5. Sequences: Lists and Tuples

Objectives

In this chapter, you'll:

- Create and initialize lists and tuples.
- Refer to elements of lists, tuples and strings.
- Sort and search lists, and search tuples.
- Pass lists and tuples to functions and methods.

Objectives (cont.)

- Use list methods to perform common manipulations, such as searching for items, sorting a list, inserting items and removing items.
- Use additional Python functional-style programming capabilities, including lambdas and the operations filter, map and reduce.

Objectives (cont.)

- Use functional-style list comprehensions to create lists quickly and easily, and use generator expressions to generate values on demand.
- Use two-dimensional lists.
- Enhance your analysis and presentation skills with the Seaborn and Matplotlib visualization libraries.

Outline

- 5.1 Introduction (05_01.html)
- 5.2 Lists (05_02.html)
- 5.3 Tuples (05_03.html)
- 5.4 Unpacking Sequences (05_04.html)
- 5.5 Sequence Slicing (05_05.html)
- 5.6 del Statement (05_06.html)
- 5.7 Passing Lists to Functions (05_07.html)

Outline (cont.)

Outline (cont.)

This content is based on Chapter 5 of the book **Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud** (https://amzn.to/2VvdnxE).

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.1 Introduction

- **Collections** are prepackaged data structures consisting of related data items.
- Examples of collections:
  - Favorite songs on your smartphone
  - Contacts list
  - A library’s books
  - Cards in a card game
  - Favorite sports team’s players
  - Stocks in an investment portfolio
  - Patients in a cancer study
  - Shopping list.
- Lists (which are modifiable) and tuples (which are not) have many common capabilities.
- Each can hold items of the same or different types.
- Lists can **dynamically resize** as necessary.
- The Intro to Data Science section uses the visualization libraries Seaborn and Matplotlib to interactively develop static bar charts containing the die frequencies.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.2 Lists

- Many of the capabilities shown in this section apply to all sequence types.

Creating a List

- **Lists** typically store **homogeneous data**, but may store **heterogeneous data**.

In [1]:

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

In [2]:

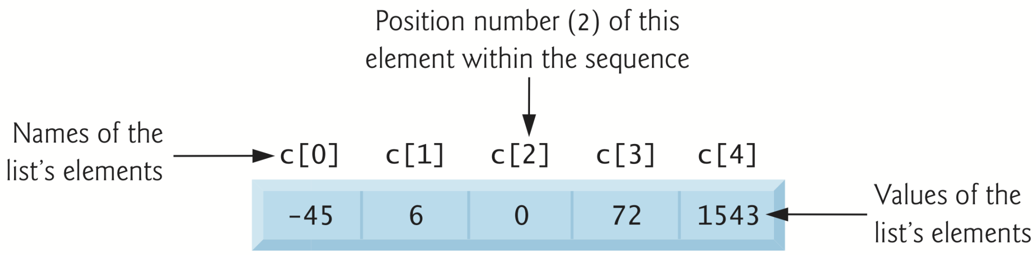
```
c
```

Out[2]:

```
[-45, 6, 0, 72, 1543]
```

Accessing Elements of a List

- Reference a list element by writing the list’s name followed by the element’s **index** enclosed in `[]` (the **subscription operator**).


Position number (2) of this element within the sequence
Names of the list’s elements
c[0]
c[1]
c[2]
c[3]
c[4]
-45
6
0
72
1543
Values of the list’s elements

In [3]:

```
c[0]
```

Out[3]:

```
-45
```

In [4]:

```
c[4]
```

Out[4]:

```
1543
```

Determining a List’s Length

In [5]:

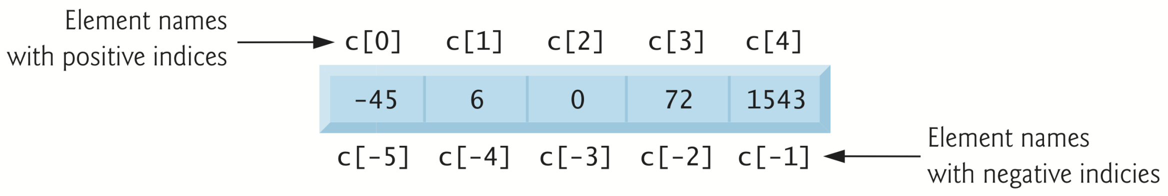
```
len(c)
```

Out[5]:

```
5
```

Accessing Elements from the End of the List with Negative Indices

- Lists can be accessed from the end by using *negative indices*:



In [6]:

```
c[-1]
```

Out[6]:

```
1543
```

In [7]:

```
c[-5]
```

Out[7]:

```
-45
```

Indices Must Be Integers or Integer Expressions

In [8]:

```
a = 1
```

In [9]:

```
b = 2
```

In [10]:

```
c[a + b]
```

Out[10]:

```
72
```

Lists Are Mutable

In [11]:

```
c[4] = 17
```

In [12]:

```
c
```

Out[12]:

```
[-45, 6, 0, 72, 17]
```

Some Sequences Are Immutable

- Python’s string and tuple sequences are immutable.

In [13]:

```
s = 'hello'
```

In [14]:

```
s[0]
```

Out[14]:

```
'h'
```

In [15]:

```
s[0] = 'H'
```

```
---------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-15-d6f3b193531a> in <module>
----> 1 s[0] = 'H'

TypeError: 'str' object does not support item assignment
```

Attempting to Access a Nonexistent Element

- Index values must be in range.

In [16]:

```
c[100]
```

```
---------------------------------------------------------------------
----
IndexError                                Traceback (most recent call l
ast)
<ipython-input-16-78fb4de297cc> in <module>
----> 1 c[100]

IndexError: list index out of range
```

Using List Elements in Expressions

In [17]:

```
c[0] + c[1] + c[2]
```

Out[17]:

```
-39
```

Appending to a List with +=

- Lists can grow dynamically to accommodate new items.

In [18]:

```
a_list = []
```

In [19]:

```
for number in range(1, 6):
    a_list += [number]
```

In [20]:

```
a_list
```

Out[20]:

```
[1, 2, 3, 4, 5]
```

- When the left operand of `+=` is a list, the right operand must be an *iterable*; otherwise, a `TypeError` occurs.

In [21]:

```
letters = []
```

In [22]:

```
letters += 'Python'
```

In [23]:

```
letters
```

Out[23]:

```
['P', 'y', 't', 'h', 'o', 'n']
```

Concatenating Lists with +

- Can **concatenate** two lists, two tuples or two strings using `+` to create a *new* sequence of the same type.

In [24]:

```
list1 = [10, 20, 30]
```

In [25]:

```
list2 = [40, 50]
```

In [26]:

```
concatenated_list = list1 + list2
```

In [27]:

```
concatenated_list
```

Out[27]:

```
[10, 20, 30, 40, 50]
```

Using `for` and `range` to Access List Indices and Values

In [28]:

```
for i in range(len(concatenated_list)):
    print(f'{i}: {concatenated_list[i]}')
```

```
0: 10
1: 20
2: 30
3: 40
4: 50
```

- We'll show a safer way to access element indices and values using built-in function `enumerate`.

Comparison Operators

- Can compare entire lists element-by-element.

In [29]:

```
a = [1, 2, 3]
```

In [30]:

```
b = [1, 2, 3]
```

In [31]:

```
c = [1, 2, 3, 4]
```

In [32]:

```
a == b
```

Out[32]:

```
True
```

In [33]:

```
a == c
```

Out[33]:

```
False
```

In [34]:

```
a < c
```

Out[34]:

```
True
```

In [35]:

```
c >= b
```

Out[35]:

```
True
```

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.3 Tuples

Creating Tuples

- To create an empty tuple, use empty parentheses.

In [1]:

```
student_tuple = ()
```

In [2]:

```
student_tuple
```

Out[2]:

```
()
```

In [3]:

```
len(student_tuple)
```

Out[3]:

```
0
```

- Pack a tuple by separating its values with commas.

In [4]:

```
student_tuple = 'John', 'Green', 3.3
```

In [5]:

```
student_tuple
```

Out[5]:

```
('John', 'Green', 3.3)
```

In [6]:

```
len(student_tuple)
```

Out[6]:

```
3
```

- When you output a tuple, Python always displays its contents in parentheses.
- Parentheses are optional when creating a tuple.

In [7]:

```
another_student_tuple = ('Mary', 'Red', 3.3)
```

In [8]:

```
another_student_tuple
```

Out[8]:

```
('Mary', 'Red', 3.3)
```

- A comma is required to create a one-element tuple.

In [9]:

```
a_singleton_tuple = ('red',)  # note the comma
```

In [10]:

```
a_singleton_tuple
```

Out[10]:

```
('red',)
```

Accessing Tuple Elements

- You generally access tuple elements directly rather than iterating over them.

In [11]:

```
time_tuple = (9, 16, 1)
```

In [12]:

```
time_tuple
```

Out[12]:

```
(9, 16, 1)
```

In [13]:

```
time_tuple[0] * 3600 + time_tuple[1] * 60 + time_tuple[2]
```

Out[13]:

```
33361
```

Adding Items to a String or Tuple

- `+=` can be used with strings and tuples, even though they’re *immutable*.
- Creates new objects.

In [14]:

```
tuple1 = (10, 20, 30)
```

In [15]:

```
tuple2 = tuple1
```

In [16]:

```
tuple2
```

Out[16]:

```
(10, 20, 30)
```

In [17]:

```
tuple1 += (40, 50)
```

In [18]:

```
tuple1
```

Out[18]:

```
(10, 20, 30, 40, 50)
```

In [19]:

```
tuple2
```

Out[19]:

```
(10, 20, 30)
```

Appending Tuples to Lists

In [20]:

```
numbers = [1, 2, 3, 4, 5]
```

In [21]:

```
numbers += (6, 7)
```

In [22]:

```
numbers
```

Out[22]:

```
[1, 2, 3, 4, 5, 6, 7]
```

Tuples May Contain Mutable Objects

In [23]:

```
student_tuple = ('Amanda', 'Blue', [98, 75, 87])
```

In [24]:

```
student_tuple[2][1] = 85
```

In [25]:

```
student_tuple
```

Out[25]:

```
('Amanda', 'Blue', [98, 85, 87])
```

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.4 Unpacking Sequences

- Can unpack any sequence’s elements by assigning the sequence to a comma-separated list of variables (of the appropriate length).

In [1]:

```
student_tuple = ('Amanda', [98, 85, 87])
```

In [2]:

```
first_name, grades = student_tuple
```

In [3]:

```
first_name
```

Out[3]:

```
'Amanda'
```

In [4]:

```
grades
```

Out[4]:

```
[98, 85, 87]
```

In [5]:

```
first, second = 'hi'
```

In [6]:

```
print(f'{first}  {second}')
```

```
h  i
```

In [7]:

```
number1, number2, number3 = [2, 3, 5]
```

In [8]:

```
print(f'{number1}  {number2}  {number3}')
```

```
2  3  5
```

In [9]:

```
number1, number2, number3 = range(10, 40, 10)
```

In [10]:

```
print(f'{number1}  {number2}  {number3}')
```

```
10  20  30
```

Swapping Values Via Packing and Unpacking

In [11]:

```
number1 = 99
```

In [12]:

```
number2 = 22
```

In [13]:

```
number1, number2 = (number2, number1)
```

In [14]:

```
print(f'number1 = {number1}; number2 = {number2}')
```

```
number1 = 22; number2 = 99
```

Accessing Indices and Values Safely with Built-in Function enumerate

- Preferred way to access an element's index *and* value is the built-in function **`enumerate`**.
- Receives an iterable and creates an iterator that, for each element, returns a tuple containing the element's index and value.
- Built-in function **`list`** creates a list from a sequence.

In [15]:

```
colors = ['red', 'orange', 'yellow']
```

In [16]:

```
list(enumerate(colors))
```

Out[16]:

```
[(0, 'red'), (1, 'orange'), (2, 'yellow')]
```

- Built-in function **`tuple`** creates a tuple from a sequence.

In [17]:

```
tuple(enumerate(colors))
```

Out[17]:

```
((0, 'red'), (1, 'orange'), (2, 'yellow'))
```

In [18]:

```
for index, value in enumerate(colors):
    print(f'{index}: {value}')
```

```
0: red
1: orange
2: yellow
```

Creating a Primitive Bar Chart

```
# fig05_01.py
"""Displaying a bar chart"""
numbers = [19, 3, 15, 7, 11]

print('\nCreating a bar chart from numbers:')
print(f'Index{"Value":>8}   Bar')

for index, value in enumerate(numbers):
    print(f'{index:>5}{value:>8}   {"*" * value}')
```

In [19]:

```
run fig05_01.py
```

```
Creating a bar chart from numbers:
Index   Value   Bar
    0      19   *******************
    1       3   ***
    2      15   ***************
    3       7   *******
    4      11   ***********
```

- The expression `python "*" * value` creates a string consisting of `value` asterisks.
- When used with a sequence, the multiplication operator (`*`) *repeats* the sequence.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.5 Sequence Slicing

- Can **slice** sequences to create new sequences of the same type containing *subsets* of the original elements.
- Slice operations that do *not* modify a sequence work identically for lists, tuples and strings.

Specifying a Slice with Starting and Ending Indices

In [1]:

```
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
```

In [2]:

```
numbers[2:6]
```

Out[2]:

```
[5, 7, 11, 13]
```

Specifying a Slice with Only an Ending Index

- Starting index `0` is assumed.

In [3]:

```
numbers[:6]
```

Out[3]:

```
[2, 3, 5, 7, 11, 13]
```

In [4]:

```
numbers[0:6]
```

Out[4]:

```
[2, 3, 5, 7, 11, 13]
```

Specifying a Slice with Only a Starting Index

- Assumes the sequence's length as the ending index.

In [5]:

```
numbers[6:]
```

Out[5]:

```
[17, 19]
```

In [6]:

```
numbers[6:len(numbers)]
```

Out[6]:

```
[17, 19]
```

Specifying a Slice with No Indices

In [7]:

```
numbers[:]
```

Out[7]:

```
[2, 3, 5, 7, 11, 13, 17, 19]
```

- Though slices create new objects, slices make **shallow copies** of the elements.
- In the snippet above, the new list's elements refer to the *same objects* as the original list's elements.

Slicing with Steps

In [8]:

```
numbers[::2]
```

Out[8]:

```
[2, 5, 11, 17]
```

Slicing with Negative Indices and Steps

In [9]:

```
numbers[::-1]
```

Out[9]:

```
[19, 17, 13, 11, 7, 5, 3, 2]
```

In [10]:

```
numbers[-1:-9:-1]
```

Out[10]:

```
[19, 17, 13, 11, 7, 5, 3, 2]
```

Modifying Lists Via Slices

- Can modify a list by assigning to a slice.

In [11]:

```
numbers[0:3] = ['two', 'three', 'five']
```

In [12]:

```
numbers
```

Out[12]:

```
['two', 'three', 'five', 7, 11, 13, 17, 19]
```

In [13]:

```
numbers[0:3] = []
```

In [14]:

```
numbers
```

Out[14]:

```
[7, 11, 13, 17, 19]
```

In [15]:

```
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
```

In [16]:

```
numbers[::2] = [100, 100, 100, 100]
```

In [17]:

```
numbers
```

Out[17]:

```
[100, 3, 100, 7, 100, 13, 100, 19]
```

In [18]:

```
id(numbers)
```

Out[18]:

```
4391419592
```

In [19]:

```
numbers[:] = []
```

In [20]:

```
numbers
```

Out[20]:

```
[]
```

In [21]:

```
id(numbers)
```

Out[21]:

```
4391419592
```

- Deleting `numbers`' contents is different from assigning `numbers` a *new* empty list `[]`.
- Identities are different, so they represent separate objects in memory.

In [22]:

```
numbers = []
```

In [23]:

```
numbers
```

Out[23]:

```
[]
```

In [24]:

```
id(numbers)
```

Out[24]:

```
4391542152
```

- When you assign a new object to a variable, the original object will be **garbage collected** if no other variables refer to it.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.6 del Statement

Deleting the Element at a Specific List Index

In [1]:

```
numbers = list(range(0, 10))
```

In [2]:

```
numbers
```

Out[2]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [3]:

```
del numbers[-1]
```

In [4]:

```
numbers
```

Out[4]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Deleting a Slice from a List

In [5]:

```
del numbers[0:2]
```

In [6]:

```
numbers
```

Out[6]:

```
[2, 3, 4, 5, 6, 7, 8]
```

In [7]:

```
del numbers[::2]
```

In [8]:

```
numbers
```

Out[8]:

```
[3, 5, 7]
```

Deleting a Slice Representing the Entire List

In [9]:

```
del numbers[:]
```

In [10]:

```
numbers
```

Out[10]:

```
[]
```

Deleting a Variable from the Current Session

In [11]:

```
del numbers
```

In [12]:

```
numbers
```

```
---------------------------------------------------------------------
----
NameError                                 Traceback (most recent call l
ast)
<ipython-input-12-6a54518a0c2c> in <module>
----> 1 numbers

NameError: name 'numbers' is not defined
```

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.7 Passing Lists to Functions

Passing an Entire List to a Function

In [1]:

```
def modify_elements(items):
    """"Multiplies all element values in items by 2."""
    for i in range(len(items)):
        items[i] *= 2
```

In [2]:

```
numbers = [10, 3, 7, 1, 9]
```

In [3]:

```
modify_elements(numbers)
```

In [4]:

```
numbers
```

Out[4]:

```
[20, 6, 14, 2, 18]
```

Passing a Tuple to a Function

- When you pass a tuple to a function, attempting to modify the tuple’s immutable elements results in a `TypeError`.

In [5]:

```
numbers_tuple = (10, 20, 30)
```

In [6]:

```
numbers_tuple
```

Out[6]:

```
(10, 20, 30)
```

```
In [7]:
```

```
modify_elements(numbers_tuple)
```

```
--------------------------------------------------------------------
----
TypeError                              Traceback (most recent call l
ast)
<ipython-input-7-9339741cd595> in <module>
----> 1 modify_elements(numbers_tuple)

<ipython-input-1-aa30cff7ee99> in modify_elements(items)
      2     """"Multiplies all element values in items by 2."""
      3     for i in range(len(items)):
----> 4         items[i] *= 2

TypeError: 'tuple' object does not support item assignment
```

A Note Regarding Tracebacks

- A traceback shows code that led to an exception.
- When an exception occurs in a single-line snippet, it's always preceded by `---->  1` , indicating that line 1 (the snippet's only line) caused the exception.
- Multiline snippets like a function definition show consecutive line numbers starting at 1.
- The last line of code shown with `---->` caused the exception.

This content is based on Chapter 5 of the book **Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud** (https://amzn.to/2VvdnxE).

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.8 Sorting Lists

Sorting a List in Ascending Order

- List method `sort` *modifies* a list.

In [1]:

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

In [2]:

```
numbers.sort()
```

In [3]:

```
numbers
```

Out[3]:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sorting a List in Descending Order

In [4]:

```
numbers.sort(reverse=True)
```

In [5]:

```
numbers
```

Out[5]:

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

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*.

In [6]:

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

In [7]:

```
ascending_numbers = sorted(numbers)
```

In [8]:

```
ascending_numbers
```

Out[8]:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [9]:

```
numbers
```

Out[9]:

```
[10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

In [10]:

```
letters = 'fadgchjebi'
```

In [11]:

```
ascending_letters = sorted(letters)
```

In [12]:

```
ascending_letters
```

Out[12]:

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

In [13]:

```
letters
```

Out[13]:

```
'fadgchjebi'
```

In [14]:

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

In [15]:

```
ascending_colors = sorted(colors)
```

In [16]:

```
ascending_colors
```

Out[16]:

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

In [17]:

```
colors
```

Out[17]:

```
('red', 'orange', 'yellow', 'green', 'blue')
```

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.9 Searching Sequences

- **Searching** is the process of locating a particular **key** value.

List Method index

- Searches through a list from index 0 and returns the index of the *first* element that matches the search key.
- `ValueError` if the value is not in the list.

In [1]:

```
numbers = [3, 7, 1, 4, 2, 8, 5, 6]
```

In [2]:

```
numbers.index(5)
```

Out[2]:

```
6
```

Specifying the Starting Index of a Search

In [3]:

```
numbers *= 2
```

In [4]:

```
numbers
```

Out[4]:

```
[3, 7, 1, 4, 2, 8, 5, 6, 3, 7, 1, 4, 2, 8, 5, 6]
```

In [5]:

```
numbers.index(5, 7)
```

Out[5]:

```
14
```

Specifying the Starting and Ending Indices of a Search

- Look for the value `7` in the range of elements with indices `0` through `3`.

In [6]:

```
numbers.index(7, 0, 4)
```

Out[6]:

```
1
```

Operators `in` and `not in`

- Operator `in` tests whether its right operand's iterable contains the left operand's value.

In [7]:

```
1000 in numbers
```

Out[7]:

```
False
```

In [8]:

```
5 in numbers
```

Out[8]:

```
True
```

- Operator `not in` tests whether its right operand's iterable does *not* contain the left operand's value.

In [9]:

```
1000 not in numbers
```

Out[9]:

```
True
```

In [10]:

```
5 not in numbers
```

Out[10]:

```
False
```

Using Operator `in` to Prevent a `ValueError`

In [11]:

```
key = 1000
```

In [12]:

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

```
1000 not found
```

Built-In Functions `any` and `all`

- Built-in function **`any`** returns `True` if any item in its iterable argument is `True`.
- Built-in function **`all`** returns `True` if all items in its iterable argument are `True`.
- Nonzero values are `True` and 0 is `False`.
- Non-empty iterable objects also evaluate to `True`, whereas any empty iterable evaluates to `False`.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.10 Other List Methods

In [1]:

```
color_names = ['orange', 'yellow', 'green']
```

Inserting an Element at a Specific List Index

In [2]:

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

In [3]:

```
color_names
```

Out[3]:

```
['red', 'orange', 'yellow', 'green']
```

Adding an Element to the End of a List

In [4]:

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

In [5]:

```
color_names
```

Out[5]:

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

Adding All the Elements of a Sequence to the End of a List

- Equivalent to `+=` .

In [6]:

```
color_names.extend(['indigo', 'violet'])
```

In [7]:

```
color_names
```

Out[7]:

```
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

In [8]:

```
sample_list = []
```

In [9]:

```
s = 'abc'
```

In [10]:

```
sample_list.extend(s)
```

In [11]:

```
sample_list
```

Out[11]:

```
['a', 'b', 'c']
```

In [12]:

```
t = (1, 2, 3)
```

In [13]:

```
sample_list.extend(t)
```

In [14]:

```
sample_list
```

Out[14]:

```
['a', 'b', 'c', 1, 2, 3]
```

- Parentheses are required for the tuple argument below, because `extend` expects **one iterable argument**.

In [15]:

```
sample_list.extend((4, 5, 6))  # note the extra parentheses
```

In [16]:

```
sample_list
```

Out[16]:

```
['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

Removing the First Occurrence of an Element in a List

- `ValueError` occurs if `remove`'s argument is not in the list.

In [17]:

```
color_names.remove('green')
```

In [18]:

```
color_names
```

Out[18]:

```
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet']
```

Emptying a List

In [19]:

```
color_names.clear()
```

In [20]:

```
color_names
```

Out[20]:

```
[]
```

Counting the Number of Occurrences of an Item

In [21]:

```
responses = [1, 2, 5, 4, 3, 5, 2, 1, 3, 3,
             1, 4, 3, 3, 3, 2, 3, 3, 2, 2]
```

In [22]:

```
for i in range(1, 6):
    print(f'{i} appears {responses.count(i)} times in responses')
```

```
1 appears 3 times in responses
2 appears 5 times in responses
3 appears 8 times in responses
4 appears 2 times in responses
5 appears 2 times in responses
```

Reversing a List’s Elements

- Method **`reverse`** reverses the contents of a list in place.

In [23]:

```
color_names = ['red', 'orange', 'yellow', 'green', 'blue']
```

In [24]:

```
color_names.reverse()
```

In [25]:

```
color_names
```

Out[25]:

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

Copying a List

- Method `copy` returns a *new* list containing a *shallow* copy.

In [26]:

```
copied_list = color_names.copy()
```

In [27]:

```
copied_list
```

Out[27]:

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

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.11 Simulating Stacks with Lists

- Python does not have a built-in stack type.
- Can think of a stack as a constrained list.
- *Push* using list method `append` .
- *Pop* using list method **`pop`** with no arguments to get items in last-in, first-out (LIFO) order.

In [1]:

```
stack = []
```

In [2]:

```
stack.append('red')
```

In [3]:

```
stack
```

Out[3]:

```
['red']
```

In [4]:

```
stack.append('green')
```

In [5]:

```
stack
```

Out[5]:

```
['red', 'green']
```

In [6]:

```
stack.pop()
```

Out[6]:

```
'green'
```

In [7]:

```
stack
```

Out[7]:

```
['red']
```

In [8]:

```
stack.pop()
```

Out[8]:

```
'red'
```

In [9]:

```
stack
```

Out[9]:

```
[]
```

In [10]:

```
stack.pop()
```

```
--------------------------------------------------------------------
----
IndexError                                Traceback (most recent call l
ast)
<ipython-input-10-415460d3b717> in <module>
----> 1 stack.pop()

IndexError: pop from empty list
```

- Also can use a list to simulate a **queue**.
- Items are retrieved from queues in **first-in, first-out (FIFO) order**.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.12 List Comprehensions

- Concise way to create new lists.
- Replaces using `for` to iterate over a sequence and create a list.

In [1]:

```
list1 = []
```

In [2]:

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

In [3]:

```
list1
```

Out[3]:

```
[1, 2, 3, 4, 5]
```

Using a List Comprehension to Create a List of Integers

In [4]:

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

In [5]:

```
list2
```

Out[5]:

```
[1, 2, 3, 4, 5]
```

- **`for` clause** iterates over the sequence produced by `range(1, 6)`.
- For each `item`, the list comprehension evaluates the expression to the left of the `for` clause and places the expression's value in the new list.

Mapping: Performing Operations in a List Comprehension's Expression

- Mapping is a common functional-style programming operation that produces a result with the *same* number of elements as the original data being mapped.

In [6]:

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

In [7]:

```
list3
```

Out[7]:

```
[1, 8, 27, 64, 125]
```

Filtering: List Comprehensions with `if` Clauses

- Another common functional-style programming operation is **filtering** elements to select only those that match a condition.
- Typically produces a list with *fewer* elements than the data being filtered.

In [8]:

```
list4 = [item for item in range(1, 11) if item % 2 == 0]
```

In [9]:

```
list4
```

Out[9]:

```
[2, 4, 6, 8, 10]
```

List Comprehension That Processes Another List’s Elements

- The `for` clause can process any iterable.

In [10]:

```
colors = ['red', 'orange', 'yellow', 'green', 'blue']
```

In [11]:

```
colors2 = [item.upper() for item in colors]
```

In [12]:

```
colors2
```

Out[12]:

```
['RED', 'ORANGE', 'YELLOW', 'GREEN', 'BLUE']
```

In [13]:

```
colors
```

Out[13]:

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

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.13 Generator Expressions

- Like list comprehensions, but create iterable **generator objects** that produce values **on demand**.
- Known as **lazy evaluation**.
- For large numbers of items, creating lists can take substantial memory and time.
- **Generator expressions** can reduce memory consumption and improve performance if the whole list is not needed at once.

In [1]:

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

In [2]:

```
for value in (x ** 2 for x in numbers if x % 2 != 0):
    print(value, end='  ')
```

```
9  49  1  81  25
```

In [3]:

```
squares_of_odds = (x ** 2 for x in numbers if x % 2 != 0)
```

In [4]:

```
squares_of_odds
```

Out[4]:

```
<generator object <genexpr> at 0x110430408>
```

- Output indicates that `square_of_odds` is a **generator object** that was created from a **generator expression (`<genexpr>`).**
- **Built-in function `next`** receives a generator or iterator and returns the next item.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.14 Filter, Map and Reduce

- Built-in `filter` and `map` functions also perform filtering and mapping.

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))
```

Out[3]:

```
[3, 7, 1, 9, 5]
```

- Functions are objects that you can assign to variables, pass to other functions and return from functions.
- Functions that receive other functions as arguments are a functional-style capability called **higher-order functions**.
- `filter`'s first argument must be a function that receives one argument and returns `True` if the value should be included in the result.
- Higher-order functions may also return a function as a result.
- `filter` returns an iterator, so `filter`'s results are not produced until you iterate through them—lazy evaluation.

In [4]:

```
[item for item in numbers if is_odd(item)]
```

Out[4]:

```
[3, 7, 1, 9, 5]
```

Using a `lambda` Rather than a Function

- For simple functions like `is_odd` that `return` only a *single expression's value*, you can use a **lambda expression** (or simply a **lambda**) to define the function inline.

In [5]:

```
list(filter(lambda x: x % 2 != 0, numbers))
```

Out[5]:

```
[3, 7, 1, 9, 5]
```

- A lambda expression is an _anonymous function
- Begins with the **`lambda`** keyword followed by a comma-separated parameter list, a colon (**`:`**) and an expression.
- A `lambda` *implicitly* returns its expression’s value.

Mapping a Sequence’s Values to New Values

In [6]:

```
numbers
```

Out[6]:

```
[10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

In [7]:

```
list(map(lambda x: x ** 2, numbers))
```

Out[7]:

```
[100, 9, 49, 1, 81, 16, 4, 64, 25, 36]
```

- Function `map` ’s first argument is a function that receives one value and returns a new value.
- equivalent list comprehension:

In [8]:

```
[item ** 2 for item in numbers]
```

Out[8]:

```
[100, 9, 49, 1, 81, 16, 4, 64, 25, 36]
```

Combining `filter` and `map`

In [9]:

```
list(map(lambda x: x ** 2,
         filter(lambda x: x % 2 != 0, numbers)))
```

Out[9]:

```
[9, 49, 1, 81, 25]
```

- Equivalent list comprehension:

In [10]:

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

Out[10]:

```
[9, 49, 1, 81, 25]
```

Reduction: Totaling the Elements of a Sequence with `sum`

- Reductions process a sequence’s elements into a single value.
    - E.g., `len`, `sum`, `min` and `max`.
- Can create custom reductions using the `functools` module (https://docs.python.org/3/library/functools.html)’s `reduce` function.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.15 Other Sequence Processing Functions

Finding the Minimum and Maximum Values Using a Key Function

In [1]:

```
'Red' < 'orange'
```

Out[1]:

```
True
```

- `'R'` “comes after” `'o'` in the alphabet, so you might expect `'Red'` to be less than `'orange'` and the condition above to be `False`.
- Strings are compared by their characters’ underlying *numerical values*, and lowercase letters have *higher* numerical values than uppercase letters.
- Confirm with built-in function **`ord`**:

In [2]:

```
ord('R')
```

Out[2]:

```
82
```

In [3]:

```
ord('o')
```

Out[3]:

```
111
```

In [4]:

```
colors = ['Red', 'orange', 'Yellow', 'green', 'Blue']
```

- Assume that we’d like to determine the minimum and maximum strings using *alphabetical* order.
- Can specify sort order with the `key` argument.

In [5]:

```
min(colors, key=lambda s: s.lower())
```

Out[5]:

```
'Blue'
```

In [6]:

```
max(colors, key=lambda s: s.lower())
```

Out[6]:

```
'Yellow'
```

Iterating Backwards Through a Sequence

In [7]:

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

In [8]:

```
reversed_numbers = [item ** 2 for item in reversed(numbers)]
```

In [9]:

```
reversed_numbers
```

Out[9]:

```
[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.
- Receives any number of iterables and returns an iterator that produces tuples containing the elements at the same index in each.

In [10]:

```
names = ['Bob', 'Sue', 'Amanda']
```

In [11]:

```
grade_point_averages = [3.5, 4.0, 3.75]
```

In [12]:

```
for name, gpa in zip(names, grade_point_averages):
    print(f'Name={name}; GPA={gpa}')
```

```
Name=Bob; GPA=3.5
Name=Sue; GPA=4.0
Name=Amanda; GPA=3.75
```

- Shortest argument determines the number of tuples produced.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

5.16 Two-Dimensional Lists

- Lists can contain other lists as elements.
- Typical use is to represent **tables** of values consisting of information arranged in **rows** and **columns**.
- To identify a particular table element, we specify *two* indices—the first identifies the element's row, the second the element's column.

Creating a Two-Dimensional List

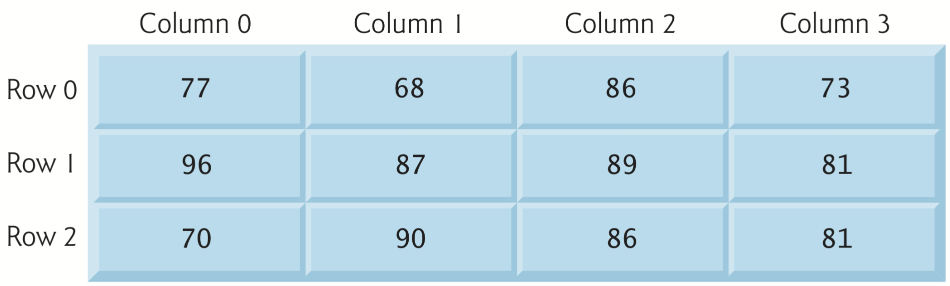
In [1]:

```
a = [[77, 68, 86, 73], [96, 87, 89, 81], [70, 90, 86, 81]]
```

Writing the list as follows makes its row and column tabular structure clearer:

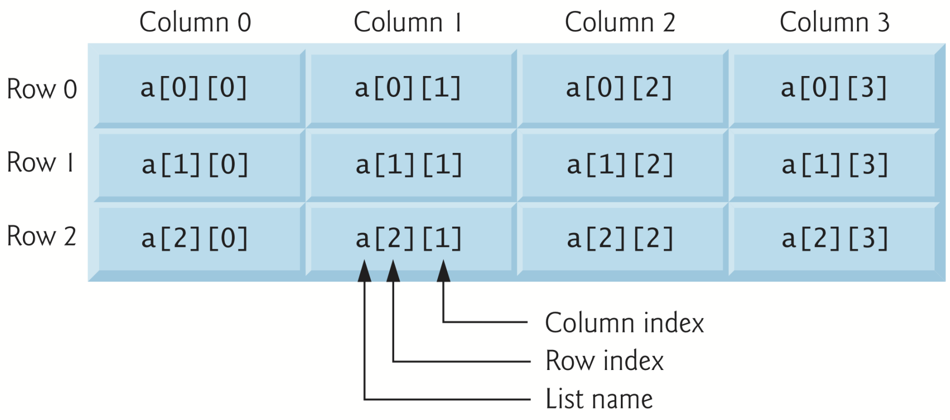
```
a = [[77, 68, 86, 73],  # first student's grades
     [96, 87, 89, 81],  # second student's grades
     [70, 90, 86, 81]]  # third student's grades
```

Illustrating a Two-Dimensional List



	Column 0	Column 1	Column 2	Column 3
Row 0	77	68	86	73
Row 1	96	87	89	81
Row 2	70	90	86	81

Identifying the Elements in a Two-Dimensional List



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Output the rows of the preceding two-dimensional list.

In [2]:

```
for row in a:
    for item in row:
        print(item, end=' ')
    print()
```

```
77 68 86 73
96 87 89 81
70 90 86 81
```

How the Nested Loops Execute

In [3]:

```
for i, row in enumerate(a):
    for j, item in enumerate(row):
        print(f'a[{i}][{j}]={item} ', end=' ')
    print()
```

```
a[0][0]=77  a[0][1]=68  a[0][2]=86  a[0][3]=73
a[1][0]=96  a[1][1]=87  a[1][2]=89  a[1][3]=81
a[2][0]=70  a[2][1]=90  a[2][2]=86  a[2][3]=81
```

- Outer `for` statement iterates over the list’s ows one row at a time.
- During each iteration of the outer `for` statement, the inner `for` statement iterates over *each* column in the current row.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.