M30299 – Programming
Lecture 15 – Using Lists

Matthew Poole & Nadim Bakhshov
moodle.port.ac.uk

School of Computing
University of Portsmouth

2020/21

Introduction to lecture

- Sometimes programs need to process large **collections** of data of the same type; for example:
  - marks from a student's modules;
  - temperatures for each day in a year; and
  - words in a document.
- In this lecture, we'll study **lists**, which hold collections of data.
- (We have already used simple examples of lists of integers:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

as the basis of `for` loops.)

Lists

- For some other languages (e.g. Java), we use the term "array" instead of "list" (some details of lists and arrays are different).
- Let's begin by defining some lists; below we create a list of ints and a list of strings:

```
>>> fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> seasons = ["spring", "summer", "autumn", "winter"]
>>> type(fibonacci)
<class 'list'>
>>> type(seasons)
<class 'list'>
```

List literals and list indexing

- The most basic list operation is **indexing**. Lists are indexed in the same way as strings, starting with position 0.
- Lists can also be indexed from the end using negative indices.

```
>>> seasons[0]
'spring'
>>> fibonacci[8]
21
>>> seasons[-1]
'winter'
>>> fibonacci[-2]
21
```

A simple application of a list

- Recall the `daysInMonth` function from worksheet P6.
- This function took a month and a year (both ints) as parameters, and returned the number of days in that month.
- The original function included a nested `if`-statement and a long Boolean expression. A possibly simpler solution is to use a list:

```
def daysInMonth(month, year):
    numDays = [31,28,31,30,31,30,31,31,30,31,30,31]
    if month == 2 and isLeapYear(year):
        return 29
    else:
        return numDays[month - 1]
```

Basic list operations

- Like strings, lists have operators for concatenation (+) and repetition (*):

```
>>> fibonacci + [55, 89]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> [10, 4] * 3
[10, 4, 10, 4, 10, 4]
```

- We can find the length of a list with the built-in function `len`:

```
>>> len(seasons)
4
```

List slicing

- Again like strings, we can do list **slicing** to get a sub-list from a list:

```
>>> fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> fibonacci[3:7]
[2, 3, 5, 8]
>>> seasons = ["spring", "summer", "autumn", "winter"]
>>> seasons[1:-1]
['summer', 'autumn']
>>> seasons[2:]
['autumn', 'winter']
>>> seasons[:-2]
['spring', 'summer']
```

Iteration though a list

- We can an **iterate** or **loop** through the elements of any list using a `for` loop:

```
>>> for season in seasons:
        print(season, end=" ")
spring summer autumn winter
```

```
>>> for i in fibonacci:
        print(i, end=" ")
0 1 1 2 3 5 8 13 21 34
```

Iteration though a list

- Sometimes it is useful to iterate through the **indices** of a list.
- To do this, we combine `range` with `len`:

```
>>> for i in range(len(fibonacci)):
        print("Fibonacci no.", i+1, "=", fibonacci[i])
Fibonacci no. 1 = 0
Fibonacci no. 2 = 1
Fibonacci no. 3 = 1
Fibonacci no. 4 = 2
   ...
Fibonacci no. 10 = 34
```

Membership checking

- We check whether a value appears in a list using the `in` operator:

```
>>> "winter" in seasons
True
>>> "december" in seasons
False
>>> 4 in fibonacci
False
```

Membership checking

- As an example application of membership checking, the following function uses a list to help with validation of user input:

```
def getSeason():
    seasons = ["spring", "summer", "autumn", "winter"]
    while True:
        season = input("Enter a season: ")
        if season in seasons:
            break
        print("Invalid season!")
    return season
```

Changing an element of a list

- An important difference between strings and lists is that lists are **mutable**; that is, we can **alter** their elements.
- We change list elements using assignment statements with list indexing:

```
>>> shopping = ["jam", "eggs", "margarine", "sugar"]
>>> shopping[2] = "butter"
>>> shopping
['jam', 'eggs', 'butter', 'sugar']
>>> shopping[-1] = "flour"
>>> shopping
['jam', 'eggs', 'butter', 'flour']
```

List methods

- There are many other operations on lists which take the form of **methods**. Let's have a look at some examples:
- The `append` method adds a value to the end of a list:

```
>>> shopping.append("eggs")
>>> shopping
['jam', 'eggs', 'butter', 'flour', 'eggs']
```

- The `remove` method removes the first occurrence of an element:

```
>>> shopping.remove("eggs")
>>> shopping
['jam', 'butter', 'flour', 'eggs']
```

List methods

- The `index` method returns the position of the first occurrence of a value:

```
>>> shopping.index("flour")
2
```

- The `sort` method sorts a list into order:

```
>>> shopping.sort()
>>> shopping
['butter', 'eggs', 'flour', 'jam']
```

Example: creating a simple dictionary

- As a simple example application of some of the concepts seen so far, consider the following programming problem:

  Write a function `makeDictionary` that allows the user to create a simple dictionary (an ordered word list): she should be able to enter words in any order, and they should then be displayed in alphabetical order.

- A reasonable first stab at a (pseudo-code) algorithm is:

```
make an empty word list
while there are more words:
    get a word from the user
    add the word to the end of the word list
sort the word list
display each word in the word list
```

Example: creating a simple dictionary

- Creating a list of words (strings) one-by-one is a variation on the accumulator pattern we've seen before:
  - we initialise a list `words` to `[]` (the empty list);
  - we add words to this list using the `append` method.
- We can use a sentinel loop to allow the user to enter the words and signal the end (the empty string is a good sentinel value).
- When the user has finished entering words, we can sort the list `words` using the `sort` method.
- Finally, we can display the words by iterating through the list using a `for` loop.

Example: creating a simple dictionary

- Coding this in Python gives:

```
def makeDictionary():
    words = []
    while True:
        word = input("Enter word (ret to exit): ")
        if word == "":
            break
        words.append(word)
    words.sort()
    for word in words:
        print(word)
```

Lists of objects

- We can store any kind of data in lists, not just numbers and strings.
- For example, in the practicals we have been using **objects** of the graphics types `Line`, `Circle`, `Point`, etc.
- We can easily create a list of `Point` objects:

```
>>> dots = [Point(10,20), Point(30,20), Point(50,20)]
```

- Suppose that we wanted to display all the points in this list in a graphical window `win`; this can be done using a `for` loop:

```
>>> for dot in dots:
        dot.draw(win)
```

Lists of mixed-type

- Note that a single list can contain data of different types. E.g.,

```
>>> shopping[2] = 42
>>> shopping
['butter', 'eggs', 42, 'jam',]
```

- Mixing data of different types in a list can make code difficult to understand; we won't look at further examples.
- More usefully, lists can be **nested**; for example:

```
>>> matrix = [[1, 2], [3, 4]]
>>> matrix[1][0]
3
```