INTPROG
Introduction to Programming

moodle.port.ac.uk

Practical Worksheet P09: Using Lists

Introduction

This worksheet is intended to get you started using lists to represent collections of data. Work through the worksheet at your own pace and, as always, make sure you study each segment of code and try to predict its effects before you enter it, and afterwards make sure you understand what it has done and why. Furthermore, feel free to experiment with the shell until you fully understand each concept.

Lists and list indexing

We will experiment with lists using Pyzo/IEP's shell. As we experiment with lists, we'll notice that they share most (but not all) of their properties with strings.

Let's begin by making a list of integers and assigning it to a variable. We then ***access the elements*** of the list using its ***indices***, which start at 0:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
primes[0]
primes[4]
primes[7]
primes[8]
```

We can also index lists using negative indices, where -1 is the final position in the list:

```
primes[-1]
primes[-3]
primes[-8]
primes[-9]
```

Obviously, we can also make lists of data of other types, including strings:

```
verbs = ["run", "walk", "sleep", "eat"]
verbs[1]
```

and floats:

```
temperatures = [14.5, 8.0, -2.5, 15.0]
temperatures[-2]
```

We note that all these lists are of the same type; i.e., try:

```
type(primes)
type(verbs)
type(temperatures)
```

Other basic list operations

The following basic list operations correspond with those that we saw for strings. Firstly there is list repetition:

```
[2, 4, 6, 8] * 3
verbs * 2
[0] * 10
```

List concatenation gives a new list formed from the elements of two existing lists:

```
primes2 = primes + [23, 29]
primes2
primes
```

The built-in function `len` gives the length of a list:

```
len(primes)
len(primes2)
```

Finally, list ***slicing*** allows us to access a ***sub-list*** (some consecutive elements) of a list:

```
primes[3:6]
primes[1:-2]
primes[:-1]
primes[4:]
primes[:]
```

We see that `primes[m:n]` gives a list containing the elements of `primes` starting at position `m` and ending one short of position `n`. If either `m` or `n` is omitted, then the beginning or the end of the list is assumed.

Iterating (looping) through a list

We have already seen (i.e. when using `range`) that we can iterate though a list using a `for` loop:

```
for prime in primes:
    print(prime, "is a prime number")
```

Sometimes we need to use a `for` loop to iterate though the ***indices*** of a list, and then access the list's elements using list indexing. To do this, we use `len` and `range`; for example:

```
for day in range(len(temperatures)):
    print("The temperature for day", day + 1, end=" ")
    print("was", temperatures[day], "degrees")
```

Membership checking

We can check whether a given value is a member of a list using the `in` operator; for example:

```
5 in primes
6 in primes
"walk" in verbs
"cat" in verbs
```

We see that `in` gives its result as a Boolean value. The `in` operator is often useful for checking the validity of user input, as the following example shows:

```
def readPrime():
    primes = [2, 3, 5, 7, 11, 13, 17, 19]
    while True:
        myPrime = eval(input("Enter a prime number less than 20: "))
```

```
        if myPrime in primes:
            break
print("You entered", myPrime)
```

Changing the elements of a list

A major difference between lists and strings is that individual elements of lists can be changed using assignment statements (we say that lists are ***mutable***, but strings are ***immutable***). For example, let's change one of the elements of the verbs list:

```
verbs[1] = "talk"
```

and a couple of the temperatures:

```
temperatures[2] = 25.0
temperatures[0] = temperatures[1] + 20
```

Experiment with updating a list in this way until you are fully familiar with how it works.

Further list operations

There are many other list operations that take the form of methods; we'll look at a few examples. Firstly, we can extend a list (i.e. add an extra element at the end) using the `append` method:

```
primes.append(23)
verbs.append("drink")
```

We can find out at which position an element first occurs in a list using the `index` method:

```
verbs.index("sleep")
primes.index(11)
```

We can count the number of times a value occurs in a list using the `count` method:

```
verbs.count("sleep")
[3, 5, 3, 4].count(3)
```

We can insert an extra element into a list at any position using `insert`:

```
verbs.insert(2, "sit")
```

We can remove the first occurrence of a value from a list using `remove`:

```
verbs.remove("eat")
```

We can sort a list into numerical or lexicographical order using the `sort` method:

```
temperatures.sort()
verbs.sort()
```

Finally, we can reverse a list using the `reverse` method:

```
verbs.reverse()
```

Experiment with these methods until you fully understand what each of them does. (Note that the unit textbook covers a few extra list methods.)

Lists of mixed-type

We have seen an example of a list of integers (`primes`), a list of floats (`temperatures`) and a list of strings (`verbs`). We should note that Python only has one list type (as we

demonstrated earlier), and that any data element can appear in any list (i.e. lists can contain a mixture of data of different types). For example, the following is allowed:

```
temperatures[2] = "Hot"
```

We won't provide an example for such multi-typed lists. Indeed, their use is rarely required and usually considered poor programming style.

Lists of other objects

Recall from lecture P04 that, as well as the built-in types (`int`, `float`, `string` etc), programs can operate on many other types of data, such as `Point`, `Line`, `Polygon` and `Rectangle` as defined in the graphics module.

We can easily make lists of such objects. For example, the following code creates a list containing two `Point` objects:

```
from graphics import *
pointList = [Point(10, 30), Point(50, 20)]
```

We can now access these `Point` objects and their methods; e.g:

```
pointList[1].getX()
pointList[0].getY()
```

Programming exercises

Your attempted solutions to these exercises should be written to a file named pract09.py. If you attempt exercises 6–8, these should be written in their own files.

1. Using a list, write a function `displayDate` that takes three integer parameters representing a day, month and year, and outputs the date as a (readable) string. For example, the call `displayDate(14, 2, 2011)` should display: `14 Feb 2011`. (Hint: put strings representing all the months into a list, and index this list.)

2. Write a function `wordLengths` that takes a list of strings as a parameter, and displays each of the words alongside its length. E.g, the call `wordLengths([["bacon", "jam", "spam"])` should result in the following output:

   ```
   bacon 5
   jam 3
   spam 4
   ```

3. Write a function `drawHexagon` that allows the user to draw a hexagon (a shape with six vertices (points)) on a graphics window. The function should allow the user to click on their chosen locations for the hexagon's vertices, and then draw the hexagon (filled in red). (Hint: A hexagon is a Polygon, and you can make a `Polygon` object from a list of `Point` objects.)

4. A test at a university is marked out of 5. Write a function `testMarks` that allows a lecturer to input the unit marks of all her students, providing a suitable way for her to say she's finished entering marks. The function should then output the number of students who have achieved each possible mark. For example, if the lecturer enters the marks 4, 0, 4, 1, 1, 4, 3 and 5, the function should output:

   ```
   1 student(s) got 0 marks
   ```

```
2 student(s) got 1 marks
0 student(s) got 2 marks
1 student(s) got 3 marks
3 student(s) got 4 marks
1 student(s) got 5 marks
```

5. Write a function `drawBarChart` that takes a list of integers as a parameter, and draws a simple downwards-facing bar chart of # symbols for the data in the list. For example, the call `drawBarChart([3, 1, 2])` should result in the following output:

```
# # #
#   #
#
```

6. Write a version street2.py of the street.py function from worksheet P08 (exercise 5). In this new version, the user should be asked for the door colour of each individual house before the row of houses is drawn.

7. Recall the one-dimensional random walk program from worksheet P08 that simulated several random walks in order to estimate the expected distance away from the start point. Write a similar program tracedwalk.py that simulates a single random walk that begins in the centre point of a pavement that is `n` steps (or squares) long (the value of `n` should be input be the user). The program should keep a count of how many times the walker steps upon each square of the pavement. The walk ends when the walker steps off either end of the pavement. For example, if the pavement is 5 squares long, the walker begins on square 3. If the random walk takes steps forward, backward, forward, forward, forward, then the program should report:

```
Square   Steps
   1       0
   2       0
   3       1
   4       2
   5       1
```

8. [Harder] Write a two-dimensional version tracedwalk2d.py of the above program. This should simulate a single random walk of the form considered in worksheet P08, where the walker begins in the central square of a square two dimensional grid of dimensions 9 by 9 steps, and where the walk ends when the walker steps off the grid. Output the results in tabular form. (Hint: you may need to use nested (two-dimensional) lists.)