Tutorial II.III - Explore and Exploit

Programming: Everyday Decision-Making Algorithms

Introduction

Welcome to the final tutorial of Part II! We'll tie together everything we've learned so far - variables, lists, conditions, and loops - to tackle a classic explore-exploit problem: the one-armed bandit (slot machine). This is a perfect example of balancing exploration (trying different strategies) with exploitation (sticking to what works).

Section 1 - Setting Up Our Casino

First, let's create a simple slot machine simulator using what we learned about variables and lists. In order to do this, we'll need to import the `random` module, which we can do using the `import` statement. Don't worry about the details of this statement for now - we'll cover it in more detail in a future tutorial.

```
import random

# Initial setup
money = 100 # Starting money
bet = 10 # How much we bet each time
payouts = [0, 5, 10, 20, 50] # Different possible wins

print("Possible outcomes:")
for payout in payouts:
    print(f"  Win ${payout}")
```

```
Possible outcomes:
  Win $0
  Win $5
  Win $10
  Win $20
  Win $50
```

In this setup, we're creating a simple slot machine where: - You start with $100 - Each play costs $10 - There are 5 possible payouts: $0 (loss), $5 (small win), $10 (break even), $20 (good win), and $50 (jackpot) - The `random.choice()` function simulates pulling the lever by randomly selecting one of these payouts

> **Note**
>
> In real slot machines, the probabilities of different payouts aren't equal. Higher payouts are much less likely to occur than lower ones. Our simplified version gives each payout an equal chance.

To simulate playing the slot machine once, we can start as follows:

```
# Simulate playing the slot machine once
win = random.choice(payouts)
money += win
print(f"You bet ${bet}")
print(f"You won ${win}!")
print(f"Your new balance is ${money}")
```

```
You bet $10
You won $20!
Your new balance is $120
```

> **Tip**
>
> `+=` is a shorthand for "add and assign". So `money += 10` is equivalent to `money = money + 10`. This also works for other arithmetic operations, like `-=`, `*=`, and `/=`.

Exercise 1.1 - Multiple Attempts

Now let's use a loop to play multiple times and track our results. Write a loop that:

1. Plays the slot machine 10 times
2. Subtracts the bet amount each time
3. Adds a random payout
4. Stores the results in a list

```
money = 100 # Reset our money
bet = 10 # How much we bet each time
results = [] # List to store how much money we have after each play

# YOUR CODE BELOW
```

```
# Check your results
assert len(results) == 10, "You should have 10 results in your list"
assert all(isinstance(x, (int, float)) for x in results), "All results should be numbers"
print(f"You played the slot machine {len(results)} times")
print(f"Your final balance is ${results[-1]}")
print(f"Unfortunately, the payout rates are not realistic here...")
```

> **Tip**
>
> Use the `append()` method to add an element to a list. For example, `results.append(money)` will add the current value of `money` to the `results` list.

Exercise 1.2 - Playing with a Budget

Now let's simulate playing the slot machine multiple times with a budget. We'll use a `while` loop to play until we run out of money. This is more realistic than playing a fixed number of times, as real players need to stop when they're out of money!

The `machine_payout_rate` represents how generous the machine is:

- The function `random.random()` generates a random number between 0 and 1.
- A rate of 2.0 means you get back exactly what you bet (on average)
- Our rate of 1.5 means you could win up to 1.5 times your bet but on average you'll get back 0.75 times your bet

```
# Compute the payout for the current play
machine_payout_rate = 1.5
payout = random.random() * machine_payout_rate * bet
```

Use a `while` loop to play the slot machine until you run out of money, and store the amount of money you have after each play in the `results` list. Further, print out the amount of money you have after each play with a formatted string.

```
import random

machine_payout_rate = 1.5 # Payout rate for the machine
bet = 10 # Bet size
money = 100 # Our starting money
results = [] # Store results of each play here

# YOUR CODE BELOW
```

```
# Check your results
assert len(results) > 0, "You should have at least one result"
assert all(isinstance(x, (int, float)) for x in results), "All results should be numbers"
assert results[-1] <= bet, "Should stop playing when money is less than bet amount"
assert all(abs(results[i] - results[i-1]) <= bet * machine_payout_rate for i in range(1,
 ↪  len(results))), "Each play should not change money by more than maximum possible
 ↪  payout"
print(f"You played the slot machine {len(results)} times")
```

Section 2 - Nested Loops

Now let's use nested loops to try different slot machines with different payout rates. This is where exploration comes in! Nested loops are a way to repeat a loop multiple times, and we can use them to try different options. It's essentially a loop inside another loop.

> Tip
>
> Nested loops are like a clock: the outer loop is like the hour hand (moves slowly), and the inner loop is like the minute hand (completes a full cycle for each move of the hour hand).

For example, `for machine in range(3)` will run a loop 3 times. We can use nested loops to try all combinations of two different options.

```
for machine in range(3):
    for payout in range(4):
        print(f"Machine Nr. {machine}, Payout Nr. {payout}")
```

```
Machine Nr. 0, Payout Nr. 0
Machine Nr. 0, Payout Nr. 1
Machine Nr. 0, Payout Nr. 2
Machine Nr. 0, Payout Nr. 3
Machine Nr. 1, Payout Nr. 0
Machine Nr. 1, Payout Nr. 1
Machine Nr. 1, Payout Nr. 2
Machine Nr. 1, Payout Nr. 3
Machine Nr. 2, Payout Nr. 0
Machine Nr. 2, Payout Nr. 1
Machine Nr. 2, Payout Nr. 2
Machine Nr. 2, Payout Nr. 3
```

Exercise 2.1 - Simulating Multiple Machines

The setup is the same as in Exercise 1.2, but we're now comparing three different machines, each with its own payout rate. Use a `while` loop for each machine and a `for` loop to try all machines. As the bank always wins, we're not interested in the results for each machine, but rather in the number of times we can play each machine before we run out of money.

Store the number of plays in a list called `plays` with three elements, one for each machine. The goal is to find out which machine lets us play the longest with our budget. This is a common exploration strategy: try each option several times to understand its characteristics.

```
import random
# Different machines might have different payout rates
machine_1_payout_rate = 1.2
machine_2_payout_rate = 1.5
machine_3_payout_rate = 1.8

bet = 10 # Bet size at each play
budget = 100 # Our starting money for each machine!
plays = [0, 0, 0] # List to store number of plays for each machine

# YOUR CODE BELOW
```

```
# Check your results
assert len(plays) == 3, "Should have results for three machines"
assert all(isinstance(x, int) and x >= 0 for x in plays), "Number of plays should be
 ↪  non-negative integers"
assert all(x <= budget/bet * 6 for x in plays), "Number of plays seems unreasonably high
 ↪  given budget"
print(f"You played the slot machine {plays[0]} times on machine 1")
print(f"You played the slot machine {plays[1]} times on machine 2")
print(f"You played the slot machine {plays[2]} times on machine 3")
```

Exercise 2.2 - Exploring the Machines

Now let's find out which machine is the best and which one is the worst by comparing the number of plays for each machine. Store the index of the best machine in `best_machine` and the index of the worst machine in `worst_machine`. You can use the `max(plays)` function to find the maximum number of plays in the `plays` list and the `plays.index(max(plays))` function to find the index of the maximum number of plays.

```
# YOUR CODE BELOW
```

```
# Check your results
assert isinstance(best_machine, int) and 0 <= best_machine <= 2, "Best machine should be
 ↪  0, 1, or 2"
assert isinstance(worst_machine, int) and 0 <= worst_machine <= 2, "Worst machine should
 ↪  be 0, 1, or 2"
assert best_machine != worst_machine, "Best and worst machine cannot be the same"
print(f"The best machine is machine {best_machine}")
print(f"The worst machine is machine {worst_machine}")
```

Conclusion

Excellent work! You've successfully combined multiple programming concepts to tackle a real explore-exploit problem:

- Variables and types for tracking money and bets
- Lists for storing payouts and results
- Loops (including nested loops) for testing different machines
- Conditions for finding the best machine

This is a simplified version of how casinos and gambling companies might analyze their games, though in reality, they use much more sophisticated algorithms! Remember though, that the casino always wins!

Solutions

You will likely find solutions to most exercises online. However, we strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.

Later, you will find the solutions to these exercises online in the associated GitHub repository, but we will also quickly go over them next week. To access the solutions, click on the Github button on the lower right and search for the folder with today's lecture and tutorial. Alternatively, you can ask ChatGPT or Claude to explain them to you. But please remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities.

That's it for part II! Next week, we'll learn about functions and how to use them to make our code more efficient and reusable.