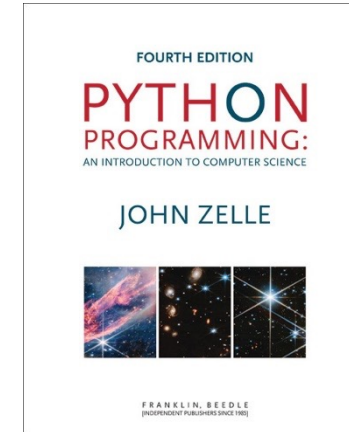


Python Programming: An Introduction To Computer Science



Chapter 9 Data Collections



Objectives

- To understand the use of lists (arrays) to represent a sequential collection of data.
- To be familiar with the functions and methods available for manipulating Python lists.
- To understand the use of tuples for grouping a set of related values.
- To be familiar with Python dictionaries as a data structure for storing non-sequential collections.



Objectives

- To be able to write programs that use lists and tuples to structure and manipulate collections of information.



Example Problem: Simple Statistics

- Many programs deal with large collections of similar information.
 - Words in a document
 - Students in a course
 - Data from an experiment
 - Customers of a business
 - Graphics objects drawn on the screen
 - Cards in a deck



Example Problem: Simple Statistics

Let's review some code we wrote in chapter 7:

```
# average4.py
#     A program to average a set of numbers
#     Illustrates sentinel loop using empty string as sentinel

def main():
    sum = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = float(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("\nThe average of the numbers is", sum / count)
```



Example Problem: Simple Statistics

- This program allows the user to enter a sequence of numbers, but the program itself doesn't keep track of the numbers that were entered – it only keeps a running total.
- Suppose we want to extend the program to compute not only the mean, but also the median and standard deviation.



Example Problem: Simple Statistics

- The *median* is the data value that splits the data into equal-sized parts.
- For the data [2, 4, 6, 9, 13], the median is 6, since there are two values greater than 6 and two values that are smaller.
- One way to determine the median is to store all the numbers, sort them, and identify the middle value.



Example Problem: Simple Statistics

- The *standard deviation* is a measure of how spread out the data is relative to the mean.
- If the data is tightly clustered around the mean, then the standard deviation is small. If the data is more spread out, the standard deviation is larger.
- The standard deviation is a yardstick to measure/express how exceptional a value is.



Example Problem: Simple Statistics

- The standard deviation is

$$s = \sqrt{\frac{\sum(\bar{x} - x_i)^2}{n - 1}}$$

- Here \bar{x} is the mean, x_i represents the i^{th} data value and n is the number of data values.
- The expression $(\bar{x} - x_i)^2$ is the square of the “deviation” of an individual item from the mean.



Example Problem: Simple Statistics

- The numerator is the sum of these squared “deviations” across all the data.
- Suppose our data was [2, 4, 6, 9, 13].
 - The mean (\bar{x}) is 6.8
 - The numerator of the standard deviation is

$$(6.8 - 2)^2 + (6.8 - 4)^2 + (6.8 - 6)^2 + (6.8 - 9)^2 + (6.8 - 13)^2 = 74.8$$

$$s = \sqrt{\frac{74.8}{5 - 1}} = \sqrt{18.7} = 4.32$$



Example Problem: Simple Statistics

- As you can see, calculating the standard deviation not only requires the mean (which can't be calculated until all the data is entered), but also each individual data element!
- We need some way to remember these values as they are entered



Python Lists

- We need a way to store and manipulate an entire collection of numbers.
- We can't just use a bunch of variables, because we don't know many numbers there will be.
- What do we need? Some way of combining an entire collection of values into one object.
- We've already done something like this before...



Python Lists

```
>>> list(range(10))  
      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
>>> "This is an ex-parrot!".split()  
      ['This', 'is', 'an', 'ex-parrot!']
```

- Both of these familiar functions return a collection of values denoted by the enclosing square brackets.
- Lists are the most common way of handling collections of data in a Python program.



Lists and Arrays as Sequences

- Python lists are ordered sequences of items. For instance, a sequence of n numbers might be called S :
$$S = s_0, s_1, s_2, s_3, \dots, s_{n-1}$$
- Specific values in the sequence can be referenced using *subscripts*, e.g. the first item is denoted with the subscript 0 (s_0)
- By using numbers as subscripts, mathematicians can succinctly summarize computations over items in a sequence using subscript variables.

$$\sum_{i=0}^{n-1} s_i$$



Lists and Arrays as Sequences

- Suppose the sequence is stored in a variable `s`. We could write a loop to calculate the sum of the items in the sequence like this:

```
sum = 0
for i in range(n):
    sum = sum + s[i]
```

- Almost all computer languages have a sequence structure like this, sometimes called an *array*.



Lists and Arrays as Sequences

- A list or array is a sequence of items where the entire sequence is referred to by a single name (i.e. `s`) and individual items can be selected by indexing (i.e. `s[i]`).
- In other programming languages, arrays are generally a fixed size, meaning that when you create the array, you have to specify how many items it can hold.
- Arrays are generally also *homogeneous*, meaning they can hold only one data type.



Lists and Arrays as Sequences

- Python lists are dynamic. They can grow and shrink on demand.
- Python lists are also *heterogeneous*, a single list can hold arbitrary data types.
- Python lists are mutable sequences of arbitrary objects.



Lists Operations

Operator	Meaning
<code><seq> + <seq></code>	Concatenation
<code><seq> * <int-expr></code>	Repetition
<code><seq>[]</code>	Indexing
<code>len(<seq>)</code>	Length
<code><seq>[:]</code>	Slicing
<code>for <var> in <seq>:</code>	Iteration
<code><expr> in <seq></code>	Membership (Boolean)



Lists Operations

- Except for the membership check, we've used these operations before on strings.
- The membership operation can be used to see if a certain value appears anywhere in a sequence.

```
>>> lst = [1,2,3,4]
>>> 3 in lst
True
```



Lists Operations

- The summing example from earlier can be written like this:

```
sum = 0
for x in s:
    sum = sum + x
```

- Unlike strings, lists are mutable:

```
>>> lst = [1, 2, 3, 4]
>>> lst[3]
4
>>> lst[3] = "Hello"
>>> lst
[1, 2, 3, 'Hello']
>>> lst[2] = 7
>>> lst
[1, 2, 7, 'Hello']
```



Lists Operations

- Lists can be created by listing items inside square brackets.

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

```
food = ["spam", "eggs", "back bacon"]
```

```
silly = [1, "spam", 4, "U"]
```

```
empty = []
```

- A list of identical items can be created using the repetition operator. This command produces a list containing 50 zeroes:

```
zeroes = [0] * 50
```



Lists Operations

```
# month2.py
# A program to print the month abbreviation, given its number.

def main():
    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    n = int(input("Enter a month number (1-12): "))

    print(f"The month abbreviation is {months[n-1]}.")
```



Lists Operations

- In this program there is a list of strings called `months` to use as the lookup table.
- This line of code is split over two lines – Python knows the list isn't finished until the `"]"` is encountered. This makes the code more readable.
- Lists, like strings, are indexed beginning with 0. `months[0]` is `"Jan"`. The `n`th month is at position `n-1`.



Lists Operations

- It would be trivial to modify this program to print out the entire month name. Just change the lookup list!

```
months = ["January", "February", "March", "April",  
          "May", "June", "July", "August",  
          "September", "October", "November", "December"]
```




Lists Methods

Method	Meaning
<code><list>.append(x)</code>	Add element x to end of list.
<code><list>.sort()</code>	Sort (order) the list. A comparison function may be passed as a parameter.
<code><list>.reverse()</code>	Reverse the list.
<code><list>.index(x)</code>	Returns index of first occurrence of x.
<code><list>.insert(i, x)</code>	Insert x into list at index i.
<code><list>.count(x)</code>	Returns the number of occurrences of x in list.
<code><list>.remove(x)</code>	Deletes the first occurrence of x in list.
<code><list>.pop(i)</code>	Deletes the i^{th} element of the list and returns its value.



Lists Methods

```
>>> lst = []
>>> lst.append("lists")
>>> lst
['lists']
>>> lst.append("are")
>>> lst.append("fun")
>>> lst
['lists', 'are', 'fun']
>>> lst.sort()
>>> lst
['are', 'fun', 'lists']
```

```
>>> lst
['lists', 'fun', 'are']
>>> lst.index("fun")
1
>>> lst.insert(0, "fun")
>>> lst
['fun', 'lists', 'fun', 'are']
>>> lst.count("fun")
2
>>> lst.remove("fun")
>>> lst
['lists', 'fun', 'are']
```



Lists Methods

- Most list methods either modify the list (e.g. `append`, `sort`, `remove`, `extend`) or leave the list unchanged and return a value (e.g. `count` and `index`).
 - However, the `pop` method actually does both!
- When you want to remove a specific valued item from a list, `remove` does the job ,whereas `pop` removes the item from a given position.
- Calling `pop` without a parameter (e.g. `lst.pop()`) will always remove the last item from the list.



Lists Methods

- Using `append` is the most common and efficient way of adding an item to an existing list.
- It is often used to accumulate a list one item at a time.



Lists Methods

- Here is a fragment of code using a sentinel loop to build a list of positive numbers typed by the user:

```
nums = []  
x = float(input('Enter a number: '))  
while x >= 0:  
    nums.append(x)  
    x = float(input('Enter a number: '))
```



Lists Methods

- Basic list principles
 - A list is a sequence of items stored as a single object.
 - Items in a list can be accessed by indexing, and sublists can be accessed by slicing.
 - Lists are mutable; individual items or entire slices can be replaced through assignment statements.
 - Lists support a number of convenient and frequently used methods.
 - Lists will grow and shrink as needed.



Statistics with Lists

- One way we can solve our statistics problem is to store the data in a list.
- We could then write a series of functions that take a list of numbers and calculates the mean, standard deviation, and median.
- Let's rewrite our earlier program to use lists to find the mean.



Pythonic List Manipulation

- Python provides *list comprehensions* as a simple, direct way of creating lists.
- Suppose instead of using the sentinel loop in `getNumbers`, we would like to get all of the numbers in a single line of input, similar to the decoder program in Chapter 8.



Pythonic List Manipulation

- [`<expr>` for `<variable>` in `<sequence>`]
- Semantically, this creates a new list, with items formed by evaluating the expression for each value of the variable as it iterates over the sequence.
- List comprehensions are handy for building lists out of other sequences and using them produces more concise, readable, and efficient solution than writing the equivalent accumulator loop.



Pythonic List Manipulation

- Another trick: make use of functions that take a list as an input parameter.
 - E.g. to find the maximum value in a list of numbers:
`maximum = max(nums)`
 - There are also built in functions for minimum (`min`) and `sum`.



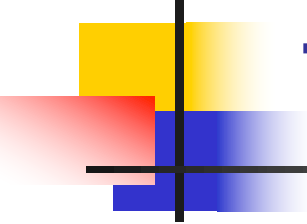
Pythonic List Manipulation

- There is one more twist on list comprehensions
 - You can filter items in the list with an if-clause
 - `[<expression> for <variable> in <sequence> if <condition>]`
- To see how this can be useful, let's extend our example with one more function.
- Extreme values are called "outliers", and sometimes we want to identify those values. One measure that's sometimes used is that any value more than 3 standard deviations from the mean is considered an outlier.



Other Data Structures

- Python lists allow us to store a collection of data as a sequence of items.
- In computer science, a way of organizing and storing data is called a *data structure*.
- Selecting or designing an appropriate data structure is often a crucial step in solving real-world computing problems.



Tuples

- A tuple looks like a list except it is enclosed in parentheses `()` instead of square brackets `[]`.
- A tuple is another sort of sequence, which means it is indexable and sliceable.
- Tuples are *immutable* – the items can't be changed.
- If the contents of a sequence won't change after it's created, using a tuple is more efficient than using a list.



Tuples

```
>>> bp = (120, 80)
>>> type(bp)
<class 'tuple'>
>>> bp[0]
120
>>> bp[1]
80
>>> systolic, diastolic = bp
>>> systolic
120
>>> diastolic
80
```



Tuples

- Did you notice the simultaneous assignment?
- Another example:

```
>>> pair = 3, 4
```

```
>>> pair
```

```
(3, 4)
```

```
>>> x, y = pair
```

```
>>> x
```

```
3
```

```
>>> y
```

```
4
```



Dictionaries

- While dictionaries aren't covered in this book, we briefly discuss them here since they show up so frequently "in the wild".
- Dictionaries store collections.
 - Lists allow us to store and retrieve items from sequential collections. We do lookups by its index, or position.
 - A mapping is collection that allows us to look up information based on arbitrary keys.



Dictionaries

- When would this be useful?
 - Looking up data based on student ID numbers
 - Locate someone based on phone number
 - Get a list of users based on zip code
- In programming terms, these are examples of *key-value* pairs. We access the *value* (student information) based on some *key* (their ID number).



Tuples

- Dictionaries are created by listing key-value pairs inside of curly braces. Keys and values are joined with a ":" and commas separate pairs.
- `passwd = {"guido": "superprogrammer",
"turing": "genius", "bill": "monopoly"}`

```
>>> passwd["guido"]  
      'superprogrammer'
```



Dictionaries

- In general, `<dictionary>[<key>]` returns the object associated with the given key.

- Dictionaries are mutable.

```
>>> passwd["bill"] = "bluescreen"
```

```
>>> passwd
```

```
    {'turing': 'genius', 'guido': 'superprogrammer',  
    'bill': 'bluescreen'}
```

- Notice that the dictionary prints out in a different order than it was created. Mappings are unordered.