CE151

Introduction to Programming
Part 4
Sequences, Strings and For Loops

Arrays and Sequences 1

In part 3 we saw an example of the processing of exam marks. Suppose that we wished to be able to perform several different forms of processing, e.g. average mark, determine how many students obtained over 70%, determine what percentage of the class passed. We would not want to have to input the marks again for each new task so we need to be able to store the marks in variables. If we wished to use a separate variable for each mark we would need to declare about 100 variables and write code to process each one – this is clearly not reasonable.

Hence we wish to store the entire collection of marks in a single variable – in most languages we would do this using an ***array***.

Arrays and Sequences 2

An array (as used in most programming languages) is a collection of data items, each of which has the same type. Arrays are stored in the computer using consecutive memory locations; the number of items must be specified when an array is created. We can then access an item in the array by specifying its position, e.g. `marks[5]` might be used to access the element at position 5 in an array called `marks`.

Python does not have a built-in array type (although one is available from a library module); instead it has various different kinds of ***sequence types***. To store the marks we would use one of these types: `list`.

Lists 1

Lists in Python are more versatile than the array types present in languages such as C and Java: the size of a list is not fixed and we can add more values at any time.

We can create a list by specifying its initial contents, e.g.

```
months = [ "Jan", "Feb", "Mar", "Apr", "May",
           "Jun", "Jul", "Aug", "Sep", "Oct",
           "Nov", "Dec" ]
daysInMonth = [ 31, 29, 31, 30, 31, 30, 31,
                31, 30, 31, 30, 31 ]
```

As seen in these examples it is acceptable to write the list over more than one line (with line breaks after commas) – indentation of continuation lines is optional but makes the code more readable.

Lists 2

We can use the subscripting operator `[]` to access an element of a list: `a[n]` refers to the element at location `n` in the list `a`. Indexing of locations starts at 0, so `months[0]` has the value `"Jan"` and `months[1]` has the value `"Feb"` . If we try to access `months[n]` where `n` has a value greater than 11 a runtime error will occur and the program will terminate.

We can also use the subscripting operator with a negative subscript: `a[-1]` refers to the last element in the list `a`, `a[-2]` refers to the penultimate element, and so on. If we try to access `months[-n]` where `n` has a value greater than 12 , once again a runtime error will occur.

[The operand inside the square brackets may be any expression with a value of type `int`.]

Lists 3

In the two lists that we have introduced all of the elements had the same type: `months` is a list of strings and `daysInMonth` is a list of integers. It is permissible in Python to mix items of different types in the same list:

```
sillyList = [ "Bart", 99.9, "Monty", 17 ]
```

However, lists with items of different types should not be used other than in special circumstances – there are other data types that are more appropriate for storing a collection of values of different types. We often need to write loops to process the elements of a list one at a time, with the same code being used to process each element, and cannot do this if the elements have different types.

Lists 4

If we were inputting exam marks into a list we would need to initialise the list to be empty and add the marks one at a time as they are input.

To create an empty list and assign it to the variable **`marks`** we would use

```
marks = []
```

We can then add items to the end of the list using the **`append`** ***method***:

```
mark = float(input("Next mark: "))
marks.append(mark)
```

(A method is like a function but it is applied to an object using the **`.`** operator.)

Lists 5

We will now write a loop to input values from the keyboard and store them in a list of exam marks. As in part 3 we will use a negative number to indicate the end of the input.

```
marks = []

print("Enter marks")
print("Use a negative number to finish")

mark = float(input("First mark: "))

while mark >= 0.0 :
  marks.append(mark)
  mark = float(input("Next mark: "))
```

Lists 6

When processing the marks we need to know how many elements are in the list. It would have been possible to use a variable to count the marks as they were being entered, but this is not necessary – we can use the `len` function to determine the length of a sequence (i.e. how many elements it contains).

On the next slide we present a loop that will calculate and output the average mark and the highest mark – note that we need to check if the list is empty since the user might have entered a negative number for the first mark.

Note that the subscript for the last element will be one less than the length of the list so we use `i < size` as the condition for the loop.

Lists 7

```
total = maximum = 0.0
size = len(marks)

if size == 0 :
  print("The list is empty")
else :
  i = 0
  while i < size :
    total = total + marks[i]
    if marks[i] > maximum : maximum = marks[i]
    i = i+1

  print("The average is", round(total/size, 1))

  print("The highest mark is", maximum)
```

Lists 8

We can replace an element of a list with another item by assigning a value to the element using the subscripting notation, for example `daysInMonth[1] = 28`.

Suppose that the examiners have decided that the examination was too hard and every student's mark should be scaled upwards by 10%. The following loop could be used to perform this task.

```
i = 0
while i < size :
  if marks[i] < 100.0 :
    marks[i] = marks[i] * 1.1
    if marks[i] >= 100.0 : marks[i] = 99.9
    # we don't want to give any student 100%
    # unless he/she actually got 100
  i = i+1
```

Lists 9

We can delete an item from a list using a `del` statement, for example `del sillyList[2]`. Having deleted this item the item that was previously at location 3 will now be at location 2.

The following code will delete from the list of marks all marks less than 40

```
i = 0
while i < len(marks) :
   if marks[i] < 40.0 : del marks[i]
   else : i = i+1
```

If we delete an item, we must make sure that we next examine the item that now occupies the same location, so we must not increment the counter.

Strings 1

The `list` type in Python is just one of the sequence types. There are three others: `str`, `tuple` and `range`. (In addition there are three more types, `set`, `frozenset` and `dict`, that have similar characteristics to these, but are not classified as sequence types since we cannot apply all of the operations defined on sequences to them.)

We have already encountered strings at the end of part 1 and when using functions like `print` and `input`. Strings are sequences of characters and have the type `str`.

Strings 2

We can access individual characters in a string using the subscript notation. However, since the Python language has no character type the value of `s[n]`, when the variable `s` refers to a string and `n` is a valid subscript, is a string of length 1. (This is unlike lists, where, if `a` was a list of integers, the value of `a[n]` would be an integer, not a list.)

The `len` function may take any sequence as its argument, so we can use it to obtain the length of a string. Hence to process the characters in a string one at a time we could use a loop with the same structure as those seen for processing marks.

Strings 3

The code fragment presented here will input a line of text into a string and count the number of spaces in that string.

```
myString = input("Enter a line of text")

spaces = 0
i = 0
size = len(myString)

while i < size :
  if myString[i] == ' ' : spaces = spaces + 1
  i = i+1

print("The line contained", spaces, "spaces")
```

Strings 4

The code fragment presented here will input a line of text into a string and output the characters in reverse order. We loop through the characters starting from the end of the string and append them in turn to a new string that is finally output.

```
myString = input("Enter a line of text")
outString = ""
i = len(myString)-1
while i >= 0 :
  outString = outString + myString[i]
  i = i - 1
print(outString)
```

Strings 5

Unlike lists, strings are ***immutable*** – the contents of a string can ***not*** be changed. Hence code such as

```
myString = "Hello, Bert"
# want to change name to Bart
myString[8] = 'a'
```

is not allowed, and there is no `append` method for strings.

Note that on the previous slide, when we perform the assignment `outString = outString + myString[i]`, the concatenation generates a new string and makes the variable `outString` refer to it; it does not change the contents of an existing string.

String Literals 1

A ***string literal*** is a string written in quotes (`'` or `"`) in a program. If a string literal is too long to display in a window we may need to break it up over more than one line; in this case the first line must end with `\`:

```
myString = "this is a long string and will not \
fit on the slide"
```

If we were to print the value of `myString` the output would appear as

```
this is a long string and will not fit on the slide
```

Note that the string contains no newline character and if we were to indent the continuation line, the indentation spaces would be part of the string.

String Literals 2

Suppose we wish to store as a single string the following heading:

```
**********
*  PAGE 1  *
**********
```

If we tried to write this as a string literal on one line of code with embedded newline characters we would need to use

```
"**********\n*  PAGE 1  *\n**********"
```

It is difficult to see what the output would look like and several attempts might be necessary before the correct string is typed.

To make situations like this easier to handle Python has a notation for multiple-line string literals.

String Literals 3

A string literal occupying several lines is written using `'''` or `"""`. (We have already used this notation for documentation – a documentation string is indeed a string; it is its position in the code that determines that it is documentation.)

Hence we can store the header string from the previous slide in a variable called `header` using

```
header = """
*****************
*    PAGE 1     *
*****************
"""
```

Using this notation the newline characters (including those adjacent to the quotes) are included in the string.

Slices 1

Subscripting is used for obtaining a single item from a sequence; a similar notation, called ***slicing***, is used to obtain a sequence of adjacent items. The value of an expression of the form `s[m:n]` (where `s` can be any sequence) is a sequence containing the elements of `s` from location `m` up to but ***not*** including location `n`.

For example, if `s1` refers to the string `"Dead parrot"`, the value of `s1[0:4]` would be `"Dead"` and the value of `s1[5:11]` would be `"parrot"`.

The value of a slice is always a sequence so if `a` is a list of numbers `a[3:4]` would be a list containing one number, not a number.

Slices 2

We can use negative numbers for either operand; these are handled in the same way as on slide 5.

We can also omit either operand (or both): the value of `s[m:]` will be a slice running from location `m` to the end of the sequence, and the value of `s[:n]` will be a slice running from the beginning of the sequence to the element located immediately before location `n`.

Hence `s1[:2]` will have the value `"De"` and `s1[8:]` will have the value `"rot"`.

`s[:-2]` will be a slice containing all of the elements of `s` except the last two.

Slices 3

The use of slicing will not generate a runtime error if either of the operands denotes an invalid location. If the second operand is too large or the first operand is too small the value obtained is the same as we would get if the argument had been omitted, so `s1[8:35]` will have the value `"rot"` and the value of `s1[:35]` will be the entire string.

If the first operand is too large, the second is too small, or the index (location in the sequence) specified by first operand is larger than the index specified by the second one, the value will be an empty sequence, so `s1[33:35]`, `s1[-25:-19]` and `s1[6:3]`, will all produce empty strings.

Slices 4

Slicing can provide a convenient way of handling leading or trailing zeroes in numeric output when we want a fixed format. For example, suppose we have three integer variables holding a day, a month and a year and want to output their values as a date in the format 12/04/98. To generate leading zeroes when necessary we could convert the day and month into strings, then append each of these strings to `"0"`. In this example we would get `"012"` and `"04"` from the numbers 12 and 4 – the required output in all cases is a slice containing the last two characters of the string.

We assume that the year contains 4 digits so all we need to do for the year is obtain a slice containing the last two digits of the year string.

Slices 5

The code to print the date in the format described on the last slide is presented here.

```
dString = "0" + str(day)
mString = "0" + str(month)
yString = str(year)
print(dString[-2:] + "/" + mString[-2:] +\
                                "/" + yString[-2:])
```

Note that to convert a number (or any other data item) to a string we use the function **str**.

Slices 6

In the same way that we can assign values to list elements using subscripts we can assign new values to slices of lists. (This does not apply to slices of strings, since strings are immutable.)

Assume that the variable `a` refers to the list `[1,2,3,4 5]`. After performing the assignment `a[1:3] = [6,7]` the list will hold `[1,6,7,4,5]`.

The right-hand operand does not have to have the same length as the slice. After performing the assignment `a[1:4] = [8,9]` the list will hold `[1,8,9,5]`.

We can use a slice in a `del` statement. For example `del a[1:4]` will remove from the list the items at location 1, 2 and 3.

Example – Trimming Strings 1

When a user is asked to supply a string as input, particularly when using a graphical user interface, the string may contain redundant spaces at the start and end. Before processing the input, it is often necessary to remove these.

On the next slide we present a function to perform this task. It will use two loops to find the locations of the first and last non-space characters in the string and then use slicing to obtain the trimmed string.

To avoid generating an unnecessary copy of the string when there are no leading or trailing spaces we simply return the original string if no trimming is necessary.

Example – Trimming Strings 2

```
def trim(s) :
  """
  trims leading and trailing spaces from s
  argument: s:str
  returns trimmed string:str
  """

  begin = 0
  while begin < len(s) and s[begin] == ' ' :
    begin = begin+1

  end = len(s)-1
  while end >= 0 and s[end] == ' ' : end = end-1

  if begin == 0 and end == len(s)-1 : return s
  else : return s[begin:end+1]
```

Example – Trimming Strings 3

Observe that in the function on the previous slide it is necessary to check whether the value of `begin` reaches `len(s)`. This would happen if the string contained nothing other than spaces. If `begin` did reach `len(s)` a runtime error would occur if we tried to access `s[begin]`. Since the `and` operator does not evaluate its second operand unless the first has the value `True` our code prevents any attempt to access `s[begin]` in this case.

We performed a similar check in the second loop. An alternative strategy would be to immediately return an empty string if `begin` reaches `len(s)`.

Another approach for checking whether the string comprised just spaces would be to compare it with `' '*len(s)` .

Lists and Assignment

Consider the following code fragment

```
a = [1, 2, 3]
b = a
a[2] = 4
b[1] = 7
print(a, b)
```

The output generated would not be `[1 ,2 ,4] [1, 7, 3]` but instead would be `[1, 7, 4] [1, 7, 4]`.

The assignment has made the variable `b` refer to the same list as the variable `a` so the two subscripted assignments modify the same list.

To make `b` refer to a copy of the list we should use `b = a[:]`.

Sequence Operators

It has been seen in the lab exercises that the operators `+` and `*` can be applied to strings to perform concatenation. These operators can in fact be used with any sequence type.

If `s1` and `s2` refer to sequences (of the same type) then `s1+s2` is a new sequence obtained by appending a copy of `s2` to the end of a copy of `s1`, so for example `[2,3]+[4,5]` gives `[2,3,4,5]`.

If `s` refers to a sequence and `n` refers to an integer then `s*n` and `n*s` both produce a sequence obtained by concatenating `n` copies of `s`, so for example `[1,2]*3` gives `[1,2,1,2,1,2]`.

Searching in Sequences 1

The `in` operator can be used to determine whether an item occurs within a sequence. For lists this must be a single item, so `x in a` has the value `True` if the item `x` occurs in the list `a`, and `False` if it does not occur.

For strings the item for which we are searching may be a single-character string or a longer string, so `s1 in s2` has the value `True` if and only if the string `s1` occurs as a substring of the string `s2`.

There is also a `not in` operator: `x not in a` has the value `True` if `x` does not occur in `a`.

Searching in Sequences 2

We present, as an example, code to extract the first word from a line of text input by the user. (Here we assume that a word is any sequence of non-space characters.) The string into which the line has been input is first trimmed, then we use the `in` (or `not in`) operator to determine whether there are any spaces left. If there are no spaces, the entire string is one word so this is the desired result. Otherwise we must locate the first space and then use slicing to obtain the substring comprising the characters before that space.

Searching in Sequences 3

```
line = input("Type a line of text: ")
line = trim(line)   # trim was defined on slide 28

if line == "" : print("No words on line")
else :
  if ' ' not in line : firstWord = line
  else :
    # line has been trimmed so first character
    #         cannot be a space
    pos = 1
    while line[pos] != ' ' : pos = pos + 1
    firstWord = line[:pos]

  print("The first word is", firstWord)
```

For Loops 1

As already seen, when accessing the elements of a sequence of length `n` one-by-one we can typically use a loop of the form

```
i = 0;

while i<n :
{ access a[i]
  i = i+1
}
```

Early procedural languages such as Algol, Basic and Pascal provided a structure known as a ***for loop*** to allow this to be done in a more convenient way using syntax such as

```
FOR i = 0 TO n-1 DO
  access a[i]
```

For Loops 2

The C family of languages introduced a more versatile form of the loop. In the Python language the `for` loop is even more versatile, allowing us to use it to access the contents of other data structures.

The general syntax of Python's `for` loop is

```
for variable in expression_list : statement
```

or

```
for variable in expression_list :
    statements
```

There are various different kinds of expression list that can be used; the simplest kind, and the only one that we will consider now, is a sequence.

For Loops 3

Using a loop of the form

```
for i in s :
    statements
```

where `s` is a sequence, the statement suite will be executed once for each item in the sequence, with the variable `i` referring to that item, in order from the first item to the last item.

Hence we can use the following loop to print all of the items in a sequence, one per line:

```
for i in s : print(i)
```

For Loops 4

If the sequence is a string the variable will refer to a string containing a single character. The following code will count the number of vowels in the string referred to by **myStr**:

```
vowels = 0
for c in myStr :
   if c in "aeiouAEIOU" : vowels = vowels+1
```

On the next slide we present a new version of the code from slide 10, using a **for** loop instead of a **while** loop. We observe that it is more concise.

For Loops 5

```
total = maximum = 0.0
size = len(marks)

if size == 0 :
  print("The list is empty")
else :
  for mark in marks :
    total = total + mark
    if mark > maximum : maximum = mark

  print("The average is", round(total/size, 1))

  print("The highest mark is", maximum)
```