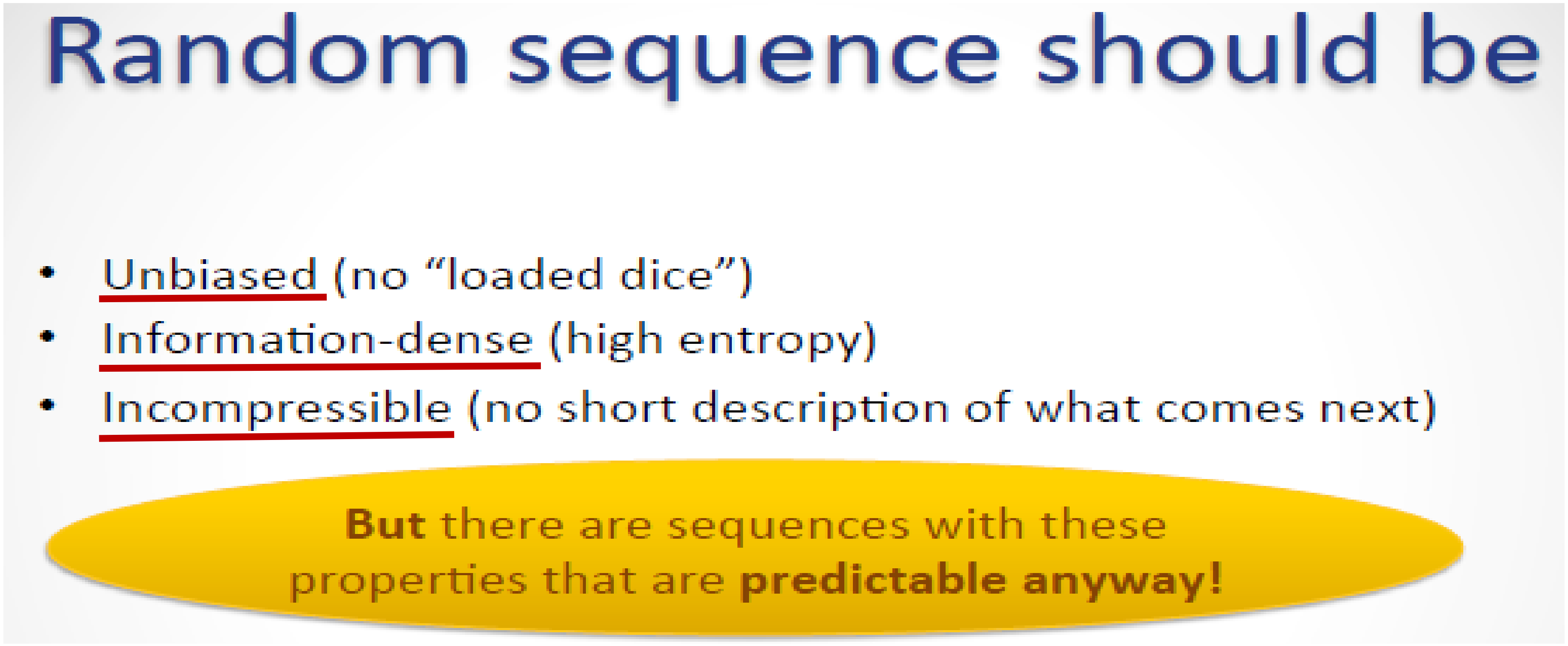
Ch 16: Simulations in Python

Table of Contents

Randomness in Computing

- Determinism: input ➔ predictable output

- Sometimes we want unpredictable outcomes
  - Internet Gambling
  - State Lotteries
  - Simulation (Weather, Evolution, Finance)
  - Selecting samples from large data sets
  - Monte Carlo Methods
  - Randomized Algorithms
  - Cryptography (BitCoin, Secure Internet)
  - Games
- We use the word “randomness” for unpredictability
  - Having no pattern



Random sequence should be

- Unbiased (no “loaded dice”)
- Information-dense (high entropy)
- Incompressible (no short description of what comes next)

But there are sequences with these properties that are **predictable anyway!**

Entropy is a measure of disorder or randomness

True Random Sequences

- Precomputed random sequences. For example, *A Million Random Digits with 100,000 Normal Deviates (1955)*: A 400 page reference book by the RAND corporation
  - 2500 random digits on each page
  - Generated from random electronic pulses
- True Random Number Generators (TRNG)
  - Extract randomness from physical phenomena such as atmospheric noise, times for radioactive decay
- Drawbacks:
  - Physical process might be biased (produce some values more frequently)
  - Expensive
  - Slow

RAND Corporation ("**R**esearch **AN**d **D**evelopment") is an American nonprofit global policy think tank created in 1948 by Douglas Aircraft Company to offer research and analysis to the United States Armed Forces

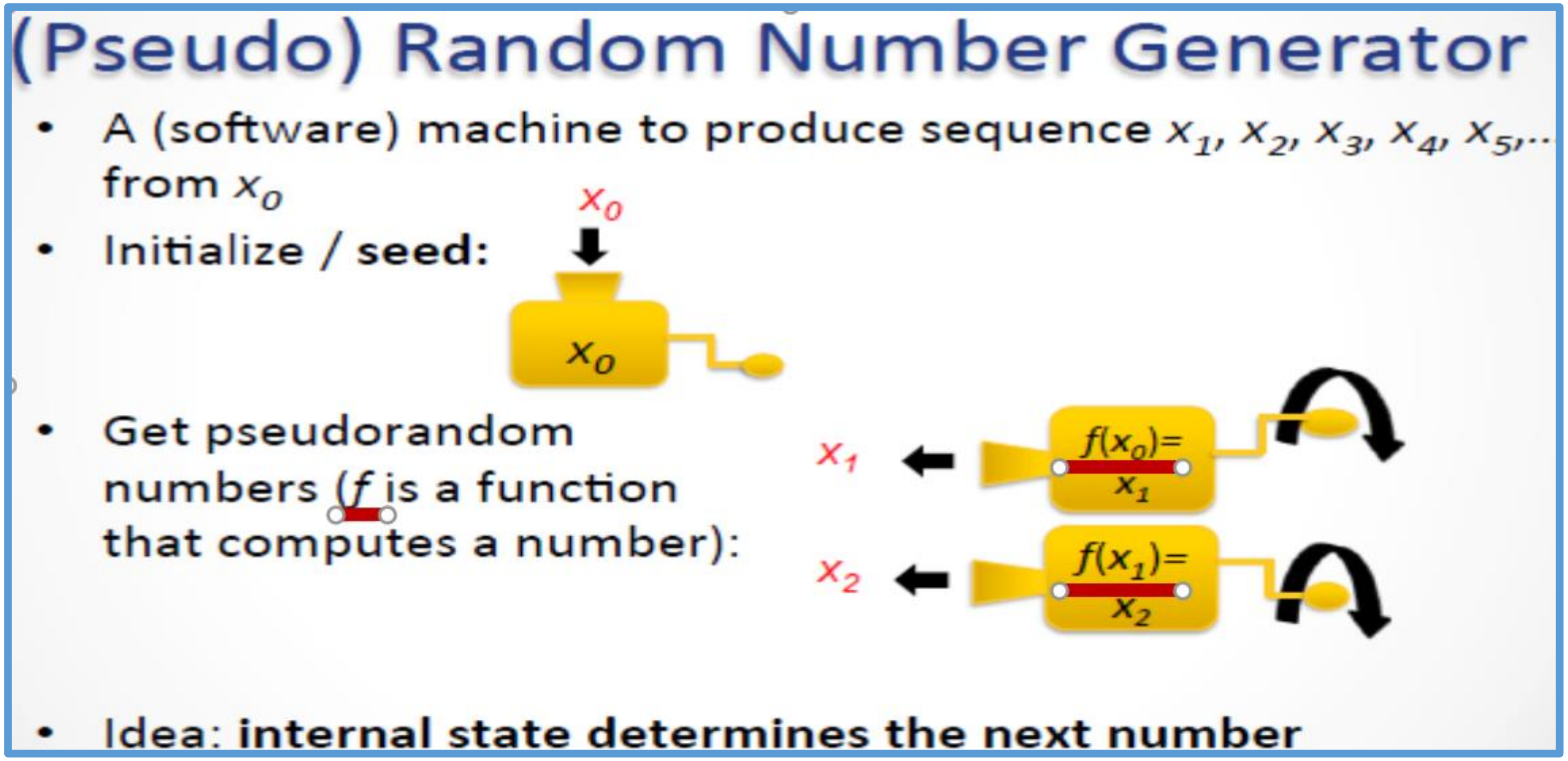
The company has grown to assist other governments, international organizations, private companies and foundations, with a host of defense and non-defense issues, including healthcare.

RAND has approximately 1,850 employees. Its American locations include: Santa Monica, California (headquarters); Arlington, Virginia; Pittsburgh, Pennsylvania; the San Francisco Bay Area; and Boston.

Pseudo Random Sequences

- Pseudo Random Number Generator (PRNG)
  - Algorithm that produces a sequence that looks random
    - Passes some randomness tests

- The sequence cannot be really random!
  - Because an algorithm produces known output, by definition

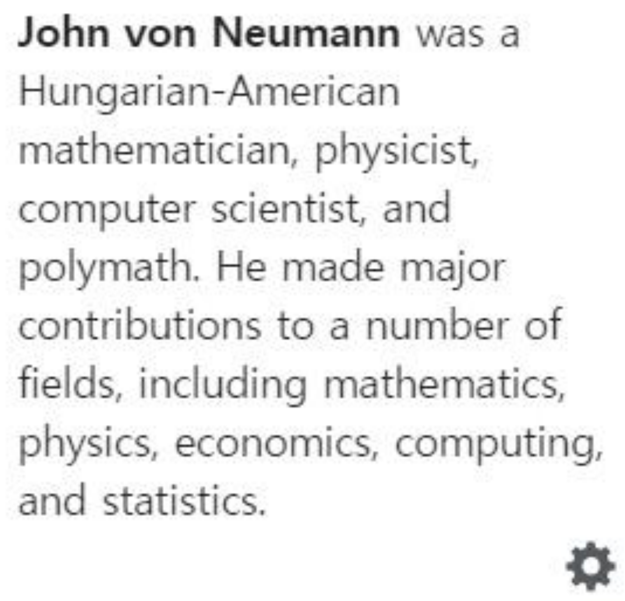


(Pseudo) Random Number Generator

- A (software) machine to produce sequence $x_1, x_2, x_3, x_4, x_5, \ldots$ from x_0
- Initialize / seed:
- Get pseudorandom numbers (f is a function that computes a number):
- Idea: internal state determines the next number

Inventor of PRNG

- An early computer-based PRNG, suggested by John von Neumann in 1946, is known as the middle-square method



John von Neumann was a Hungarian-American mathematician, physicist, computer scientist, and polymath. He made major contributions to a number of fields, including mathematics, physics, economics, computing, and statistics.

SIMPLE PRNGs with LCG formula

- *Linear congruential generator formula:* $x_{i+1} = (a\ x_i + c)\ \%\ m$
- a, c, and m are constants
- Good enough for many purposes
- ...if a, c, and m are properly chosen

- Linear Congruential Generator (LCG) Example

```
# a = 1, c = 7, m = 12
current_x = 0              # global internal state / seed
def prng_seed(s) :         # seed the generator
    global current_x
    current_x = s
def prng1(n):              # LCG
   return (n + 7) % 12
def prng() :               # state updater
   global current_x
   current_x = prng1(current_x)
   return current_x
```

```
>>> prng_seed(1)
>>> prng( ) for I in range(10)
```

First 12 numbers: 1, 8, 3, 10, 5, 0, 7, 2, 9, 4, 11, 6

Does this look random to you?

Example PRNG

```
def prng1(n):                    # LCG
    return (n + 7) % 12)  # a = 1, c = 7, m = 12
```

```
>>> prng_seed(5)
>>> prng1( ) for I in range(20)
```

- First 20 numbers:

```
5, 0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10,
5, 0, 7, 2, 9, 4, 11, 6
```

Random-looking?

- What do you think the next number in the sequence is?
- **Moral: just eyeballing the sequence not a good test of randomness!**
- This generator has a **period** that is too short: it repeats too soon.
- (What else do you notice if you look at it for a while?)

Another PRNG

```
def prng2(n):
  return (n + 8) % 12 # a=1, c=8, m=12
```

```
>>> prng_seed(8)
>>> [ prng2( ) for I in range(12) ]
```

```
>>> [ prng() for n in range(12) ]
[ prng() for n in range(12) ]
[8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4, 0]
```

Random-looking?

Moral: choice of a, c, and m crucial!

PRNG Period

- Let’ s define the PRNG *period* as the number of values in the sequence before it repeats.

```
def prng1(n):
    return (n + 7) % 12)
```

```
5, 0, 7, 2, 9, 4, 11, 6, 1, 8, 3,
10, 5, 0, 7, 2, 9, 4, 11, 6, ...
```

prng1, period = 12

next number = (last number + 7) mod 12

```
def prng2(n):
    return (n + 8) % 12
```

```
8, 4, 0, 8, 4, 0, 8, 4, 0, 8, ...
```

prng2, period = 3

next number = (last number + 8) mod 12

We want the longest period we can get!

Picking the Constants a, c, m [1/2]

- Large value for *m*, and appropriate values for *a* and *c* that work with this *m*

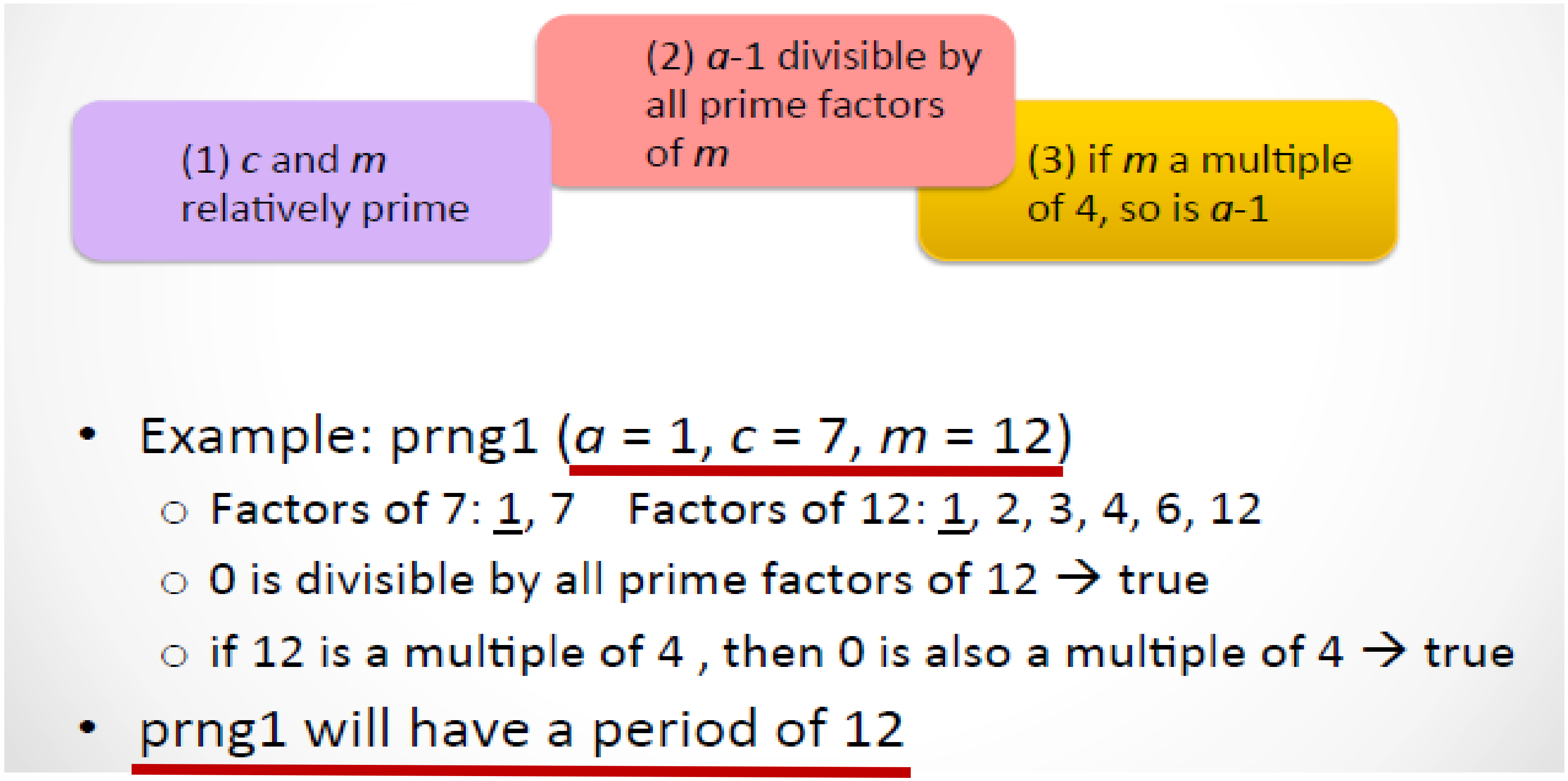
  $\Rightarrow$ a very long sequence before numbers begin to repeat.
- Maximum period is *m*

- The LCG will have a period of *m* (the maximum) if and only if:
  - *c* and *m* are *relatively prime* (i.e. the only positive integer that divides both *c* and *m* is 1)
  - *a*-1 is divisible by all prime factors of *m*
  - if *m* is a multiple of 4 , then *a*-1 is also a multiple of 4
- (*Number theory* tells us so)

Hull, T. E., Dobell, A. R. (July 1962). "Random Number Generators". *SIAM Review.*

Picking the Constants a, c, m [2/2]

- *Linear congruential generator formula:*
  $x_{i+1} = (a\ x_i + c)\ \%\ m$



(1) *c* and *m* relatively prime

(2) *a*-1 divisible by all prime factors of *m*

(3) if *m* a multiple of 4, so is *a*-1

- Example: prng1 ($\underline{a = 1, c = 7, m = 12}$)
  - Factors of 7: $\underline{1}$, 7 Factors of 12: $\underline{1}$, 2, 3, 4, 6, 12
  - 0 is divisible by all prime factors of 12 → true
  - if 12 is a multiple of 4 , then 0 is also a multiple of 4 → true
- $\underline{\text{prng1 will have a period of 12}}$

Exercise for you

- *Linear congruential generator formula:* $x_{i+1} = (a\ x_i + c)\ \%\ m$

(1) c and m relatively prime

(2) a-1 divisible by all prime factors of m

(3) if m a multiple of 4, so is a-1

$$\underline{x_{i+1} = (5x_i + 3) \text{ modulo } 8}$$

$x_0 = 4$ $a = 5$ $c = 3$ $m = 8$

- What is the period of this generator? Why?
- Compute x_1, x_2, x_3 for this LCG formula.

LCGs in the Real World

- *Linear congruential generator formula:* $x_{i+1} = (a\, x_i + c) \% m$
- a, c, and m are constants

- glibc (used by the compiler gcc for the C language): $a = 1103515245$, $c = 12345$, $m = 2^{32}$
- *Numerical Recipes* (popular book on numerical methods and analysis): $a = 1664525$, $c = 1013904223$, $m = 2^{32}$
- Random class in Java: $a = 25214903917$, $c = 11$, $m = 2^{48}$

Some pitfalls of PRNGs

- Still, Predictable Seed
  - Example: famous Netscape security flaw caused by using system time
- Still, Repeated Seed
  - When running many applications at the same time
- Hidden Correlations

Finding Hidden Correlations [1/2]

P. Hellekalek / Mathematics and Computers in Simulation 46 (1998) 485–505

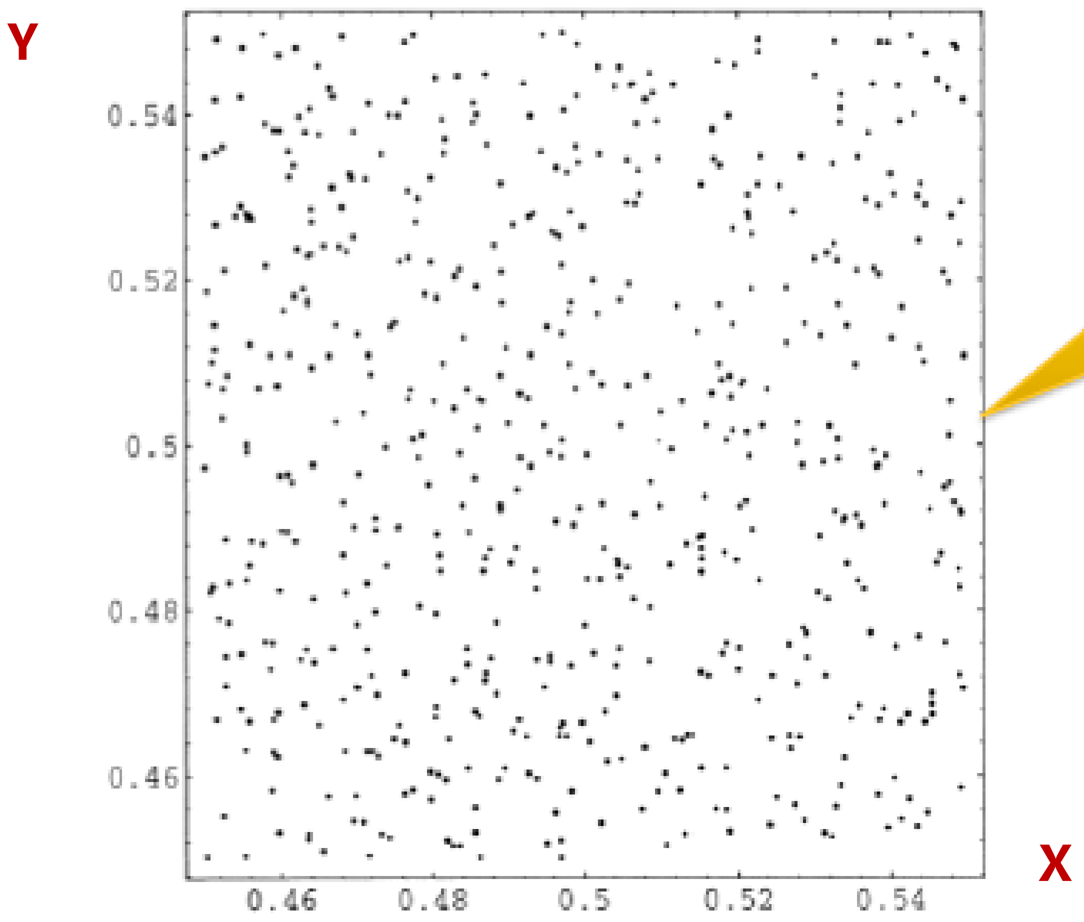


Fig. 1. LCG(2^{31}, 65539, 0, 1) Dimension 2: Zoom into the unit interval.

Looking good!

Finding Hidden Correlations [2/2]

P. Hellekalek/Mathematics and Computers in Simulation 46 (1998) 485–505

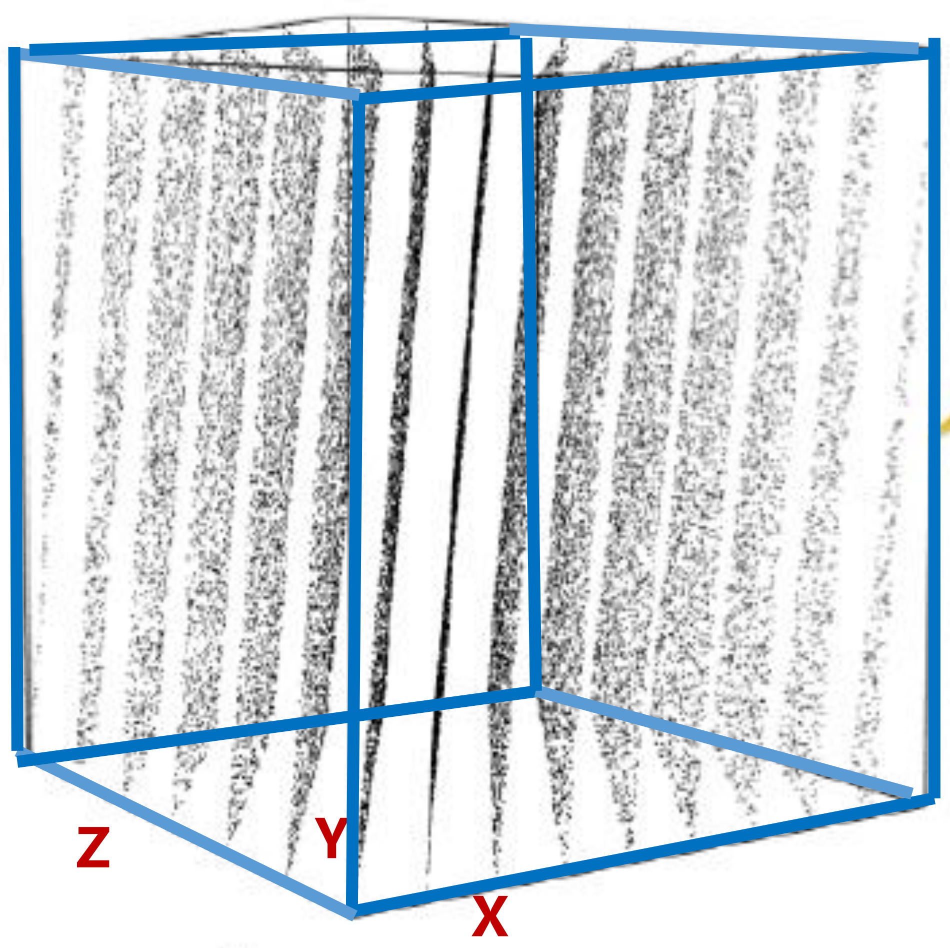
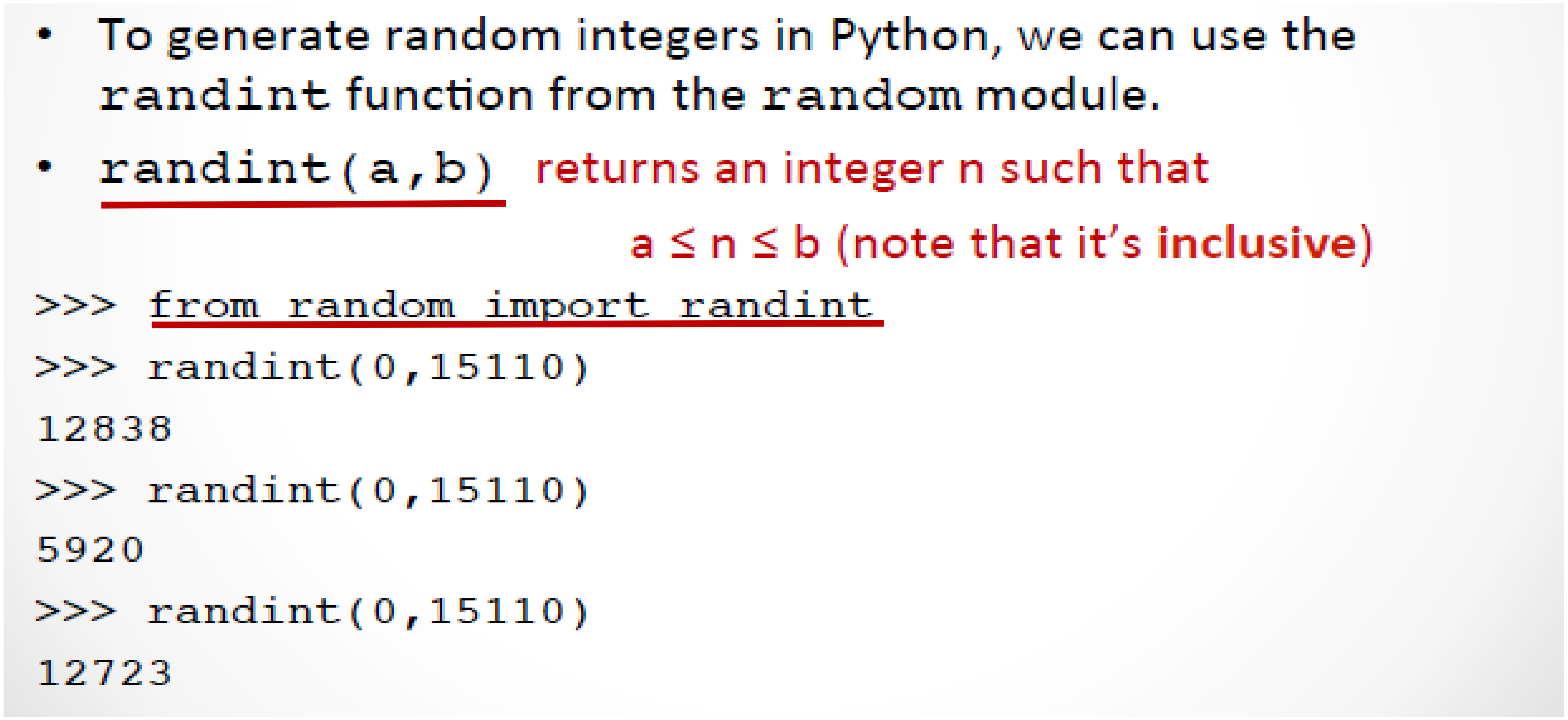


Fig. 2. LCG(2^{31}, 65539, 0, 1) Dimension 3: The 15 planes.

“random” Module in Python [1/4]

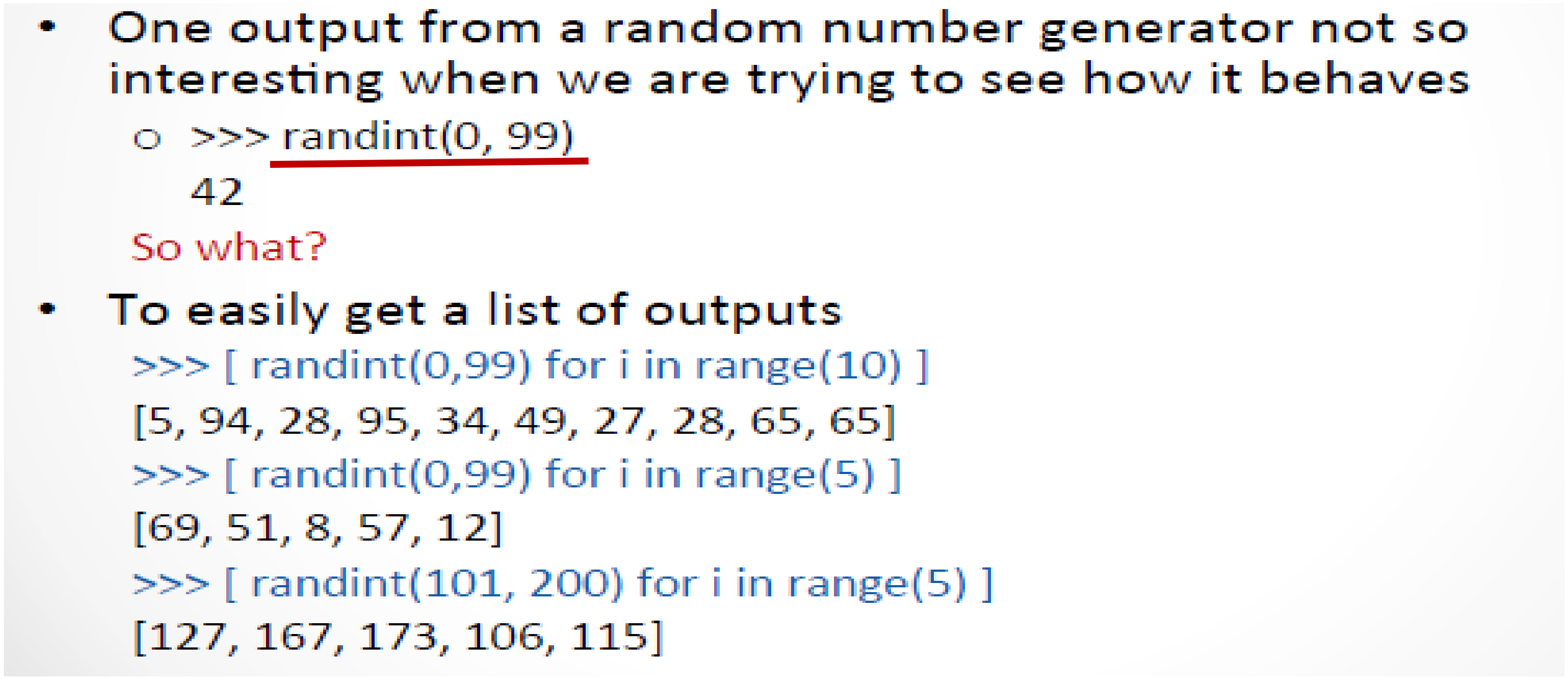


- To generate random integers in Python, we can use the `randint` function from the `random` module.
- `randint(a,b)` returns an integer n such that

  a ≤ n ≤ b (note that it’s **inclusive**)

```
>>> from random import randint
>>> randint(0,15110)
12838
>>> randint(0,15110)
5920
>>> randint(0,15110)
12723
```

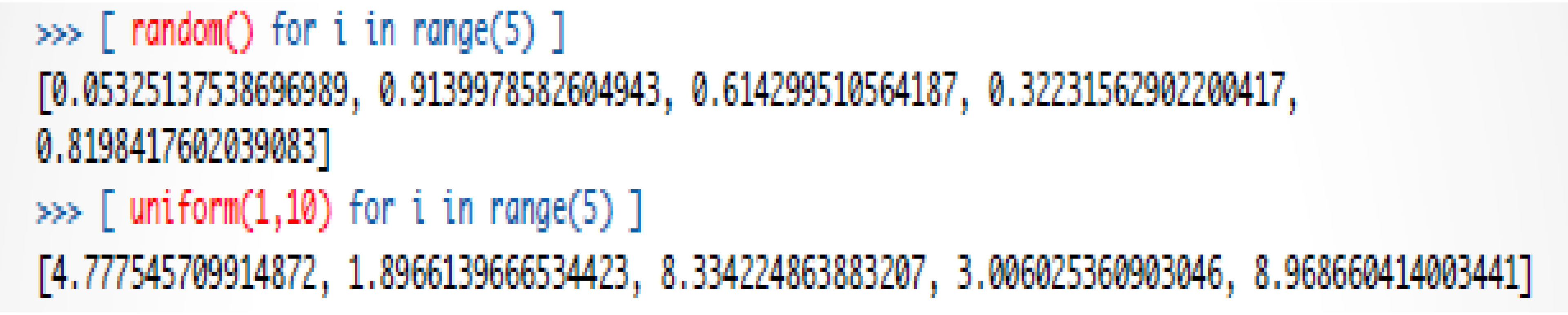
“random” Module in Python [2/4]



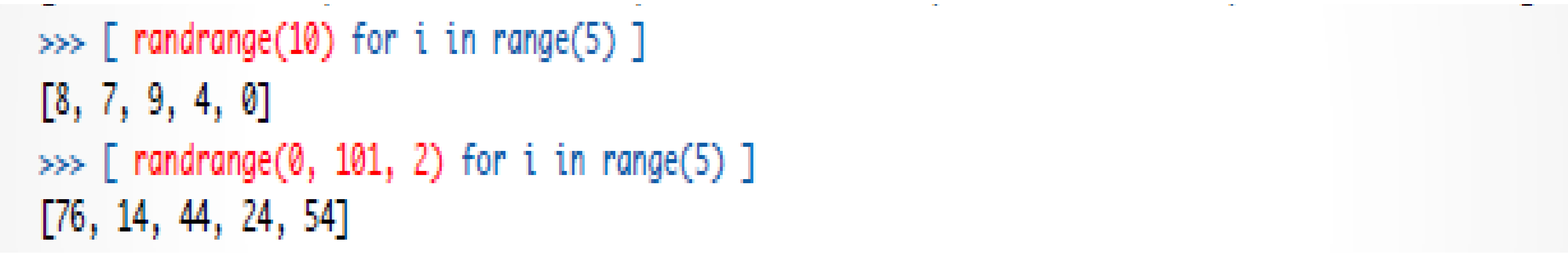
- One output from a random number generator not so interesting when we are trying to see how it behaves
  ```
  o >>> randint(0, 99)
    42
  ```
  So what?
- To easily get a list of outputs
  ```
  >>> [ randint(0,99) for i in range(10) ]
  [5, 94, 28, 95, 34, 49, 27, 28, 65, 65]
  >>> [ randint(0,99) for i in range(5) ]
  [69, 51, 8, 57, 12]
  >>> [ randint(101, 200) for i in range(5) ]
  [127, 167, 173, 106, 115]
  ```

Simulation에 필요한 Random Number들을 List로

“random” Module in Python [3/4]

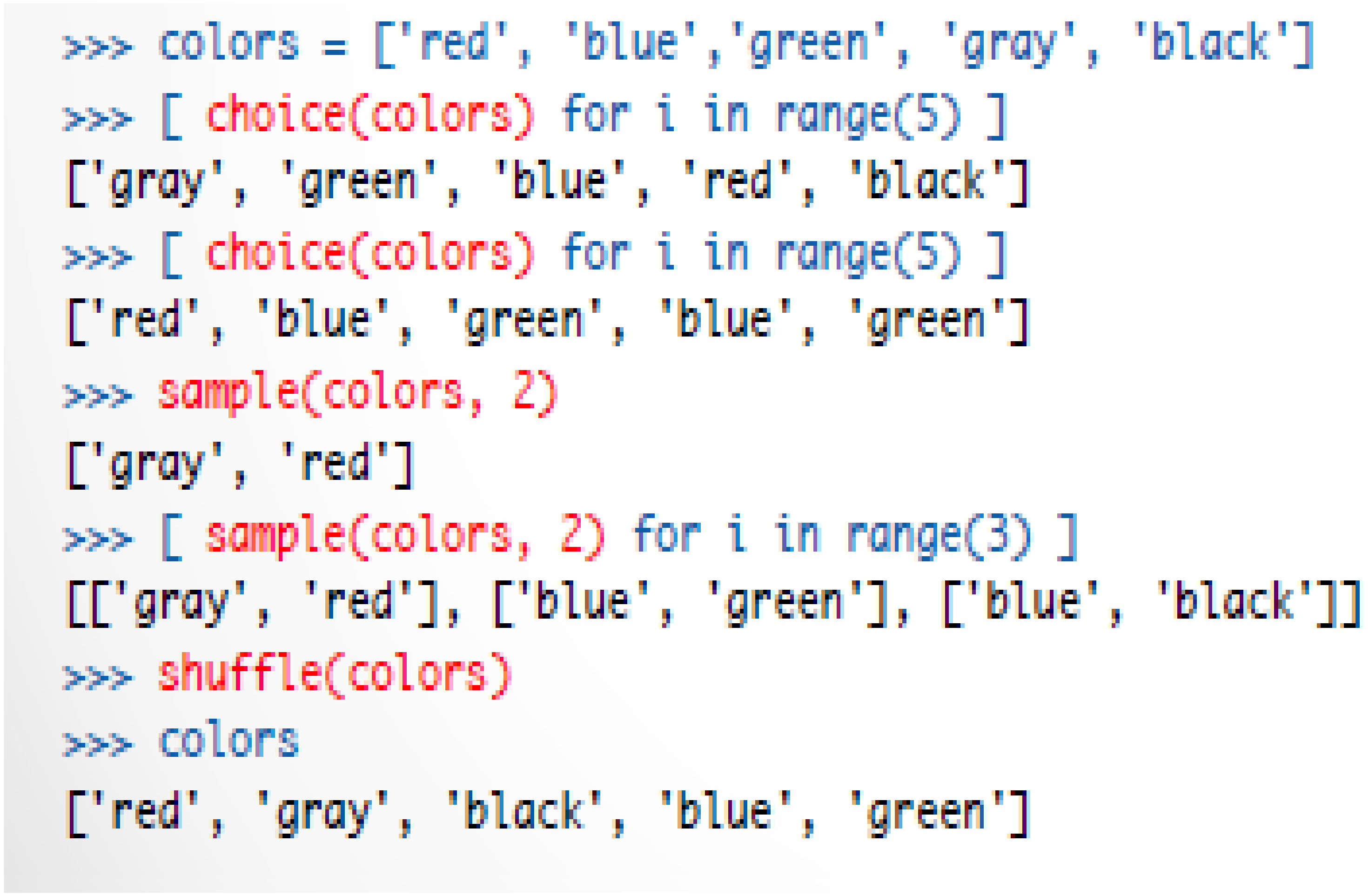


```
>>> [ random() for i in range(5) ]
[0.05325137538696989, 0.9139978582604943, 0.614299510564187, 0.3223156290220417,
0.8198417602039083]
>>> [ uniform(1,10) for i in range(5) ]
[4.777545709914872, 1.8966139666534423, 8.334224863883207, 3.006025360903046, 8.968660414003441]
```



```
>>> [ randrange(10) for i in range(5) ]
[8, 7, 9, 4, 0]
>>> [ randrange(0, 101, 2) for i in range(5) ]
[76, 14, 44, 24, 54]
```

“random” Module in Python [4/4]



```
>>> colors = ['red', 'blue','green', 'gray', 'black']
>>> [ choice(colors) for i in range(5) ]
['gray', 'green', 'blue', 'red', 'black']
>>> [ choice(colors) for i in range(5) ]
['red', 'blue', 'green', 'blue', 'green']
>>> sample(colors, 2)
['gray', 'red']
>>> [ sample(colors, 2) for i in range(3) ]
[['gray', 'red'], ['blue', 'green'], ['blue', 'black']]
>>> shuffle(colors)
>>> colors
['red', 'gray', 'black', 'blue', 'green']
```

중요

Major Functions in Random Module

- randint(n1, n2) ➔ n1 ~ n2사이의 random integer
- randrange(n1, n2) ➔ n1 ~ n2사이의 random integer
  - n2는 포함이 안됨, randrange(n1, n2, step)

- random( ) ➔ 0 ~ 1사이의 random float number
- uniform(n1, n2) ➔ n1 ~ n2사이의 random float number

- choice(L) ➔ L중에서 1개 선택
- sample(L, n) ➔ L중에서 n개 선택

- shuffle(L) ➔ L을 shuffling

Adjust Range with prng( )

- Suppose we have a LCG with period *n* (*n* is very large)
- ... but we want to play a game involving dice (each side of a die has a number of spots from 1 to 6)
- How do we take an integer between 0 and *n*, and obtain an integer between 1 and 6?
  - Forget about our LCG and use `randint(?,?)`
  - *Great, but how did they do that?*

what values should we use?

randint(1, 6)

prng( )를 이용해서 만들어 본다면!!!

- Specifically: our LCG is the Linear Congruential Generator of glib (period = 2^{31} = 2147483648)
- We call `prng()` and get numbers like 1533190675, 605224016, 450231881, 1443738446, ...
- We define:

```
def roll_die():
    roll = prng() % 6 + 1
    assert 1 <= roll and roll <= 6
    return roll
```

- What's the smallest possible value for `prng() % 6 ?`
- The largest possible?

Random Range with prng( )   [1/2]

- Instead of rolling dice, we want to pick a random (US) presidential election year between 1788 and 2012
  - election years always divisible by 4
- We still have the same LCG with period 2147483648. What do we do?
  - Forget about our LCG and use `randrange(1788, 2013, 4)`
  - *Great, but how did they do that?*

prng( )를 이용해서 만들어 본다면!!!

- Remember, `prng( )` gives numbers like
  1533190675, 605224016, 450231881, 1443738446, ...

```
def election_year() :
    year = ?
    assert 1788 <= year and year <= 2012 and year % 4 == 0
    return year
```

Random Range with prng( ) [2/2]

- First: think *how many numbers are there in the range we want? That is, how many elections from 1788 to 2012?*
  - 2012 – 1788? No!
  - (2012 – 1788) / 4? Not quite! (there's one extra)
  - (2012 – 1788) / 4 + 1 = 57 elections
  - So let's randomly generate a number from 0 to 56 inclusive:

```
def election_year() :
    election_number = prng() % ( (2012 - 1788) // 4  + 1)
    assert 0 <= election_number and election_number <= 56
    year = ?
    assert 1788 <= year and year <= 2012 and year % 4 == 0
    return year
```

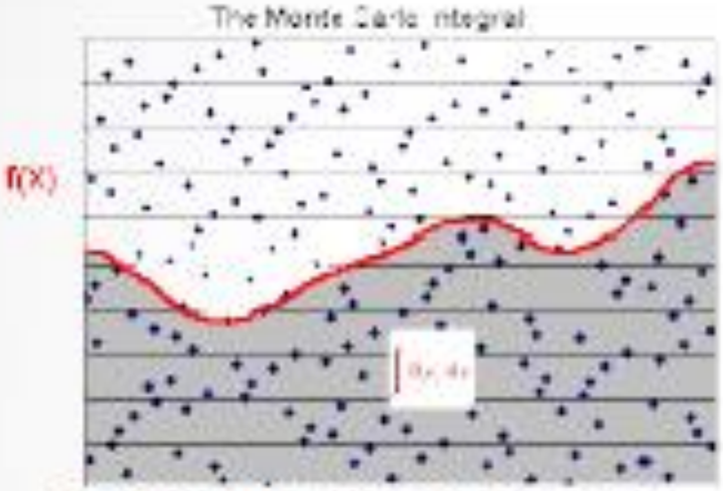
- Okay, but now we have random integers from 0 through 56
  - good, since there have been 57 elections
  - bad, since we want years, not election numbers 0 ... 56

```
def election_year() :
    election_number = prng() % ( (2012 - 1788) // 4  + 1)
    assert 0 <= election_number and election_number <= 56
    year = election_number * 4 + 1788
    assert 1788 <= year and year <= 2012 and year % 4 == 0
    return year
```

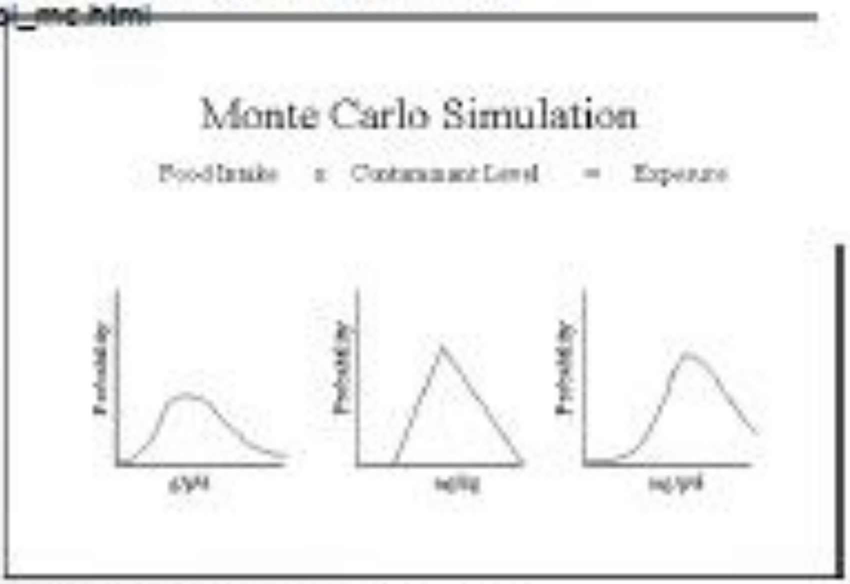
```
>>> [ election_year() for i in range(10) ]
[1976, 1912, 1796, 1800, 1984, 1852, 1976, 1804, 1992, 1972]
```

```
>>> randrange(1788, 2012, 4)
```

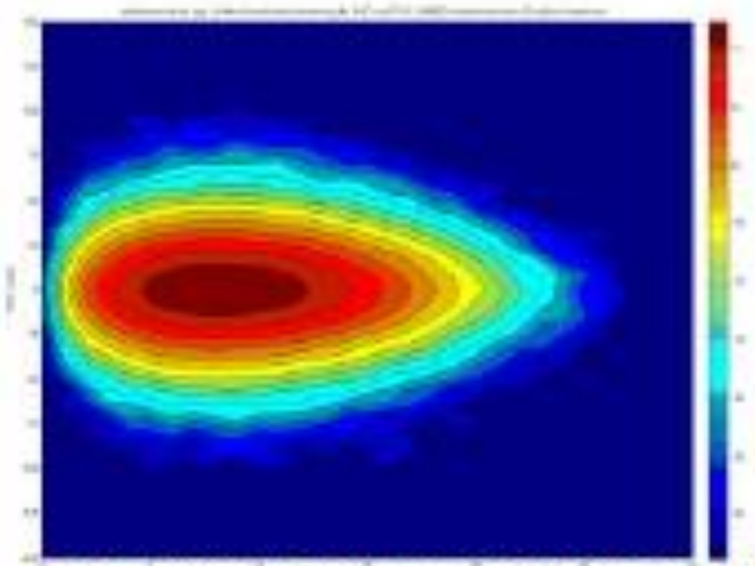
Some Applications using Random Numbers



http://marcoagd.usuarios.rdc.puc-rio.br/quasi_mc.html



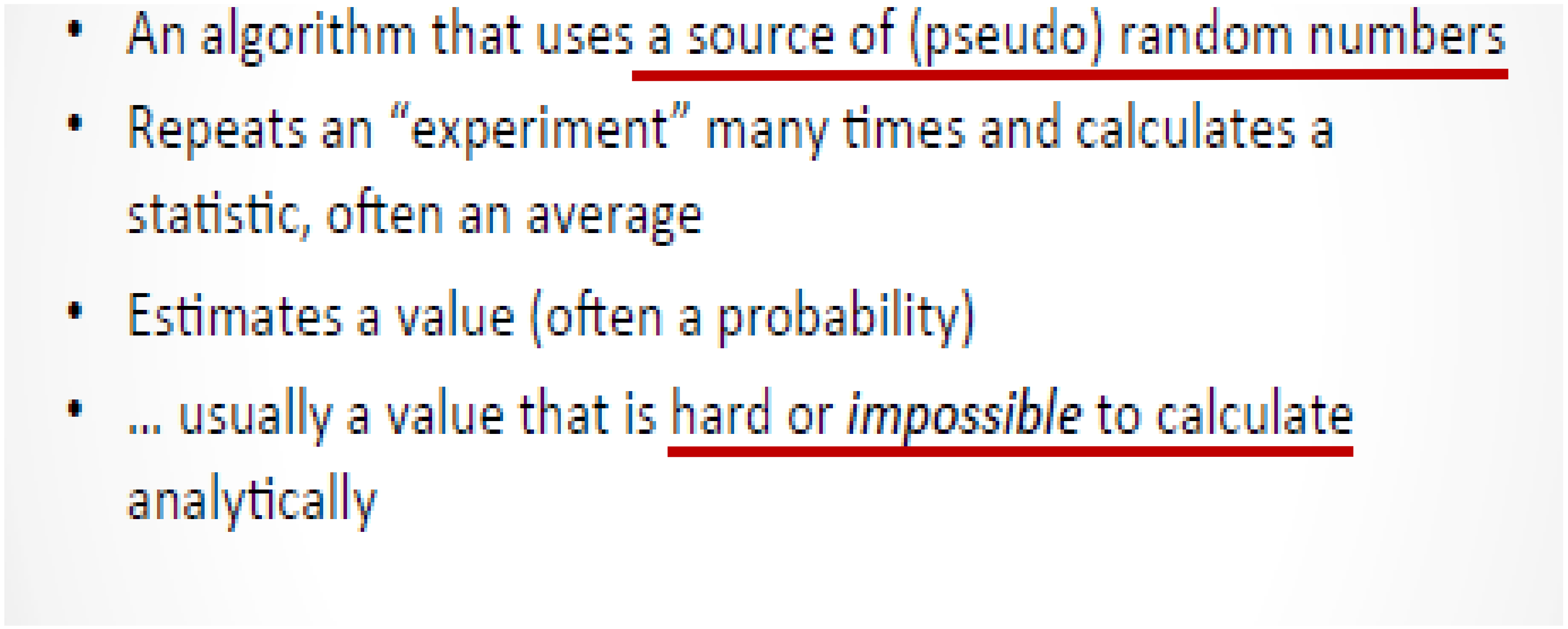
US Food and Drug Administration



Dr.-Ing. Matthias Westhäuser. Statistical Analysis of Fiber Optical Systems using Multicanonical Monte Carlo Methods (http://www.hft.e-technik.tu-dortmund.de/forschung/projekt.php?id=18&lang=en)

Table of Contents

What is a Monte Carlo Method?



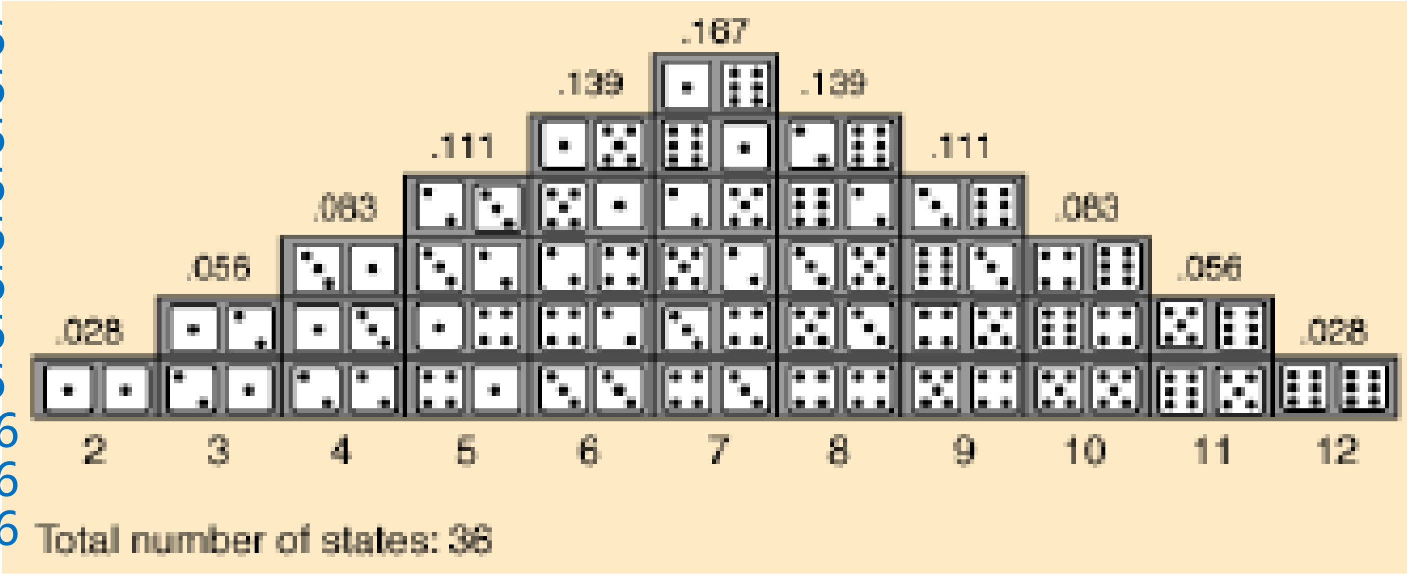
- An algorithm that uses a source of (pseudo) random numbers
- Repeats an "experiment" many times and calculates a statistic, often an average
- Estimates a value (often a probability)
- … usually a value that is hard or *impossible* to calculate analytically

Analytical Modeling VS Simulation

Dice Statistics Problem [1/2]

- We can **analyze** throwing a pair of dice and get the following probabilities for the sum of the two dice:

Prob(sum = 2) :	1 / 36
Prob(sum = 3) :	2 / 36
Prob(sum = 4) :	3 / 36
Prob(sum = 5) :	4 / 36
Prob(sum = 6) :	5 / 36
Prob(sum = 7) :	6 / 36
Prob(sum = 8) :	5 / 36
Prob(sum = 9) :	4 / 36
Prob(sum = 10) :	3 / 36
Prob(sum = 11) :	2 / 36
Prob(sum = 12) :	1 / 36

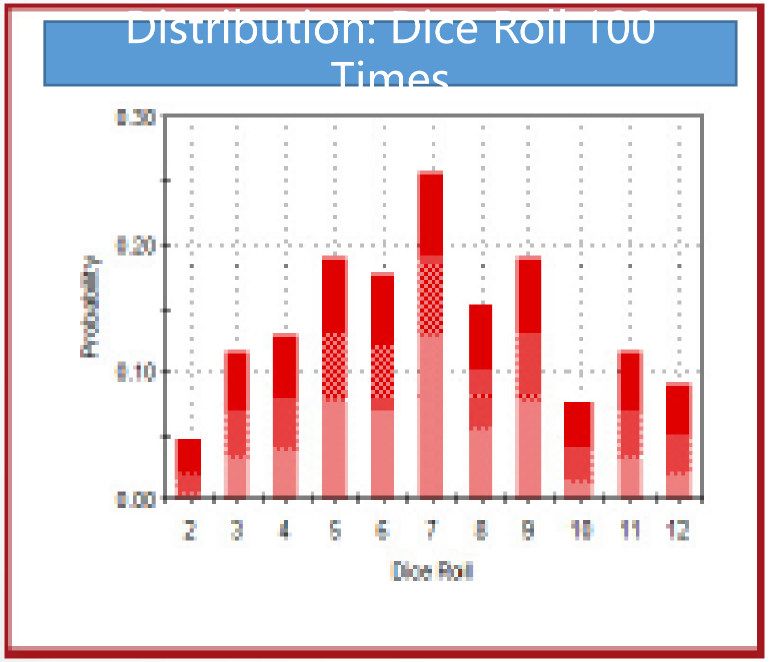


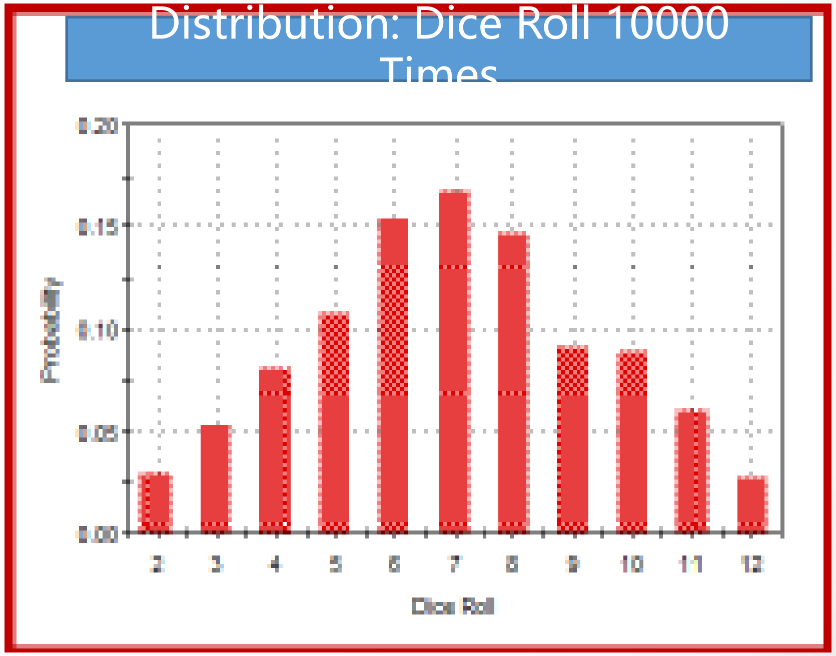
Prob(x) =

(x-1) / 36	2 <= x <= 7
(12 – ((x-1) mod 11)) / 36	8 <= x <= 12
0	Otherwise

Dice Statistics Problem [2/2]

- … **or** we can throw a pair of dice 100 times and record what happens, or 10000 times for a more accurate estimate.

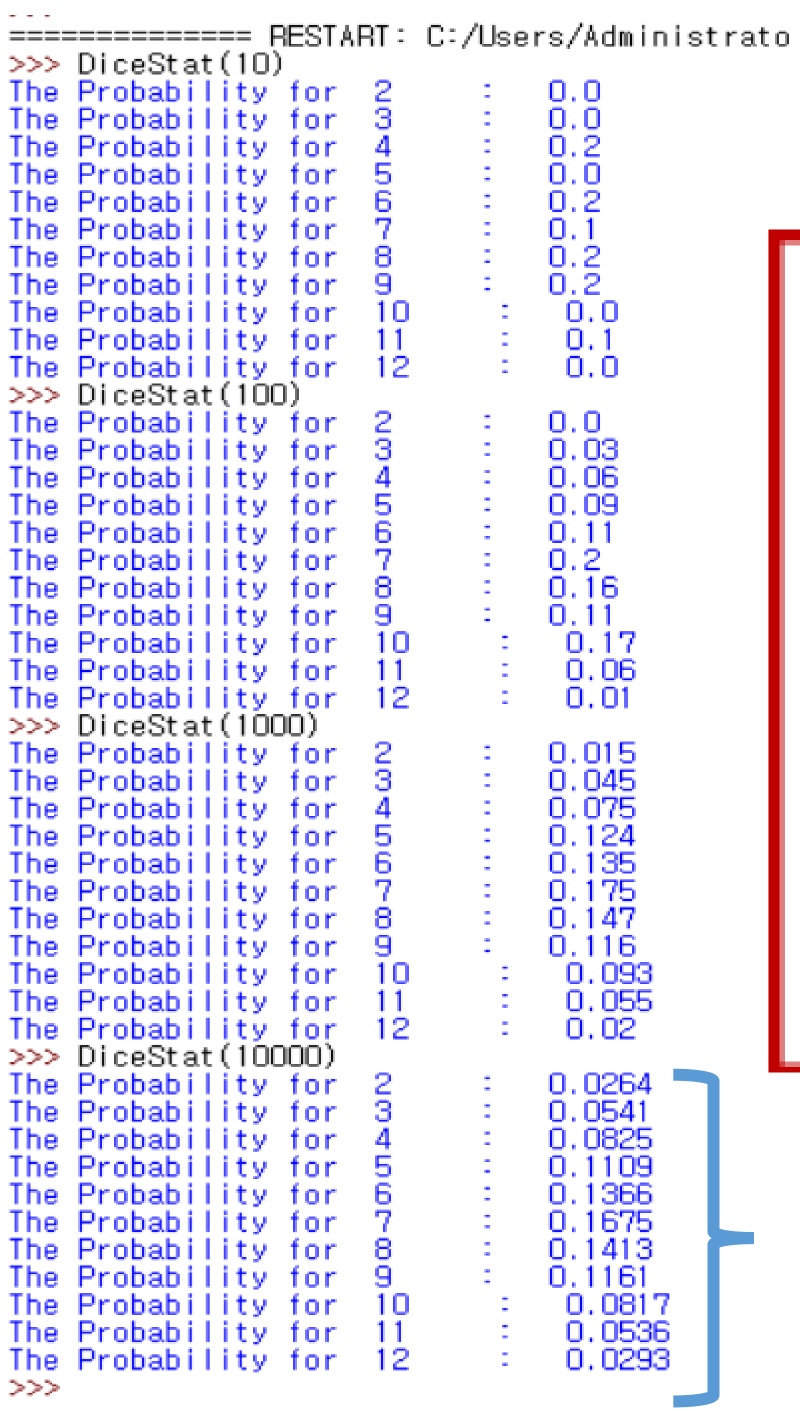




Dice Statistics Problem: Code

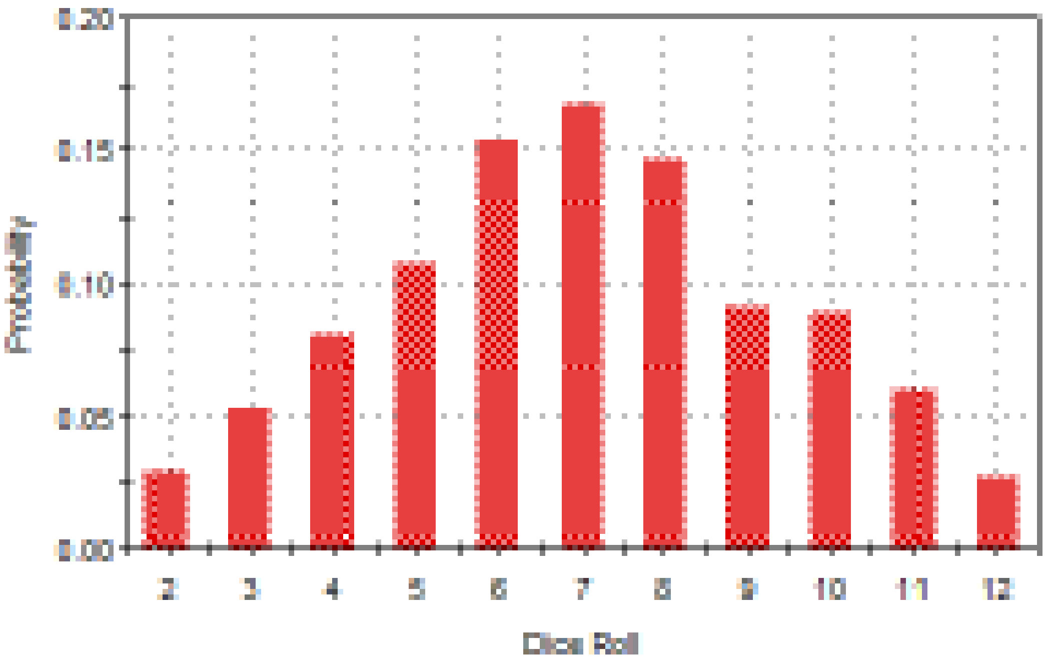
```
def DiceStat(trials):
    count_list   = [0,0,0,0,0,0,0,0,0,0,0]
    count_prob = [0,0,0,0,0,0,0,0,0,0,0]

    for i in range(trials):
        value1 = roll( )
        value2 = roll( )
        dice_index = value1 + value2  -  2
        count_list[dice_index]  =  count_list[dice_index]  +  1

    for j in range(0,11):
       count_prob[j] = count_list[j] / trials
       print("The probability for ",   j+2,  ": ",  count_prob[j])
```

```
def roll():
    from random import randint
    return randint(1,6)
```



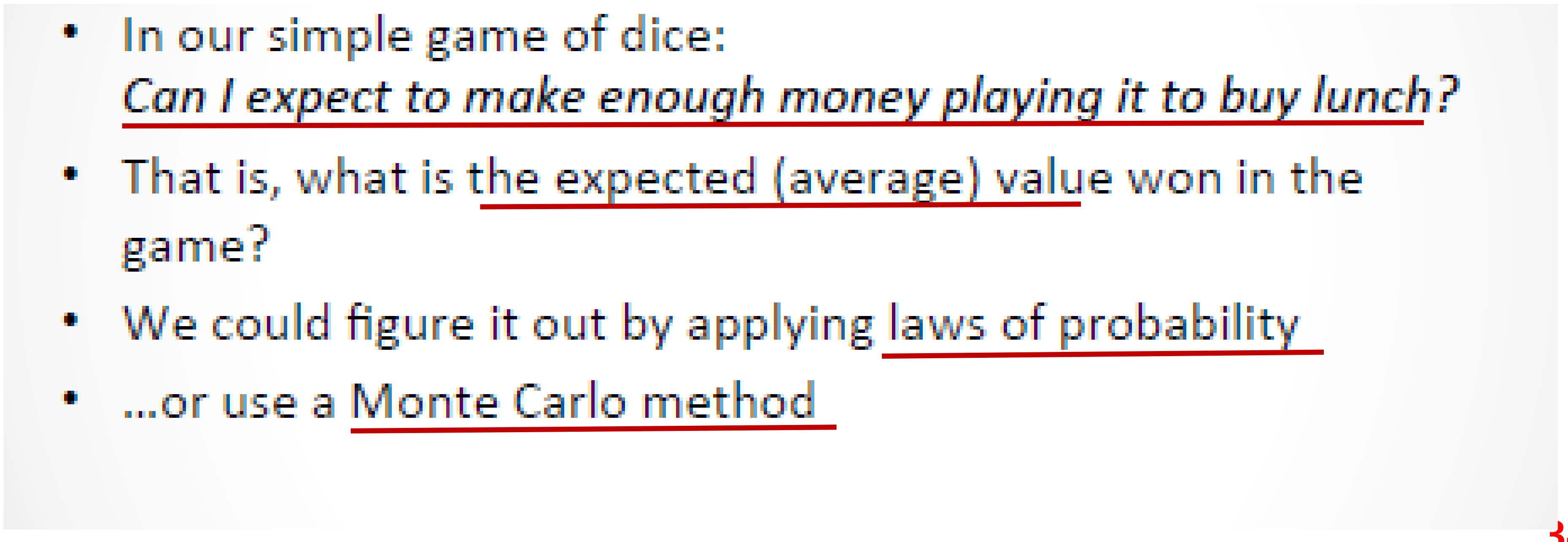
```
============== RESTART: C:/Users/Administrato
>>> DiceStat(10)
The Probability for 2     :   0.0
The Probability for 3     :   0.0
The Probability for 4     :   0.2
The Probability for 5     :   0.0
The Probability for 6     :   0.2
The Probability for 7     :   0.1
The Probability for 8     :   0.2
The Probability for 9     :   0.2
The Probability for 10     :   0.0
The Probability for 11     :   0.1
The Probability for 12     :   0.0
>>> DiceStat(100)
The Probability for 2     :   0.0
The Probability for 3     :   0.03
The Probability for 4     :   0.06
The Probability for 5     :   0.09
The Probability for 6     :   0.11
The Probability for 7     :   0.2
The Probability for 8     :   0.16
The Probability for 9     :   0.11
The Probability for 10     :   0.17
The Probability for 11     :   0.06
The Probability for 12     :   0.01
>>> DiceStat(1000)
The Probability for 2     :   0.015
The Probability for 3     :   0.045
The Probability for 4     :   0.075
The Probability for 5     :   0.124
The Probability for 6     :   0.135
The Probability for 7     :   0.175
The Probability for 8     :   0.147
The Probability for 9     :   0.116
The Probability for 10     :   0.093
The Probability for 11     :   0.055
The Probability for 12     :   0.02
>>> DiceStat(10000)
The Probability for 2     :   0.0264
The Probability for 3     :   0.0541
The Probability for 4     :   0.0825
The Probability for 5     :   0.1109
The Probability for 6     :   0.1366
The Probability for 7     :   0.1675
The Probability for 8     :   0.1413
The Probability for 9     :   0.1161
The Probability for 10     :   0.0817
The Probability for 11     :   0.0536
The Probability for 12     :   0.0293
>>>
```

Distribution: Dice Roll 10000 Times



The Hungry Dice Player Problem [1/3]

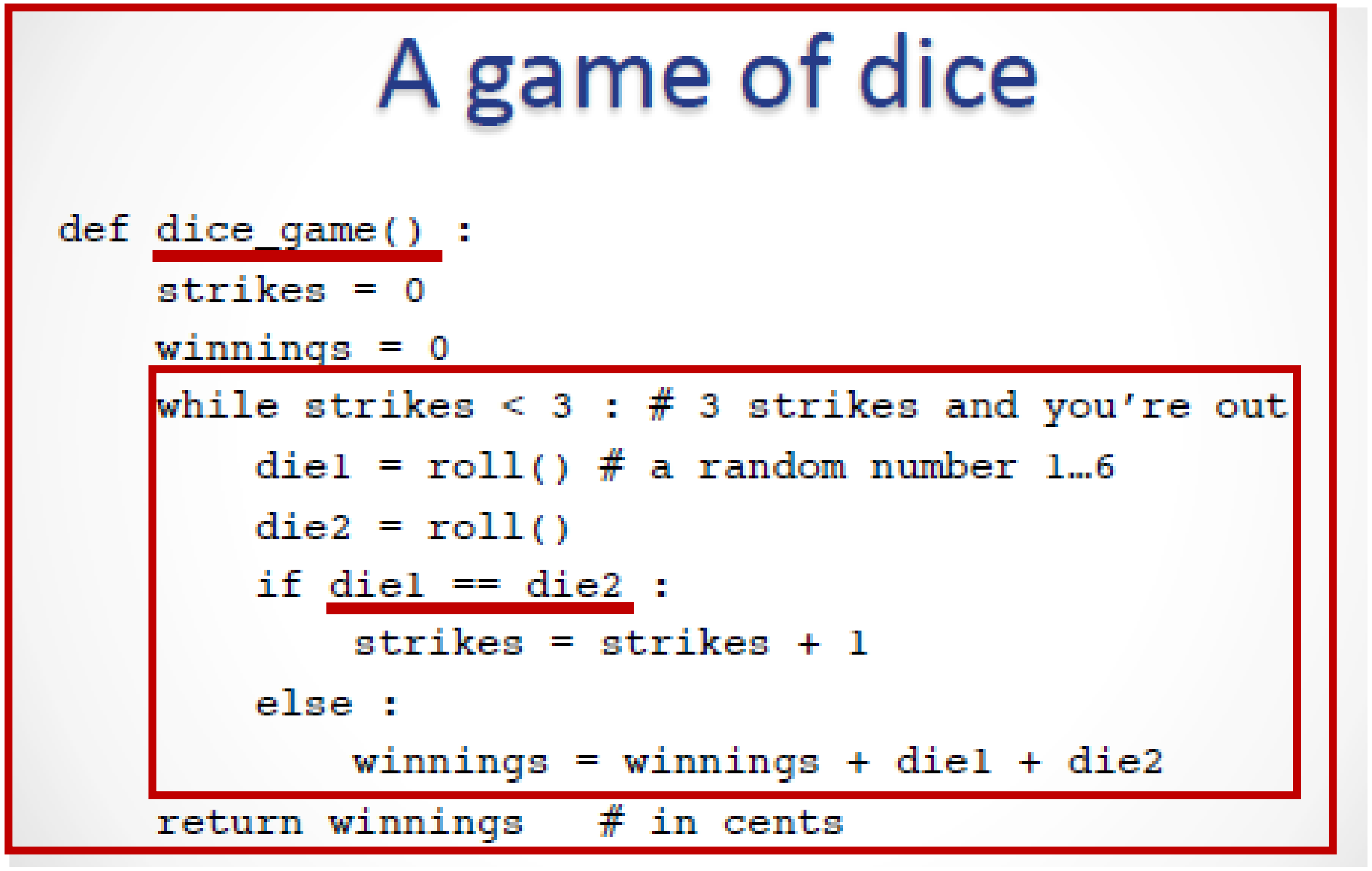
주사위 두개를 던져서 같은 값이 세번 나오기전까지 던져진 주사위 두개의 윗면에 나온 숫자들 만큼의 cent를 모아서 lunch를 사먹는 게임



- In our simple game of dice:
  Can I expect to make enough money playing it to buy lunch?
- That is, what is the expected (average) value won in the game?
- We could figure it out by applying laws of probability
- ...or use a Monte Carlo method

The Hungry Dice Player Problem [2/3]

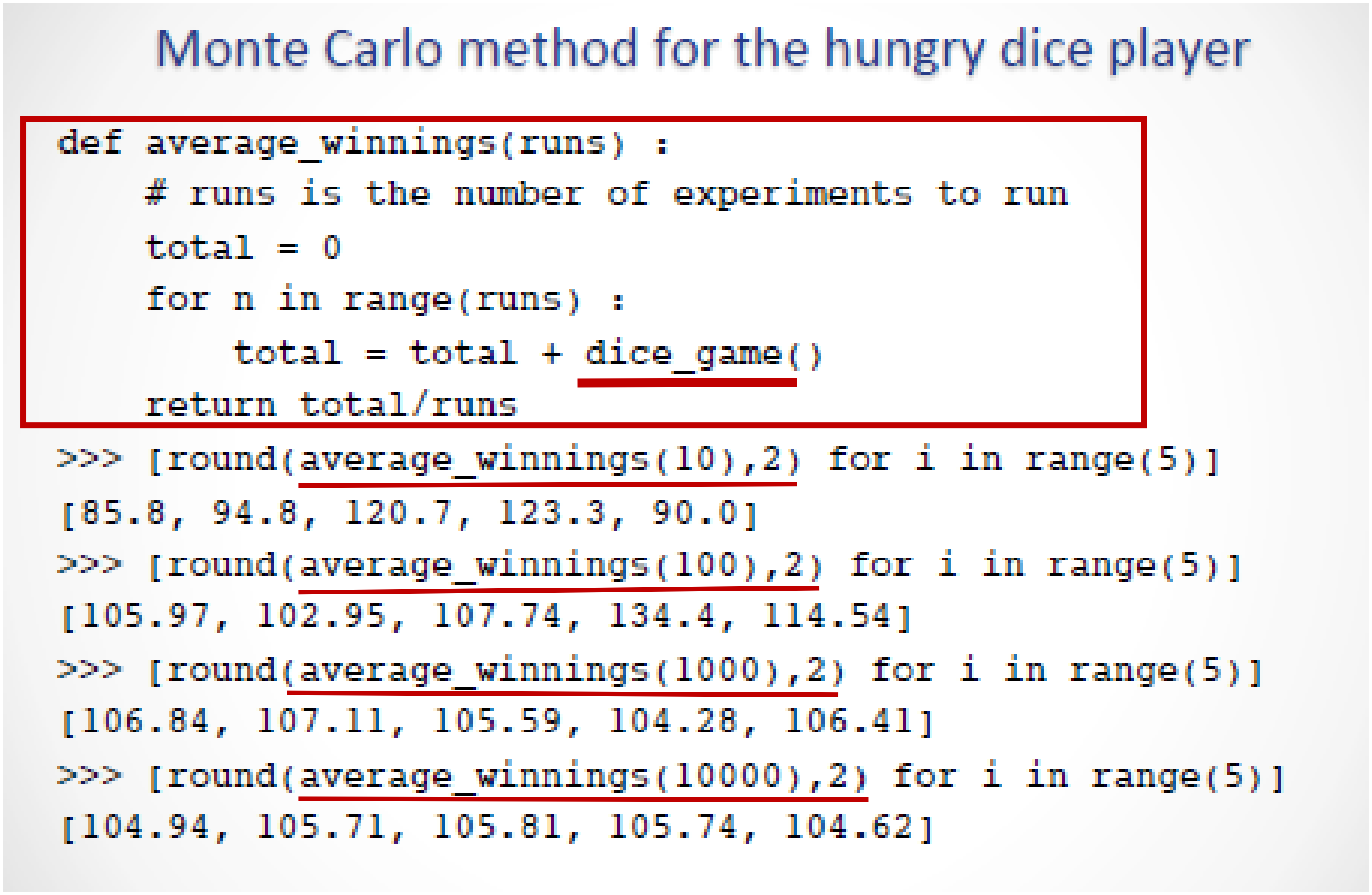
```
def roll():
    from random import randint
    return randint(1,6)
```



A game of dice

```
def dice_game() :
    strikes = 0
    winnings = 0
    while strikes < 3 : # 3 strikes and you're out
        die1 = roll() # a random number 1…6
        die2 = roll()
        if die1 == die2 :
            strikes = strikes + 1
        else :
            winnings = winnings + die1 + die2
    return winnings    # in cents
```

The Hungry Dice Player Problem [3/3]



Monte Carlo method for the hungry dice player

```
def average_winnings(runs) :
    # runs is the number of experiments to run
    total = 0
    for n in range(runs) :
        total = total + dice_game()
    return total/runs
>>> [round(average_winnings(10),2) for i in range(5)]
[85.8, 94.8, 120.7, 123.3, 90.0]
>>> [round(average_winnings(100),2) for i in range(5)]
[105.97, 102.95, 107.74, 134.4, 114.54]
>>> [round(average_winnings(1000),2) for i in range(5)]
[106.84, 107.11, 105.59, 104.28, 106.41]
>>> [round(average_winnings(10000),2) for i in range(5)]
[104.94, 105.71, 105.81, 105.74, 104.62]
```

참고

통밥으로 문제풀기

- 태종, 정조, 세종, 문종, 단종의 생일 맞추기 문제를 만약 통밥 (Wild Guess)으로 푼다면 평균 몇 개의 생일을 맞출수 있을까?

· 태종	(1) 4월 4일
· 정조	(2) 1월 27일
· 세종	(3) 3월 10일
· 문종	(4) 6월8일
· 단종	(5) 9월 16일

- Trial 1: (3), (2), (1), (4), (5)
- Trial 2: (4), (3), (1), (5), (2)

……

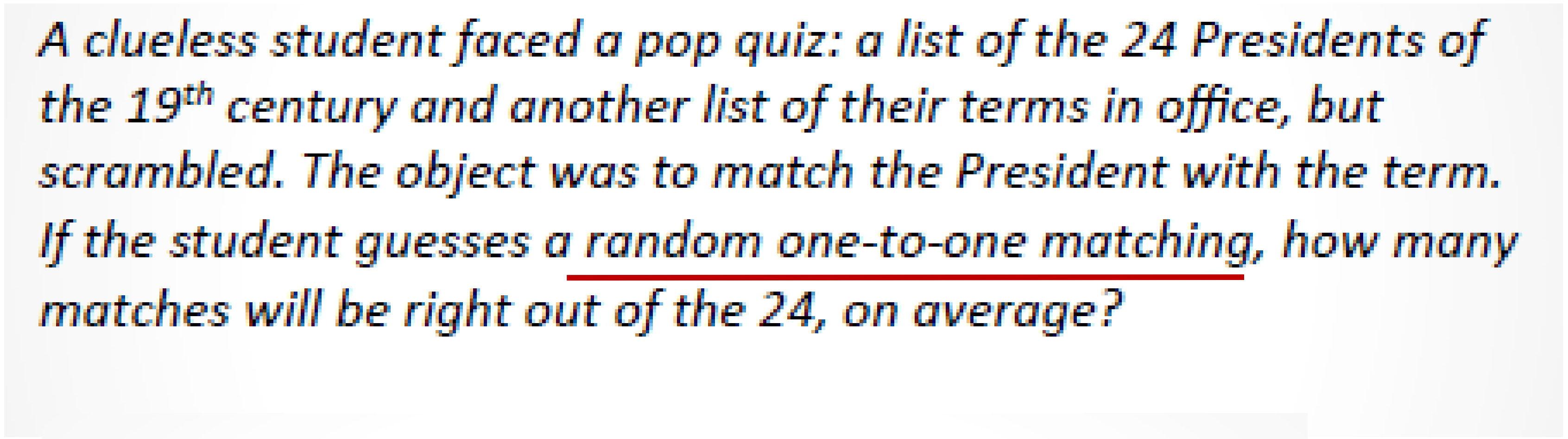
- Trial 10: (2), (4), (5), (1), (3)

정답: (4), (1), (5), (2), (3)

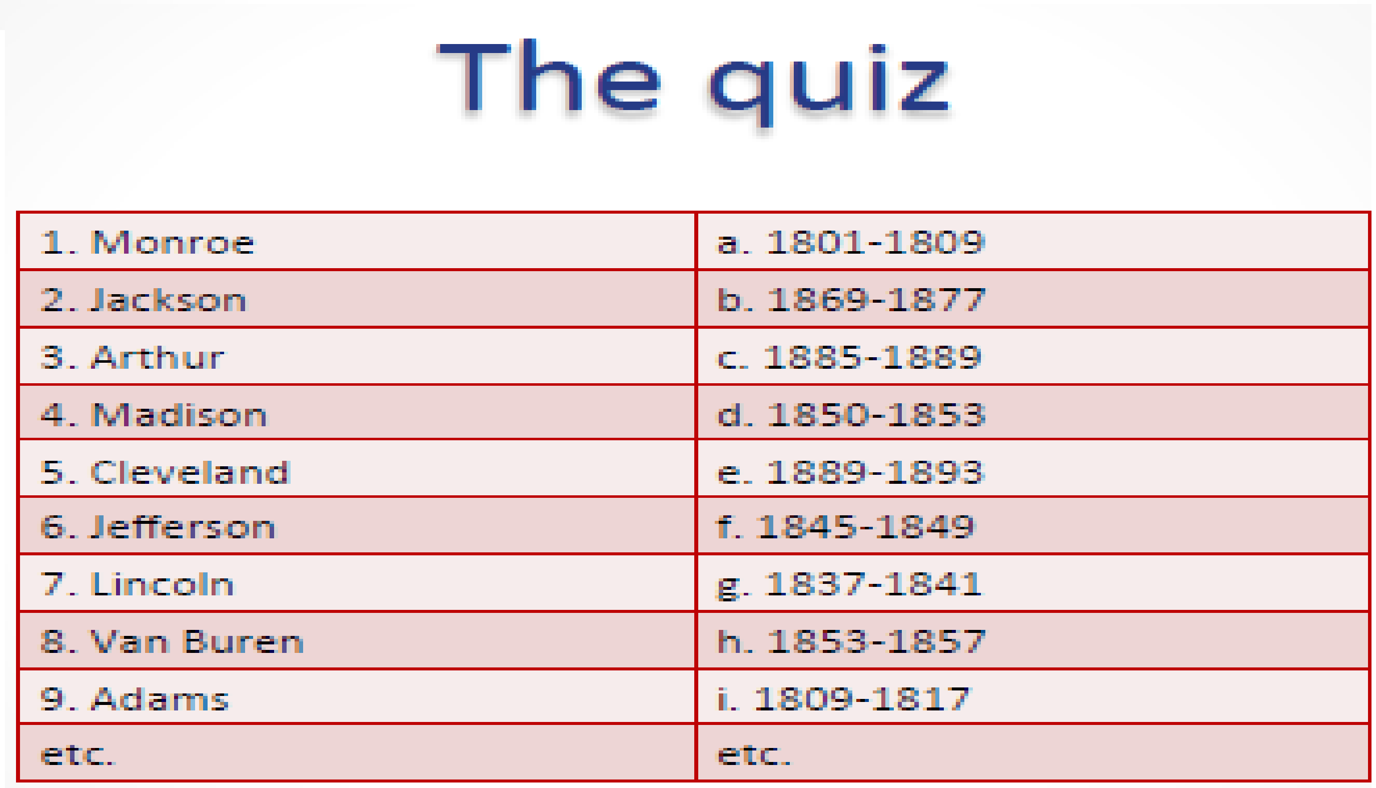
방법1: (1,2,3,4,5) 를 shuffle하고 정답과 비교해서 hit수를 합산

방법2: (1,2,3,4,5) 를 shuffle하고 (1,2,3,4,5)와 비교해서 hit수를 합산

The Clueless Student Problem [1/8]



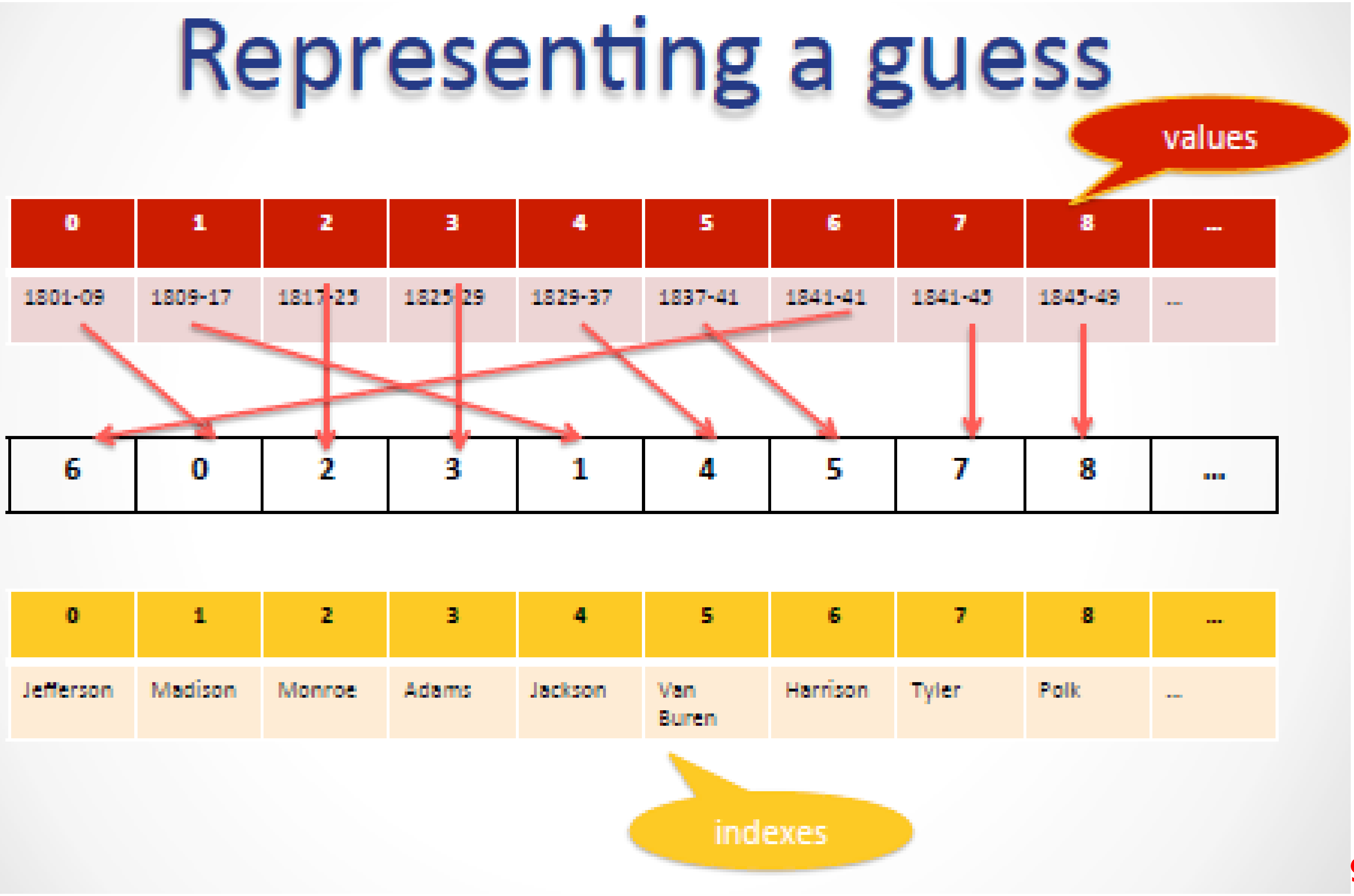
A clueless student faced a pop quiz: a list of the 24 Presidents of the 19th century and another list of their terms in office, but scrambled. The object was to match the President with the term. If the student guesses a random one-to-one matching, how many matches will be right out of the 24, on average?



The quiz

1. Monroe	a. 1801-1809
2. Jackson	b. 1869-1877
3. Arthur	c. 1885-1889
4. Madison	d. 1850-1853
5. Cleveland	e. 1889-1893
6. Jefferson	f. 1845-1849
7. Lincoln	g. 1837-1841
8. Van Buren	h. 1853-1857
9. Adams	i. 1809-1817
etc.	etc.

The Clueless Student Problem [2/8]



Representing a guess

values

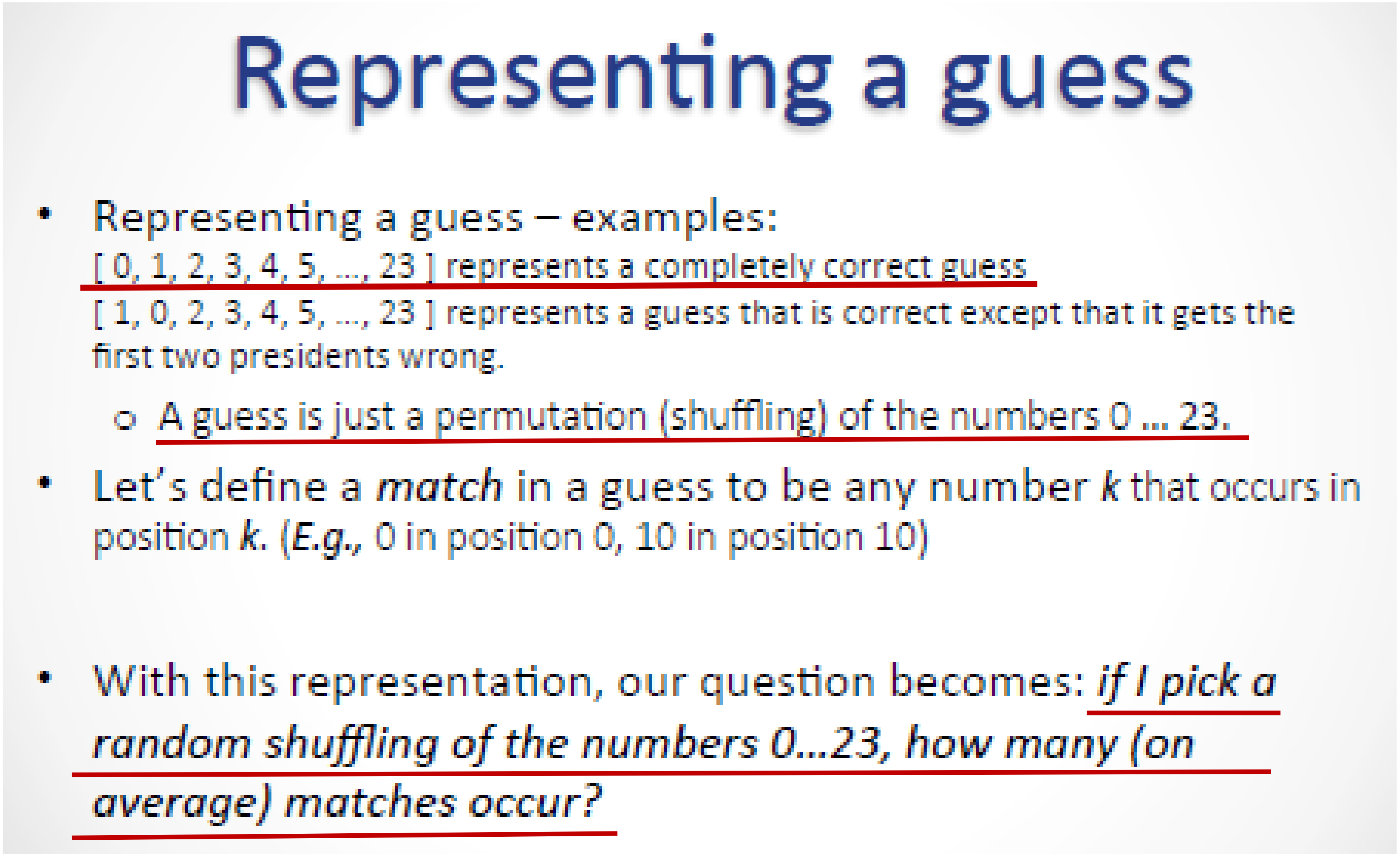
0	1	2	3	4	5	6	7	8	…
1801-09	1809-17	1817-25	1825-29	1829-37	1837-41	1841-41	1841-45	1845-49	…

6	0	2	3	1	4	5	7	8	…

0	1	2	3	4	5	6	7	8	…
Jefferson	Madison	Monroe	Adams	Jackson	Van Buren	Harrison	Tyler	Polk	…

indexes

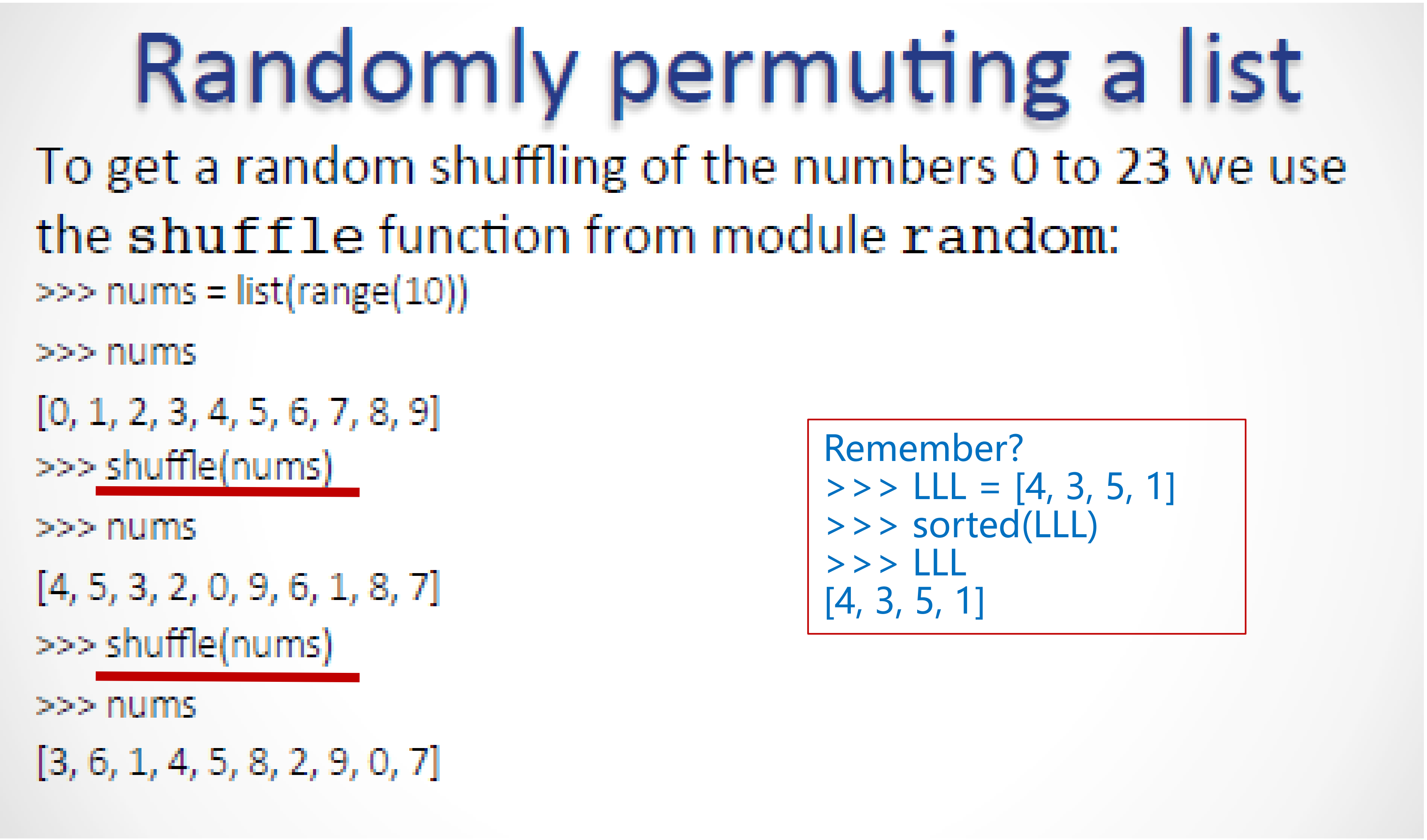
The Clueless Student Problem [3/8]



Representing a guess

- Representing a guess – examples:
  [0, 1, 2, 3, 4, 5, ..., 23] represents a completely correct guess
  [1, 0, 2, 3, 4, 5, ..., 23] represents a guess that is correct except that it gets the first two presidents wrong.
  - A guess is just a permutation (shuffling) of the numbers 0 ... 23.
- Let's define a *match* in a guess to be any number *k* that occurs in position *k*. (*E.g.*, 0 in position 0, 10 in position 10)
- With this representation, our question becomes: *if I pick a random shuffling of the numbers 0...23, how many (on average) matches occur?*

The Clueless Student Problem [4/8]



Randomly permuting a list

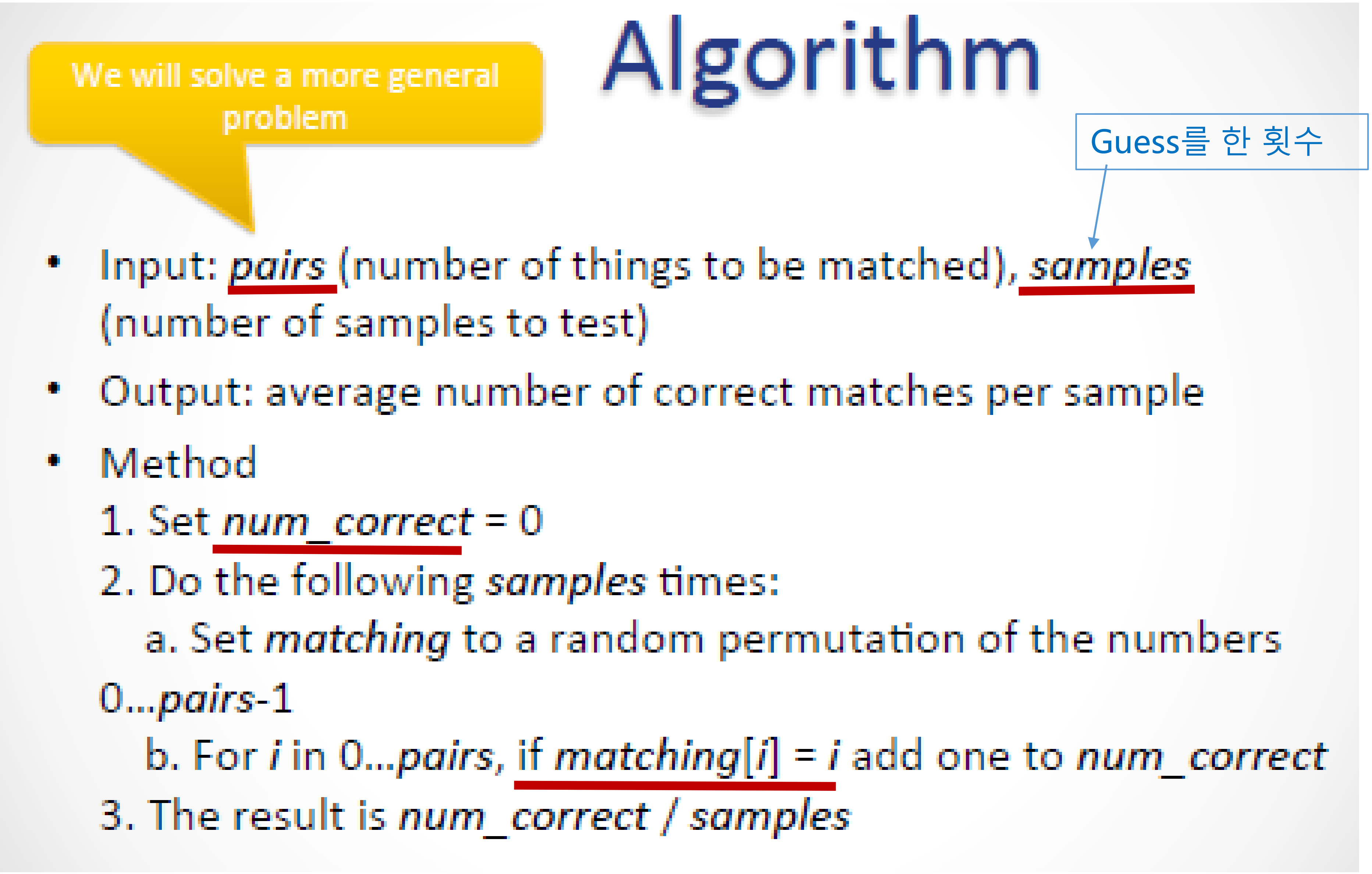
To get a random shuffling of the numbers 0 to 23 we use the `shuffle` function from module `random`:

```
>>> nums = list(range(10))
>>> nums
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> shuffle(nums)
>>> nums
[4, 5, 3, 2, 0, 9, 6, 1, 8, 7]
>>> shuffle(nums)
>>> nums
[3, 6, 1, 4, 5, 8, 2, 9, 0, 7]
```

Remember?

```
>>> LLL = [4, 3, 5, 1]
>>> sorted(LLL)
>>> LLL
[4, 3, 5, 1]
```

The Clueless Student Problem [5/8]



We will solve a more general problem

Algorithm

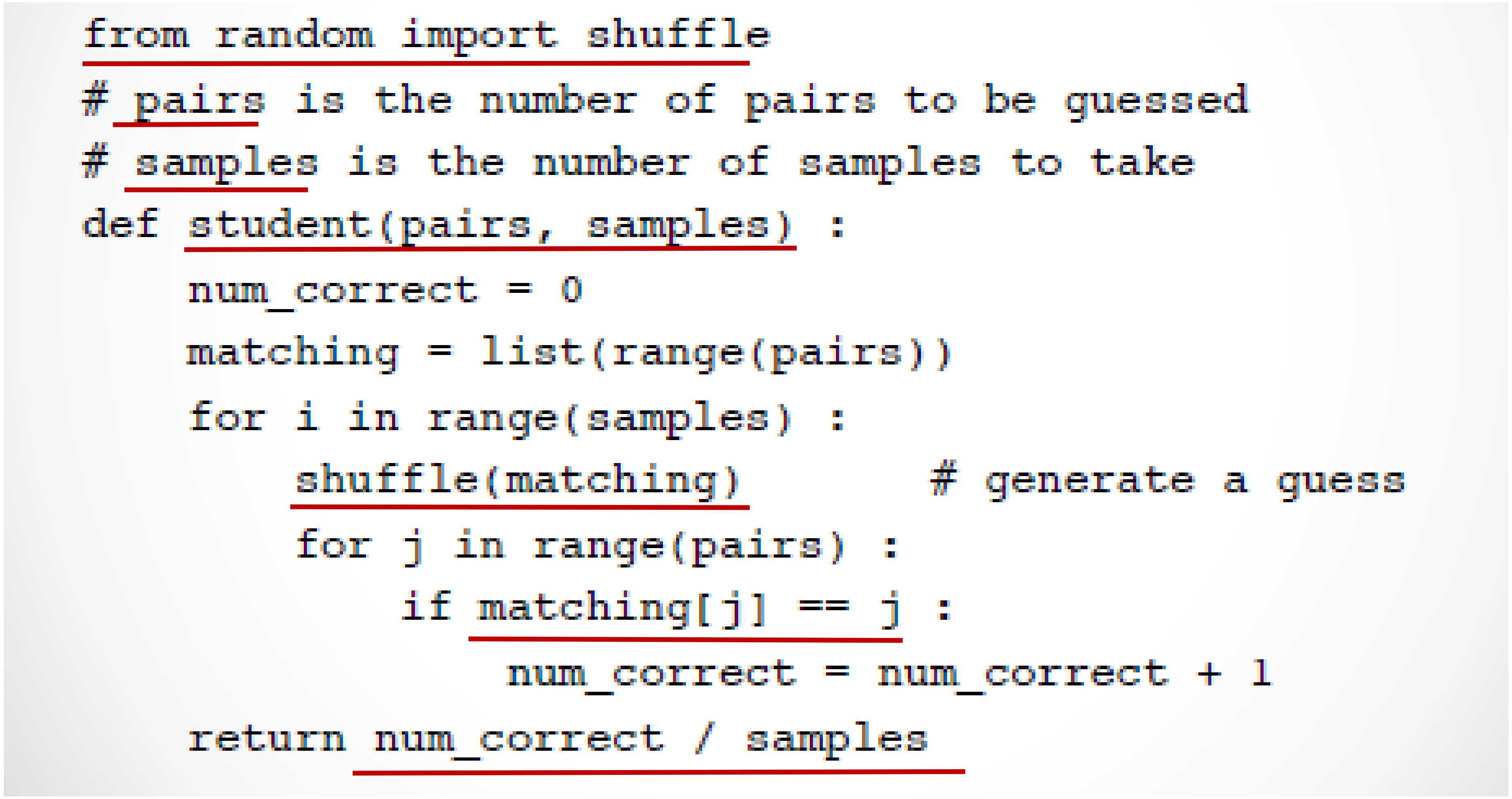
Guess를 한 횟수

- Input: *pairs* (number of things to be matched), *samples* (number of samples to test)
- Output: average number of correct matches per sample
- Method

1. Set *num_correct* = 0
2. Do the following *samples* times:
    a. Set *matching* to a random permutation of the numbers 0...*pairs*-1
    b. For *i* in 0...*pairs*, if *matching*[*i*] = *i* add one to *num_correct*
3. The result is *num_correct* / *samples*

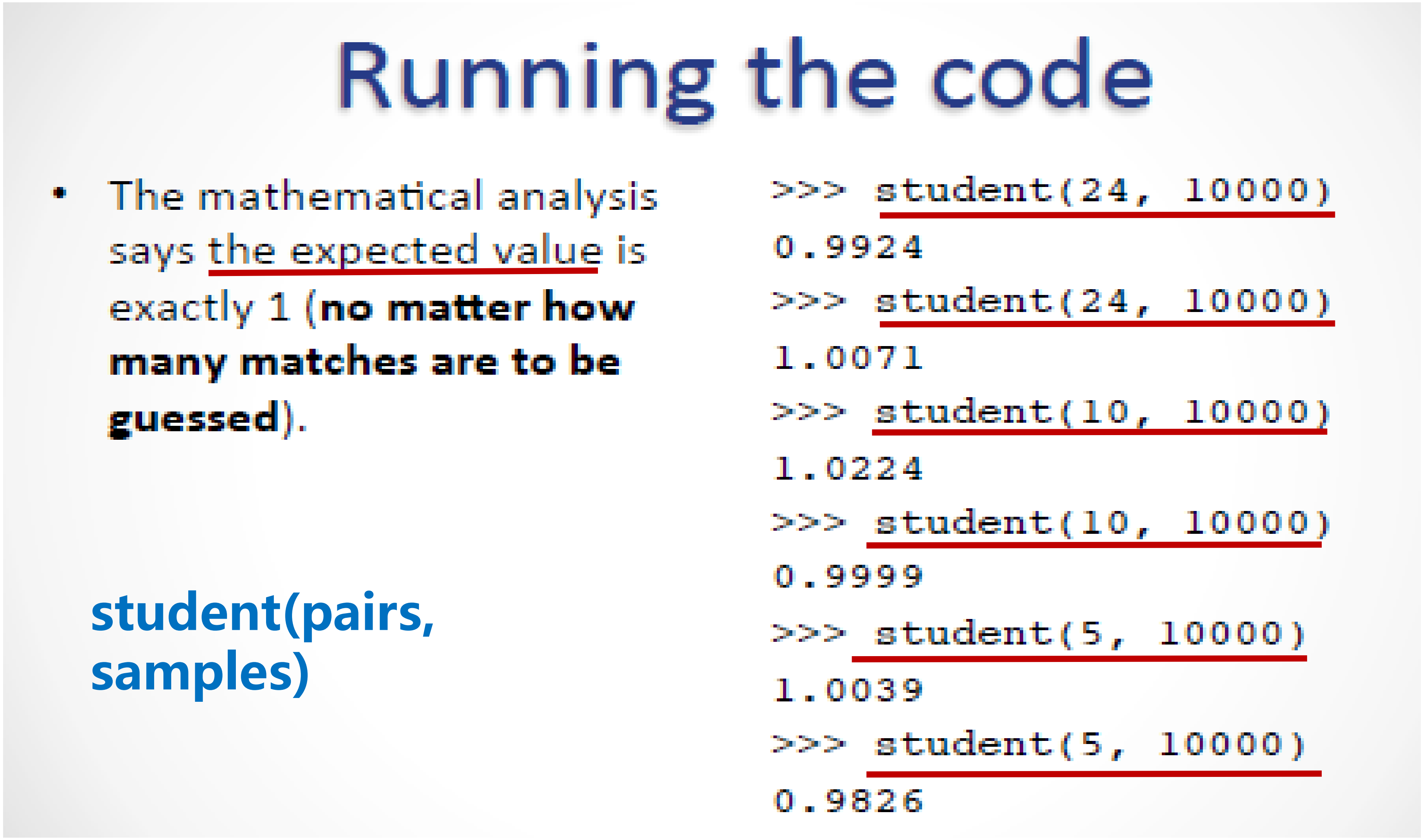
The Clueless Student Problem [6/8]

Code for the Clueless Student



```
from random import shuffle
# pairs is the number of pairs to be guessed
# samples is the number of samples to take
def student(pairs, samples) :
    num_correct = 0
    matching = list(range(pairs))
    for i in range(samples) :
        shuffle(matching)         # generate a guess
        for j in range(pairs) :
            if matching[j] == j :
                num_correct = num_correct + 1
    return num_correct / samples
```

The Clueless Student Problem [7/8]



Running the code

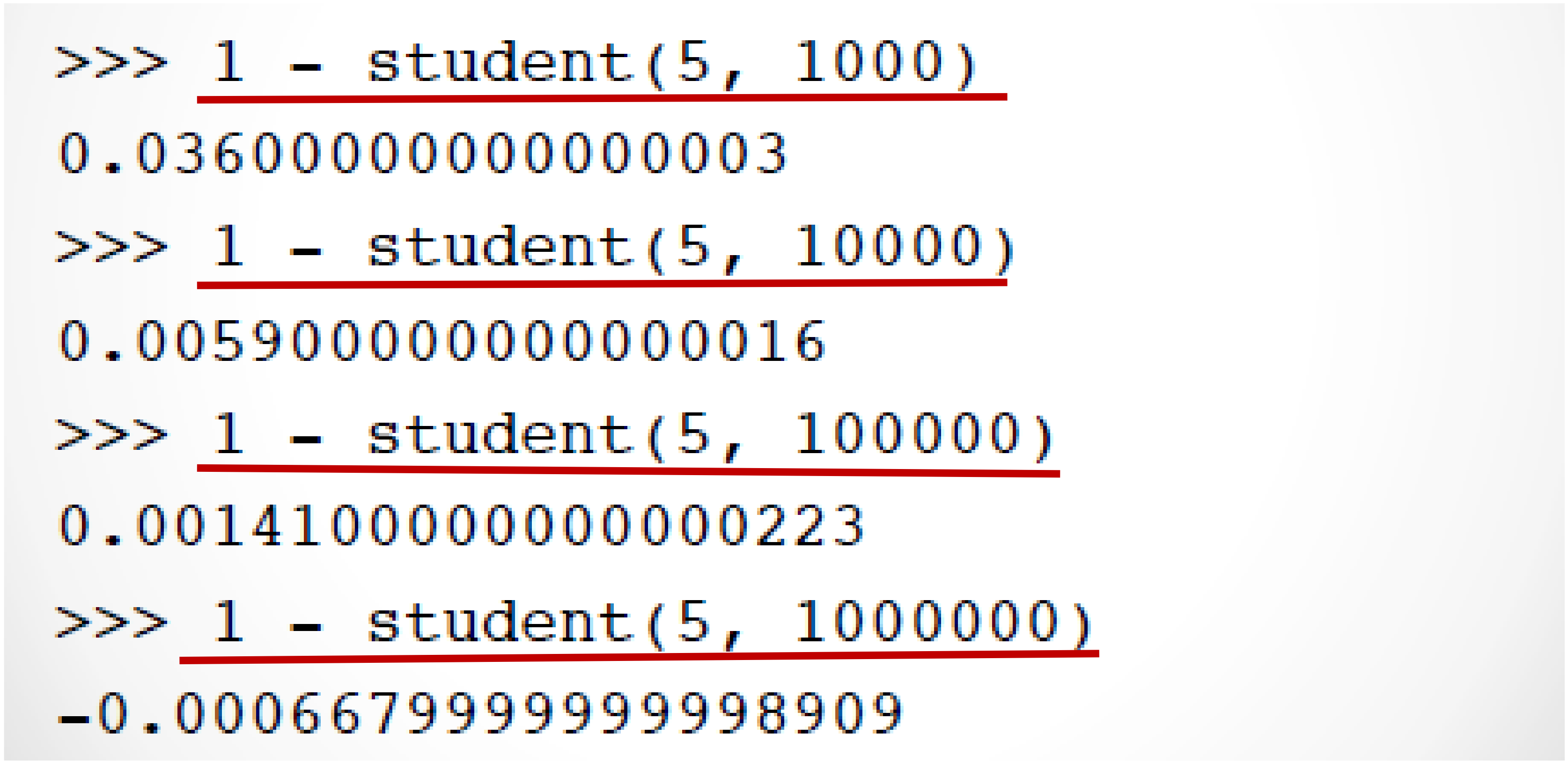
- The mathematical analysis says the expected value is exactly 1 (**no matter how many matches are to be guessed**).

student(pairs, samples)

```
>>> student(24, 10000)
0.9924
>>> student(24, 10000)
1.0071
>>> student(10, 10000)
1.0224
>>> student(10, 10000)
0.9999
>>> student(5, 10000)
1.0039
>>> student(5, 10000)
0.9826
```

The Clueless Student Problem [8/8]

More Samples ➔ Smaller Error



```
>>> 1 - student(5, 1000)
0.03600000000000003
>>> 1 - student(5, 10000)
0.005900000000000016
>>> 1 - student(5, 100000)
0.0014100000000000223
>>> 1 - student(5, 1000000)
-0.0006679999999998909
```

The expected value of student( ) was 1

The Umbrella Quandary Problem [1/2]

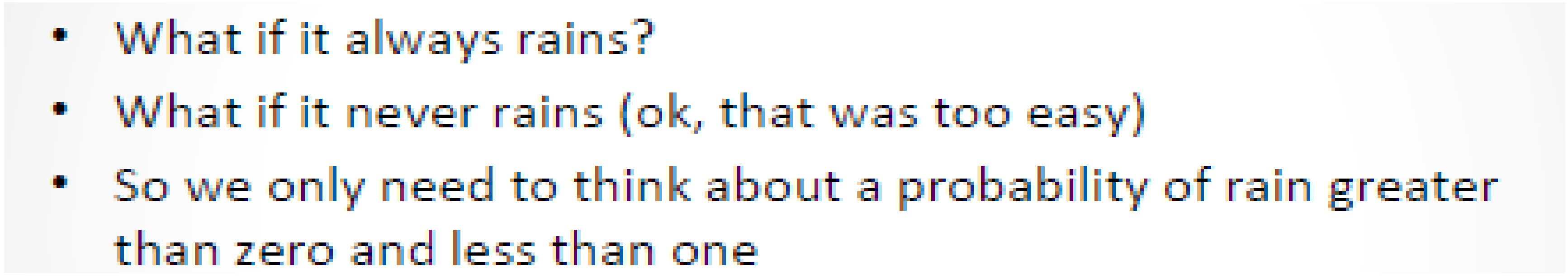
- Mr. X walks between home and work every day
- He likes to keep an umbrella at each location
- But he always forgets to carry one if it's not raining
- If the probability of rain is *p*, how many trips can he expect to make before he gets caught in the rain? (Assuming that if it's not raining when he starts a trip, it doesn't rain during the trip.)

비올 확률이 p%일때 비안맞고 진행된 trip 횟수를 구하라!

Home
Office

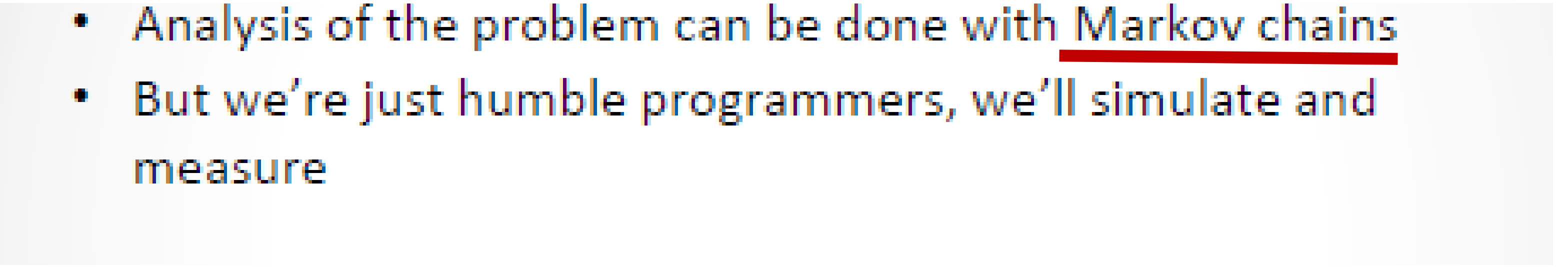
The Umbrella Quandary Problem [2/2]

The Trivial Cases



- What if it always rains?
- What if it never rains (ok, that was too easy)
- So we only need to think about a probability of rain greater than zero and less than one

Solving the umbrella quandary



- Analysis of the problem can be done with Markov chains
- But we're just humble programmers, we'll simulate and measure

Analytical Modeling VS Simulation

The Umbrella Quandary Problem : Algorithm [1/2]

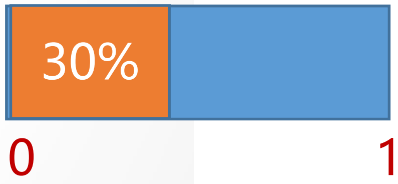
Simulating an event with a given probability

- In contrast to the clueless student problem we're given a probability of an event
- We want to simulate that the event happens, with the given probability *p* (where *p* is a number between 0 and 1)
- Technique: get a random float between 0 and 1; if it's less than *p* simulate that the event happened

```
if random() < p :
    raining = True
```

비올 확률이 p percent일때
실제로 p percent만 raining variable이 True를 가지게 된다!

random( ) 은 uniform!



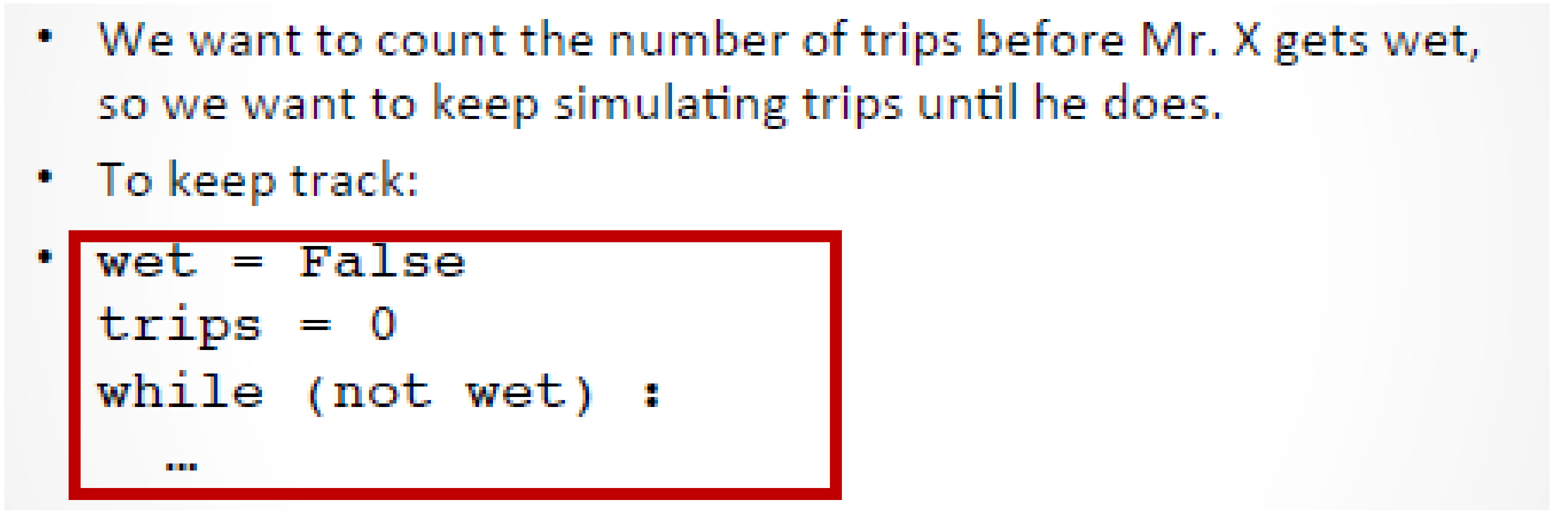
Representing home, work, and umbrellas

- Use 0 for home, 1 for work, and a two-element list for the number of umbrellas at each location
- How should we initialize?

-
  ```
  location = 0
  umbrellas = [1, 1]
  ```

The Umbrella Quandary Problem : Algorithm [2/2]

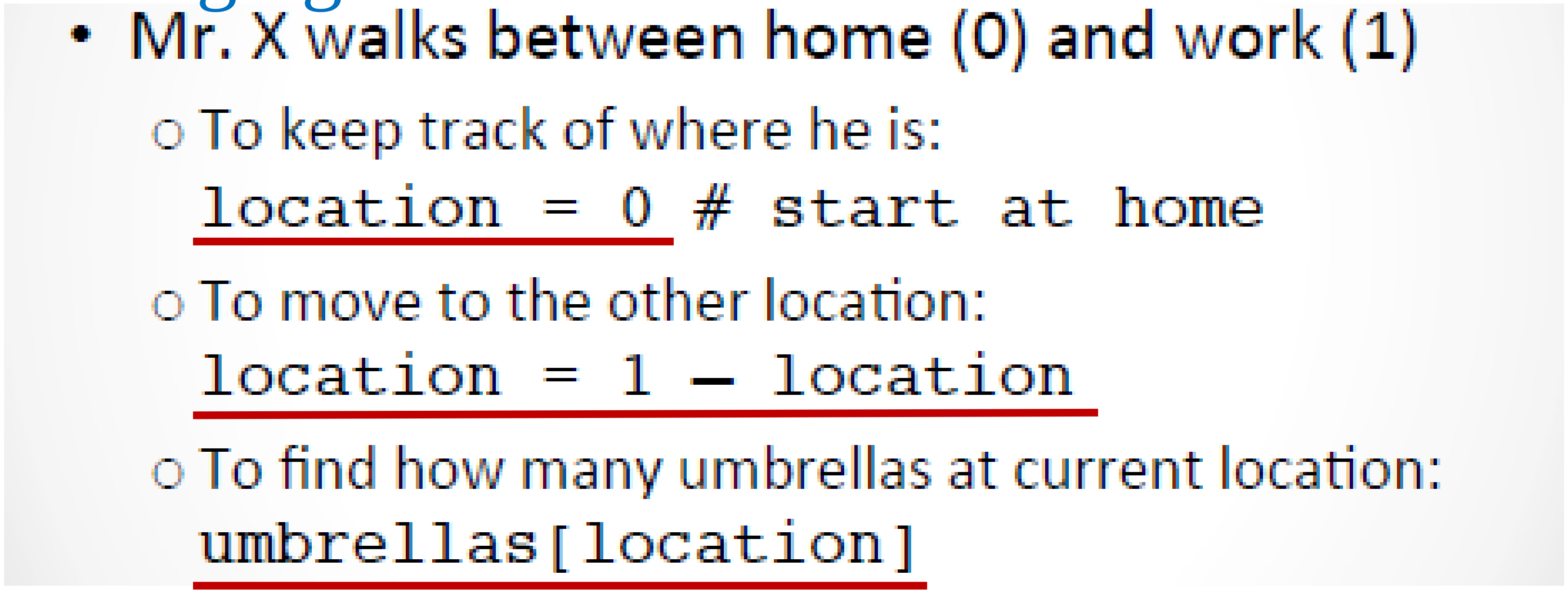
Figuring out when to stop



- We want to count the number of trips before Mr. X gets wet, so we want to keep simulating trips until he does.
- To keep track:

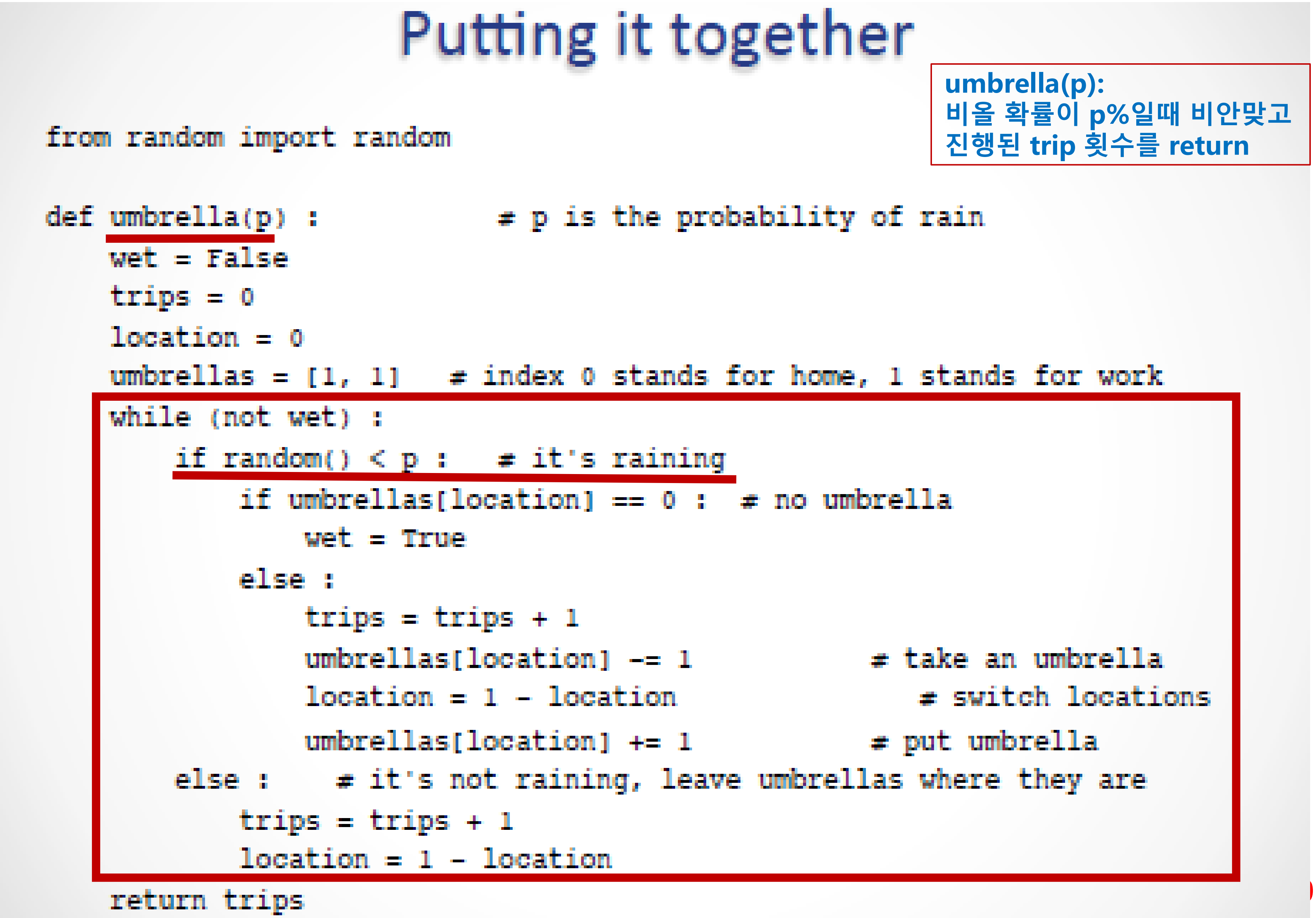
-
  ```
  wet = False
  trips = 0
  while (not wet) :
    …
  ```

Changing locations



- Mr. X walks between home (0) and work (1)
  - To keep track of where he is:
    ```
    location = 0 # start at home
    ```
  - To move to the other location:
    ```
    location = 1 – location
    ```
  - To find how many umbrellas at current location:
    ```
    umbrellas[location]
    ```

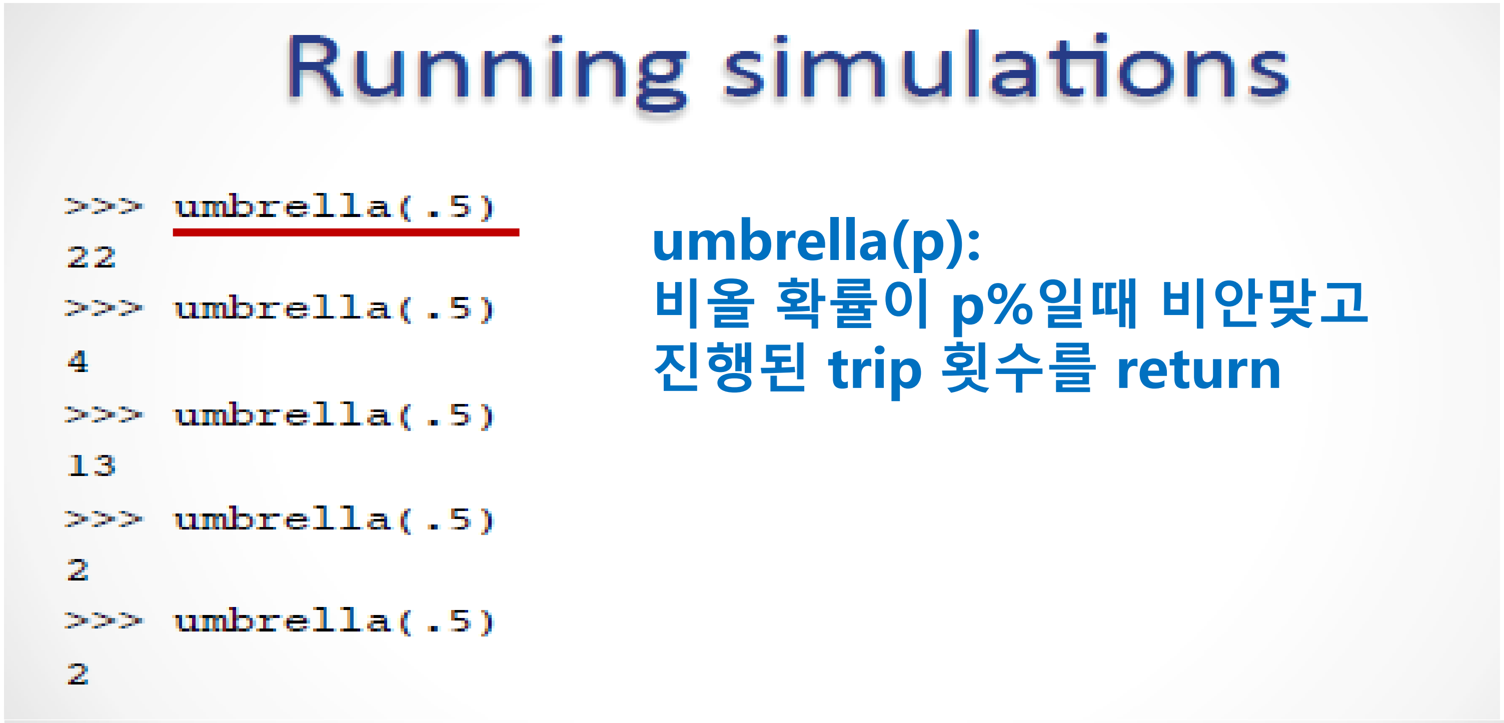
The Umbrella Quandary Problem : Python Code



Putting it together

umbrella(p):
비올 확률이 p%일때 비안맞고
진행된 trip 횟수를 return

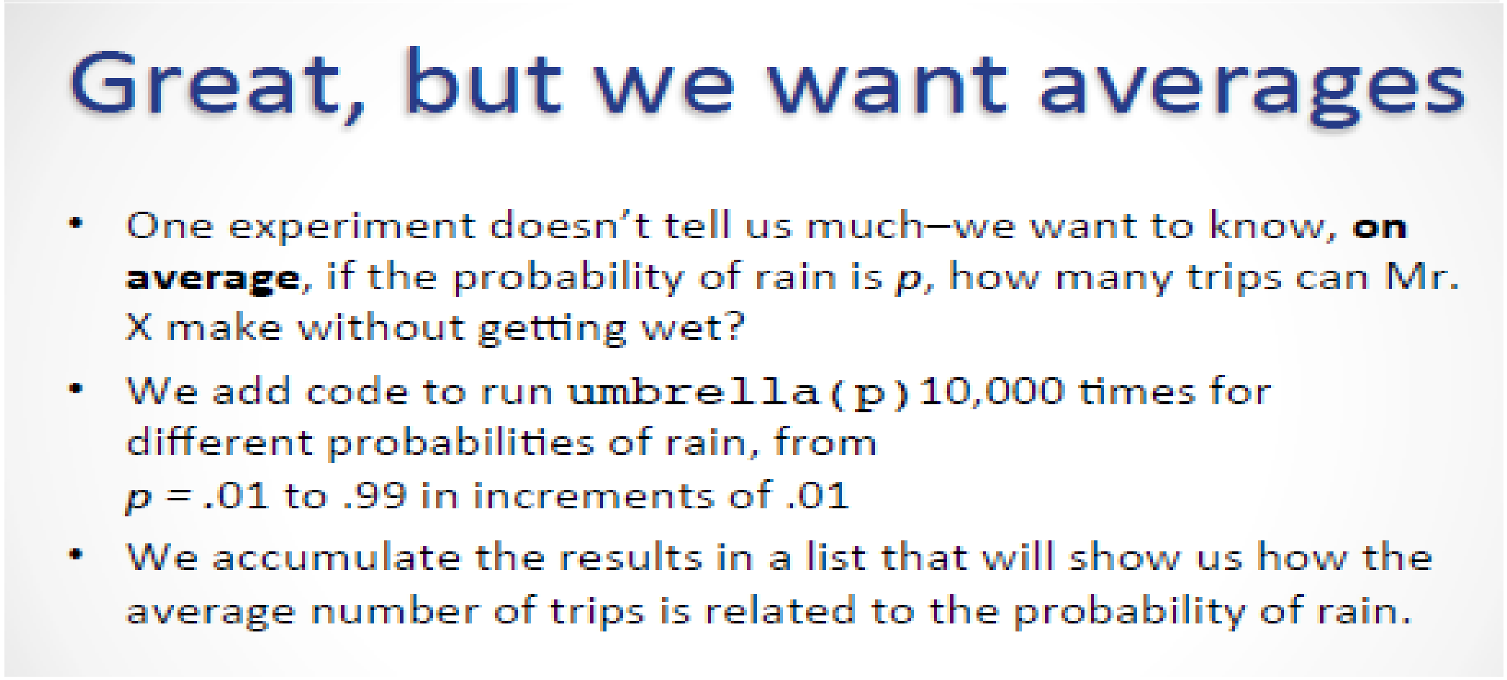
```
from random import random

def umbrella(p) :             # p is the probability of rain
    wet = False
    trips = 0
    location = 0
    umbrellas = [1, 1]   # index 0 stands for home, 1 stands for work
    while (not wet) :
        if random() < p :   # it's raining
            if umbrellas[location] == 0 :  # no umbrella
                wet = True
            else :
                trips = trips + 1
                umbrellas[location] -= 1          # take an umbrella
                location = 1 - location             # switch locations
                umbrellas[location] += 1          # put umbrella
        else :   # it's not raining, leave umbrellas where they are
            trips = trips + 1
            location = 1 - location
    return trips
```



Running simulations

```
>>> umbrella(.5)
22
>>> umbrella(.5)
4
>>> umbrella(.5)
13
>>> umbrella(.5)
2
>>> umbrella(.5)
2
```

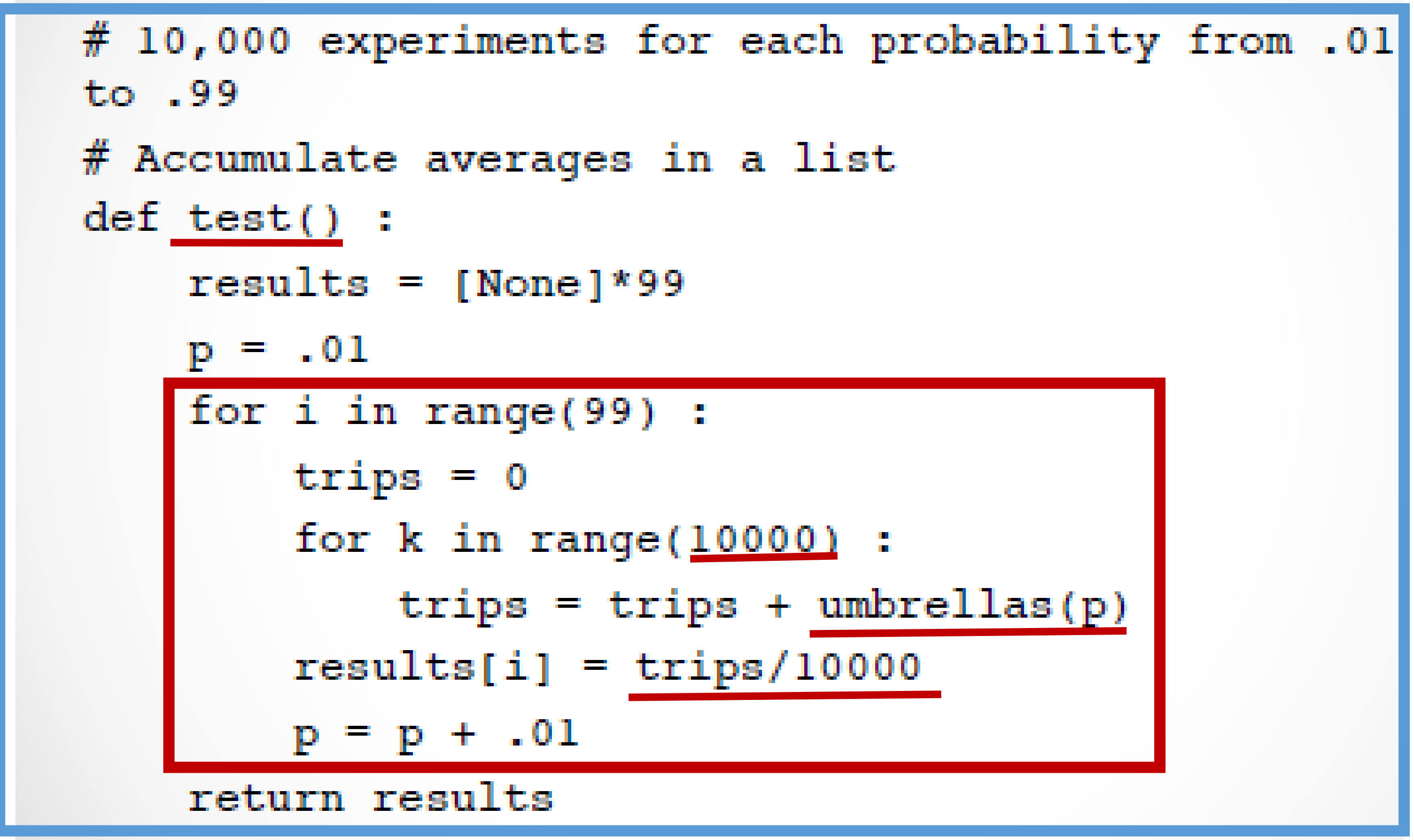
umbrella(p):
비올 확률이 p%일때 비안맞고
진행된 trip 횟수를 return



Great, but we want averages

- One experiment doesn't tell us much—we want to know, **on average**, if the probability of rain is *p*, how many trips can Mr. X make without getting wet?
- We add code to run `umbrella(p)` 10,000 times for different probabilities of rain, from *p* = .01 to .99 in increments of .01
- We accumulate the results in a list that will show us how the average number of trips is related to the probability of rain.

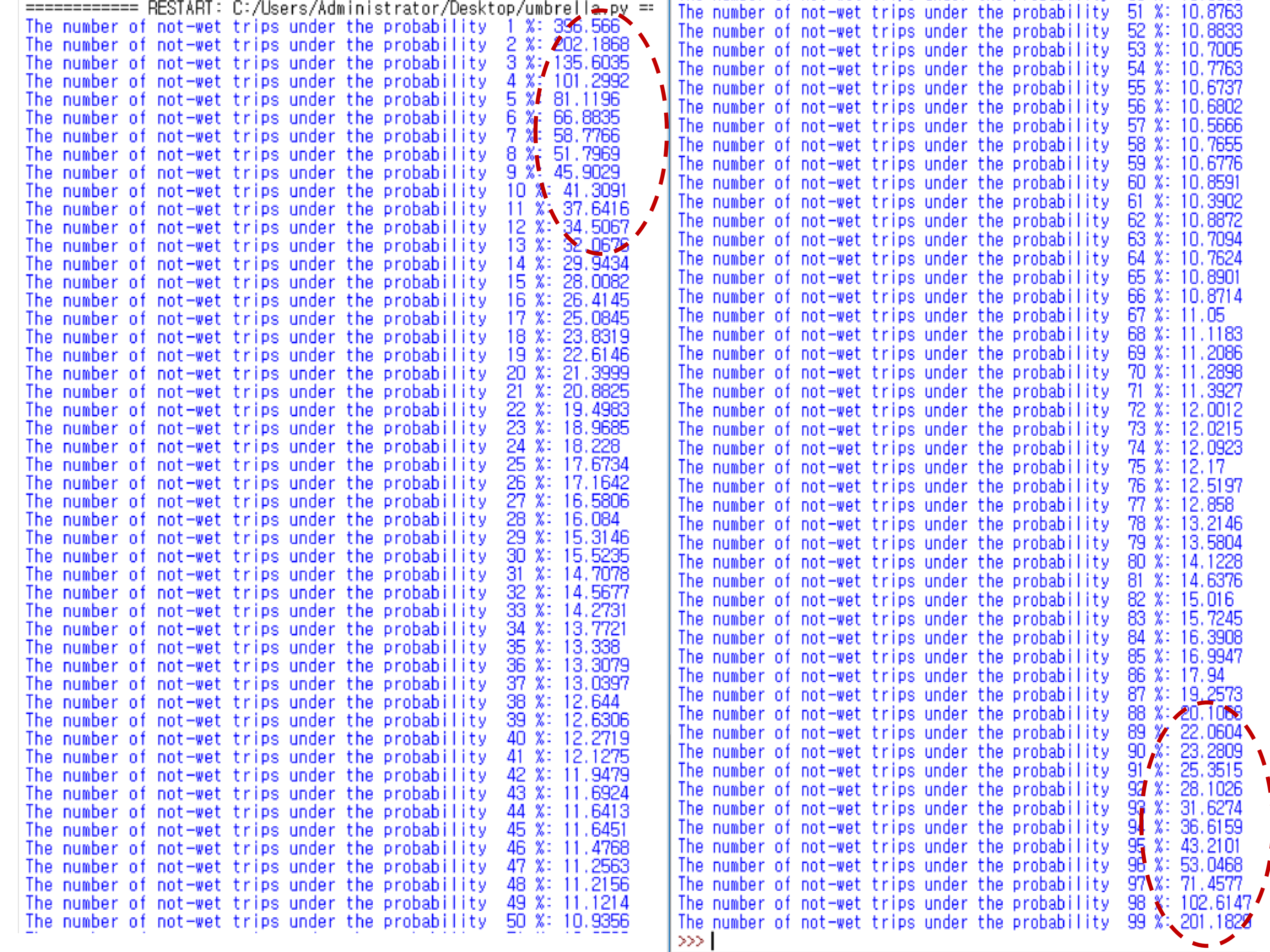
Running the experiments



```
# 10,000 experiments for each probability from .01
to .99
# Accumulate averages in a list
def test() :
    results = [None]*99
    p = .01
    for i in range(99) :
        trips = 0
        for k in range(10000) :
            trips = trips + umbrellas(p)
        results[i] = trips/10000
        p = p + .01
    return results
```

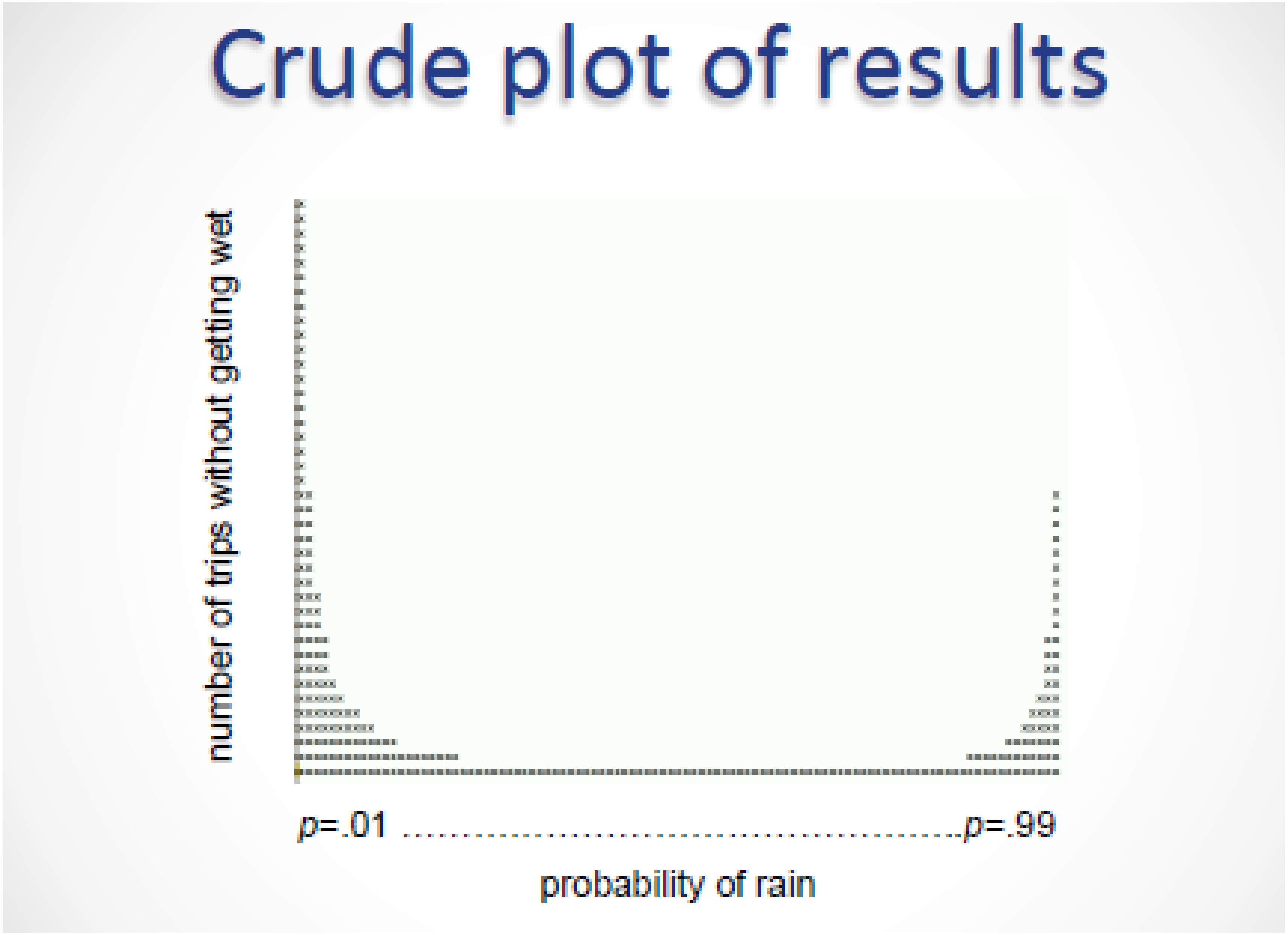
비올 확률 (0.1 ~ 0.99) 총 99경우에서 각 경우마다 평균 몇번의 trip이 가능한지를 보여주는 list

```
probability_list = test()
for i in range(1, 100):
    print(“The number of non-wet trips under the probability ”, j , “%: ”, probability_list[j-1] )
```



```
============ RESTART: C:/Users/Administrator/Desktop/umbrella.py ==
The number of not-wet trips under the probability  1 %: 396.566
The number of not-wet trips under the probability  2 %: 202.1868
The number of not-wet trips under the probability  3 %: 135.6035
The number of not-wet trips under the probability  4 %: 101.2992
The number of not-wet trips under the probability  5 %: 81.1196
The number of not-wet trips under the probability  6 %: 66.8835
The number of not-wet trips under the probability  7 %: 58.7766
The number of not-wet trips under the probability  8 %: 51.7969
The number of not-wet trips under the probability  9 %: 45.9029
The number of not-wet trips under the probability  10 %: 41.3091
The number of not-wet trips under the probability  11 %: 37.6416
The number of not-wet trips under the probability  12 %: 34.5067
The number of not-wet trips under the probability  13 %: 32.0676
The number of not-wet trips under the probability  14 %: 29.9434
The number of not-wet trips under the probability  15 %: 28.0082
The number of not-wet trips under the probability  16 %: 26.4145
The number of not-wet trips under the probability  17 %: 25.0845
The number of not-wet trips under the probability  18 %: 23.8319
The number of not-wet trips under the probability  19 %: 22.6146
The number of not-wet trips under the probability  20 %: 21.3999
The number of not-wet trips under the probability  21 %: 20.8825
The number of not-wet trips under the probability  22 %: 19.4983
The number of not-wet trips under the probability  23 %: 18.9685
The number of not-wet trips under the probability  24 %: 18.228
The number of not-wet trips under the probability  25 %: 17.6734
The number of not-wet trips under the probability  26 %: 17.1642
The number of not-wet trips under the probability  27 %: 16.5806
The number of not-wet trips under the probability  28 %: 16.084
The number of not-wet trips under the probability  29 %: 15.3146
The number of not-wet trips under the probability  30 %: 15.5235
The number of not-wet trips under the probability  31 %: 14.7078
The number of not-wet trips under the probability  32 %: 14.5677
The number of not-wet trips under the probability  33 %: 14.2731
The number of not-wet trips under the probability  34 %: 13.7721
The number of not-wet trips under the probability  35 %: 13.338
The number of not-wet trips under the probability  36 %: 13.3079
The number of not-wet trips under the probability  37 %: 13.0397
The number of not-wet trips under the probability  38 %: 12.644
The number of not-wet trips under the probability  39 %: 12.6306
The number of not-wet trips under the probability  40 %: 12.2719
The number of not-wet trips under the probability  41 %: 12.1275
The number of not-wet trips under the probability  42 %: 11.9479
The number of not-wet trips under the probability  43 %: 11.6924
The number of not-wet trips under the probability  44 %: 11.6413
The number of not-wet trips under the probability  45 %: 11.6451
The number of not-wet trips under the probability  46 %: 11.4768
The number of not-wet trips under the probability  47 %: 11.2563
The number of not-wet trips under the probability  48 %: 11.2156
The number of not-wet trips under the probability  49 %: 11.1214
The number of not-wet trips under the probability  50 %: 10.9356
The number of not-wet trips under the probability  51 %: 10.8763
The number of not-wet trips under the probability  52 %: 10.8833
The number of not-wet trips under the probability  53 %: 10.7005
The number of not-wet trips under the probability  54 %: 10.7763
The number of not-wet trips under the probability  55 %: 10.6737
The number of not-wet trips under the probability  56 %: 10.6802
The number of not-wet trips under the probability  57 %: 10.5666
The number of not-wet trips under the probability  58 %: 10.7655
The number of not-wet trips under the probability  59 %: 10.6776
The number of not-wet trips under the probability  60 %: 10.8591
The number of not-wet trips under the probability  61 %: 10.3902
The number of not-wet trips under the probability  62 %: 10.8872
The number of not-wet trips under the probability  63 %: 10.7094
The number of not-wet trips under the probability  64 %: 10.7624
The number of not-wet trips under the probability  65 %: 10.8901
The number of not-wet trips under the probability  66 %: 10.8714
The number of not-wet trips under the probability  67 %: 11.05
The number of not-wet trips under the probability  68 %: 11.1183
The number of not-wet trips under the probability  69 %: 11.2086
The number of not-wet trips under the probability  70 %: 11.2898
The number of not-wet trips under the probability  71 %: 11.3927
The number of not-wet trips under the probability  72 %: 12.0012
The number of not-wet trips under the probability  73 %: 12.0215
The number of not-wet trips under the probability  74 %: 12.0923
The number of not-wet trips under the probability  75 %: 12.17
The number of not-wet trips under the probability  76 %: 12.5197
The number of not-wet trips under the probability  77 %: 12.858
The number of not-wet trips under the probability  78 %: 13.2146
The number of not-wet trips under the probability  79 %: 13.5804
The number of not-wet trips under the probability  80 %: 14.1228
The number of not-wet trips under the probability  81 %: 14.6376
The number of not-wet trips under the probability  82 %: 15.016
The number of not-wet trips under the probability  83 %: 15.7245
The number of not-wet trips under the probability  84 %: 16.3908
The number of not-wet trips under the probability  85 %: 16.9947
The number of not-wet trips under the probability  86 %: 17.94
The number of not-wet trips under the probability  87 %: 19.2573
The number of not-wet trips under the probability  88 %: 20.1068
The number of not-wet trips under the probability  89 %: 22.0604
The number of not-wet trips under the probability  90 %: 23.2809
The number of not-wet trips under the probability  91 %: 25.3515
The number of not-wet trips under the probability  92 %: 28.1026
The number of not-wet trips under the probability  93 %: 31.6274
The number of not-wet trips under the probability  94 %: 36.6159
The number of not-wet trips under the probability  95 %: 43.2101
The number of not-wet trips under the probability  96 %: 53.0468
The number of not-wet trips under the probability  97 %: 71.4577
The number of not-wet trips under the probability  98 %: 102.6147
The number of not-wet trips under the probability  99 %: 201.1829
>>>
```

The Umbrella Quandary Problem : Plot the Simulation



중요

Major Functions in Random Module

- randint(n1, n2) ➔ n1 ~ n2사이의 random integer
- randrange(n1, n2) ➔ n1 ~ n2사이의 random integer
  - n2는 포함이 안됨, randrange(n1, n2, step)

- random( ) ➔ 0 ~ 1사이의 random float number
- uniform(n1, n2) ➔ n1 ~ n2사이의 random float number

- choice(L) ➔ L중에서 1개 선택
- sample(L, n) ➔ L중에서 n개 선택

- shuffle(L) ➔ L을 shuffling