The Monty Hall problem 蒙提霍尔问题

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming, trimester 1, 2019

随机的

```
In [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:

```
In [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!


Out[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:

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

int 放在前面，直接整型

模拟

```
How many games should I simulate? 10


Out[3]: 10
```

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

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

How many games should I simulate? ten
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-3-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**:

我们不应该假设用户输入必然是有效的。而且如果不是那不应该导致int（）结束程序。 调用int（）从用户输入的字符串暂时生成一个整数可以放在一个try语句中，其关联的except语句将是
当且仅当字符串使int（）引发except语句要捕获的类型的异常时才执行

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

Out[5]: 10

Good input!
```

```
try:
   tp = int(input('enter temp))
```

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

Bad input!
```

整数 while语句

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`:

中断语句 条件

观察下面的代码在用户输入10，然后是1o，然后是10.0，和最终10

```
In [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.')
```

模拟

```
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!
```

如果对int（）的调用成功，则用户提供的输入仍然无效，因为它是0或严格为负数。 为了处理这种情况，可以在try语句中测试int（）返回的数字是否严格为正，如果不是，则使用raise并显式生成ValueError异常，与int（）隐式引发ValueError异常具有相同的效果：让代码的执行跳转到关联的except语句。 当用户输入10，然后是-10，然后是0，最后是10时，观察下面的代码是如何执行的

无效的

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 explicitly 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`:

执行

```
In [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!
```

表达

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.

大写字母

str类的title（）方法允许用户检查字符串是否是以字母开头的单词大写字母。单词的Python定义是：以字母或下划线开头的字符序列，可能后跟下划线，字母和数字。 Python标识符（变量名称，函数.....）必须是精确意义上的单词

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

```
Out[9]: True
```

最后一个不是单词，是一句话

```
Out[9]: True

Out[9]: True

Out[9]: True

Out[9]: False

Out[9]: False

Out[9]: False
```

把大写字母变成小写字母

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:

小写字母

str类的lower（）方法允许从字符串S获取一个字符串，其中S中的所有大写字母都已转换为小写

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

Out[10]: 'y'

Out[10]: 'yes'

Out[10]: 'yes'

Out[10]: 'yes'

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

使用istitle（）检查输入是否为大写单词，如果是，请使用lower（）获取小写版本，这是一种数据规范化形式

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*; 正常化
- check whether the input, or its lowercase version in case it is a capitalised word, is one of `yes`, `y`, `no` and `n`. 检查输入或其小写版本是否为大写单词，是yes，y，no和n之一

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):

我们可以改为创建一个集合，而不是从yes，y，no和n中列出一个列表。 这更自然，因为顺序无关紧要（同样，检查对象是否是集合的成员比检查它是否是列表的成员更有效

```
In [11]: # Checking for membership in a list
         'y' in ['yes', 'y', 'no', 'n']
         'Y' in ['yes', 'y', 'no', 'n']
         # Checking for membership in a set
         'y' in {'yes', 'y', 'no', 'n'}
         'Y' in {'yes', 'y', 'no', 'n'}

Out[11]: True

Out[11]: False

Out[11]: True

Out[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:

大括号用于文字词典和文字集。 除了空集与空字典之外，没有潜在的冲突；{}表示空字典，而不是空集：

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

```
Out[12]: dict
```
字典

```
Out[12]: set
```
集

```
Out[12]: dict

Out[12]: set
```

在这里，用户输入也可能无效；再一次，设置一个无限循环是很自然的，一旦用户提供有效输入代码就会突破。 而不是没有或确实生成异常的代码，if语句的条件评估为True或False可以控制否退出循环用一个休息声明。 当用户输入NO，然后noooo，然后是No！，最后是No时，观察下面代码是如何执行的

无效的

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`:

```
In [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.lower() in {'no', 'n'}:
                 print("I keep in mind you don't want to switch.")
                 break
             print('Your input is incorrect, try again.')
```

参赛者

```
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:

为了模拟游戏，我们需要随机选择获胜的门，即打开包含汽车的房间的门。 使用字符“A”，“B”和“C”作为门标签，问题归结为随机选择其中一个字符串。 我们可以将它们放在一个列表中并使用choice（）函数
来自随机模块，这是可能的，因为我们以前从后者导入前者，以便在均匀分布之后进行随机选择；在场景后面，choice（）随机生成一个介于0和2之间的数字，并返回带有该索引的列表成员。 这就是为什么我们使门的标签成为列表的成员而不是集合的成员：因为集合的成员是无序的，它们没有关联的索引，因此choice（）不能应用于集合。 随机运行以下单元格会产生以下3种可能结果之一：

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

随机生成，随机选择 A B C

```
w for w in dir(list) if w[0] != '_'
```

```
Out[14]: 'C'
```

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:

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

```
Out[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:

```
In [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
```

```
Out[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:

给定具有n个不同元素的列表L，更有效的是，至少当n大时，随机选择0和n-1之间的数字i并返回并移除索引i的L的成员，而不是让选择（）在场景后面生成这样的数字i，用索引i返回L的成员，然后让remove（）通过从头开始扫描L来查找L中的位置已经“丢失”的元素，element by 元素，直到找到元素并可以删除。 来自随机模块的randrange（）函数允许随机生成0和作为参数提供的数字n之间的数字，其中包括0和n。 随机运行以下单元格会产生以下3种可能结果之一：

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

随机产生

```
Out[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 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:

列表类的pop（）方法弹出列表的最后一个元素并返回它，除非给出一个自然数最多等于列表长度减1作为参数，在这种情况下它会删除元素 在列表中的那个索引处并返回它。 随机运行以下单元格会产生6种可能结果之一

```
In [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
```

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

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

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

In [18]: doors = ['A', 'B', 'C']

         doors.pop(1), doors

Out[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:

所以randrange（）和pop（）提供了一个很好的选择来确定参赛者选择的门并将其从门列表中删除。

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

Out[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`:

将事物放在一起，随机运行以下单元格会产生12种可能的结果之一。
代码使用else语句来处理关联的if语句的条件求值为False的情况：

```
In [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
```

```
Out[21]: 'C'
```

```
Out[21]: 'B'
```

```
Out[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.

剩下的

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:

```
In [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
```

```
Out[22]: 'C'
```

```
Out[22]: 'C'
```

```
Out[22]: 'A'
```

```
Out[22]: 'B'
```

模拟

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:

括号

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

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

         f()
```

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

```
Out[23]: 1
```

```
Out[23]: (2, 20)
```

```
Out[23]: 2
```

```
Out[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:

```
In [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
```

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

参数 默认值

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:

位置参数

```
In [26]: # x and y are parameters without default values,
         # u and v are parameters with 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 $n-1$:

在范围类的帮助下，for语句是在给定次数下执行一段代码的自然选择。 当给定一个自然数n作为参数时，range（）返回一个iterable，即一个对象，iter（）函数可以应用于该对象以返回一个迭代器，该迭代器被设计为产生0到n之间的所有自然数 - 1：

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

```
Out[26]: 0
```

```
Out[26]: 1
```

```
Out[26]: 2
```

```
Out[26]: 3
```

```
---------------------------------------------------------------------------

StopIteration                             Traceback (most recent call last)

<ipython-input-25-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:

以下语句的工作原理如下：在场景后面，从range（4）创建一个迭代器，然后一次又一次地调用next（），返回一个赋值给变量i的值，并可能在for的主体中使用 声明，直到生成StopIteration，导致for语句优雅地停止执行

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

等于

```
i is equal to 0
i is equal to 1
i is equal to 2
i is equal to 3
```

如果播放的游戏多于显示的数量，一个好的设计是打印出一个带有三个点的线，表示输出只是部分的。 我们为此目的使用了elif语句

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:

模拟

```
In [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:

为了输出正在播放的游戏信息，使用格式化的字符串很方便；它们前面带有f，并且可以包含成对的花括号，这些花括号表示要用其值替换的表达式。 此外，虽然字符串可以与+运算符显式连接，但是当它们除了空格字符之外，它们也可以隐式连接，包括可能是新行：

```
In [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`:

通过使用冒号跟带有冒号的格式化字符串的花括号内的表达式，可以使用特殊语法来控制应如何显示表达式的值。 特别是，如果表达式的计算结果为浮点值，那么小数点后显示的位数可以通过跟随冒号跟随点，然后是所需的位数来设置，然后f：

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

```
Out[30]: '0.38317757009345793'

Out[30]: '0'

Out[30]: '0.4'
```

```
Out[30]: '0.38'

Out[30]: '0.383'

Out[30]: '0.3832'

Out[30]: '0.38317757009345793095533849737O'
```

基本上 模拟

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:

初始化，预赋值

- 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.

补全

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

```
In [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('Contestant has won '
              f'{nb_of_wins / nb_of_games * 100:.2f}% of games.'
             )
```

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

```
In [32]: simulate()

How many games should I simulate? 10_000
Should the contestant switch? yes
Starting the simulation with the contestant switching doors.

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

        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 C and wins.

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

        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 C and looses.

…

Contestant has won 66.27% of games.
```

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

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

How many games should I simulate? 10_000
Should the contestant switch? no
Starting the simulation with the contestant not switching doors.

        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 A and looses.

        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 A and looses.

        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 C and looses.

…

Contestant has won 34.03% of games.
```