Equivalence Testing in Python

equivalence_partition():

- Partitions a collection of objects into equivalence classes based on a given relation.
- Iterates through the objects and checks whether they belong to an existing class or form a new one.

equivalence_enumeration():

- Extends equivalence_partition() by also enumerating the equivalence classes (assigning indices).

check_equivalence_partition():

- Validates the correctness of the equivalence partition by ensuring all objects in the same class satisfy the equivalence relation.

test_equivalence_partition():

- Demonstrates partitioning on the range [-3, 4] with the relation (x - y) % 4 == 0.
- Prints the resulting equivalence classes and partitions.

Output:

```
george.koridze@MBP-GK-QQXJPGK7P4 ~ % /usr/local/bin/python3 /Users/george.koridze/Desktop/Essex/SSD/ePortfolio/Unit5/equivalence.py
{1, -3}
{2, -2}
{3, -1}
{0, 4}
-3 : {1, -3}
-2 : {2, -2}
-1 : {3, -1}
0 : {0, 4}
1 : {1, -3}
2 : {2, -2}
3 : {3, -1}
4 : {0, 4}
```

Experimentation:

Change the Range

Modify range(-3, 5) to range(-10, 10).

```python
def test_equivalence_partition():
    relation = lambda x, y: (x - y) % 4 == 0
    classes, partitions = equivalence_partition(
        range(-10, 10),
        relation
    )
```

Output:

Summary:

Key Observations:

- The code accurately partitions based on the equivalence relation.
- Modifying the relation changes how classes are formed.
- Larger ranges or complex relations may increase computational overhead.

Security Implications:

- Input validation is crucial to prevent unexpected errors.
- Relations that involve heavy computation (e.g., factorials, large-scale checks) could lead to performance bottlenecks.