

AM205: Introduction to the command line

In the AM205 command-line workshop, we covered a wide range of basic commands for the Linux terminal environment, and we played several rounds of the unique number game on an Amazon Cloud server. A short writeup should be submitted to Canvas by September 22 at 5pm, consisting of the following¹

1. Submit the command you used for Exercise 2(a) to solve the first code, involving substituting prime and non-prime numbers. (*You may optionally submit the decoded messages for second and third codes.*)
2. Submit a strategy for the unique number game (Exercise 3, listed below).

As for all AM205 group activities, you can submit solutions in groups of 1–3. If you submit in a group, only one member needs to upload a writeup and list the names of groupmates on the submission.²

The unique number game is described in more detail in the following sections. You can get the associated files in two places:

1. On the Amazon Cloud server which is accessible via SSH at 3.89.49.217. The files are in the `/shared/ung_files` directory.
2. In the [AM205 group activities Git repository](#), linked to from the front page of the AM205 website.

While you can develop your strategy for the unique number game on your own computer, we recommend that you try running it on the Amazon Cloud server to practice Linux terminal commands. **Note that the temporary Amazon Cloud server will be terminated in the evening of October 1, so please ensure any files are copied off it by then.**

The unique number game

In the unique number game, there are n players and each must choose an integer between 0 and 99. The winner is the person who chose the smallest number that was not chosen by anyone else. For example, if $n = 5$ the players might make the following choices:

Player	A	B	C	D	E
Choice	0	5	2	3	0

In this example, players A and E both chose zero so they invalidate each other. Player C made the lowest remaining choice and is therefore the winner. In certain rare cases, such as if B and D switched their choices to two, a game may result in no winner.

¹Since exercise 1 was very brief, you aren't required to submit anything for this.

²Since this was an introductory workshop we expect most students will find it easiest to write solutions themselves. Subsequent workshops will likely require more group interaction.

If the unique number game is played with human players, then choosing a number requires some psychology about what the other players will pick. Lower numbers are more desirable, but they are risky choices since there may be a greater probability that others will also choose them. While at the University of Cambridge, Chris played this game twice with $n = 100$ undergraduate friends. The winning number in the first game was two, and the winning number in the second game was thirteen. For many repeated games, we expect that a mixed-strategy **Nash equilibrium** may exist, where different numbers are chosen with different probabilities.

The game

We are going to play many repeated unique number games in AM205. Your job is to write a Python³ function that implements a strategy in this repeated game. The function should be called `play` and should accept five inputs `n`, `hist`, `lw`, `lc`, `y`—these will be discussed in more detail later. A simple strategy, given in `fixed_choice.py`, would be just to pick a fixed number:

```
def play(n, hist, lw, lc, y):  
  
    # Just always return 2  
    return 2
```

Alternatively, as in `random_choice.py`, one could choose the numbers from 0 to 11 with equal probability:

```
from random import randint  
  
def play(n, hist, lw, lc, y):  
  
    # Return a random number between 0 and 11  
    return randint(0, 11)
```

Your function should be called `firstname_lastname.py` and should be uploaded to Canvas with your assignment.

Once we have collected all of the programs we will run a hundred sets of ten million rounds each. The number of wins for each player will be counted across all 10^9 rounds.

Testing a strategy

We have provided you with a program called `game_test.py` that can be used to test different strategies against each other. It is currently set up to simulate 20,000 rounds with $n = 36$ players. 35 of the players use the `random_choice.py` strategy, and the last player uses a very basic strategy in `test_strategy.py`. Currently the player using the test strategy only wins a small fraction of approximately 0.92% of the rounds, much less than

³If you are not familiar with Python, the teaching staff will help to translate your function.

an equal share of $\frac{1}{36} = 2.78\%$ of the rounds. You can try modifying this function to do better.

More available information

The function `play` will be passed additional information about the current state of the game and what has happened in previous rounds. If you wish, you can make use of some of this information to devise a strategy. The five inputs are as follows:

- `n` The total number of players. This will likely be between 35 and 40.
- `hist` A list of length 100, containing the total number of choices of each number across all previous rounds that have been played so far.
- `lw` The winning number in the previous round. If there was no winner in the previous round this will be set to 100. In the very first round, before anything has been played, this will be set to zero.
- `lc` A list of length `n` of all of the players' choices in the previous round. In the very first round, before anything has been played, all entries will be set to zero. Each player will occupy the same entry in the list throughout the game; for example, by repeatedly examining `lc[5]` you could see the entire history of one particular player.
- `y` Your position within the `lc` list. Hence `lc[y]` will be equal to your choice in the previous round.

Three sample strategies that make use of some of this information are provided for you to review:

- `prev.py` If the previous winning choice was `lw`, then that may be a good number in general. This function returns `lw-1`, `lw`, or `lw+1` with equal probability.
- `histo.py` This function chooses the lowest number whose probability of occurrence in the previous games is less than $\frac{1}{n}$, and adds a small random displacement.
- `mean.py` This function calculates the mean of the winning choices in all previous games and chooses that, rounded to the nearest integer.

Exercise 3

Devise a strategy for the unique number game following the description and rules above. Anything is acceptable, and you can even copy verbatim or modify one of the basic examples discussed above. Submit it with name `firstname_lastname.py`.

Technicalities

The following technicalities should not be relevant for any basic strategy but may come into play if you want to try something more elaborate:

1. We will review all submissions prior to playing the game. If we notice any functions that are liable to crash, we may contact you or make minor fixes.
2. Designing strategy that involves cooperation beforehand with other students is allowed. For example, if you restrict your choice to odd numbers, and your friend restricts his/her choice to even numbers, then at least you know that you will never invalidate each other's choices. However, this a small advantage that may be outweighed by many other factors. Within the game, active communication between two strategies, such as by sharing data, is not allowed. The positions of each player within 1c will be randomized, so there will be no straightforward way to identify your friend.
3. If your function returns a value less than zero, it will be treated as choosing zero. If your function returns a value greater than 99, it will be treated as choosing 99.
4. The function can make use of any of the standard Python libraries, as well as SciPy and NumPy. However, the routine that you submit should be pure Python—calling some other pre-compiled code is not allowed.
5. There will be a limit on how long your function can take. If a function (a) takes longer than 0.1 s more than 100 times in any set of rounds, (b) ever takes longer than 10 s, or (c) crashes, it will be disqualified and the game will be reset to not include this function. The hundred sets of rounds will be run on a variety of Linux and Mac workstations.
6. The games will be run on machines with at least two gigabytes of memory. In the unlikely event that the games run out memory, functions that consume the most memory will be disqualified and the game will be reset.
7. The teaching staff may disqualify any entry that is deemed to be subverting the rules or not in the spirit of the game.