Lecture 12: Set, Lists and Comprehensions

Swakkhar Shatabda

B.Sc. in Data Science
Department of Computer Science and Engineering
United International University

January 17, 2024

Slicing a List

- To make a slice, you specify the index of the first and last elements you want to work with.
- As with the range() function, Python stops one item before the second index you specify.

```
players = ['charles', 'martina', 'michael', 'florence', 'eli'
print(players[0:3])
```

- If you omit the first index in a slice, Python automatically starts your slice at the beginning of the list.
- if you want all items from the third item through the last item, you can start with index 2 and omit the second index.
- if we want to output the last three players on the roster, we can use the slice players[-3:].

Copying a List

- To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index ([:]).

```
my_foods = ['pizza', 'falafel', 'carrot cake']
friend_foods = my_foods[:]
my_foods.append('cannoli')
friend_foods.append('ice cream')
print("My favorite foods are:")
print(my_foods)
print("My friend's favorite foods are:")
print(friend_foods)
```

Sets

- A set is an unordered collection of unique values.
- Sets may contain only immutable objects, like strings, ints, floats and tuples that contain only immutable elements.
- Though sets are iterable, they are not sequences and do not support indexing and slicing with square brackets, []. Dictionaries also do not support slicing.

```
colors = {'red', 'orange', 'yellow', 'green', 'red', 'blue'}
print(colors)
```

- An important use of sets is duplicate elimination, which is automatic when creating a set.

Sets

- You can determine the number of items in a set with the built-in len function
- You can check whether a set contains a particular value using the in and not in operators
- Sets are iterable, so you can process each set element with a for loop.

```
colors = {'red', 'orange', 'yellow', 'green', 'red', 'blue'}
print(colors)
print(len(colors))
print('red' in colors)
print('violet' not in colors)
for c in colors:
    print(c.upper(), end=' ')
```

Sets

- You can create a set from another collection of values by using the built-in set function.

```
numbers = list(range(10)) + list(range(5))
print(numbers)
s = set(numbers)
print(s)
```

- Sets are mutable.
  - Set method add inserts its argument if the argument is not already in the set; otherwise, the set remains unchanged.
  - Set method remove removes its argument from the set—a KeyError occurs if the value is not in the set.
  - Method discard also removes its argument from the set but does not cause an exception if the value is not in the set.

Operations

- Various operators and methods can be used to compare sets. The following sets contain the same values, so == returns True and != returns False.

```
print({1, 3, 5} == {3, 5, 1})
print({1, 3, 5} != {3, 5, 1})
```

- The < operator tests whether the set to its left is a proper subset of the one to its right—that is, all the elements in the left operand are in the right operand, and the sets are not equal.
- The <= operator tests whether the set to its left is an improper subset of the one to its right—that is, all the elements in the left operand are in the right operand, and the sets might be equal.

Operations

- The $>$ operator tests whether the set to its left is a proper superset of the one to its right—that is, all the elements in the right operand are in the left operand, and the left operand has more elements.
- The $>=$ operator tests whether the set to its left is an improper superset of the one to its right—that is, all the elements in the right operand are in the left operand, and the sets might be equal
- You may also check for an improper subset with the set method issubset and for an improper superset with the set method issuperset.
- The union of two sets is a set consisting of all the unique elements from both sets (| operator).
- The intersection of two sets is a set consisting of all the unique elements that the two sets have in common (& operator).
- The difference between two sets is a set consisting of the elements in the left operand that are not in the right operand (- operator).

List Comprehensions

- List comprehensions— a concise and convenient notation for creating new lists.
- List comprehensions can replace many for statements that iterate over existing sequences and create new lists.

```
list1 = []
for item in range(1, 6):
    list1.append(item)

list2 = [i for i in range(1,6)]
```

Filter and Mapping

- A list comprehension's expression can perform tasks, such as calculations, that map elements to new values (possibly of different types).

```
list3 = [item ** 3 for item in range(1, 6)]
```

- Another common functional-style programming operation is filtering elements to select only those that match a condition. This typically produces a list with fewer elements than the data being filtered.
- To do this in a list comprehension, use the if clause.

```
list4 = [item for item in range(1, 11) if item % 2 == 0]
colors = ['red', 'orange', 'yellow', 'green', 'blue']
olors2 = [item.upper() for item in colors]
```

Generators

- A generator expression is similar to a list comprehension, but creates an iterable generator object that produces values on demand.
- This is known as lazy evaluation. List comprehensions use greedy evaluation—they create lists immediately when you execute them.
- For large numbers of items, creating a list can take substantial memory and time. Generator expressions can reduce your program's memory consumption and improve performance if the whole list is not needed at once.
- Generator expressions are defined in parentheses instead of square brackets.

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
for value in (x ** 2 for x in numbers if x % 2 != 0):
    print(value, end='  ')
```

Zip

- Built-in function zip enables you to iterate over multiple iterables of data at the same time.

```
names = ['Bob', 'Sue', 'Amanda']
grade_point_averages = [3.5, 4.0, 3.75]
for name, gpa in zip(names, grade_point_averages):
    print(f'Name={name}; GPA={gpa}')
```

- zip can be used for list comprehension

```
[(a + b) for a, b in zip([10, 20, 30], [1, 2, 3])]
```

Dictionary Comprehensions

- Dictionary comprehensions provide a convenient notation for quickly generating dictionaries, often by mapping one dictionary to another.

```
grades = {'Sue': [98, 87, 94], 'Bob': [84, 95, 91]}
grades2 = {k: sum(v) / len(v) for k, v in grades.items()}
print(grades2)
```

Set Comprehensions

- Like dictionary comprehensions, you define set comprehensions in curly braces.

```
numbers = [1, 2, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 10]
evens = {item for item in numbers if item % 2 == 0}
```

Homework

In check-writing systems, it's crucial to prevent alteration of check amounts. One common security method requires that the amount be written in numbers and spelled out in words as well. Even if someone can alter the numerical amount of the check, it's tough to change the amount in words. Create a dictionary that maps numbers to their corresponding word equivalents. Write a script that inputs a numeric check amount that's less than 1000 and uses the dictionary to write the word equivalent of the amount. For example, the amount 112.43 should be written as ONE HUNDRED TWELVE AND 43/100