The Monty Hall problem

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

[1]:
```
from random import choice
from random import randrange
```

A contestant and a host are at a place that gives access to three rooms, whose doors are all closed before the game starts. A car is in one room, a goat in another, while the third room is empty. The host knows what is in each room, while the contestant doesn't. The contestant is asked to chose one of the doors. Following that choice, the host opens another door, which is that of a room that does not contain the car. The contestant is then asked to either stick to his choice or "switch", that is, choose the third door (e.g., the door which is neither the one he chose in the first place nor the one that the host opened). The host opens the door as requested by the contestant, who wins what is in the room, if anything. The question is: what is the contestant's best strategy – to switch or not to switch?

The contestant wishes to win the car, not the goat. He has one chance out of three to, in the first place, choose the door of the room that contains the car.

- If he does not switch, he then has one chance out of three to win.
- If he switches then he loses in case the room he chose in the first place contains the car. But he wins otherwise. Indeed, out of both doors that are still closed, one contains the car, so the host has to open the other one (that is, if the room whose door was chosen in the first place contains nothing, then the host opens the door of the room that contains the goat, and if the room whose door was chosen in the first place contains the goat, then the host opens the door of the room that contains nothing). So in both cases, the second door that the host opens following the decision of the contestant to switch is that of the room that contains the car. Hence by switching, the contestant has two chances out of three to win.

Hence it is best to switch.

We aim to simulate the playing of this game, with either strategy, switching or not, to check that when it is played with the chosen strategy a large enough number of times, then the experimental results support the conclusions of the previous reasoning. More precisely, the user should be asked and express whether he wants to switch and how many times the game should be played. By default, if the user requests to play at most 6 games, then the details of each game (which door the contestant choses in the first place, which door the host opens first, and which door he opens next that determines the outcome of the game), will be output; if the user requests more games to be played, then the details of the first 6 games only will be output. It should be possible to set the default of 6 to another value. In any case, the percentage of times the contestant won the game should be eventually displayed.

The `input()` function takes a string as argument to prompt and get input from the user, say `I`

`agree!`; the function returns that string after Carriage return has been pressed:

[2]:
```
input('Better have a space at the end of the prompt message: ')
```

```
Better have a space at the end of the prompt message: I agree!
```

[2]:
```
'I agree!'
```

If we prompt the user for how many games should be played, `int()` can yield an integer from the string, say `'10'`, returned by `input()`. There is no need to store the string, we can get the integer directly:

[3]:
```
int(input('How many games should I simulate? '))
```

```
How many games should I simulate? 10
```

[3]:
```
10
```

In case user input is invalid, say `ten`, then `int()` raises a `ValueError`:

[4]:
```
int(input('How many games should I simulate? '))
```

```
How many games should I simulate? ten
```

```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)

<ipython-input-4-f940614e97ee> in <module>
----> 1 int(input('How many games should I simulate? '))


ValueError: invalid literal for int() with base 10: 'ten'
```

We should not presume that user input is necessarily valid. Moreover, in case it is not, that should not cause `int()` to bring the program to an end. A call to `int()` to tentatively produce an integer from the string input by the user can be placed within a `try` statement, whose associated `except` statement will be executed if and only if the string makes `int()` raise an exception of the **type** that the `except` statement is meant to **catch**:

[5]:
```
try:
    int('10')
    print('Good input!')
except ValueError:
    print('Bad input!')
```

[5]:
```
10
```

```
Good input!
```

[6]:
```
try:
    int('ten')
    print('Good input!')
except ValueError:
    print('Bad input!')
```

```
Bad input!
```

If `int()` fails to produce an integer from the string input by the user, then the program should notify him that his input is invalid and prompt him again, and possibly many times again, until the input does not cause `int()` to fail. One way to achieve this is to make use of a `while` statement, here designed to be executed forever because it tests a condition that always evaluates to `True` (it suffices to make `True` itself be the condition), but with a `break` statement in the body to **exit** the loop as soon as the user provides valid input. Observe how the code below executes when the user inputs `ten`, then `1o`, then `10.0`, and finally `10`:

[7]:
```
while True:
    try:
        nb_of_games = int(input('How many games should I simulate? '))
        print('Ok, will do!')
        break
    except ValueError:
        print('Your input is incorrect, try again.')
```

```
How many games should I simulate? ten
Your input is incorrect, try again.
How many games should I simulate? 1o
Your input is incorrect, try again.
How many games should I simulate? 10.0
Your input is incorrect, try again.
How many games should I simulate? 10
Ok, will do!
```

In case the call to `int()` succeeds, the input provided by the user can still be invalid, because it is 0 or a strictly negative number. To deal with this situation, one can, within the `try` statement, test whether the number returned by `int()` is strictly positive, and in case it is not, use `raise` and implicitly generate a `ValueError` exception, which has the same effect as `int()` implicitly raising a `ValueError` exception: let the execution of the code **jump** to the associated `except` statement. Observe how the code below executes when the user inputs `ten`, then `-10`, then `0`, and finally `10`:

[8]:
```
while True:
    try:
        nb_of_games = int(input('How many games should I simulate? '))
        if nb_of_games <= 0:
            raise ValueError
```

```
        print('Ok, will do!')
        break
    except ValueError:
        print('Your input is incorrect, try again.')
```

```
How many games should I simulate? ten
Your input is incorrect, try again.
How many games should I simulate? -10
Your input is incorrect, try again.
How many games should I simulate? 0
Your input is incorrect, try again.
How many games should I simulate? 10
Ok, will do!
```

After he expressed how many games should be played, the user should then be prompted and answer whether he wants the contestant to switch or not. To be faithful to the game, this question should be asked for every game, right after the host has opened the first door, but since we are interested in statistical results, the question will be asked once and for all before playing the first game, and all games will be played accordingly. We can be more or less flexible in which answers we accept. We could be strict and accept nothing but either `yes` or `no`. We could be more flexible, and accept `y` and `n` as well. And we could be even more flexible and also accept `Yes`, `Y`, `No` and `N`. Let us go for that.

The `title()` method of the `str` class allows one to check whether a string is a word that starts with a capital letter. The Python definition of a word is: a sequence of characters that starts either with a letter or an underscore, and that is possibly followed with underscores, letters, and digits. Python **identifiers** (names of variables, functions…) have to be words in this precise sense.

[9]:
```
'Y'.istitle()
'Yes'.istitle()
'Yes_'.istitle()
'Yes_12_'.istitle()
'yes'.istitle()
'yEs'.istitle()
'Yes, by all means, yes!'.istitle()
```

[9]: True

[9]: True

[9]: True

[9]: True

[9]: False

[9]: False

[9]: False

The `lower()` method of the `str` class allows one to get from a string S a string where all uppercase letters in S have been converted to lowercase:

[10]:
```
'Y'.lower()
'Yes'.lower()
'yes'.lower()
'yEs'.lower()
'Yes, by all means, yes!'.lower()
```

[10]: `'y'`

[10]: `'yes'`

[10]: `'yes'`

[10]: `'yes'`

[10]: `'yes, by all means, yes!'`

If we want to accept as input `yes`, `y`, `no` and `n` as well as their capitalised forms, we can:

- check with `istitle()` whether the input is a capitalised word and in case it is, use `lower()` to get a lowercase version, a form of data *normalisation*;
- check whether the input, or its lowercase version in case it is a capitalised word, is one of `yes`, `y`, `no` and `n`.

Rather than making a list out of `yes`, `y`, `no` and `n`, we can instead create a **set**. This is more natural as order does not matter (also, checking whether an object is a member of a set is more efficient than checking whether it is a member of a list). The `in` operator does not only allow one to check for membership of keys in dictionaries; it can be used for other kinds of membership, and with strings, for being a substring:

[11]:
```
# Checking for membership in a list or a tuple
'yes' in ['yes', 'y', 'no', 'n']
'ye' in ('yes', 'y', 'no', 'n')
# Checking for membership in a set
'y' in {'yes', 'y', 'no', 'n'}
'Y' in {'yes', 'y', 'no', 'n'}
# Checking for being a substring in a string
'' in 'maybe', 'y' in 'maybe', 'ayb' in 'maybe'
'ab' in 'maybe'
```

[11]: `True`

[11]: `False`

[11]: `True`

[11]: `False`

[11]: `(True, True, True)`

[11]: `False`

Curly braces are used for both literal dictionaries and literal sets. There is no potential conflict, except for empty set versus empty dictionary; `{}` denotes an empty dictionary, not an empty set:

[12]:
```
# Singleton dictionary and set, respectively
type({'one': 1})
type({'one'})
# Empty dictionary and set, respectively
type({})
type(set())
```

[12]: `dict`

[12]: `set`

[12]: `dict`

[12]: `set`

Here too, user input could be invalid; again, it is natural to set an infinite loop which code breaks out from as soon as the user provides valid input. Rather than code that does not or does generate an exception, the evaluation of the condition of an `if` statement to `True` or `False` can control whether to exit the loop with a `break` statement. Observe how the code below executes when the user inputs `NO`, then `noooo`, then `No!`, and finally `No`:

[13]:
```
while True:
    contestant_switches = input('Should the contestant switch? ')
    if contestant_switches.istitle():
        contestant_switches = contestant_switches.lower()
    if contestant_switches in {'yes', 'y'}:
        print('I keep in mind you want to switch.')
        break
    if contestant_switches in {'no', 'n'}:
        print("I keep in mind you don't want to switch.")
        break
    print('Your input is incorrect, try again.')
```

```
Should the contestant switch? NO
Your input is incorrect, try again.
Should the contestant switch? noooo
Your input is incorrect, try again.
Should the contestant switch? No!
Your input is incorrect, try again.
Should the contestant switch? No
I keep in mind you don't want to switch.
```

To simulate a game, we need to randomly choose the winning door, namely, the door that opens to the room that contains the car. Using the characters `'A'`, `'B'` and `'C'` as door labels, the problem boils down to randomly choosing one of those strings. We can put them in a list and use the `choice()` function from the `random` **module**, which is possible because we previously **imported** the former from the latter, to make a random selection following the uniform distribution; behind the scene, `choice()` randomly generates a number between 0 and 2 and returns the member of the list with that index. That is why we make the labels of the doors the members of a list rather than the members of a set: as the members of a set are unordered, they have no associated index, hence `choice()` cannot be applied to a set. Running the following cell randomly produces one of 3 possible outcomes:

[14]:
```
doors = ['A', 'B', 'C']
# Randomly returns one of 'A', 'B' or 'C'
choice(doors)
```

[14]:
```
'A'
```

The contestant is asked to choose a door. To make it easier to determine the two doors that the host will then open, it is convenient to remove from `doors` the label of the door first chosen by the contestant; that is something the `remove()` method of the `list` class can do:

[15]:
```
doors = ['A', 'B', 'C']
doors.remove('B')
doors
```

[15]:
```
['A', 'C']
```

The door chosen by the contestant should also be chosen randomly. One option is to use both `choice()` and `remove()`. Running the following cell randomly produces one of 3 possible outcomes:

[16]:
```
doors = ['A', 'B', 'C']
# Randomly assigns one of 'A', 'B' or 'C'
first_chosen_door = choice(doors)
doors.remove(first_chosen_door)
first_chosen_door, doors
```

[16]:
```
('C', ['A', 'B'])
```

Given a list L with n distinct elements, it is more effective to, at least when n is large, randomly select a number i between 0 and $n-1$ and return and remove the member of L of index i, rather than let `choice()` generate behind the scene such a number i, return the member of L with index i, and then let `remove()` look for that element, whose location in L has been "lost", by scanning L starting from the beginning, element by element, until the element is found and can be removed. The `randrange()` function from the `random` module allows one to randomly generate a number between 0 and the number n provided as argument, with 0 included and n excluded. Running the following cell randomly produces one of 3 possible outcomes:

[17]:
```
# Randomly returns one of 0, 1 or 2
randrange(3)
```

[17]: 2

The `pop()` method of the `list` class pops out the last element of a list and returns it, unless it is given a natural number at most equal to the length of the list minus 1 as argument, in which case it removes the element at that index from the list and returns it. Running the following cell randomly produces one of 6 possible outcomes:

[18]:
```
doors = ['A', 'B', 'C']
# Returns and removes from doors 'A', 'B' and 'C'
# one after the other in a random order
doors.pop(), doors
doors.pop(), doors
doors.pop(), doors
```

[18]: ('C', ['A', 'B'])

[18]: ('B', ['A'])

[18]: ('A', [])

[19]:
```
doors = ['A', 'B', 'C']

doors.pop(1), doors
```

[19]: ('B', ['A', 'C'])

So `randrange()` and `pop()` offer a good alternative to determine the door chosen by the contestant and remove it from the list of doors. Running the following cell randomly produces one of 3 possible outcomes:

[20]:
```
doors = ['A', 'B', 'C']
# 3 possible outcomes
first_chosen_door = doors.pop(randrange(3))
first_chosen_door, doors
```

[20]: ('B', ['A', 'C'])

After the contestant has chosen a door, one can determine which door the host opens for the first time.

- Case 1: the contestant did not chose the winning door. Since the label of the room chosen by the contestant was removed from `doors`, `doors` contains the label of the winning door as well as the third label. The host has no choice, he has to open the room with that third label. This is easily achieved by removing the label of the winning door from `doors`, which then becomes a list with a single element, with an index of 0.
- Case 2: the contestant chose the winning door. Then `doors` contains the labels of both rooms that do not contain the car. The host can open either, and he does it randomly. We can also remove its label from `doors`.

Putting things together, running the following cell randomly produces one of 12 possible outcomes. The code makes use of an `else` statement, to deal with the case where the condition of the associated `if` statement evaluates to `False`:

[21]:
```
doors = ['A', 'B', 'C']
# 3 possible outcomes
winning_door = choice(doors)
# 3 possible outomes
first_chosen_door = doors.pop(randrange(3))

if first_chosen_door == winning_door:
    # 2 possible outcomes
    opened_door = doors.pop(randrange(2))
else:
    doors.remove(winning_door)
    opened_door = doors[0]

winning_door
# Possibly identical to previous
first_chosen_door
# Necessarily different to previous two
opened_door
```

[21]: `'C'`

[21]: `'C'`

[21]: `'A'`

After the host has opened a door for the first time, knowing whether the contestant switches or not, one can determine which door the latter asks the former to open.

- Case 1: the contestant does not switch. Then the answer is immediate, and does not depend on whether or not the winning door was chosen in the first place.
- Case 2: the contestant switches.
  - Subcase 1: the contestant chose the winning door in the first place. Then, following that choice, the label of the winning door was removed from `doors`, and the label of the door that the host opened was one of the remaining two; that label was then also removed from `doors`, leaving `doors` with a single label, which is that of the door that the contestant now wants to be opened.
  - Subcase 2: the contestant did not choose the winning door in the first place. Then we know that the second door that the host will open is the winning door.

At this stage, whether the contestant switches or not matters, so we use and adapt the code that lets the user experiment with one or the other strategy. Putting things together, running the following cell randomly produces one of 24 possible outcomes:

[22]:
```
# 2 possible outcomes
while True:
```

```
    contestant_switches = input('Should the contestant switch? ')
    if contestant_switches.istitle():
        contestant_switches = contestant_switches.lower()
    if contestant_switches in {'yes', 'y'}:
        contestant_switches = True
        print('I keep in mind you want to switch.')
        break
    if contestant_switches.lower() in {'no', 'n'}:
        contestant_switches = False
        print("I keep in mind you don't want to switch.")
        break
    print('Your input is incorrect, try again.')

doors = ['A', 'B', 'C']
# 3 possible outcomes
winning_door = choice(doors)
# 3 possible outomes
first_chosen_door = doors.pop(randrange(3))

if not contestant_switches:
    second_chosen_door = first_chosen_door
if first_chosen_door == winning_door:
    # 2 possible outcomes
    opened_door = doors.pop(randrange(2))
    if contestant_switches:
        second_chosen_door = doors[0]
else:
    doors.remove(winning_door)
    opened_door = doors[0]
    if contestant_switches:
        second_chosen_door = winning_door

winning_door
# Possibly identical to previous
first_chosen_door
# Necessarily different to previous two
opened_door
# Same as first_chosen_door if contestant does not switch
# Otherwise, the unique one which is neither
# first_chosen_door nor opened_door
second_chosen_door
```

```
Should the contestant switch? Yes
I keep in mind you want to switch.
```

[22]: 'A'

[22]: 'B'

[22]: 'C'

[22]: 'A'

To better organise the code, we create two functions, one to let the user input how many games should be played and whether he wants the contestant to switch, another one to run the simulation. The latter will call the former. We start with the first function. A function returns one and only one value, but lists or tuples allow one to package many values into one. The syntax of tuples where parentheses are left implicit is particularly appropriate for the code to look "as if" many values were returned, values that can be directly **unpacked** as the function is called:

[23]:
```
def f():
    return 1

def g():
    # Returns one value, a tuple
    return 2, 20

f()
g()
u, v = g()
u
v
```

[23]: 1

[23]: (2, 20)

[23]: 2

[23]: 20

The function that lets the user input how many games should be played and whether he wants the contestant to switch returns one of 4 possible outcomes:

[24]:
```
def set_simulation():
    while True:
        try:
            nb_of_games =\
                    int(input('How many games should I simulate? '))
            if nb_of_games <= 0:
                raise ValueError
            break
        except ValueError:
            print('Your input is incorrect, try again.')
    while True:
```

```
        contestant_switches = input('Should the contestant switch? ')
        if contestant_switches.istitle():
            contestant_switches = contestant_switches.lower()
        if contestant_switches in {'yes', 'y'}:
            contestant_switches = True
            print('I keep in mind you want to switch.')
            break
        if contestant_switches.lower() in {'no', 'n'}:
            contestant_switches = False
            print("I keep in mind you don't want to switch.")
            break
        print('Your input is incorrect, try again.')
    return nb_of_games, contestant_switches

set_simulation()
```

```
How many games should I simulate? 10_000
Should the contestant switch? N
I keep in mind you don't want to switch.
```

[24]:
```
(10000, False)
```

By default, we want to display the details of only the first 6 games, in case the user requests more games to be played. Functions can be defined with some or all **parameters** taking **default** values. All parameters without default values should precede all parameters with default values. When calling the function, all **positional** arguments should precede all **keyword** arguments. The order of the keyword arguments is irrelevant:

[25]:
```
# x and y are parameters without default values,
# u and v are parameters without default values
def f(x, y, u=10, v=20):
    print(x, y, u, v)

# 2 positional arguments
f(1, 2)
# 3 positional arguments
f(1, 2, 4)
# 4 positional arguments
f(1, 2, 3, 4)
# 1 positional argument, 2 keyword arguments
f(1, v=4, y=2)
# 3 keyword arguments
f(u=3, y=2, x=1)
```

```
1 2 10 20
1 2 4 20
1 2 3 4
1 2 10 4
```

```
1 2 3 20
```

A `for` statement is a natural choice for executing a piece of code a given number of times, with the help of the `range` class. When it is given a natural number n as argument, `range()` returns an **iterable**, that is, a kind of object to which the `iter()` function can be applied to return an iterator designed to yield all natural numbers between 0 and $n-1$:

[26]:
```
x = iter(range(4))
next(x)
next(x)
next(x)
next(x)
next(x)
```

[26]:
```
0
```

[26]:
```
1
```

[26]:
```
2
```

[26]:
```
3
```

```
      ␣
↪---------------------------------------------------------------------------

       StopIteration                                 Traceback (most recent call␣
↪last)

       <ipython-input-26-b76478931f55> in <module>
         4 next(x)
         5 next(x)
    ----> 6 next(x)


       StopIteration:
```

The following statement works as follows: behind the scene, an iterator is created from `range(4)`, then `next()` is called again and again, returning a value that is assigned to the variable `i` and possibly used in the body of the `for` statement, until a `StopIteration` is generated, causing the `for` statement to gracefully stop execution:

[27]:
```
for i in range(4):
    print('i is equal to', i)
```

```
i is equal to 0
i is equal to 1
```

```
i is equal to 2
i is equal to 3
```

In case more games are played than displayed, a good design is to print out a line with three dots to indicate that the output is only partial. We make use of an `elif` statement for that purpose:

[28]:
```
def simulate(nb_of_games_to_display=3):
    for i in range(nb_of_games):
        if i < nb_of_games_to_display:
            print('Display all details of game number', i)
        elif i == nb_of_games_to_display:
            print('...')

nb_of_games = 2
simulate()
print()

nb_of_games = 3
simulate()
print()

nb_of_games = 4
simulate()
```

```
Display all details of game number 0
Display all details of game number 1

Display all details of game number 0
Display all details of game number 1
Display all details of game number 2

Display all details of game number 0
Display all details of game number 1
Display all details of game number 2
...
```

To output information about the game as it is being played, it is convenient to use **formatted** strings; they are preceded with `f` and can contain pairs of curly braces that surround expressions meant to be replaced with their values. Also, though strings can be explicitly concatenated with the `+` operator, they can also be implicitly concatenated when they are separated with nothing but space characters, including possibly new lines:

[29]:
```
x = 10
u = 4.5
v = 10
print(f'x is equal to {x}.'
      ' That is not all: '
      f'{u} divided by {v} equals {u / v}.'
     )
```

```
x is equal to 10. That is not all: 4.5 divided by 10 equals 0.45.
```

By following the expression within the curly braces of a formatted string with a colon, one can use special syntax to control how the value of the expression should be displayed. In particular, if the expression evaluates to a **floating point** value, then the number of digits to be displayed after the decimal point can be set by following the colon with a dot, then the desired number of digits, then `f`:

[30]:
```
x = 123 / 321
f'{x}'
f'{x:.0f}'
f'{x:.1f}'
f'{x:.2f}'
f'{x:.3f}'
f'{x:.4f}'
f'{x:.30f}'
```

[30]: `'0.38317757009345793'`

[30]: `'0'`

[30]: `'0.4'`

[30]: `'0.38'`

[30]: `'0.383'`

[30]: `'0.3832'`

[30]: `'0.383177570093457930955338497370'`

We essentially have all the code for the function that runs the simulation. We need to keep track of how many games the contestant wins. For this purpose, a variable `nb_of_wins` can be defined to play the role of a counter. Initialised to `0`, it should be incremented if:

- the door that the contestant chooses in the first place is the winning door and the contestant does not switch, or
- the door that the contestant chooses in the first place is not the winning door and the contestant switches.

Putting everything together complemented with `print()` statements:

[31]:
```
def simulate(nb_of_games_to_display=6):
    nb_of_games, contestant_switches = set_simulation()
    print('Starting the simulation with the contestant', end=' ')
    if not contestant_switches:
        print('not ', end='')
    print('switching doors.\n')
    nb_of_wins = 0
    for i in range(nb_of_games):
```

```
        doors = ['A', 'B', 'C']
        winning_door = choice(doors)
        if i < nb_of_games_to_display:
            print('\tContestant does not know it, but car '
                  f'happens to be behind door {winning_door}.'
                 )
        first_chosen_door = doors.pop(randrange(3))
        if i < nb_of_games_to_display:
            print(f'\tContestant chooses door {first_chosen_door}.')
        if not contestant_switches:
            second_chosen_door = first_chosen_door
        if first_chosen_door == winning_door:
            opened_door = doors.pop(randrange(2))
            if contestant_switches:
                second_chosen_door = doors[0]
            else:
                nb_of_wins += 1
        else:
            doors.remove(winning_door)
            opened_door = doors[0]
            if contestant_switches:
                second_chosen_door = winning_door
                nb_of_wins += 1
        if i < nb_of_games_to_display:
            print(f'\tGame host opens door {opened_door}.')
            print(f'\tContestant chooses door {second_chosen_door}', end=' ')
            print('and wins.\n') if second_chosen_door == winning_door\
                                  else print('and looses.\n')
        elif i == nb_of_games_to_display:
            print('...\n')
    print(f'Contestant won {nb_of_wins / nb_of_games * 100:.2f}% of games.')
```

We can experiment with the default number of games to display:

[32]:
```
simulate()
```

```
How many games should I simulate? 10_000
Should the contestant switch? yes
I keep in mind you want to switch.
Starting the simulation with the contestant switching doors.

        Contestant does not know it, but car happens to be behind door A.
        Contestant chooses door B.
        Game host opens door C.
        Contestant chooses door A and wins.

        Contestant does not know it, but car happens to be behind door B.
        Contestant chooses door C.
```

```
        Game host opens door A.
        Contestant chooses door B and wins.

        Contestant does not know it, but car happens to be behind door B.
        Contestant chooses door A.
        Game host opens door C.
        Contestant chooses door B and wins.

        Contestant does not know it, but car happens to be behind door B.
        Contestant chooses door A.
        Game host opens door C.
        Contestant chooses door B and wins.

        Contestant does not know it, but car happens to be behind door C.
        Contestant chooses door B.
        Game host opens door A.
        Contestant chooses door C and wins.

        Contestant does not know it, but car happens to be behind door A.
        Contestant chooses door B.
        Game host opens door C.
        Contestant chooses door A and wins.

…

Contestant won 67.15% of games.
```

We can experiment changing the default number of games to display:

[33]:
```
simulate(3)
```

```
How many games should I simulate? 10_000
Should the contestant switch? no
I keep in mind you don't want to switch.
Starting the simulation with the contestant not switching doors.

        Contestant does not know it, but car happens to be behind door C.
        Contestant chooses door A.
        Game host opens door B.
        Contestant chooses door A and looses.

        Contestant does not know it, but car happens to be behind door A.
        Contestant chooses door A.
        Game host opens door B.
        Contestant chooses door A and wins.

        Contestant does not know it, but car happens to be behind door C.
        Contestant chooses door C.
```

```
        Game host opens door A.
        Contestant chooses door C and wins.

…

Contestant won 33.31% of games.
```