Tutorial II.II - Loops

Programming: Everyday Decision-Making Algorithms

Introduction

Loops are a fundamental concept in programming that allow us to repeat a set of instructions multiple times. They're essential for tasks that require processing large datasets, iterating through complex sequences, or automating repetitive processes. In the context of clinical trials, loops help us systematically analyze multiple treatments or patient outcomes.

i Note

If a cell is marked with `YOUR CODE BELOW`, you are expected to write your code in that cell.

Section 1 - For Loops

A `for` loop allows us to iterate through a sequence of items. In clinical trials, this is particularly useful when we need to process multiple treatments or patient outcomes. The basic syntax is:

```python
for item in sequence:
    # do something with item
```

Think of it like reviewing patient charts: instead of manually checking each chart, you can automate the process to review them one by one systematically. This is especially valuable when dealing with large clinical trials or multiple treatment groups. Example with treatment options:

```python
treatments = ["Standard Drug", "New Drug A", "New Drug B"]
for treatment in treatments:
    print(f"Evaluating efficacy of {treatment}")
```

```
Evaluating efficacy of Standard Drug
Evaluating efficacy of New Drug A
Evaluating efficacy of New Drug B
```

Range in For Loops

When you need to iterate a specific number of times, the `range()` function is invaluable. It generates a sequence of numbers. It can take up to three arguments:

- `start`: The starting number (inclusive)
- `stop`: The ending number (exclusive)
- `step`: The difference between each number in the sequence

```
# Visual representation of range
print("range(5):", list(range(5)))
print("range(1, 6):", list(range(1, 6)))
print("range(0, 10, 2):", list(range(0, 10, 2)))
```

```
range(5): [0, 1, 2, 3, 4]
range(1, 6): [1, 2, 3, 4, 5]
range(0, 10, 2): [0, 2, 4, 6, 8]
```

We can then use this sequence in a `for` loop to iterate through the numbers.

```
# Simulate 5 trial phases
for phase in range(5):  # 0 to 4
    print(f"Starting Phase {phase + 1}")
```

```
Starting Phase 1
Starting Phase 2
Starting Phase 3
Starting Phase 4
Starting Phase 5
```

```
# Range with start, stop, and step
for dose in range(100, 501, 100):  # 100 to 500 in steps of 100
    print(f"Testing dosage: {dose}mg")
```

```
Testing dosage: 100mg
Testing dosage: 200mg
Testing dosage: 300mg
Testing dosage: 400mg
Testing dosage: 500mg
```

Break and Continue Statements

- `break`: Immediately exits the loop
- `continue`: Skips to the next iteration

```
efficacy_scores = [65, 85, 45, 82, 58]
for score in efficacy_scores:
    if score < 50:  # Skip low efficacy treatments
        continue
    print(f"Treatment efficacy: {score}%")
    if score >= 85:  # We found a highly effective treatment!
        print("Found a promising treatment - consider focusing on this
one!")
        break
```

```
Treatment efficacy: 65%
Treatment efficacy: 85%
Found a promising treatment - consider focusing on this one!
```

Tuple Unpacking

Tuple unpacking is a powerful feature that allows you to assign multiple variables in a single statement. It's often used in `for` loops to access elements from a list or tuple. For example:

```
treatments = [("Standard Drug", 75), ("New Drug A", 85), ("Alternative B",
68), ("New Drug C", 92)]
for name, efficacy in treatments:
    print(f"Treatment: {name}, Efficacy: {efficacy}%")
```

```
Treatment: Standard Drug, Efficacy: 75%
Treatment: New Drug A, Efficacy: 85%
Treatment: Alternative B, Efficacy: 68%
Treatment: New Drug C, Efficacy: 92%
```

Exercise 1.1 - Count Adverse Events

Track the number of treatments that had too many adverse events (more than 5) with a `for` loop. Add 1 to `high_adverse_count` for each treatment that had more than 5 adverse events.

```
adverse_events = [3, 7, 2, 6, 8, 1, 4]
high_adverse_count = 0

# YOUR CODE BELOW
```

```
# Assertions to verify your solution
assert isinstance(high_adverse_count, int), "high_adverse_count should be
an integer"
assert high_adverse_count == 3, "There should be exactly 3 treatments with
high adverse events"
print("Success! You've correctly counted the high adverse event
treatments.")
```

Exercise 1.2 - Calculate Average Treatment Efficacy

Calculate the average efficacy of the treatments using a `for` loop. First, add each rate to the total. Then, calculate the average by dividing the total by the number of treatments.

```
efficacy_rates = [78, 72, 85, 65, 80]
average_efficacy = 0
total = 0
# YOUR CODE BELOW
```

```
# Assertions to verify your solution
assert isinstance(average_efficacy, (int, float)), "average_efficacy should
be a number"
assert average_efficacy == 76, "average_efficacy should be 76"
print("Success! Your average calculation is correct.")
```

Tip

- Add each rate to the total
- Calculate the average by dividing the total by the number of treatments
- Note that `len(efficacy_rates)` gives you the count of treatments

Exercise 1.3 - Find Promising Treatments

Add all treatments with an efficacy rate of at least 80% to the list `promising_treatments`.

```
treatments = [("Standard Drug", 75), ("New Drug A", 85), ("Alternative B",
68), ("New Drug C", 92)]
promising_treatments = []

# YOUR CODE BELOW
```

```
# Assertions to verify your solution
assert len(promising_treatments) == 2, "Should find exactly 2 promising
treatments"
assert all(isinstance(t, str) for t in promising_treatments),
"promising_treatments should contain strings"
assert "New Drug A" in promising_treatments, "New Drug A should be in
promising_treatments"
print("Success! You've correctly identified the promising treatments.")
```

Tip

- Each treatment is represented as a tuple: `(name, efficacy_rate)`
- You need to check if the efficacy rate (second element) is ≥ 80%
- If it meets the criteria, add the treatment name (first element) to `promising_treatments`
- You could use tuple unpacking in your loop for cleaner code

Section 2 - While Loops

While loops continue executing as long as a condition is true. They're perfect for situations where we don't know in advance how many iterations we'll need, like finding an effective treatment. Think of them as running clinical trials until you either find a breakthrough treatment or run out of research funding.

```python
resources = 100  # Starting research resources
treatments_tested = 0
current_best = 0

while resources > 0:
    print(f"Resources remaining: {resources}")
    resources -= 20  # Each trial phase costs 20 units
    treatments_tested += 1

print(f"Tested {treatments_tested} treatment variations")
```

```
Resources remaining: 100
Resources remaining: 80
Resources remaining: 60
Resources remaining: 40
Resources remaining: 20
Tested 5 treatment variations
```

We can also use them without specific conditions at the beginning:

```python
trials = 0
while True:
    if trials > 5:
        break
    print(f"Trial {trials}")
    trials += 1
```

```
Trial 0
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
```

i Note

While loops are particularly useful when:

- You don't know how many iterations you'll need
- You're looking for a specific outcome
- You need to stop based on resource constraints

Be careful with while loops! An infinite loop can crash your program:

```
while True:  # Dangerous!
    print("This will never end")
```

Warning

Always ensure:

- The condition will eventually become false
- Resources or counters are properly updated
- You have a way to break out if needed

Exercise 2.1 - Breakthrough Treatment

Use a `while` loop to find the first treatment with an efficacy score of at least 88. Count the number of trials it takes to find this treatment and store it in the variable `trials`. Save the efficacy score of the breakthrough treatment in the variable `current_efficacy`. Note, that the `efficacy_scores` list is indexed from 0!

```
efficacy_scores = [72, 68, 78, 88, 71, 65, 59]
current_efficacy = 0
trials = 0

# YOUR CODE BELOW
```

```
# Assertions to verify your solution
assert trials == 4, "Should take exactly 4 trials to find the breakthrough"
assert current_efficacy == 88, "Should find efficacy score of 88"
print("Success! You've found the breakthrough treatment efficiently.")
```

Exercise 2.2 - Resource Management

Continue trials until either resources are depleted or a breakthrough (efficacy >= 90) is found. Count the number of trials it takes to find this treatment and store it in the variable `trials_conducted`. Save the efficacy score of the breakthrough treatment in the variable `current_best`. Note, that the `efficacy_scores` list is indexed from 0!

```
resources = 100
cost_per_trial = 25
current_best = 0
trials_conducted = 0
efficacy_scores = [75, 82, 88, 92, 85, 78]  # Simulated trial results

# YOUR CODE BELOW
```

```
# Assertions to verify your solution
assert trials_conducted == 4, "Should conduct 4 trials"
```

```python
assert current_best >= 90, "Should find a treatment with >= 90 efficacy"
assert resources >= 0, "Should not exceed resource limit"
print("Success! You've managed resources effectively.")
```

Tip

- Track both resources and efficacy simultaneously
- Stop if either:
  - Resources fall below cost_per_trial
  - A breakthrough treatment (≥ 90% efficacy) is found
- Remember to deduct costs for each trial

Conclusion

Excellent work! You've learned about two powerful loop structures in Python through the lens of clinical trials and treatment exploration:

- `for` loops help us:
  - Process data systematically
  - Analyze outcomes efficiently
  - Work with structured data like treatment efficacy rates
- `while` loops are perfect for:
  - Resource-constrained scenarios
  - Finding certain outcomes
  - Situations with unknown iteration counts

In the context of clinical trials: - `for` loops are like reviewing a predetermined set of patient data - `while` loops are like continuing research until you either find a successful treatment or exhaust your resources

These loop structures are fundamental tools, allowing us to process large datasets and make informed decisions in the context of the explore-exploit trade-off.

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.

Continue to the next tutorial to tie all previous concepts together in the context of the explore-exploit problem in a casino!