823G5 Programming in Python

Lab 2: Code constructs and using lists

Aim

To familiarise yourself with

- `if` statements
- `while` loops
- `for` loops
- lists and list operations
- list comprehension

Exercises

Download the `Lab2Exercises` ZIP file for example solutions. The code is a set of Python source code file (`.py`). You will need to open a new PyCharm project and import the `.py` files into it. Refer to the Lab 1 instructions if you cannot remember how to do this.

Extra hints: working with strings

There are several useful operators for working with strings. You can use the `==` operator to check whether two strings have the same value. Use the `is` operator to determine whether two variables refer to the same object (this is a subtle distinction). Here is an example to illustrate the point:

```
s1 = 'red'
s2 = 'blue'

# == tests whether the value of two strings are equal
# is tests whether the two variables refer to the same object string

# Here s1 and s2 are not equal in value, and they are not the
# same object

if s1 == s2:
    print('Equal')
else:
    print('Not equal')
```

```
if s1 is s2:
    print('Same object')
else:
    print('Not the same object')

s3 = s2

# Here s2 and s3 are equal in value, and
# refer to the same object

if s3 == s2:
    print('Equal in value')
else:
    print('Not equal in value')

if s3 is s2:
    print('Same object')
else:
    print('Not the same object')

# Here s4 and s2 are equal in value, but do not
# refer to the same object

s4 = 'Blue'
s4 = s4.lower()

if s4 == s2:
    print('Equal in value')
else:
    print('Not equal in value')

if s4 is s2:
    print('Same object')
else:
    print('Not the same object')
```

You can also use comparison operators such as < and > to determine string ordering based on lexicographical rules. As case forms part of the rules of lexicographical ordering, you find the methods `upper()` and `lower()` useful. They transform a string to upper and lower case respectively. Remember that strings are immutable so a copy is returned:

```
words = ['aardvark', 'GNU', 'penGuin', 'FIsh']
w1 = input('Enter a word to compare: ')
w1 = w1.lower()

for w in words:
    if w1 > w.lower():
        print(w1, 'comes after', w, 'in the dictionary')
    elif w1 == w.lower():
        print(w1, 'is in the dictionary')
        break
    else:
        print(w1, 'comes before', w, 'in the dictionary')
```

There are also sorting and ordering operations available for lists of strings:

```
# Note sorting takes place in situ ...

words = ['aardvark', 'gnu', 'penguin', 'fish']
words.sort()
print(words) # Forwards alphabetic order

words.sort(key=str.lower, reverse=True)
print(words) # reverse alphabetic order
```

Coding challenges 1

1) Allow the user to enter a number. Use an `if` statement to display whether the number is negative, equal to zero or positive.
2) The `range()` function can be used to create a sequence of numbers (see the code examples for this). Allow the user to enter a number and then iterate over the range to determine whether the number the user entered is present in the sequence.
3) An arithmetic series of numbers is defined by an initial value and a common difference. For example, if the starting value is 2 and the common difference is 3, the arithmetic sequence would begin 2, 5, 8, 11, 14, … Allow the user to enter values for the initial value and common difference. Write code to determine the sum of the first 10 elements in the arithmetic sequence.
4) Write code to simulate the simple children's game "Fuzz Buzz". The rules are simple; start counting upwards from 1 in single increments. If the number is divisible by 5, display "Fuzz". if the number is divisible by 6, display "Buzz". If the number is divisible by both, display "Fuzz Buzz". Otherwise just display the number. Play the game from 1 through to 50 inclusive.

Some suggested answers are provided in Canvas.

Coding challenge 2

For this challenge we will create a bubble (swap) sorter that can sort words in a list into alphanumerical order. A bubble sorter operates by swapping pairs of elements until the complete ordering is achieved.

The bubble sort process for a list of N items is based on a series of iterations of swap sorting. An iteration works by comparing two items and determining

whether they need to be swapped. Here is an example of a single iteration of the process for a list of 6 elements ($N = 6$):

gnu	aardvark	horse	donkey	eagle	emu	swap
aardvark	gnu	horse	donkey	eagle	emu	no swap
aardvark	gnu	horse	donkey	eagle	emu	swap
aardvark	gnu	donkey	horse	eagle	emu	swap
aardvark	gnu	donkey	eagle	horse	emu	swap
aardvark	gnu	donkey	eagle	emu	horse	iteration done!

This results in a maximum of $N - 1$ swap pairs. In order to guarantee that the list is ordered, we need to repeat this iteration process a maximum of $N - 1$ times. This is needed as the first element in the sorted list might have started at the right hand most position and needs to make its way to the start of the list.

Using two loops, create a bubble sorter that can sort a Python list. Your sorter should ignore case in performing the sort.

A suggested answer is provided on Canvas.

Coding challenge 3

Let's start by adding some more coding basics, in particular the notion of a list comprehension.

A list comprehension provides a concise way to create lists. Common application are to make new lists where each element is the result of some operation applied to each member of another sequence or iterable thing, or to create a sub-sequence of those elements that satisfy a certain condition.
Say, for example, we had list of numbers [1,2,3,4] and we wanted to generate a list of the squares of those values. That requires squaring each list element in turn. We could do that using a loop, but a list comprehension provides a more concise form:

```
x = [1, 2, 3, 4]
squares = [y**2 for y in x]
print(squares)
```

A list comprehension consists of square brackets followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from

evaluating the expression in the content of the `for` and `if` clauses that follow it. For example, finding the pairings of non-identical values between two lists:

```
x = [1, 2, 3]
y = [2, 3, 4, 5]

z = [(a,b) for a in x for b in y if a != b]
print(z)

# Gives [(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4),
# (2, 5), (3, 2), (3, 4), (3, 5)]
```

Using this knowledge:

1) Create a list comprehension and use it to display only the even numbers in a given list.
2) Create a list comprehension that produces a new list `b` from a given list of strings `a` consisting of only those elements in `a` that have 3 or fewer characters.
3) Create a list comprehension that produces a new list `b` from a given list of strings `a` consisting of only those elements in `a` that begin with a specific letter.
4) Create a list comprehension that produces a new list b from a given list of strings a consisting of only those elements in a that begin with a specific letter and have 3 of fewer characters.

A suggested answer is provided on Canvas.

Dr. Benjamin Evans
B.D.Evans@sussex.ac.uk