CH-5 Sequences: Lists and Tuples

September 2024

Introduction

- Collections are prepackaged data structures consisting of related data items.
- Examples of collections include your favorite songs on your smartphone, your contacts list, a library's books, your cards in a card game, your favorite sports team's players, the stocks in an investment portfolio, patients in a cancer study and a shopping list.
- Python's built-in collections enable you to store and access data conveniently and efficiently.

Lists

- Ordered sequence of information, accessible by index
- Lists typically store homogeneous data, that is, values of the same data type.

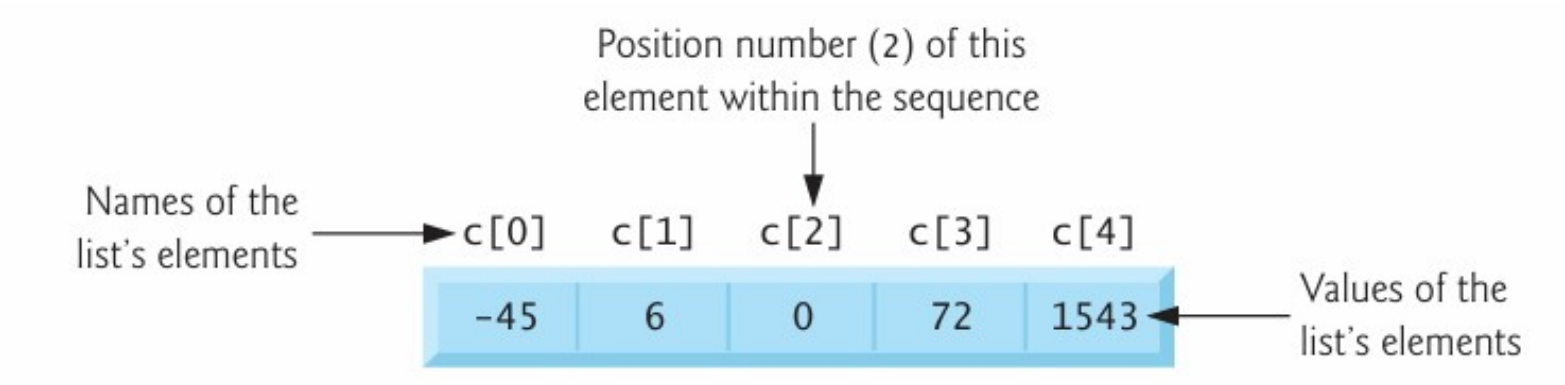
```
In [1]: c = [-45, 6, 0, 72, 1543]

In [2]: c
Out[2]: [-45, 6, 0, 72, 1543]
```

- They also may store heterogeneous data, that is, data of many different types.

```
['Mary', 'Smith', 3.57, 2022]
# student's first name, last name, grade point
average and graduation year
```

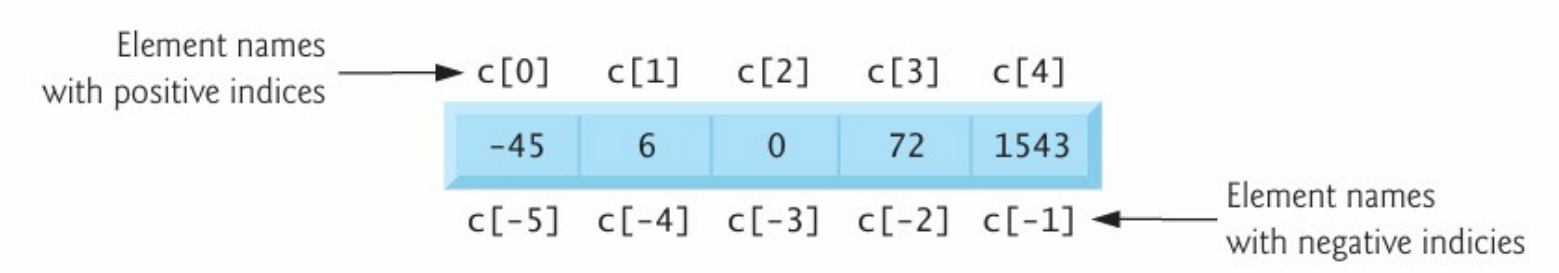
Lists

- Accessing Elements of a List- list's name followed by the element's index (or position number) enclosed in square brackets ([ ], known as the subscription operator)



Lists

- Accessing Elements from the End of the List with Negative Indices-



- Determining a List's Length- use built in len *function*
- Indices Must Be Integers or Integer Expressions. Using a non-integer index value causes a `TypeError`.
- Lists are mutable!- their elements can be modified. Assigning to an element at an index changes the value.

```
In [1]: L = [1,2,3,4,5]
In [2]: L[2] = 100
In [3]: L
Out[3]: [1, 2, 100, 4, 5]
Note that this is the same object L.
```

Lists

- List elements may be used as variables in expressions:

```
In [4]: L[1] + L[2]
Out[4]: 102
```

- Appending to a List with $+=$
  - $+=$ allows dynamic list expansion by adding each element from an iterable on the right side.
  - Examples of valid iterables: lists, tuples, and strings.

```
In [1]: L = [ ]

In [2]: for i in range(1,6):
...:      L += [i]
...:
In [3]: L
Out[3]: [1, 2, 3, 4, 5]
```

Lists

Appending to a List with +=

- Strings can also be appended character by character:

```
In [1]: letters = [ ]

In [2]: letters += 'Python'

In [3]: letters
Out[3]: ['P', 'y', 't', 'h', 'o', 'n']
```

- Attempting `L += non_iterable` (e.g., an integer) raises a `TypeError`.

Lists

Concatenating Lists with +

- You can concatenate two lists, two tuples or two strings using the + operator.
- The result is a new sequence of the same type containing the left operand's elements followed by the right operand's elements.

```
In [4]: L1 = [10, 20, 30]

In [5]: L2 = [40, 50, 60, 70]

In [6]: L1 + L2
Out[6]: [10, 20, 30, 40, 50, 60, 70]
```

- A TypeError occurs if the + operator's operands are difference sequence types.

Lists

Using for and range to Access List Indices and Values

```
In [1]: L = [10, 20, 30, 40, 50]

In [2]: for i in range(len(L)):
   ...:     print(f" {i}: {L[i]}")
   ...:
0: 10
1: 20
2: 30
3: 40
4: 50
```

Comparison Operators

```
In [1]: L1 = [10, 20, 30]
In [2]: L2 = [40, 50, 60, 70]
In [3]: L1==L2
Out[3]: False
```

Tuples

- an ordered sequence of elements, can mix element types
- cannot change element values, immutable
- represented with parentheses
- Creating Tuples
  - To create an empty tuple, use empty parentheses:

```
In [4]: t = () # empty tuple
```

  - You can pack a tuple by separating its values with commas:

```
In [5]:  student_tuple = 'John', 'Green', 3.3

In [6]: student_tuple
Out[6]: ('John', 'Green', 3.3)
```

Tuples

- Creating Tuples
  - The following is a tuple with one element only:

```
In [7]: s_t = 'red',
In [9]: s_t
Out[9]: ('red',)
```

- Accessing Tuple Elements

  Like list indices, tuple indices start at 0.

```
In [14]: t_t = (5, 20, 30)

In [15]: t_t
Out[15]: (5, 20, 30)

In [16]: s_c = t_t[0]*3600 + t_t[1]*60 + t_t[2]

In [17]: s_c # seconds
Out[17]: 18430
```

Tuples

- Adding Items to a String or Tuple
  - As with lists, the += augmented assignment statement can be used with strings and tuples, even though they're immutable.

```
In [18]: tuple1 = (10,20,30)

In [19]: tuple1 += (40,50)

In [20]: tuple1
Out[20]: (10, 20, 30, 40, 50)
```

  - For a string or tuple, the item to the right of += must be a string or tuple, respectively— mixing types causes a `TypeError`.

Tuples

- Appending Tuples to Lists
  - You can use $+=$ to append a tuple to a list:

```
In [21]: L = [1,2,3,4,5]

In [22]: t = (6,7,8)

In [23]: L += t

In [24]: L
Out[24]: [1, 2, 3, 4, 5, 6, 7, 8]
```

Tuples

- Tuples May Contain Mutable Objects

```
In [23]: st_t = ('Amanda', 'Blue', [98, 75, 87])
```

  - Even though the tuple is immutable, its list element is mutable:

```
In [26]: st_t
Out[26]: ('Amanda', 'Blue', [98, 75, 87])

In [27]: st_t[2]
Out[27]: [98, 75, 87]

In [28]: st_t[2][0] = 5

In [29]: st_t
Out[29]: ('Amanda', 'Blue', [5, 75, 87])
```

Unpacking Sequences

- You can unpack any sequence's elements by assigning the sequence to a comma-separated list of variables.
- A `ValueError` occurs if the number of variables to the left of the assignment symbol is not identical to the number of elements in the sequence on the right.

```
In [17]: student_t = ['Amanda', [98, 85, 87]]
In [18]: name, grades = student_t

In [19]: name
Out[19]: 'Amanda'

In [20]: grades
Out[20]: [98, 85, 87]
```

Unpacking Sequences

- The following code unpacks a string, a list and a sequence produced by range:

```
In [1]: f,s = 'HI'

In [2]: print(f, s)
H I

In [3]: f,s,t = [1,2,3]

In [4]: print(f,s,t)
1 2 3

In [5]: f,s,t = range(1,10,3) # start = 1, end =
    10, step = 3

In [6]: print(f,s,t)
1 4 7
```

Unpacking Sequences

- You can swap two variables' values using sequence packing and unpacking:

```
In [7]: a = 5

In [8]: b = 10

In [9]: a,b
Out[9]: (5, 10)

In [10]: a,b = b,a

In [11]: a,b
Out[11]: (10, 5)
```

Unpacking Sequences

Accessing Indices and Values Safely with Built-in Function enumerate:

- The preferred mechanism for accessing an element's index and value is the built-in function enumerate.
- This function receives an iterable and creates an iterator that, for each element, returns a tuple containing the element's index and value

```
In [12]: colors = ['red', 'green', 'blue']

In [13]: list(enumerate(colors))
Out[13]: [(0, 'red'), (1, 'green'), (2, 'blue')]

In [14]: tuple(enumerate(colors))
Out[14]: ((0, 'red'), (1, 'green'), (2, 'blue'))

In [15]: for index, value in enumerate(colors):
    ...:     print(f" {index}: {value}", end=' ')
    ...:
 0: red  1: green  2: blue
```

Unpacking Sequences

Creating a Primitive Bar Chart

```
# creating a bar chart
numbers = [2,5,3,6,7]
print('Creating a bar chart')
print(f"{'index':<5} {'value':<5} {'bar':<10}")
for index, value in enumerate(numbers):
    print(f"{index:<5} {value:<5} {'*'*value:<10}")
```

Output:

```
Creating a bar chart
index value bar
0     2     **
1     5     *****
2     3     ***
3     6     ******
4     7     *******
```

Sequence Slicing

- You can slice sequences to create new sequences of the same type containing subsets of the original elements.
- Slice operations can modify mutable sequences—those that do not modify a sequence work identically for lists, tuples and strings.
- Specifying a Slice with Starting and Ending Indices:
  - The slice copies elements from the **starting index** to the left of the colon up to, but not including, the **ending index** to the right of the colon.
  - The original list is not modified.

```
In [1]: L = [1,2,3,4,5,6,7]

In [2]: L[2:5]
Out[2]: [3, 4, 5]
```

Sequence Slicing

- Specifying a Slice with Only an Ending Index:
  - If you omit the starting index, 0 is assumed.
  - $[0:5] \equiv [:5]$

```
In [3]: L[0:5]
Out[3]: [1, 2, 3, 4, 5]

In [4]: L[:5]
Out[4]: [1, 2, 3, 4, 5]
```

- Specifying a Slice with Only a Starting Index:
  - If you omit the ending index, Python assumes the sequence's length.

```
In [5]: L[2:]
Out[5]: [3, 4, 5, 6, 7]
```

Sequence Slicing

- Specifying a Slice with No Indices:
  - Omitting both the start and end indices copies the entire sequence.

```
In [6]: L[:]
Out[6]: [1, 2, 3, 4, 5, 6, 7]
```

  - Though slices create new objects, slices make shallow copies of the elements—that is, they copy the elements' references but not the objects they point to.
  - So, in the snippet above, the new list's elements refer to the same objects as the original list's elements, rather than to separate copies.
- Slicing with Steps:

```
In [7]: L[1:6:2]
Out[7]: [2, 4, 6]
```

Sequence Slicing

- Slicing with Negative Indices and Steps:
  - You can use a **negative step** to select slices in **reverse order**.

```
In [8]: L[::-1]
Out[8]: [7, 6, 5, 4, 3, 2, 1]

#This is equivalent to L[-1:-8:-1]
In [10]: L[-1:-8:-1]
Out[10]: [7, 6, 5, 4, 3, 2, 1]
```

- Modifying Lists Via Slices:
  - You can modify a list by assigning to a slice of it—the rest of the list is unchanged.

```
In [14]: L[0:4] = ['A','B','C','D']

In [15]: L
Out[15]: ['A', 'B', 'C', 'D', 5, 6, 7]
```

Sequence Slicing

- Modifying Lists Via Slices:
  - First k elements of the list can be deleted by assigning an empty list to the k-element slice:

```
In [1]: L = [1,2,3,4,5,6,7]

In [2]: L[0:4] = []

In [3]: L
Out[3]: [5, 6, 7]
```

  - You can delete all the elements in the list, leaving the existing list empty.

```
In [7]: L = [1,2,3,4,5]

In [8]: L[:]=[]

In [9]: L
Out[9]: []
```

Sequence Slicing

- Modifying Lists Via Slices:
  - Deleting contents keeps the list object's identity.
  - Assigning a new list creates a different object in memory.

```
In [16]: L
Out[16]: [10, 4, 10]

In [17]: id(L)
Out[17]: 1884662403648

In [18]: L[:] = []

In [19]: L
Out[19]: []

In [20]: id(L)
Out[20]: 1884662403648

In [21]: L = []

In [22]: id(L)
Out[22]: 1884662936000
```

del Statement

- The del statement also can be used to remove elements from a list and to delete variables from the interactive session.
- You can remove the element at any valid index or the element(s) from any valid slice.
- Deleting the Element at a Specific List Index

```
In [2]: L = list(range(1,11))

In [3]: L
Out[3]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [4]: del L[-5]

In [5]: L
Out[5]: [1, 2, 3, 4, 5, 7, 8, 9, 10]
```

del Statement

- Deleting a Slice from a List

```
In [8]: del L[-1:-3:-1]

In [9]: L
Out[9]: [1, 2, 3, 4, 5, 7, 8]

In [10]: del L[::2]

In [11]: L
Out[11]: [2, 4, 7]
```

- Deleting a Slice Representing the Entire List

```
In [12]: del L[:]

In [13]: L
Out[13]: []
```

- Deleting a Variable from the Current Session

```
In [14]: del L
```

Passing Lists to Functions

- Passing Lists to a Function
  - Lists are mutable; functions receive a reference to the original list.
  - Modifying list elements within a function alters the original list.
- Passing Tuples to a Function
  - Tuples are immutable; attempting to modify elements causes a TypeError.
  - However, mutable elements (e.g., lists) within a tuple can still be modified in a function.

Passing Lists to Functions

```
L = ['a', 'b', 'c']
T = ('a', 'b', 'c')
def modify_sequence(s):
    for i in range(len(s)):
        s[i]*=2
    return s
```

```
print(modify_sequence(L))
Output: === RESTART: C:/Users/jps15/OneDrive/Desktop/
    Python/a51.py ==
['aa', 'bb', 'cc']
```

```
print(modify_sequence(T))
Output: TypeError: 'tuple' object does not support
    item assignment
```

- Recall that tuples may contain mutable objects, such as lists. Those objects still can be modified when a tuple is passed to a function.

Sorting Lists

- Sorting arranges data in ascending or descending order.
- A key computing task, essential for data organization and retrieval.
- Extensively studied in computer science, especially in data structures and algorithms.
- Critical for improving data processing, search efficiency, and overall program performance.
- Detailed exploration in topics like recursion, search algorithms, and Big O analysis.

Sorting Lists

- Sorting a List in Ascending Order
  - List method `sort` modifies a list to arrange its elements in ascending order:

```
In [16]: n = [1,8,3,4,9,7,6,2,5,10]

In [17]: n.sort()

In [18]: n
Out[18]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Sorting a List in Descending Order
  - To sort a list in descending order, call list method sort with the optional keyword argument `reverse` set to True (False is the default).

```
In [19]: n = [1,8,3,4,9,7,6,2,5,10]

In [20]: n.sort(reverse = True)

In [21]: n
Out[21]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Sorting Lists

- Built in Function *sorted*
  - Built-in function `sorted` returns a **new list** containing the sorted elements of its argument sequence—the original sequence is **unmodified**.
  - The built-in sorted function can be used on lists and tuples, but when applied to a string, it treats the string as a sequence of characters and returns a list of sorted characters, not a sorted string.
  - Use the optional keyword argument `reverse` with the value `True` to sort the elements in `descending order`.

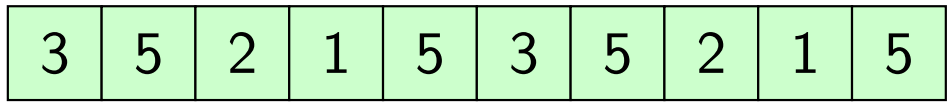
Searching Sequences

- Purpose of Searching: Searching involves locating a specific key value within a sequence (e.g., list, tuple, or string) to determine if it exists in the collection.
- Application of Searching: This technique helps quickly identify whether a desired value is present in the sequence, which is essential for data retrieval and decision-making processes.
- List Method index: List method index takes as an argument a search key—the value to locate in the list—then searches through the list from index 0 and returns the index of the **first** element that matches the search key:

```
In [7]: L = [3, 5, 2, 1, 5]

In [8]: L.index(5)
Out[8]: 1

In [9]: L.index(7)

ValueError: 7 is not in list
```

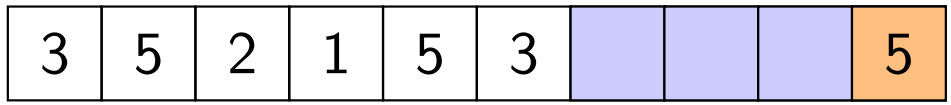
Searching Sequences

- Specifying the Starting Index of a Search
  - **Multiplying the Sequence** `L *= 2`



List after `L *= 2`

  - **Subset Search Using** `index(5, 6)`

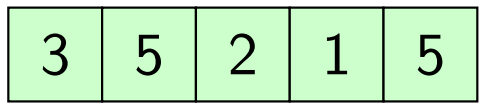


Search Result: `L.index(5, 6)` $\rightarrow$ Index 9

    - Highlights the subset from index 6 onward in blue and marks the found value 5 at index 9 in orange.

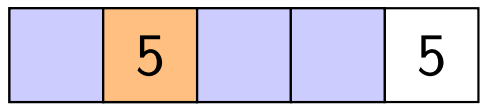
Searching Sequences

- Specifying the Starting and Ending Indices of a Search
  - **Original List:** `L = [3, 5, 2, 1, 5]`



Original List `L`

  - **Subset Search Using** `L.index(5, 0, 4)`



Search Result: `L.index(5, 0, 4)` $\rightarrow$ Index 1

    - Highlights the search subset from index 0 to 4 (not including 4) in blue, and marks the found value 5 at index 1 in orange.

Searching Sequences

- Operators `in` and `not in`
  - Operator in tests whether its right operand's iterable contains the left operand's value.
  - Similarly, operator not in tests whether its right operand's iterable does not contain the left operand's value.

```
L = [3, 5, 2, 1, 5]

# Operator in
print(1000 in L)   # Output: False
print(5 in L)      # Output: True

# Operator not in
print(1000 not in L)   # Output: True
print(5 not in L)      # Output: False
```

Searching Sequences

- Using Operator in to Prevent a `ValueError`
  - Python code:

```
key = 1000

if key in numbers:
    print(f'found {key} at index {L.index(key)
    }')
else:
    print(f'{key} not found')
```

  - In this snippet, the `in` operator is used to prevent a `ValueError` when searching for a key in the sequence `L`.
- Built-in Functions: `any` and `all`

$$\text{any(iterable)} = \begin{cases} \text{True}, & \text{if any item in iterable is True} \\ \text{False}, & \text{otherwise} \end{cases}$$

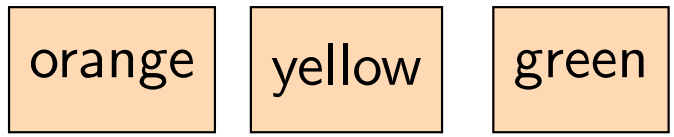
$$\text{all(iterable)} = \begin{cases} \text{True}, & \text{if all items in iterable are True} \\ \text{False}, & \text{otherwise} \end{cases}$$

Searching Sequences

- Nonzero values are considered **True**.
- Zero is considered **False**.
- Non-empty iterables evaluate to **True**.
- Empty iterables evaluate to **False**.

Other List Methods

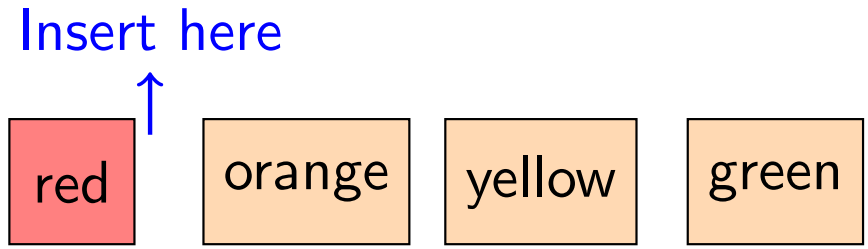
- Inserting an Element at a Specific List Index
  - **Original List:** `color_names = ['orange', 'yellow', 'green']`



Original List `color_names`

  - **Insert `'red'` at Index 0**

```
color_names.insert(0, 'red')
```

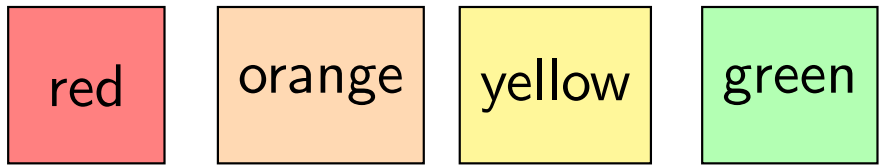


List After Insertion

  - **Updated List:** `['red', 'orange', 'yellow', 'green']`

Other List Methods

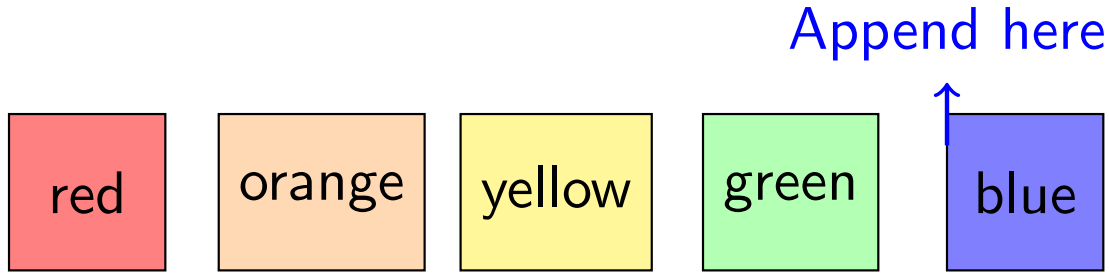
- Adding an Element to the End of a List
  - **Original List:** `color_names = ['red', 'orange', 'yellow', 'green']`



Original List `color_names`

  - **After Appending** `'blue'`

```
color_names.append('blue')
```



Updated List `['red', 'orange', 'yellow', 'green', 'blue']`

Other List Methods

- Adding All the Elements of a Sequence to the End of a List
  - Use list method `extend` to add all the elements of another sequence to the end of a list:

```
In [11]: colors = [ 'red', 'violet', 'green']

In [12]: colors.extend(['BLUE', 'BLACK'])

In [13]: colors
Out[13]: ['red', 'violet', 'green', 'BLUE', '
    BLACK']
```

  - This is the equivalent of using $+=$.

```
1 In [14]: colors.extend('ABC')
2 In [15]: colors
3 Out[15]: ['red', 'violet', 'green', 'BLUE', '
    BLACK', 'A', 'B', 'C']
4 In [16]: colors.extend((1,2,3))
5 In [17]: colors
6 Out[17]: ['red', 'violet', 'green', 'BLUE', '
    BLACK', 'A', 'B', 'C', 1, 2, 3]
```

Other List Methods

- Removing the First Occurrence of an Element in a List
  - Method remove deletes the first element with a specified value—a `ValueError` occurs if remove's argument is not in the list:

```
In [19]: colors
Out[19]: ['red', 'orange', 'yellow', 'green',
    'blue', 'orange', 'white']

In [20]: colors.remove('orange')

In [21]: colors
Out[21]: ['red', 'yellow', 'green', 'blue', '
    orange', 'white']
```

- Emptying a List
  - To delete all the elements in a list, call method clear:

```
In [22]: colors.clear()

In [23]: colors
Out[23]: []
```

Other List Methods

- Counting the Number of Occurrences of an Item
  - List method count searches for its argument and returns the number of times it is found:

```
In [24]: responses = [1,2,1,4,5,1,3,2,5,4,6,3]

In [25]: responses.count(3)
Out[25]: 2
```

- Reversing a List's Elements
  - List method reverse reverses the contents of a list in place, rather than creating a reversed copy, as we did with a slice previously:

```
In [36]: colors
Out[36]: ['red', 'orange', 'yellow', 'green',
    'blue']

In [37]: colors.reverse()
In [38]: colors
Out[38]: ['blue', 'green', 'yellow', 'orange',
     'red']
```

Other List Methods

- Copying a List
  - List method `copy` returns a *new* list containing a *shallow* copy of the original list:

```
In [48]: colors
Out[48]: ['blue', 'green', 'yellow', 'orange',
     'red']

In [49]: copied_colors = colors.copy()

In [50]: copied_colors
Out[50]: ['blue', 'green', 'yellow', 'orange',
     'red']
```

  - This is equivalent to the previously demonstrated slice operation:

```
1 copied_colors = colors[:]
```

Simulating Stacks with Lists

- Python does not have a built-in stack type, but you can think of a stack as a constrained list.
- You push using list method append, which adds a new element to the end of the list.
- You pop using list method pop with no arguments, which removes and returns the item at the end of the list.

```
In [1]: stack = []
In [2]: stack.append('green')
In [3]: stack
Out[3]: ['green']
In [4]: stack.append('red')
In [5]: stack
Out[5]: ['green', 'red']
In [6]: stack.pop()
Out[6]: 'red'
In [7]: stack
Out[7]: ['green']
In [8]: stack.pop()
Out[8]: 'green'
In [9]: stack
Out[9]: []
```

List Comprehensions

- List comprehensions provide a concise way to create new lists from existing sequences.
- Replacement for Loops: They can often replace for loops used to populate lists, making the code shorter and clearer.

```
In [18]: for i in range(1,11):
    ...:     L.append(i)
    ...:

In [19]: L
Out[19]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
#Using a List Comprehension to Create a List of
    Integers
In [20]: L_New = [ i for i in range(1,11)]

In [21]: L_New
Out[21]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

List Comprehensions

- Structure of List Comprehensions:

```
new\_list = [expression for item in iterable if
    condition]
```

  - **expression:** The operation or transformation to apply to each item.
  - **for item in iterable:** Iterates over each item in the specified iterable.
  - **if condition (optional):** A filter that only includes items for which the condition is true.

- List Comprehension That Processes Another List's Elements

```
In [22]: c = ['RED', 'BLUE', 'WHITE']

In [23]: C = [color.lower() for color in c]

In [24]: C
Out[24]: ['red', 'blue', 'white']
```

Generator Expressions

- A generator expression is similar to a list comprehension, but it creates an iterable generator object rather than a list.
- Generates items one at a time on demand, known as lazy evaluation.
- Memory Efficiency: Uses less memory than list comprehensions for large data, as items are generated only when needed.
- **Syntax:** Defined with parentheses ( ) instead of square brackets [ ].
- **Performance:** Improves performance when you don't need the entire list immediately, reducing memory and computation time.

Filter, Map and Reduce

- built-in `filter` and `map` functions for `filtering` and `mapping`, respectively
- reductions in which you process a collection of elements into a single value, such as their count, total, product, average, minimum or maximum.

Filtering a Sequence's Values with the Built-In `filter` Function

```
In [1]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [2]: def is_odd(x):
   ...:     # returns True only if x is odd
   ...:     return x%2 !=0
   ...:
   ...:
In [3]: list(filter(is_odd, numbers)) # highter-
    order function
Out[3]: [3, 7, 1, 9, 5]
```

Filter, Map and Reduce

- Function `filter` returns an iterator, so filter's results are not produced until you iterate through them. This is another example of *lazy evaluation*.
- We can obtain the same results as above by using a list comprehension with an if clause:

```
In [8]: [item for item in numbers if is_odd(item)]
Out[8]: [3, 7, 1, 9, 5]
```

Using a lambda Rather than a Function

- Lambda expressions (or simply "lambdas") are used to define small, anonymous functions inline, typically where a function is passed as an argument.
- **Syntax:** `lambda parameter_list: expression`

Filter, Map and Reduce

- Begins with lambda, followed by parameters, a colon, and an expression.
- Automatically returns the result of the expression.

```
In [10]: list(filter(lambda x: x%2!=0, numbers))
Out[10]: [3, 7, 1, 9, 5]
```

- Here, `lambda x: x % 2 != 0` is passed to filter, selecting only odd numbers from numbers.
- filter returns an iterator, so `list()` converts it to a list for display.

- **Comparison to Regular Functions:**

```
# Standard function definition
def is_odd(x):
    return x % 2 != 0

# Equivalent lambda expression
lambda x: x % 2 != 0
```

Filter, Map and Reduce

Mapping a Sequence's Values to New Values

- The `map` function applies a specified function to each item in an iterable, returning an iterator with the results.
- **Syntax:** `map(function, iterable)`
  - The first argument is a function that takes one argument and returns a value.
  - The second argument is an iterable (e.g., list, tuple) containing values to transform.

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
list(map(lambda x: x ** 2, numbers))
```

  - Here, `lambda x: x ** 2` squares each number in `numbers`.
  - `map` uses lazy evaluation, so `list()` is used to convert the iterator to a list for display.
- **Equivalent List Comprehension:** `[item ** 2 for item in numbers]`

Filter, Map and Reduce

Combining `filter` and `map` with Lambda Expressions

```
In [10]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [11]: list(map(lambda x: x**2, filter(lambda x:
     x%2!=0, numbers)))
Out[11]: [9, 49, 1, 81, 25]
```

- `filter(lambda x: x % 2 != 0, numbers)`: Filters out only the odd values in numbers.
- `map(lambda x: x ** 2, ...)`: Squares each of the filtered odd values.
- `list(...)`: Converts the resulting iterator to a list for display.

- **Equivalent List Comprehension:**

```
[x ** 2 for x in numbers if x % 2 != 0]
```

Filter, Map and Reduce

Reduction: Totaling the Elements of a Sequence with sum

- Reductions aggregate a sequence's elements into a single value using functions like len, sum, min, and max.
- Custom reductions can be created with functools.reduce.
- This approach, which is foundational in big data frameworks like Hadoop, uses filter, map, and reduce in a functional programming style.

Copy vs Deep Copy

Copy (Shallow Copy):

- A shallow copy creates a new list, but only at the top level.
- Nested elements (like lists within lists) are not copied; they are shared between the original and the copy.

```
import copy
l = [[1, 2], 3, 4]
m = copy.copy(l)  # or m = l[:]
m[0][0] = 99
```

- Here, m is a new list, but `m[0]` and `l[0]` refer to the same nested list. So, modifying `m[0][0]` affects `l[0][0]`.
- **Shallow copy:** Only copies the top-level elements.

Copy vs Deep Copy

Copy (Deep Copy):

- A deep copy creates a completely independent copy, including nested elements. Changes to m will not affect l.

```
import copy
l = [[1, 2], 3, 4]
m = copy.deepcopy(l)
m[0][0] = 99
```

- Here, m is entirely separate from l, so changing `m[0][0]` does not change l.
- **Deep Copy:** Copies all elements, including nested ones.

Iterable and Iterator

Iterable

- An **iterable** is any Python object capable of returning its elements one at a time. Examples include lists, tuples, dictionaries, sets, and strings. These can be used in loops, such as `for` loops.
- To check if an object is iterable, it must implement the `__iter__()` method, which returns an iterator. However, an iterable itself does not keep track of its current position in the sequence.

```
my_list = [1, 2, 3]  # This is an iterable.
```

Iterator

- An **iterator** is a Python object that represents a stream of data. It keeps track of the current position in that data and knows how to return the next item in the sequence.
- An iterator implements two methods:

  `__iter__()`: Returns the iterator object itself.

  `__next__()`: Returns the next element in the sequence. Raises `StopIteration` when there are no more elements.

Iterable and Iterator

- You can obtain an iterator from an iterable by using the `iter()` function.

```
my_iter = iter(my_list)  # Creates an iterator
    from the iterable `my_list`.
print(next(my_iter))  # Outputs: 1
print(next(my_iter))  # Outputs: 2
```

Other Sequence Processing Functions

Finding the Minimum and Maximum Values Using a Key Function

- By default, Python's min and max compare strings based on their character's numerical values (ASCII/Unicode).
- Lowercase letters have higher numerical values than uppercase letters, affecting lexicographical order (e.g., 'Red' <'orange' is True because R has a lower numerical value than o).
- To get alphabetic ordering, convert each string to lowercase or uppercase during comparison.
- Use the key argument in min and max to apply a transformation for comparison without altering the original list.

Other Sequence Processing Functions

- min(colors, key=lambda s: s.lower()) returns 'Blue' (alphabetically first).
- max(colors, key=lambda s: s.lower()) returns 'Yellow' (alphabetically last).
- Here, lambda s: s.lower() ensures comparison ignores case.

```
In [13]: min(colors, key = lambda s: s.lower()) Out[13]:
    'Blue'
In [14]: max(colors, key = lambda s: s.lower()) Out[14]:
    'Yellow'
```

- **Key Argument Functionality:**
  - The key argument takes a function with one parameter.
  - min and max apply this function to each element, using the results for comparison.

Other Sequence Processing Functions

Iterating Backward Through a Sequence

- Built-in function `reversed` returns an iterator that enables you to iterate over a sequence's values backward.

```
In [7]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
In [7]: reversed_numbers = [item for item in
    reversed(numbers)]
In [8]: reversed_numbers
Out[8]: [36, 25, 64, 4, 16, 81, 1, 49, 9, 100]
```

Combining Iterables into Tuples of Corresponding Elements

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

```
In [9]: names = ['Bob', 'Sue', 'Amanda']
In [10]: grade_point_averages = [3.5, 4.0, 3.75]
In [11]: for name, gpa in zip(names,
    grade_point_averages):
   ...:
  print(f'Name={name}; GPA={gpa}')
   ...:
```

Two-Dimensional Lists

- Lists can contain other lists as elements.
- A typical use of such nested (or multidimensional) lists is to represent tables of values consisting of information arranged in rows and columns.
- To identify a particular table element, we specify two indices—by convention, the first identifies the element's row, the second the element's column.
- Lists that require two indices to identify an element are called two-dimensional lists (or double-indexed lists or double-subscripted lists).
- Creating a Two-Dimensional List-
- Illustrating a Two-Dimensional List
- Identifying the Elements in a Two-Dimensional List-
- How the Nested Loops Execute-

Intro to Data Science: Simulation and Static Visualizations

Visualizing Die-Roll Frequencies and Percentages

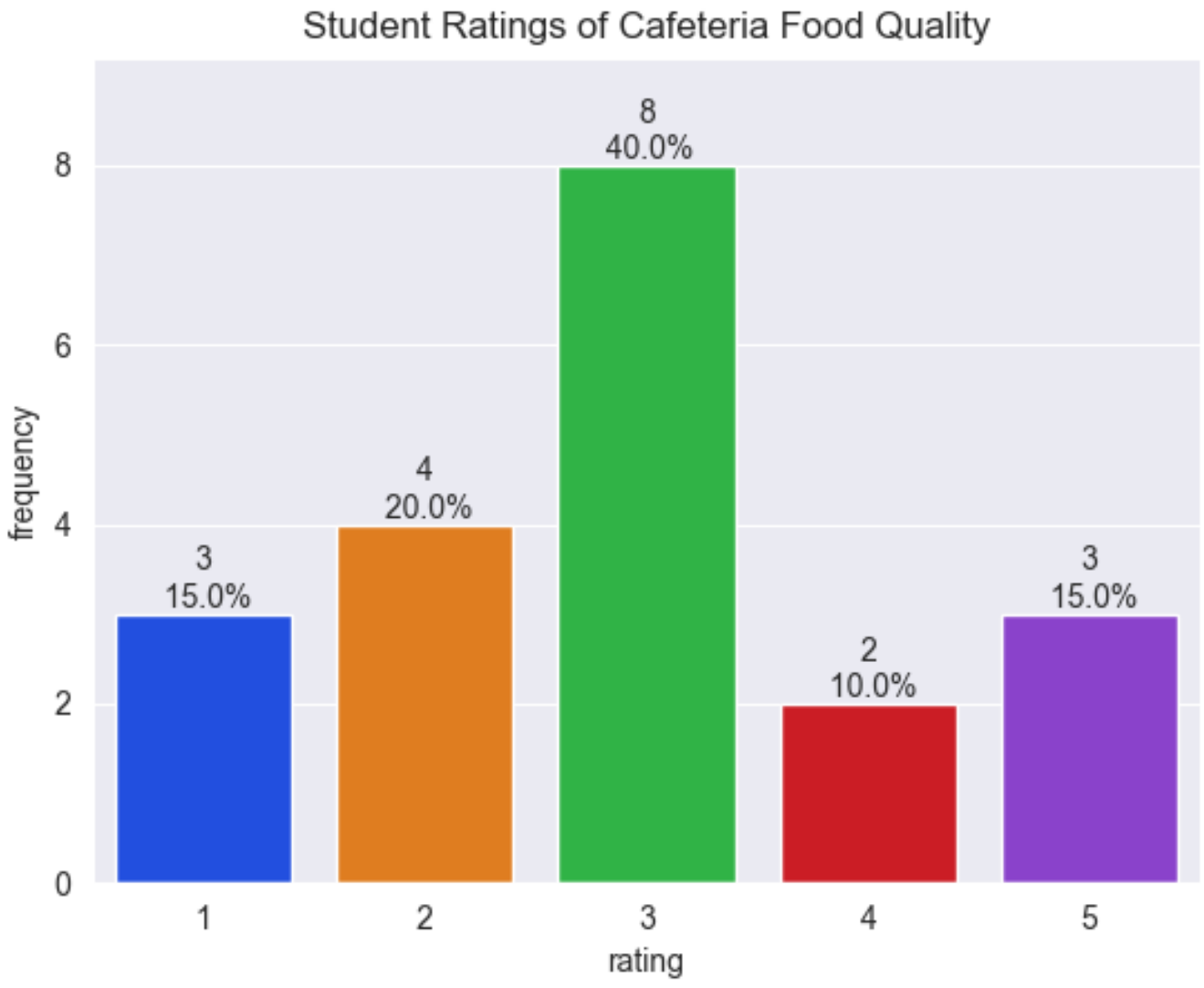
```
import matplotlib.pyplot as plt
import numpy as np
import random
import seaborn as sns
import sys

# Check if the number of rolls is provided as a
    command-line argument
try:
    num_rolls = int(sys.argv[1])  # Get the first
    argument after the script name
except (IndexError, ValueError):
    num_rolls = 6000  # Default to 6000 if no valid
    argument is provided

# Simulate die rolls
rolls = [random.randrange(1, 7) for _ in range(
    num_rolls)]

# Calculate unique values and their frequencies
values,frequencies=np.unique(rolls,return_counts=True)
```

```
# Set the plot title
title = f'Rolling a Six-Sided Die {num_rolls:,} Times'
# Set style and create the bar plot
sns.set_style("darkgrid")
axes = sns.barplot(x=values, y=frequencies, palette='
    bright')
# Set the title and labels
axes.set_title(title)
axes.set(xlabel='Die Value', ylabel='Frequency')
# Set y-axis limit for better display of text above
    bars
axes.set_ylim(top=max(frequencies) * 1.10)
# Annotate each bar with its frequency and percentage
for bar, frequency in zip(axes.patches, frequencies):
    text_x = bar.get_x() + bar.get_width() / 2.0
    text_y = bar.get_height()
    text =f'{frequency}\n{frequency / num_rolls:0.3%}'
    axes.text(text_x, text_y, text, fontsize=11, ha='
    center', va='bottom')
# Display the plot
plt.show()
```

Intro to Data Science: Survey Response Statistics Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being “awful” and 5 being “excellent.” Place the 20 responses in a list
1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 5
Determine and display the frequency of each rating. Use the built-in functions, statistics module functions to display the following response statistics:
minimum, maximum, range, mean, median, mode, variance and standard deviation.

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# List of responses (ratings) from students on a scale of 1 to 5
responses = [1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 5]

# Get the unique values (ratings) and their corresponding frequencies
values, frequencies = np.unique(responses, return_counts=True)
# Set the visual style of the plot to 'darkgrid'
sns.set_style('darkgrid')
# Title for the bar chart
title = "Student Ratings of Cafeteria Food Quality"
```

```
# Create a bar plot with Seaborn
axes = sns.barplot(x=values, y=frequencies, palette='bright')
# Set the title of the chart
axes.set_title(title)
# Set labels for the x and y axes
axes.set(xlabel='Rating', ylabel='Frequency')
# Set y-axis limit slightly above max frequency for spacing
axes.set_ylim(top=max(frequencies) * 1.15)

# Annotate each bar with frequency and percentage of total responses
for bar, frequency in zip(axes.patches, frequencies):
    # x position for annotation (center of each bar)
    x_t = bar.get_x() + bar.get_width()/2
    # y position for annotation (top of each bar)
    y_t = bar.get_height()
    # Text with count and percentage
    text = f"{frequency}\n{frequency / len(responses):0.1%}"
    # Display text above each bar
    axes.text(x_t, y_t, text, fontsize=10, ha='center', va='bottom')

# Show the plot
plt.show()
```


Student Ratings of Cafeteria Food Quality
frequency
rating
0
2
4
6
8
1
2
3
4
5
3
15.0%
4
20.0%
8
40.0%
2
10.0%
3
15.0%

```python
import statistics as s
responses = [1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 5]
# Statistical measures
minimum = min(responses)
maximum = max(responses)
data_range = maximum - minimum
mean = s.mean(responses)
median = s.median(responses)
mode = s.mode(responses)
variance = s.variance(responses)
std_dev = s.stdev(responses)

# Display statistics
print("Minimum:", minimum)
print("Maximum:", maximum)
print("Range:", data_range)
print("Mean:", mean)
print("Median:", median)
print("Mode:", mode)
print(f"Variance: {variance:0.2f}")
print(f"Standard Deviation: { std_dev:.2f}")
```

Exercises

Q.1 What, if anything, is wrong with each of the following code segments?

```
a) day, high_temperature = ('Monday', 87, 65)
b) numbers = [1, 2, 3, 4, 5]
   numbers[10]
c) name = 'amanda'
   name[0] = 'A'
d) numbers = [1, 2, 3, 4, 5]
   numbers[3.4]
e) student_tuple=('Amanda','Blue',[98, 75, 87])
   student_tuple[0] = 'Ariana'
f) ('Monday', 87, 65) + 'Tuesday'
g) 'A' += ('B', 'C')
h) x = 7
   del x
   print(x)
i) numbers = [1, 2, 3, 4, 5]
   numbers.index(10)
j) numbers = [1, 2, 3, 4, 5]
   numbers.extend(6, 7, 8)
```

Exercises

Q.2 (Iteration Order) Create a 2-by-3 list, then use a nested loop to:

a) Set each element's value to an integer indicating the order in which it was processed by the nested loop.

b) Display the elements in tabular format. Use the column indices as headings across the top, and the row indices to the left of each row.

Q.3 (IPython Session: Slicing) Create a string called alphabet containing 'abcdefghijklmnopqrstuvwxyz', then perform the following separate slice operations to obtain:

a) The first half of the string using starting and ending indices.

b) The first half of the string using only the ending index.

c) The second half of the string using starting and ending indices.

d) The second half of the string using only the starting index.

e) Every second letter in the string starting with 'a'.

f) The entire string in reverse.

g) Every third letter of the string in reverse starting with 'z'

Exercises

Q.4 (Functions Returning Tuples) Define a function `rotate` that receives three arguments and returns a tuple in which the first argument is at index 1, the second argument is at index 2 and the third argument is at index 0. Define variables a, b and c containing 'Doug', 22 and 1984. Then call the function three times. For each call, unpack its result into a, b and c, then display their values.

Q.5 (Duplicate Elimination) Create a function that receives a list and returns a (possibly shorter) list containing only the unique values in sorted order. Test your function with a list of numbers and a list of strings.

Exercises

Q.6 (Sorting Letters and Removing Duplicates) Insert 20 random letters in the range 'a' through 'f' into a list. Perform the following tasks and display your results:

a) Sort the list in ascending order.

b) Sort the list in descending order.

c) Get the unique values sort them in ascending order.