determine if the *Player* will continue or stop playing. This can be used for a player who is placing no bets while waiting; for example, a player who is waiting for the Roulette wheel to spin red seven times in a row before betting on black.
3. The *Player* can throw an exception when they are done playing. This is an exceptional situation: it occurs exactly once in each simulation. However, it is a well-defined condition, and doesn't deserve to be called "exceptional" . It is merely a terminating condition for the game.

We recommend adding a method to *Player* to indicate when *Player* is done playing. This gives the most flexibility, and it permits *Game* to cycle until the player withdraws from the game.

A consequence of this decision is to rework the *Game* class to allow the player to exit. This is relatively small change to interrogate the *Player* before asking the player to place bets.

Note: Design Evolution

In this case, these were situations which we didn't discover during the initial design. It helped to have some experience with the classes in order to determine the proper allocation of responsibilities. While design walkthroughs are helpful, an alternative is a "prototype", a piece of software that is incomplete and can be disposed of. The earlier exercise created a version of *Game* that was incomplete, and a version of `PlayerStub` that will have to be disposed of.

13.1.5 Creating Bets from Outcomes

Generally, a *Player* will have a few *Outcome*s on which they are betting. Many systems are similar to the Martingale system, and place bets on only one of the *Outcome*s. These *Outcome* objects are usually created during player initialization. From these *Outcome*s, the *Player* can create the individual *Bet* instances based on their betting strategy.

Since we're currently using the *Wheel* as a repository for all legal *Outcome* instances, we'll need to provide the *Wheel* to the *Player*.

This doesn't generalize well for Craps or Blackjack. We'll need to revisit this design decision. In the long run, we'll need to find another kind of factory for creating proper *`Outcome`* instances.

13.2 Player Design

class `Player`

We'll design the base class of *`Player`* and a specific subclass, *`Martingale`*. This will give us a working player that we can test with.

`Player` places bets in Roulette. This an abstract class, with no actual body for the `placeBets()` method. However, this class does implement the basic `win()` method used by all subclasses.

13.2.1 Fields

`Player.`**`stake`**
The player's current stake. Initialized to the player's starting budget.

`Player.`**`roundsToGo`**
The number of rounds left to play. Initialized by the overall simulation control to the maximum number of rounds to play. In Roulette, this is spins. In Craps, this is the number of throws of the dice, which may be a large number of quick games or a small number of long-running games. In Craps, this is the number of cards played, which may be large number of hands or small number of multi-card hands.

`Player.`**`table`**
The *`Table`* used to place individual *`Bet`*s. The *`Table`* contains the current *`Wheel`* from which the player can get `Outcomes` used to build *`Bet`*s.

13.2.2 Constructors

`Player.`**`__init__`**`(`*self*, *table*`)`
Constructs the *`Player`* with a specific *`Table`* for placing *`Bet`*s.

Parameters **`table`** (Table) – the table to use

Since the table has access to the *`Wheel`*, we can use this wheel to extract :class'Outcome' objects.

13.2.3 Methods

`Player.`**`playing`**`(`*self*`)` → boolean
Returns `true` while the player is still active.

`Player.`**`placeBets`**`(`*self*`)`
Updates the *`Table`* with the various *`Bet`* s.

When designing the *`Table`*, we decided that we needed to deduct the amount of a bet from the stake when the bet is created. See the Table *Roulette Table Analysis* for more information.

`Player.`**`win`**`(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which won

Notification from the *`Game`* that the *`Bet`* was a winner. The amount of money won is available via `bet` method `winAmount()`.

`Player.`**`lose`**`(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which won

Notification from the *Game* that the *Bet* was a loser. Note that the amount was already deducted from the stake when the bet was created.

13.3 Martingale Player Design

class Martingale

Martingale is a *Player* who places bets in Roulette. This player doubles their bet on every loss and resets their bet to a base amount on each win.

13.3.1 Fields

Martingale.**lossCount**

The number of losses. This is the number of times to double the bet.

Martingale.**betMultiple**

The the bet multiplier, based on the number of losses. This starts at 1, and is reset to 1 on each win. It is doubled in each loss. This is always equal to $2^{lossCount}$.

13.3.2 Methods

Martingale.**placeBets**(*self*)

Updates the *Table* with a bet on “black”. The amount bet is $2^{lossCount}$, which is the value of *betMultiple*.

Martingale.**win**(*self*, *bet*)

Parameters **bet** (Bet) – The bet which won

Uses the superclass *win()* method to update the stake with an amount won. This method then resets *lossCount* to zero, and resets *betMultiple* to 1.

Martingale.**lose**(*self*, *bet*)

Parameters **bet** (Bet) – The bet which won

Uses the superclass *lose()* to do whatever bookkeeping the superclass already does. Increments *lossCount* by 1 and doubles *betMultiple*.

13.4 Player Deliverables

There are six deliverables for this exercise. The new classes must have Python docstrings.

- The *Player* abstract superclass. Since this class doesn’t have a body for the `placeBets()`, it can’t be unit tested directly.
- A revised *Passenger57* class. This version will be a proper subclass of *Player*, but still place bets on black until the stake is exhausted. The existing unit test for *Passenger57* should continue to work correctly after these changes.
- The *Martingale* subclass of *Player*.
- A unit test class for *Martingale*. This test should synthesize a fixed list of *Outcome* s, *Bin* s, and calls a *Martingale* instance with various sequences of reds and blacks to assure that the bet doubles appropriately on each loss, and is reset on each win.

- A revised *Game* class. This will check the player's `playing()` method before calling `placeBets()`, and do nothing if the player withdraws. It will also call the player's `win()` and `lose()` methods for winning and losing bets.
- A unit test class for the revised *Game* class. Using a non-random generator for *Wheel*, this should be able to confirm correct operation of the *Game* for a number of bets.

CHAPTER

FOURTEEN

OVERALL SIMULATION CONTROL

We can now use our application to generate some more usable results. We can perform a number of simulation runs and evaluate the long-term prospects for the Martingale betting system. We want to know a few things about the game:

- How long can we play with a given budget? In other words, how many spins before we've lost our stake.
- How much we can realistically hope to win? How large a streak can we hope for? How far ahead can we hope to get before we should quit?

In the *Simulation Analysis* section, we'll look at general features of simulation.

We'll look at the resulting statistical data in *Statistical Summary*.

This will lead us to the details in *Simulator Design*. We'll note that the details of the simulator require some changes to the definition of the `Player`. We'll look at this in *Player Rework*.

We'll enumerate all of this chapter's deliverables in *Simulation Control Deliverables*.

14.1 Simulation Analysis

A `Simulator` class will be allocated a number of responsibilities:

- Create the `Wheel`, `Table` and `Game` objects.
- Simulate a number of sessions (typically 100), saving the maximum stake and length of each session.
- For each session: initialize the `Player` and `Game` , cycle the game a number of times, collect the size of the `Player` 's stake after each cycle.
- Write a final summary of the results.

We'll look at several topics:

- *Simulation Terms* will formalize some definitions.
- *Simulation Control* will look at the overall simulation process.
- We'll look at how we create a new player in *Player Initialization*.
- The most important thing is gather performance data. We'll look at this in *Player Interrogation*.

14.1.1 Simulation Terms

We'll try to stick to the following definitions. This will help structure our data gathering and analysis.

cycle

A single cycle of betting and bet resolution. This depends on a single random event: a spin of the wheel or a throw of the dice. Also known as a round of play.

session

One or more cycles. The session begins with a player having their full stake. A session ends when the play elects to leave or can no longer participate. A player may elect to leave because of elapsed time (typically 250 cycles), or they have won a statistically significant amount. A player can no longer participate when their stake is too small to make the minimum bet for the table.

game

Some games have intermediate groupings of events between an individual cycles and an entire session. Blackjack has *hands* where a number of player decisions and a number of random events contribute to the payoff. Craps has a *game*, which starts with the dice roll when the point is *off*, and ends when the point is made or the shooter gets *Craps*; consequently, any number of individual dice rolls can make up a game. Some bets are placed on the overall game, while others are placed on individual dice rolls.

14.1.2 Simulation Control

The sequence of operations for the simulator looks like this.

Controlling the Simulation

1. **Empty List of Maxima**. Create an empty maxima list. This is the maximum stake at the end of each session.
2. **Empty List of Durations**. Create an empty durations list. This is the duration of each session, measured in the number of cycles of play before the player withdrew or ran out of money.
3. **For All Sessions**. For each of 100 sessions:

   Empty List of Stake Details. Create an empty list to hold the history of stake values for this session. This is raw data that we will summarize into two metrics for the session: maximum stake and duration. We could also have two simple variables to record the maximum stake and count the number of spins for the duratioon. However, the list allows us to gather other statistics, like maximum win or maximum loss.

   While The Player Is Active.

      Play One Cycle. Play one cycle of the game. See the definition in *Roulette Game Class*.

      Save Outcomes. Save the player's current stake in the list of stake values for this session. An alternative is to update the maximum to be the larger of the current stake and the maximum, and increment the duration.

   Get Maximum. Get the maximum stake from the list of stake values. Save the maximum stake metric in the maxima list.

   Get Duration. Get the length of the list of stake values. Save the duration metric in the durations list. Durations less than the maximum mean the strategy went bust.

4. **Statistical Description of Maxima**. Compute the average and standard deviation of the values in the maxima list.
5. **Statistical Description of Durations**. Compute the average and standard deviation of values in the durations list.

Both this overall `Simulator` and the `Game` collaborate with the `Player`. The `Simulator`'s collaboration, however, initalizes the `Player` and then monitors the changes to the `Player`'s stake. We have to design two interfaces for this collaboration.

- Initialization
- Interrogation

14.1.3 Player Initialization

The *`Simulator`* will initialize a *`Player`* for 250 cycles of play, assuming about one cycle each minute, and about four hours of patience. We will also initialize the player with a generous budget of the table limit, 100 betting units. For a $10 table, this is $1,000 bankroll.

Currently, the *`Player`* class is designed to play one session and stop when their duration is reached or their stake is reduced to zero. We have two alternatives for reinitializing the *`Player`* at the beginning of each session.

1. Provide some *setters* that allow a client class like this overall simulator control to reset the `stake` and `roundsToGo` values of a *`Player`*.
2. Provide a **Factory** that allows a client class to create new, freshly initialized instances of *`Player`*.

While the first solution is quite simple, there are some advantages to creating a `PlayerFactory`. If we create an **Abstract Factory**, we have a single place that creates all *`Player`*s.

Further, when we add new player subclasses, we introduce these new subclasses by creating a new subclass of the factory. In this case, however, only the main program creates instances of *`Player`*, reducing the value of the factory. While design of a **Factory** is a good exercise, we can scrape by with adding setter methods to the *`Player`* class.

14.1.4 Player Interrogation

The *`Simulator`* will interrogate the *`Player`* after each cycle and capture the current stake. An easy way to manage this detailed data is to create a `List` that contains the stake at the end of each cycle. The length of this list and the maximum value in this list are the two metrics the *`Simulator`* gathers for each session.

Our list of maxima and durations are created sequentially during the session and summarized sequentially at the end of the session. A `list` will do everything we need. For a deeper discussion on the alternatives available in the collections framework, see *Design Decision – Choosing A Collection*.

14.2 Statistical Summary

The *`Simulator`* will interrogate the *`Player`* after each cycle and capture the current stake. We don't want the sequence of values for each cycle; we want a summary of all the cycles in the session. We can save the length of the sequence as well as the maximum of the sequence. We can then calculate aggregate performance parameters for each session.

Our objective is to run several session simulations to get averages and a standard deviations for duration and maximum stake. This means that the *`Simulator`* needs to retain these statistical samples. We will defer the detailed design of the statistical processing, and simply keep the duration and maximum values in lists for this first round of design.

14.3 Simulator Design

class `Simulator`

`Simulator` exercises the Roulette simulation with a given *`Player`* placing bets. It reports raw statistics on a number of sessions of play.

14.3.1 Fields

`Simulator.`**`initDuration`**

The duration value to use when initializing a *`Player`* for a session. A default value of 250 is a good choice here.

`Simulator.`**`initStake`**
The stake value to use when initializing a *`Player`* for a session. This is a count of the number of bets placed; i.e., 100 $10 bets is $1000 stake. A default value of 100 is sensible.

`Simulator.`**`samples`**
The number of game cycles to simulate. A default value of 50 makes sense.

`Simulator.`**`durations`**
A `List` of lengths of time the *`Player`* remained in the game. Each session of play producrs a duration metric, which are collected into this list.

`Simulator.`**`maxima`**
A `List` of maximum stakes for each *`Player`*. Each session of play producrs a maximum stake metric, which are collected into this list.

`Simulator.`**`player`**
The *`Player`*; essentially, the betting strategy we are simulating.

`Simulator.`**`game`**
The casino game we are simulating. This is an instance of *`Game`*, which embodies the various rules, the *`Table`* and the *`Wheel`*.

14.3.2 Constructors

`Simulator.`**`__init__`**`(`*self*, *game*, *player*`)`
Saves the *`Player`* and *`Game`* instances so we can gather statistics on the performance of the player's betting strategy.

Parameters

- **`game`** (Game) – The Game we're simulating. This includes the *`Table`* and *`Wheel`*.
- **`player`** (Player) – The Player. This encapsulates the betting strategy.

14.3.3 Methods

`Simulator.`**`session`**`(`*self*`)` → list

Returns `list` of stake values.

Return type list

Executes a single game session. The *`Player`* is initialized with their initial stake and initial cycles to go. An empty `List` of stake values is created. The session loop executes until the *`Player`* `playing()` returns false. This loop executes the *`Game cycle()`*; then it gets the stake from the *`Player`* and appends this amount to the `List` of stake values. The `List` of individual stake values is returned as the result of the session of play.

`Simulator.`**`gather`**`(`*self*`)`
Executes the number of games sessions in *`samples`*. Each game session returns a `List` of stake values. When the session is over (either the play reached their time limit or their stake was spent), then the length of the session `List` and the maximum value in the session `List` are the resulting duration and maximum metrics. These two metrics are appended to the *`durations`* list and the *`maxima`* list.

A client class will either display the durations and maxima raw metrics or produce statistical summaries.

14.4 Player Rework

The current design for the *`Player`* doesn't provide all the methods we need.

We'll can add two new methods: one will set the `stake` and the other will set the `roundsToGo`.

`Player.`**`setStake`**`(`*self*, *stake*`)`

> **Parameters** **`stake`** (*integer*) – the Player's initial stake

`Player.`**`setRounds`**`(`*self*, *rounds*`)`

> **Parameters** **`rounds`** (*integer*) – the Player's duration of play

14.5 Simulation Control Deliverables

There are five deliverables for this exercise. Each of these classes needs complete Python docstring comments.

- Revision to the *`Player`* class. Don't forget to update unit tests.
- The *`Simulator`* class.
- The expected outcomes from the non-random wheel can be rather complex to predict. Because of this, one of the deliverables is a demonstration program that enumerates the actual sequence of non-random spins. From this we can derive the sequence of wins and losses, and the sequence of *`Player`* bets. This will allow us to predict the final outcome from a single session.
- A unit test of the *`Simulator`* class that uses the non-random generator to produce the predictable sequence of spins and bets.
- A main application function that creates the necessary objects, runs the *`Simulator`*'s `gather()` method, and writes the available outputs to `sys.stdout`

For this initial demonstration program, it should simply print the list of maxima, and the list of session lengths. This raw data can be redirected to a file, loaded into a spreadsheet and analyzed.

CHAPTER

FIFTEEN

SEVENREDS PLAYER CLASS

This section introduces an additional specialization of the Martingale strategy. Additionally, we'll also address some issues in how an overall application is composed of individual class instances. Adding this new subclass should be a small change to the main application class.

We'll also revisit a question in the design of the overall `Table`. Should we really be checking for a minimum? Or was that needless?

In *SevenReds Player Analysis* we'll examine the general strategy this player will follow.

We'll revisit object-oriented design by composition in *Soapbox on Composition*.

In *SevenReds Design* we'll look at the design of this player. We'll need to revise the overall design for the abstract `Player`, also, which we'll look at in *Player Rework*.

These design changes will lead to other changes. We'll look at these changes in *Game Rework* and *Table Rework*.

This will lead to *SevenReds Player Deliverables*, which enumerates all of the deliverables for this chapter.

15.1 SevenReds Player Analysis

The `SevenReds` player waits for seven red wins in a row before betting black. This is a subclass of `Player`. We can create a subclass of our main `Simulator` to use this new `SevenReds` class.

We note that `Passenger57`'s betting is stateless: this class places the same bets over and over until they are cleaned out or their playing session ends.

The `Martingale` player's betting, however, is stateful. This player changes the bet based on wins and losses. The state is a loss counter than resets to zero on each win, and increments on each loss.

Our `SevenReds` player will have two states: waiting and betting. In the waiting state, they are simply counting the number of reds. In the betting state, they have seen seven reds and are now playing the Martingale system on black. We will defer serious analysis of this *stateful* betting until some of the more sophisticated subclasses of `Player`. For now, we will simply use an integer to count the number of reds.

15.1.1 Game Changer

Currently, a `Player` is not informed of the final outcome unless they place a bet. We designed the `Game` to evaluate the `Bet` instances and notify the `Player` of just their `Bet`s that were wins or losses. We will need to add a method to `Player` to be given the overall list of winning `Outcome`s even when the `Player` has not placed a bet.

Once we have updated the design of `Game` to notify `Player`, we can add the new `SevenReds` class. Note that we are can intoduce each new betting strategy via creation of new subclasses. A relatively straightforward update to our simulation main program allows us to use these new subclasses. The previously working subclasses are left in place, allowing graceful evolution by adding features with minimal rework of existing classes.

In addition to waiting for the wheel to spin seven reds, we will also follow the Martingale betting system to track our wins and losses, assuring that a single win will recoup all of our losses. This makes *`SevenReds`* a further specialization of *`Martingale`*. We will be using the basic features of *`Martingale`*, but doing additional processing to determine if we should place a bet or not.

Introducing a new subclass should be done by upgrading the main program. See *Soapbox on Composition* for comments on the ideal structure for a main program. Additionally, see the *Roulette Solution Questions and Answers* FAQ entry.

15.1.2 Table Changes

When we designed the table, we included a notion of a valid betting state. We required that the sum of all bets placed by a *`Player`* was below some limit. We also required that there be a table minimum present.

A casino has a table minimum for a variety of reasons. Most notably, it serves to distinguish casual players at low-stakes tables from "high rollers" who might prefer to play with other people who wager larger amounts.

In the rare event that a player is the only person at a roulette wheel, the croupier won't spin the wheel until a bet is placed. This is an odd thing. It's also very rare. Pragmatically, there are almost always other players, and the wheel will be spun even if a given player is not betting.

Our design for a table really should **not** have any check for a minimum bet. It's a rule that doesn't make sense for the kind of simulation we're doing.

For this particular player, it's essential that the wheel is spun even if the player has no bets in play.

15.2 Soapbox on Composition

Generally, a solution is composed of a number of objects. However, the consequences of this are often misunderstood. Since the solution is a composition of objects, it falls on the main method to do just the composition and nothing more.

Our ideal main program creates and composes the working set of objects. In this case, it should decode the command-line parameters, and use this information to create and initialize the simulation objects, then start the processing. For these simple exercises, however, we're omitting the parsing of command-line parameters, and simply creating the necessary objects directly.

A main program should, therefore, look something like the following:

```
theWheel= Wheel()
theTable= Table()
theGame= Game( theWheel, theTable )
thePlayer= SevenReds( theTable )
sim= new Simulator( theGame, thePlayer )
sim.gather()
```

We created an instance of *`Wheel`* which has the bins and outcomes. We created an instance of *`Table`* which has a place to put the bets. We've unified these two through an instance of *`Game`* which depends on the wheel and the table.

When we created the `thePlayer`, we could have used *`Martingale`* or *`Passenger57`*. The player object can use the table to get the wheel instance. This wheel instance provides the outcomes used to build bets.

We have assembled the overall simulation by composition of a Wheel, the Table and the specific Player algorithm.

The real work is done by `Simulater.gather()`. This relies on the game, table, and player to create some data we can analyze.

In some instances, the construction of objects is not done directly by the main method. Instead, the main method will use **Builders** to create the various objects. The idea is to avoid mentioning the class definitions directly. We can upgrade or replace a class, and also upgrade the **Builder** to use that class appropriately. This isolates change to the class hierarchy and a builder function.

15.3 SevenReds Design

class `SevenReds`

`SevenReds` is a *`Martingale`* player who places bets in Roulette. This player waits until the wheel has spun red seven times in a row before betting black.

15.3.1 Fields

`SevenReds.`**`redCount`**

The number of reds yet to go. This starts at `7` , is reset to `7` on each non-red outcome, and decrements by `1` on each red outcome.

Note that this class inherits `betMultiple`. This is initially `1`, doubles with each loss and is reset to one on each win.

15.3.2 Methods

`SevenReds.`**`placeBets`**`(`*self*`)`

If *`redCount`* is zero, this places a bet on black, using the bet multiplier.

`SevenReds.`**`winners`**`(`*self*, *outcomes*`)`

Parameters **`outcomes`** (*Set of Outcome*) – The *`Outcome`* set from a *`Bin`*.

This is notification from the *`Game`* of all the winning outcomes. If this vector includes red, *`redCount`* is decremented. Otherwise, *`redCount`* is reset to `7`.

15.4 Player Rework

We'll need to revise the *`Player`* class to add the following method. The superclass version doesn't do anything with this information. Some subclasses, however, will process this.

`Player.`**`winners`**`(`*self*, *outcomes*`)`

Parameters **`outcomes`** (*Set of Outcome*) – The set of *`Outcome`* instances that are part of the current win.

The game will notify a player of each spin using this method. This will be invoked even if the player places no bets.

15.5 Game Rework

We'll need to revise the *`Game`* class to extend the cycle method. This method must provide the winning bin's *`Outcome`* set.

15.6 Table Rework

We'll need to revise the *`Table`* class to remove any minimum bet rule. If there are no bets, the game should still proceed.

15.7 SevenReds Player Deliverables

There are six deliverables from this exercise. The new classes will require complete Python docstrings.

- A revision to the *Player* to add the *Player.winners()* method. The superclass version doesn't do anything with this information. Some subclasses, however, will process this.
- A revision to the *Player* unit tests.
- A revision to the *Game* class. This will call the `winners()` with the winning *Bin* instance before paying off the bets.
- A revision to the *Table* class. This will allow a table with zero bets to be considered valid for the purposes of letting the game continue.
- The *SevenReds* subclass of *Player*.
- A unit test of the *SevenReds* class. This test should synthesize a fixed list of *Outcome*s, *Bin*s and the call a *SevenReds* instance with various sequences of reds and blacks. One test cases can assure that no bet is placed until 7 reds have been seen. Another test case can assure that the bets double (following the Martingale betting strategy) on each loss.
- A main application function that creates the necessary objects, runs the *Simulator*'s `gather()` method, and writes the available outputs to `sys.stdout`

For this initial demonstration program, it should simply print the list of maxima, and the list of session lengths. This raw data can be redirected to a file, loaded into a spreadsheet and analyzed.

CHAPTER

SIXTEEN

STATISTICAL MEASURES

The *Simulator* class collects two `Lists`. One has the length of a session, the other has the maximum stake during a session. We need to create some descriptive statistics to summarize these stakes and session lengths.

This section presents two ordinary statistical algorithms: mean and standard deviation. Python 3 introduces the `statistics` module, which can slightly simplify some of the programming required for this chapter.

We'll present the details of how to compute these two statistics for those who are interested.

In *Statistical Analysis* we'll address the overall goal of gathering and analyzing statistics.

In *Some Foundations* we'll look at the Σ operator which is widely used for stastical calculation. We'll see how to implement this in Python.

In *Statistical Algorithms* we'll look specifically at mean and standard deviation.

Since we're apply stastical calculation to a list of integer values, we'll look at how we can extend the `list` in *IntegerStatistics Design*.

We'll enumerate the deliverables for this chapter in *Statistics Deliverables*.

On Statistics

In principle, we could apply basic probability theory to predict the statistics generated by this simulation. Indeed, some of the statistics we are gathering are almost matters of definition, rather than actual unknown data. We are, however, much more interested in the development of the software than we are in applying probability theory to a relatively simple casino game. As a consequence, the statistical analysis here is a little off-base.
Our goal is to build a relatively simple-minded set of descriptive statistics. We will average two metrics: the peak stake for a session and the duration of the session. Both of these values have some statistical problems. The peak stake, for instance, is already a summary number. We have to be cautious in our use of summarized numbers, since we have lost any sense of the frequency with which the peaks occured. For example, a player may last 250 cycles of play, with a peak stake of 130. We don't know if that peak occurred once or 100 times out of that 250. Additionally, the length of the session has an upper bound on the number of cycles. This forces the distribution of values to be skewed away from the predictable distribution.

16.1 Statistical Analysis

We will design a `Statistics` class with responsibility to retain and summarize a list of numbers and produce the average (also known as the mean) and standard deviation. The *Simulator* can then use this this `Statistics` class to get an average of the maximum stakes from the list of session-level measures. The *Simulator* can also apply this `Statistics` class to the list of session durations to see an average length of game play before going broke.

We have three design approaches for encapsulating processing:

- We can extend an existing class, or

- We can wrap an existing class, creating a whole new kind of thing, or
- We can delegate the statistical functions to a separate function. This is how thing currently stand in Python. We have a build-in list class and a separate `statitics` module.

A good approach is to extend the built-in list with statistical summary features. Given this new class, we can replace the original `List` of sample values with a `StatisticalList` that both saves the values and computes descriptive statistics.

This design has the most reuse potential. This can be used in a variety of contexts, and allows us the freedom to switch `List` implementation classes without any other changes.

The detailed algorithms for mean and standard deviation are provided in *Statistical Algorithms*.

16.2 Some Foundations

For those programmers new to statistics, this section covers the Sigma operator, Σ.

$$\sum_{i=0}^{n} f(i)$$

The Σ operator has three parts to it. Below it is a bound variable, *i*, and the starting value for the range, written as $i = 0$. Above it is the ending value for the range, usually something like *n*. To the right is some function to execute for each value of the bound variable. In this case, a generic function, $f(i)$ is shown. This is read as “sum *f* (*i*) for *i* in the range 0 to *n*”.

One common definition of Σ uses a closed range, including the end values of 0 and *n* . However, since this is not a helpful definition for software, we will define Σ to use a half-open interval. It has exactly *n* elements, including 0 and *n* -1; mathematically, $0 \leq i < n$.

Consequently, we prefer the following notation, but it is not often used. Since statistical and mathematical texts often used 1-based indexing, some care is required when translating formulae to programming languages that use 0-based indexing.

$$\sum_{0 \leq i < n} f(i)$$

Our two statistical algorithms have a form more like the following function. In this we are applying some function, f, to each value, x_i of an list, x.

When computing the mean, as a special case, there is no function applied to the values in the list. When computing standard deviation, the function involves subtracting and multiplying.

$$\sum_{0 \leq i < n} f(x_i)$$

We can transform this definition directly into a for loop that sets the bound variable to all of the values in the range, and does some processing on each value of the List of Integers.

This is the Python implementation of Sigma. This computes two values, the sum, `s` and the number of elements, `n`.

```
s= sum( theList )
n = len( theList )
```

When computing the standard deviation, we do something like the following

```
s= sum( f(x) for x in theList )
n = len( theList )
```

Where the `f(x)` calculuation computes the measure of deviation from the average.

16.3 Statistical Algorithms

We'll look at two important algorithms:

- *mean*, and
- *standard deviation*.

16.3.1 Mean

Computing the mean of a list of values is relatively simple. The mean is the sum of the values divided by the number of values in the list. Since the statistical formula is so closely related to the actual loop, we'll provide the formula, followed by an overview of the code.

$$\mu_x = \frac{\sum\limits_{0 \leq i < n} x_i}{n}$$

The definition of the Σ mathematical operator leads us to the following method for computing the mean:

```
sum(self)/len(self)
```

This matches the mathematical definition nicely.

16.3.2 Standard Deviation

The standard deviation can be done a few ways. We'll use the formula shown below. This computes a deviation measurement as the square of the difference between each sample and the mean.

The sum of these measurements is then divided by the number of values times the number of degrees of freedom to get a standardized deviation measurement.

Again, the formula summarizes the loop, so we'll show the formula followed by an overview of the code.

$$\sigma_x = \sqrt{\frac{\sum\limits_{0 \leq i < n} (x_i - \mu_x)^2}{n-1}}$$

The definition of the Σ mathematical operator leads us to the following method for computing the standard deviation:

We can use a generator expression for this.

```
m = mean(x)
math.sqrt( sum( (x-m)**2 for x in self ) / (len(self)-1) )
```

This seems to match the mathematical definition nicely.

16.4 IntegerStatistics Design

class IntegerStatistics(*list*)

IntegerStatistics computes several simple descriptive statistics of `Integer` values in a `List`.

This extends list with some additional methods.

16.4.1 Constructors

Since this class extends the built-in list, we'll leverage the existing constructor.

16.4.2 Methods

`IntegerStatistics.`**`mean`**`(`*self*`)`

Computes the mean of the `List` of values.

`IntegerStatistics.`**`stdev`**`(`*self*`)`

Computes the standard deviation of the `List` values.

16.5 Statistics Deliverables

There are three deliverables for this exercise. These classes will include the complete Python dostring.

- The *`IntegerStatistics`* class.
- A unit test of the *`IntegerStatistics`* class.

  Prepare some simple list (or tuple) of test data.

  The results can be checked with a spreadsheet
- An update to the overall *`Simulator`* that gets uses an *`IntegerStatistics`* object to compute the mean and standard deviation of the peak stake. It also computest the mean and standard deviation of the length of each session of play.

Here is some standard deviation unit test data.

Sample Value
10
8
13
9
11
14
6
4
12
7
5

Here are some intermediate results and the correct answers given to 6 significant digits. Your answers should be the same to the precision shown.

sum	99
count	11
mean	9.0
sum (x-m)2**	110.0
stdev	3.317

CHAPTER SEVENTEEN

RANDOM PLAYER CLASS

This section will introduce a simple subclass of `Player` who bets at random.

In *Random Player Analysis* we'll look at what this player does.

We'll turn to how the player works in *Random Player Design*.

In *Random Player Deliverables* we'll enumerate the deliverables for this player.

An important consideration is to compare this player with the player who always bets black and the player using the Martigale strategy to always bet black. Who does better? If they're all about the same, what does that say about the house edge in this game?

17.1 Random Player Analysis

One possible betting strategy is to bet completely randomly. This serves as an interesting benchmark for other betting strategies.

We'll write a subclass of `Player` which steps through all of the bets available on the `Wheel`, selecting one or more of the available outcomes at random. This `Player`, like others, will have a fixed initial stake and a limited amount of time to play.

The `Wheel` class can provide an `Iterator` over the collection of `Bin` instances. We could revise `Wheel` to provide a `binIterator()` method that we can use to return all of the `Bin`s. From each `Bin`, we will need an iterator we can use to return all of the `Outcome`s.

To collect a list of all possible `Outcome`s, we would use the following algorithm:

Locating all Outcomes

1. **Empty List of Outcomes**. Create an empty set of all `Outcome`s, `all_OC`.
2. **Get Bin Iterator**. Get the Iterator from the `Wheel` that lists all `Bin`s.
3. **For each Bin**.

   Get Outcome Iterator. Get the Iterator that lists all `Outcome`s.

   For each Outcome.

      Save Outcome. Add each `Outcome` to the set of all known outcomes, `all_OC`.

To place a random bet, we would use the following algorithm:

Placing a Random Bet

1. Get the size of the pool of all possible `Outcome`s, s.
2. Get a random number, u, from zero to the $s-1$. That is, $0 \leq u < s$. This is readily available in `random.randrange()`.
3. Return element u from the pool of `Outcome`s.

17.2 Random Player Design

class `PlayerRandom`

`PlayerRandom` is a `Player` who places bets in Roulette. This player makes random bets around the layout.

17.2.1 Fields

`PlayerRandom.`**`rng`**

A Random Number Generator which will return the next random number.

When writing unit tests, we will want to patch this with a mock object to return a known sequence of bets.

17.2.2 Constructors

`PlayerRandom.`**`__init__`**`(`*table*`)`

This uses the `super()` construct to invoke the superclass constructor using the `Table`.

Parameters **`table`** (Table) – The `Table` which will accept the bests.

This will create a `random.Random` random number generator.

It will also use the wheel associated with the table to get the set of bins. The set of bins is then used to create the pool of outcomes for creating bets.

17.2.3 Methods

`PlayerRandom.`**`placeBets`**`(`*self*`)`

Updates the `Table` with a randomly placed bet.

17.3 Random Player Deliverables

There are five deliverables from this exercise. The new classes need Python docstrings.

- Updates to the class `Bin` to return an iterator over available `Outcome`s. Updates to unittests for the class `Bin`, also.
- Updates to the `Wheel` to return an iterator over available `Bin`s. Updates to the unittests for the class `Wheel`, also.
- The `PlayerRandom` class.
- A unit test of the `PlayerRandom` class. This should use the NonRandom number generator to iterate through all possible `Outcome`s.
- An update to the overall `Simulator` that uses the `PlayerRandom`.

CHAPTER

EIGHTEEN

PLAYER 1-3-2-6 CLASS

This section will describe a player who has a complex internal state. We will also digress on the way the states can be modeled using a group of polymorphic classes.

We'll start by examining the essential 1-3-2-6 betting strategy in *Player 1-3-2-6 Analysis*.

This will lead us to considering stateful design and how to properly handle polymorphism. This is the subject of *On Polymorphism*.

We'll address some additional design topics in *Player 1-3-2-6 Questions and Answers*.

The design is spread across several sections:

- *Player1326 State Design*,
- *Player1326 No Wins Design*,
- *Player1326 One Win Design*,
- *Player1326 Two Wins Design*, and
- *Player1326 Three Wins_*.

Once we've defined the various states, the overall player can be covered in *Player1326 Design*.

We'll enumerate the deliverables in *Player 1-3-2-6 Deliverables*.

There are some more advanced topics here. First, we'll look at alternative designs in *Advanced Exercise – Refactoring*. Then, we'll look at ways to reuse state objects in *Advanced Exercise – Less Object Creation*. Finally, we'll examine the **Factory** design pattern in *Player1326 State Factory Design*.

18.1 Player 1-3-2-6 Analysis

On the Internet, we found descriptions of a betting system called the "1-3-2-6" system. This system looks to recoup losses by waiting for four wins in a row. The sequence of numbers (1, 3, 2 and 6) are the multipliers to use when placing bets after winning.

At each loss, the sequence resets to the multiplier of 1.

At each win, the multiplier is advanced. It works like this:

- After one win, the bet advances to 3×. This is done by leaving the winnings in place, and adding to them.
- After a second win, the bet is reduced to 2×, and the winnings of 4× are taken off the table.
- In the event of a third win, the bet is advanced to 6× by putting the 4× back into play.

Should there be a fourth win, the sequence resets to 1×. The winnings from the 6× bet are the hoped-for profit.

This betting system makes our player more stateful than in previous betting systems. When designing `SevenReds`, we noted that this player was stateful; that case, the state was a simple count.

In this case, the description of the betting system seems to identify four states: no wins, one win, two wins, and three wins. In each of these states, we have specific bets to place, and state transition rules that tell us what to do next. The following table summarizes the states, the bets and the transition rules.

Table 18.1: 1-3-2-6 Betting States

Current State	Bet Amount	On loss, change to:	On win, change to:
No Wins	1	One Win	No Wins
One Win	3	Two Wins	No Wins
Two Wins	2	Three Wins	No Wins
Three Wins	6	No Wins	No Wins

When we are in a given state, the table gives us the amount to bet in the *Bet* column. If this bet wins, we transition to the state named in the *On Win* column, otherwise, we transition to the state named in the *On Loss* column. We always start in the No *Wins* state.

Design Pattern. We can exploit the **State** design pattern to implement this more sophisticated player. The design pattern suggests that we create a hierarchy of classes to represent these four states. Each state will have a slightly different bet amount, and different state transition rules. Each individual state class will be relatively simple, but we will have isolated the processing unique to each state into separate classes.

One of the consequences of the **State** design pattern is that it obligates us to define the interface between the *`Player`* and the object that holds the *`Player`*'s current state.

It seems best to have the state object follow our table and provide three methods: `currentBet()`, `nextWon()`, and `nextLost()`. The *`Player`* can use these methods of the state object to place bets and pick a new state.

- A state's `currentBet()` method will construct a *`Bet`* from an *`Outcome`* that the *`Player`* keeps, and the multiplier that is unique to the state. As the state changes, the multiplier moves between 1, 3, 2 and 6.
- A state's `nextWon()` method constructs a new state object based on the state transition table when the last bet was a winner.
- A state's `nextLost()` method constructs a new state based on the state transition table when the last bet was a loser. In this case, all of the various states create a new instance of the `NoWins` object, resetting the multiplier to 1 and starting the sequence over again.

18.2 On Polymorphism

One very important note is that we never have to check the class of any object. This is so important, we will repeat it here.

Important: We don't use `isistance()`.

We use *polymorphism* and design all subclasses to have the same interface.

Python relies on *duck typing*:

"if it walks like a duck and quacks like a duck, it is a duck"

This defines class membership using a very simple rule. If an object has the requested method, then it's a member of the class.

Additionally, Python relies on the principle that it's better to seek forgiveness than ask permission.

What this translates to is an approach where an object's methods are simply invoked. If the object implements the methods, then it walked like a duck and – for all practical purposes – actually *was* a duck. If the method is not implemented, the application has a serious design problem and needs to crash.

We find that `isinstance()` is sometimes used by beginning programmers who have failed to properly delegate processing to the subclass.

Often, when a responsibility has been left out of the class hierarchy, it is allocated to the client object. The typical Pretty-Poor Polymorphism looks like the following:

Pretty Poor Polymorphism

```
class SomeClient( object ):
    def someMethod( self ):
        if isinstance(x, AClass):
            Special Case that should have been part of AClass
```

In all cases, uses of `isinstance()` must be examined critically.

Generally, we can avoid `isinstance()` tests by refactoring the special case out of the collaborating class. There will be three changes as part of the refactoring.

1. We will move the special-case the functionality into the class being referenced by `isinstance()`. In the above example, the special case is moved to `AClass`.
2. We will usually have to add default processing to the superclass of `AClass` so that all other sibling classes of `AClass` will have an implementation of the special-case feature.
3. We simply call the refactored method from the client class.

This refactoring leads to a class hierarchy that has the property of being *polymorphic*: all of the subclasses have the same interface: all objects of any class in the hierarchy are interchangable. Each object is, therefore, responsible for correct behavior. More important, a client object does not need to know which subclass the object is a member of: it simply invokes methods which are defined with a uniform interface across all subclasses.

18.3 Player 1-3-2-6 Questions and Answers

Why code the state as objects?

The reason for encoding the states as objects is to encapsulate the information and the behavior associated with that state. In this case, we have both the bet amount and the rules for transition to next state. While simple, these are still unique to each state.

Since this is a book on design, we feel compelled to present the best design. In games like blackjack, the player's state may have much more complex information or behavior. In those games, this design pattern will be very helpful. In this one case only, the design pattern appears to be over-engineered.

We will use this same design pattern to model the state changes in the Craps game itself. In the case of the Craps game, there is additional information as well as behavior changes. When the state changes, bets are won or lost, bets are working or not, and outcomes are allowed or prohibited.

Isn't it simpler to code the state as a number? We can just increment when we win, and reset to zero when we lose.

The answer to all "isn't it simpler" questions is "yes, but..." In this case, the full answer is "Yes, but what happens when you add a state or the states become more complex?"

This question arises frequently in OO programming. Variations on this question include "Why is this an entire object?" and "Isn't an object over-engineering this primitive type?" See *Design Decision – Object Identity* FAQ entry on the `Outcome` class for additional background on object identity.

Our goal in OO design is to isolate responsibility. First, and most important, we can unambiguously isolate the responsibilities for each individual state. Second, we find that it is very common that only one state changes, or new states get added. Given these two conditions, the best object model is separate state objects.

Doesn't this create a vast number of state objects?

Yes.

There are two usual follow-up questions: "Aren't all those objects a lot of memory overhead?" or "...a lot of processing overhead?"

Since Python removes unused objects, the old state definitions are removed when no longer required.

Object creation is an overhead that we can control. One common approach is to use the **Singleton** design pattern. In this case, this should be appropriate because we only want a single instance of each of these state classes.

Note that using the **Singleton** design pattern doesn't change any interfaces except the initialization of the `Player1326` object with the starting state.

Is Polymorphism necessary?

In some design patterns, like **State** and **Command**, it is essential that all subclasses have the same interface and be uniform, indistinguishable, almost anonymous instances. Because of this polymorphic property, the objects can be invoked in a completely uniform way.

In our exercise, we will design a number of different states for the player. Each state has the same interface. The actual values for the instance variables and the actual operation implemented by a subclass method will be unique. Since the interfaces are uniform, however, we can trust all state objects to behave properly.

There are numerous designs where polymorphism doesn't matter at all. In many cases, the anonymous uniformity of subclasses isn't relevant. When we move on to example other casino games, we will see many examples of non-polymorphic class hierarchies. This will be due to the profound differences between the various games and their level of interaction with the players.

18.4 Player1326 State Design

class **`Player1326State`**

`Player1326State` is the superclass for all of the states in the 1-3-2-6 betting system.

18.4.1 Fields

`Player1326State.`**`player`**

The `Player1326` player who is currently in this state. This player will be used to provide the `Outcome` that will be used to create the `Bet`.

18.4.2 Constructors

`Player1326State.`**`__init__`**`(`*self*, *player*`)`

The constructor for this class saves the `Player1326` which will be used to provide the `Outcome` on which we will bet.

18.4.3 Methods

`Player1326State.`**`currentBet`**`(`*self*`)` → Bet

Constructs a new `Bet` from the player's preferred `Outcome`. Each subclass has a different multiplier used when creating this `Bet`.

In Python, this method should return `NotImplemented`. This is a big debugging aid, it helps us locate subclasses which did not provide a method body.

`Player1326State.nextWon(self)` → Player1326State
: Constructs the new `Player1326State` instance to be used when the bet was a winner.

  In Python, this method should return `NotImplemented`. This is a big debugging aid, it helps us locate subclasses which did not provide a method body.

`Player1326State.nextLost(self)` → Player1326State
: Constructs the new `Player1326State` instance to be used when the bet was a loser. This method is the same for each subclass: it creates a new instance of `Player1326NoWins`.

  This defined in the superclass to assure that it is available for each subclass.

18.5 Player1326 No Wins Design

class **`Player1326NoWins`**

`Player1326NoWins` defines the bet and state transition rules in the 1-3-2-6 betting system. When there are no wins, the base bet value of 1 is used.

18.5.1 Methods

`Player1326NoWins.currentBet(self)` → Bet
: Constructs a new `Bet` from the player's `outcome` information. The bet multiplier is 1.

`Player1326NoWins.nextWon(self)` → Player1326State
: Constructs the new `Player1326OneWin` instance to be used when the bet was a winner.

18.6 Player1326 One Win Design

class **`Player1326OneWin`**

`Player1326OneWin` defines the bet and state transition rules in the 1-3-2-6 betting system. When there is one wins, the base bet value of 3 is used.

18.6.1 Methods

`Player1326OneWin.currentBet(self)` → Bet
: Constructs a new `Bet` from the player's `outcome` information. The bet multiplier is 3.

`Player1326OneWin.nextWon(self)` → Player1326State
: Constructs the new `Player1326TwoWins` instance to be used when the bet was a winner.

18.7 Player1326 Two Wins Design

class **`Player1326TwoWins`**

`Player1326TwoWins` defines the bet and state transition rules in the 1-3-2-6 betting system. When there are two wins, the base bet value of 2 is used.

18.7.1 Methods

`Player1326TwoWins.`**`currentBet`**`(`*self*`)` → Bet

Constructs a new *`Bet`* from the player's `outcome` information. The bet multiplier is 2.

`Player1326TwoWins.`**`nextWon`**`(`*self*`)` → Player1326State

Constructs the new *`Player1326ThreeWins`* instance to be used when the bet was a winner.

18.8 Player1326 Three Wins

class `Player1326ThreeWins`

`Player1326ThreeWins` defines the bet and state transition rules in the 1-3-2-6 betting system. When there are three wins, the base bet value of 6 is used.

18.8.1 Methods

`Player1326ThreeWins.`**`currentBet`**`(`*self*`)` → Bet

Constructs a new *`Bet`* from the player's `outcome` information. The bet multiplier is 6.

`Player1326ThreeWins.`**`nextWon`**`(`*self*`)` → Player1326State

Constructs the new *`Player1326NoWins`* instance to be used when the bet was a winner.

An alternative is to update the player to indicate that the player is finished playing.

18.9 Player1326 Design

class `Player1326`

`Player1326` follows the 1-3-2-6 betting system. The player has a preferred *`Outcome`*, an even money bet like red, black, even, odd, high or low. The player also has a current betting state that determines the current bet to place, and what next state applies when the bet has won or lost.

18.9.1 Fields

`Player1326.`**`outcome`**

This is the player's preferred *`Outcome`*. During construction, the Player must fetch this from the *`Wheel`*.

`Player1326.`**`state`**

This is the current state of the 1-3-2-6 betting system. It will be an instance of a subclass of *`Player1326State`*. This will be one of the four states: No Wins, One Win, Two Wins or Three Wins.

18.9.2 Constructors

`Player1326.`**`__init__`**`(`*self*`)`

Initializes the state and the outcome. The *`state`* is set to the initial state of an instance of *`Player1326NoWins`*.

The *`outcome`* is set to some even money proposition, for example `"Black"`.

18.9.3 Methods

`Player1326.placeBets(self)`

Updates the *`Table`* with a bet created by the current state. This method delegates the bet creation to *`state`* object's `currentBet()` method.

`Player1326.win(self, bet)`

Parameters **`bet`** (Bet) – The Bet which won

Uses the superclass method to update the stake with an amount won. Uses the current state to determine what the next state will be by calling *`state`*'s objects `nextWon()` method and saving the new state in *`state`*

`Player1326.lose(self, bet)`

Parameters **`bet`** (Bet) – The Bet which lost

Uses the current state to determine what the next state will be. This method delegates the next state decision to *`state`* object's `nextLost()` method, saving the result in *`state`*.

18.10 Player 1-3-2-6 Deliverables

There are eight deliverables for this exercise. Additionally, there is an optional, more advanced design exercise in a separate section.

- The five classes that make up the *`Player1326State`* class hierarchy.
- The *`Player1326`* class.
- A Unit test for the entire *`Player1326State`* class hierarchy. It's possible to unit test each state class, but they're so simple that it's often easier to simply test the entire hierarchy.
- A unit test of the *`Player1326`* class. This test should synthesize a fixed list of *`Outcome`*s, *`Bin`*s, and calls a *`Player1326`* instance with various sequences of reds and blacks. There are 16 different sequences of four winning and losing bets. These range from four losses in a row to four wins in a row.
- An update to the overall *`Simulator`* that uses the *`Player1326`*.

18.11 Advanced Exercise – Refactoring

Initially, each subclass of *`Player1326State`* has a unique `currentBet()` method.

This class can be simplified slightly to have the bet multiplier coded as an instance variable, `betAmount`. The `currentBet()` method can be refactored into the superclass to use the `betAmount` value.

This would simplify each subclass to be only a constructor that sets the `betAmount` multiplier to a value of 1, 3, 2 or 6.

Similarly, we could have the next state after winning defined as an instance variable, `nextStateWin`. We can initialized this during construction. Then the `nextWon()` method could also be refactored into the superclass, and would return the value of the `nextStateWin` instance variable.

The deliverable for this exercise is an alternative *`Player1326State`* class using just one class and distinct constructors that create each state object with an appropriate bet multipler and and next state win value.

18.12 Advanced Exercise – Less Object Creation

The object creation for each state change can make this player rather slow.

There are a few design pattens that can reduce the number of objects that need to be created.

1. Global objects for the distinct state objects.

   While truly global objects are usually a mistake, we can justify this by claiming that we're only creating objects that are shared among a few objects.

   The objects aren't global variables: they're global constants. There's no state change, so they aren't going to be shared improperly.

2. The **Singleton** design pattern.

   With some cleverness, this can be made transparent in Python. However, it's easiest to make this explicit, by defining a formal `instance()` method that fetches (or creates) the one-and-only instance of the class.

3. A **Factory** object that produces state objects. This **Factory** can retain a small pool of object instances, eliminating needless object construction.

The deliverables for this exercise are revisions to the *`Player1326State`* class hierarchy to implement the **Factory** design pattern. This will not change any of the existing unit tests, demonstrations or application programs.

18.12.1 Player1326 State Factory Design

`Player1326StateFactory.`**`values`**

This is a map from a class name to an object instance.

`Player1326StateFactory.`**`__init__`**`(`*self*`)`

Create a new mapping from the class name to object instance. There are only four objects, so this is relatively simple.

`Player1326StateFactory.`**`get`**`(`*self*, *name*`)` → Player1326State

Parameters **`name`** (*String*) – name of one of the subclasses of *`Player1326State`*.

CHAPTER

NINETEEN

CANCELLATION PLAYER CLASS

This section will describe a player who has a complex internal state that can be modeled using existing library classes.

In *Cancellation Player Analysis* we'll look at what this player does.

We'll turn to how the player works in *PlayerCancellation Design*.

In *Cancellation Player Deliverables* we'll enumerate the deliverables for this player.

19.1 Cancellation Player Analysis

One method for tracking the lost bets is called the "cancellation" system or the "Labouchere" system. The player starts with a betting budget allocated as a series of numbers.

The usual example sequence is `[ 1, 2, 3, 4, 5, 6 ]`.

Each bet will be the sum of the first and last numbers in the last.

In this example, the end values of 1+6 leads the player to bet 7.

When the player wins, the player cancels the two numbers used to make the bet. In the event that all the numbers are cancelled, the player has doubled their money, and can retire from the table happy.

For each loss, however, the player adds the amount of the bet to the end of the sequence; this is a loss to be recouped. This adds the loss to the amount bet to assure that the next winning bet both recoups the most recent loss and provides a gain. Multiple winning bets will recoup multiple losses, supplemented with small gains.

Example. Here's an example of the cancellation system using `[ 1, 2, 3, 4, 5, 6 ]`

1. Bet 1+6. A win. Cancel 1 and 6 leaving `[ 2, 3, 4, 5 ]`
2. Bet 2+5. A loss. Add 7 leaving `[ 2, 3, 4, 5, 7 ]`
3. Bet 2+7. A loss. Add 9 leaving `[ 2, 3, 4, 5, 7, 9 ]`
4. Bet 2+9. A win. Cancel 2 and 9 leaving `[ 3, 4, 5, 7 ]`
5. Next bet will be 3+7.

State. The player's state is a list of individual bet amounts. This list grows and shrinks; when it is empty, the player leaves the table. We can keep a `List` of individual bet amounts. The total bet will be the first and last elements of this list. Wins will remove elements from the collection; losses will add elements to the collection. Since we will be accessing elements in an arbitrary order, we will want to use an `ArrayList`. We can define the player's state with a simple list of values.

19.2 PlayerCancellation Design

class `PlayerCancellation`

PlayerCancellation uses the cancellation betting system. This player allocates their available budget into a sequence of bets that have an accelerating potential gain as well as recouping any losses.

19.2.1 Fields

`PlayerCancellation.`**`sequence`**

This `List` keeps the bet amounts; wins are removed from this list and losses are appended to this list. THe current bet is the first value plus the last value.

`PlayerCancellation.`**`outcome`**

This is the player's preferred *Outcome*.

19.2.2 Constructors

`PlayerCancellation.`**`__init__`**`(`*self*`)`

This uses the *PlayerCancellation.resetSequence()* method to initalize the sequence of numbers used to establish the bet amount. This also picks a suitable even money *Outcome*, for example, black.

19.2.3 Methods

`PlayerCancellation.`**`resetSequence`**`(`*self*`)`

Puts the initial sequence of six `Integer` instances into the *sequence* variable. These `Integer`s are built from the values 1 through 6.

`PlayerCancellation.`**`placeBets`**`(`*self*`)`

Creates a bet from the sum of the first and last values of *sequence* and the preferred outcome.

`PlayerCancellation.`**`win`**`(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which won

Uses the superclass method to update the stake with an amount won. It then removes the fist and last element from *sequence*.

`PlayerCancellation.`**`lose`**`(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which lost

Uses the superclass method to update the stake with an amount lost. It then appends the sum of the first and list elements of *sequence* to the end of *sequence* as a new `Integer` value.

19.3 Cancellation Player Deliverables

There are three deliverables for this exercise.

- The *PlayerCancellation* class.
- A unit test of the *PlayerCancellation* class. This test should synthesize a fixed list of *Outcome*s, *Bin* s, and calls a *PlayerCancellation* instance with various sequences of reds and blacks. There are 16 different sequences of four winning and losing bets. These range from four losses in a row to four wins in a row. This should be sufficient to exercise the class and see the changes in the bet amount.
- An update to the overall *Simulator* that uses the *PlayerCancellation*.

CHAPTER

TWENTY

FIBONACCI PLAYER CLASS

This section will describe a player who has an internal state that can be modeled using methods and simple values instead of state objects.

This is a variation on the Martingale System. See *Martingale Player Design* for more information.

In *Fibonacci Player Analysis* we'll look at what this player does.

We'll turn to how the player works in *PlayerFibonacci Design*.

In *Fibonacci Player Deliverables* we'll enumerate the deliverables for this player.

20.1 Fibonacci Player Analysis

A player could use the *Fibonacci Sequence* to structure a series of bets in a kind of cancellation system. The Fibonacci Sequence is

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

At each loss, the sum of the previous two bets is used, which is the next number in the sequence. In the event of a win, we revert to the basic bet.

Example. Here's an example of the Fibonacci system.

1. Bet 1. A win.
2. Bet 1. A loss. The next value in the sequence is 1.
3. Bet 1. A loss. The next value in the sequence is 2.
4. Bet 2. A loss. The next value in the sequence will be 3
5. Bet 3. In the event of a loss, the next bet is 5. Otherwise, the bet is 1.

State. In order to compute the Fibonacci sequence, we need to retain the two previous bets as the player's state. In the event of a win, we revert to the basic bet value of `1`.

In the event of a loss, we can update the two numbers to show the next step in the sequence. The player's state is just these two numeric values.

20.2 PlayerFibonacci Design

class `PlayerFibonacci`

`PlayerFibonacci` uses the Fibonacci betting system. This player allocates their available budget into a sequence of bets that have an accelerating potential gain.

20.2.1 Fields

`PlayerFibonacci.recent`
This is the most recent bet amount. Initially, this is 1.

`PlayerFibonacci.previous`
This is the bet amount previous to the most recent bet amount. Initially, this is zero.

20.2.2 Constructors

`PlayerFibonacci.__init__(`*self*`)`
Initialize the Fibonacci player.

20.2.3 Methods

`PlayerFibonacci.win(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which won

Uses the superclass method to update the stake with an amount won. It resets *`recent`* and *`previous`* to their initial values of 1 and 0.

`PlayerFibonacci.lose(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which lost

Uses the superclass method to update the stake with an amount lost. This will go “forwards” in the sequence. It updates *`recent`* and *`previous`* as follows.

$$next \leftarrow recent + previous$$
$$previous \leftarrow recent$$
$$recent \leftarrow next$$

Parameters **`bet`** (Bet) – The *`Bet`* which lost.

20.3 Fibonacci Player Deliverables

There are three deliverables for this exercise.

- The *`PlayerFibonacci`* class.
- A unit test of the *`PlayerFibonacci`* class. This test should synthesize a fixed list of *`Outcome`*s, *`Bin`*s, and calls a *`PlayerFibonacci`* instance with various sequences of reds and blacks. There are 16 different sequences of four winning and losing bets. These range from four losses in a row to four wins in a row. This should be sufficient to exercise the class and see the changes in the bet amount.
- An update to the overall *`Simulator`* that uses the *`PlayerFibonacci`*.

CHAPTER TWENTYONE

CONCLUSION

The game of Roulette has given us an opportunity to build an application with a considerable number of classes and objects. It is comfortably large, but not complex; we have built tremendous fidelity to a real-world problem. Finally, this program produces moderately interesting simulation results.

In this section we'll look at our overall approach to design in *Exploration*.

We'll look at some design principles in *The SOLID Principles*.

In *Other Design Patterns* we'll look at some of the other design patterns we've been using.

21.1 Exploration

We note that a great many of our design decisions were not easy to make without exploring a great deal of the overall application's design. We've shown how to do this exploration: design just enough but remain tolerant of our own ignorance.

There's an idealized fantasy in which a developer design an entire, complex application before writing any software. The process for creating a complete design is still essentially iterative. Some parts are designed in detail, with tolerance for future changes; then other parts are designed in detail and the two design elements reconciled. This **can** be done on paper or on a whiteboard.

With Python, however, we can write – and revise – draft programming as easily as erasing a whiteboard. It's quite easy to do incremental design by writing and revising a working base of code.

For new designers, we can't give enough emphasis to the importance of creating a trial design, exploring the consequences of that design, and then doing rework of that design. Too often, we have seen trial designs finalized into deliverables with no opportunity for meaningful rework. In *Review of Testability*, we presented one common kind of rework to support more complete testing. In *Player Class*, we presented another kind of rework to advance a design from a stub to a complete implementation.

21.2 The SOLID Principles

There are five principles of object-oriented design. We've touched on several. For more information, see https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

S Single responsibility principle. We've emphasized this heavily by trying to narrow the scope of responsibility in each class.

O Open/closed principle. The terminology here is important. A class is open to extension but closed to modification. We prefer to wrap or extend classes. We prefer not to modify or tweak a class. When we make a change to a class, we are careful to be sure that the ripple touches the entire hierarchy and all of the collaborators.

L Liskov substitution principle. In essence, this is a test for proper polymorphism. If classes are truly polymorphic, one can be substituted for another. We've generally focused on assuring this.

I Interface segregation principle. In most of what we're doing, we've kept interfaces as narrow as possible. When confronted with **Wrap vs. Extend** distinctions, the idea of interface segregation suggests that we should prefer wrapping a class because that tends to narrow the interface.

D Dependency inversion principle. This principle guides us toward creating common abstractions. Our *`Player`* is an example of this. We didn't create games and tables that depend on a specific player. We created games and tables that would work with any class that met the minimal requirements for the *`Player`* interface.

21.3 Other Design Patterns

We also feel compelled to point out the distinction between relatively active and passive classes in this design. We had several passive classes, like *`Outcome`*, *`Bet`*, and *`Table`*, which had few responsibilities beyond collecting a number of related attributes and providing simple functions.

We also had several complex, active classes, like *`Game`*, *`BinBuilder`* and all of the variations on *`Player`*. These classes, typically, had fewer attributes and more complex methods. In the middle of the spectrum is the *`Wheel`*.

We find this distinction to be an enduring feature of OO design: there are *things* and *actors*; the things tend to be passive, acted upon by the actors. The overall system behavior emerges from the collaboration among all of the objects in the system; primarily – but not exclusively – the behavior of the active classes.