Programming in Python for Business Analytics (BMAN73701)¶

Lab Session: Week 2 - Conditionals and Loops¶

If you are comfortable with the material, then you can just do the exercises, though do try to read through the examples provided as they cover some common mistakes and some of the statements required for the exercises.

Conditionals - If this then that else if this then that else that(!!)¶

Sometimes we want to execute different code based on different conditions. A common example of this (as you saw in the slides) is for printing different grades based on the score that was achieved. They are generally simple by themselves, but are often used in `for` or `while` loops, and can become complex by using many conditions. The below is long, silly example of if statements just to show some of what can be done and some potential pitfalls.

```
# Get the user input
number = input("Please provide a number: ")

# First check if they gave an integer
if int(number) != float(number):
    print("You gave a decimal!?")
    print("Now it's an integer)

# Regardless, convert it to an int (as remember it is a
string!)
```

```
number = int(number)

# Check what range the value is in
if number == 0:
    print("Zero")
elif number < 0:
    print("Don't be so negative!")
elif number > 0:
    print("Stay positive ;)")
elif number==3: # Can you spot a problem here? Try entering 3
    print("You guessed my favourite number!")

# Check what kind of number it is
if number % 2 == 0:
    print("That's an even number!")
else:
    print("What an odd number!")
```

Exercise 1: Determine the min and max value¶

This exercise is intended to be short and quick, so if you get stuck ask for help so that you can move on quickly. The example above is more complicated!

You need to write a program that asks the user for two floats (floating point numbers/decimals), and then determine which one is the minimum and which one is the maximum.

Lists¶

Before an exercise, we'll just cover some material and play around with lists first. A `list` is just something we can use to store other objects, including more lists. It's incredibly flexible and is used a lot. One thing to keep in mind is that usually a list contains items of a single data type. In Python, you can have a list that has, for example, numbers and strings, but if you are doing this then you need to re-think your code (and in most other languages this is not even allowed!).

Let's create some lists that we will play around with:

```
# Numbers in a list
list_a = [1,2,3,4]
# A list of strings
list_b = ['strings', 'in', 'a', 'list']
# A list of lists, each sublist with numbers
list_c = [[1,2,3],[4,5,6],[7,8,9]]
# An empty list to use later
list_d = []
# A range converted in a list (this will be explained more at a
later date)
list_e = list(range(10))
```

Note that none of the lists here are called "list". This is because, as we can see in the last line, `list` is actually a built-in function, just like `print` or `input`. If we name a variable "list", it will overwrite this function and we won't have access to it anymore until we restart the program. While I don't recommend this, if you want to try this and see why it causes problems, type `print="test"`. You'll find yourself no longer able to print anything! Until you know most of the functions this can be an easy mistake to make, but can be easily avoided by using more than one word to name a variable. You will need to keep this in mind when we start writing our own functions next week.

Lists have the useful property that they are *mutable*, so we can add items, remove items freely. Note that this can also be a problem as you can accidentally change a list - such as modifying a list during a for loop.

Using the variables/lists above, what output would you expect from the statements below? If you're not sure, try to guess, then try it out and see if you were right. As these are single line statements, it may be easier to do this in the console. As these are simple statements, we expect you to take a bit of time to think and then type them in - therefore the whole list below should only take 5-10 minutes.

```
# Adding items to a list
list_a.append(5)
list_a.insert(0,0)
list_a += [6]
# Removing items
list_b.remove('in')
del(list_b[0])
# Length of a list
```

```
print(len(list_c))
# Length of a sublist
print(len(list_c[0]))
# Finding where something is in a list
list_a.index(1)
list_c.index(1) # Careful!
# Moving items around
list_a.sort()
list_a.sort(reverse=True)
list_a.reverse()
# Append vs extend
list_d.append(1) # Append a single number
list_d.append([7,2]) # Append using a list of 2 numbers
list_d.extend([3,5]) # Extend using a list of 2 numbers
# Slicing
list_e[:]
list_e[:5]
list_e[5:]
list_e[-1]
# Modifying items in a list
list_e[:2] *= 2 # This is why we need to be careful multiplying
lists/other objects
list_e[2:] = list(range(2,12)) # Replace items in a list
```

That's quite a few things, but practice is key. More can be done with lists but if you're comfortable with the above that's great!

For Loops¶

These are used to iterate over something, and so we know in advance how many times we go through the loop. They are very closely linked with lists, as the two are used together very often.

As we did with lists, we'll go over a few examples that you should first try to guess the output, and then type them in and try them out. As these loops will require more than one line, it may be better to do this in a script than in the console.

```
# Simple for loop
for i in range(5):
    print(i)

# Range can have more uses
# It can take the form range(start, stop, step)
```

```
for i in range(1, 11, 2):
    print(i)

# Loop over a list of strings
for text_var in ["text", "to", "print"]:
    print(text_var)

# enumerate is a very useful function! It allows us to get a
counter (i), and the value (text_var)
for i, text_var in enumerate(["number"]*4):
    print(text_var, i)

# Assignment in and outside a for loop
a = 0
for i in range(5):
    b = 0
    a += i
    b += i
    print(a, b)
print("Final:", a, b)

# Loops and mutable objects
a_list = [1,2,3,4,5]
for val in a_list:
    a_list.remove(val)
    print(val)
    print(a_list)

# If you're unsure why the above happens, try this
a_list = [1,2,3,4,5]
for i, val in enumerate(a_list):
    del(a_list[i])
    print(i)
    print(a_list)
```

We use `i` here just for convention as a counter.

Exercise 2:¶

Write a program that asks the user for 3 things that they are interested in, and then print those things as a list. Once you've done this, modify the script so that the user can input how many things they are interested in (instead of 3), and then afterwards print all of their interests.

While Loops¶

While loops are used for when we want to repeat as long as some condition is `True`, typically for when we don't know exactly when this condition will become `False`.

Consider this program:

```
# Import the random module
import random

# Select a random number
number_to_guess = random.randint(1, 10)

# Initialise our variable outside the range of what we're
guessing
guess_val = 0

# While we have guessed incorrectly, keep trying
while number_to_guess != int(guess_val):
    guess_val = input("Guess a number between 1 and 10: ")
print("Correct!")
```

We use a while loop as we don't know how long it will take the user to guess what number was randomly selected. We know it will be at least 1 and at most 10 attempts (assuming the user is trying out each number between 1 and 10!), but that's it.

Infinite Loops¶

One problem with `while` loops is the chance to get stuck in an *infinite loop*, which is where the while loop never exits and it continues *forever*. Sometimes you can close the program (such as Spyder) and it's OK, other times you have to restart the computer. I'll show an example below of an infinite loop, so if you choose to run it on your computer be prepared to have to restart!

```
# Sometimes we use while True so that we can use an if
statement to break later
n = 0
while True:
    # Add 2 to n
    n += 2
    # This might print a lot
    print(n)
    # Break when equals to 5...which it never does!
    if n == 5:
        break
```

Manipulating and exiting loops: `continue` and `break`¶

These are two very useful ways of changing how loops work. They are used alongside an `if` statement. We use `break` to exit a (for or while) loop, and `continue` is used to skip everything else in the block and go back to the start of the next iteration of the loop. Try to think about what the following two segments of code would output before running them.

```
# Using continue and break in a for loop
for letter in "aPaYaTaHaOaN_TEXT":
    if letter == "a":
        continue
    if letter == "_":
        break
    print(letter)

# Skip certain numbers in the while loop
n = 0
while n < 10:
    n += 1
    if n % 2 == 0:
        continue
    print(n)
```

Exercise 3:¶

Write a program that iterates over the numbers from 1 to 30 (inclusive). For every multiple of 3, the program must print `Richard`. For every multiple of 5, the program must print `Manuel`. For every number that is a multiple of both, print `Cameron`. Note that some may find this difficult, but breaking the problem down into little steps and seeing what output you get is vital.

I won't show the whole expected output, but here's the first half:

```
1
2
Richard
4
Manuel
Richard
7
8
Richard
Manuel
11
Richard
13
14
Cameron
```

Dictionaries¶

A dictionary is a set of `key:value` pairs, where the keys are unique identifiers that map to their respective values. These values can be of any type (including another dictionary!).

```
# Create an empty dict
people = {}
# Create a key 'names' in the dict, with an empty list
people['names'] = []
# Define some names
names = ["Richard", "Manuel", "Cameron"]

for name in names:
    people['names'].append(name)
print(people)
```

The importance of using dictionaries effectively is deciding what to use as keys and what to use as values. We can then use this structure to quickly and easily select groups of data by using loops and conditions.

Exercise 4:¶

Create a dictionary where the key is the name of the person, and the value is another dictionary where there are two keys `"city"` and `"country"` where that person is from. You need to use at least three people for this. You do not need to use `input` for this, though you can if you want. You should do this using a loop. Some example data is below for you:

```
names = ["Richard", "Manuel", "Cameron"]
cities = ["Karlsruhe", "Cadiz", "London"]
countries = ["Germany", "Spain", "UK"]
```

Using the data above, when you print the final dictionary it should look like:

```
{'Richard': {'city': 'Karlsruhe', 'country': 'Germany'},
'Manuel': {'city': 'Cadiz', 'country': 'Spain'}, 'Cameron':
{'city': 'London', 'country': 'UK'}}
```

As the amount of data grows and the complexity, dictionaries can be incredibly useful and efficient. With the data in a structured format we can do more informative things with it. Looping through the dictionary (let's call it `people`), print the city of all the people from one particular country. Using the example above, if we wanted to print only the cities of the people from Germany, the output would simply be:

In [5]:

```
for person in people.values():
    if person['country'] == "Germany":
        print(person['city'])
```

```
Karlsruhe
```

Exercise 5:¶

In this exercise you need to write a program that simulates a given number of flips or coins. We've provided some steps below to build up the required code bit-by-bit.

1. To simulate a coin flip we need to generate random 0s (heads) or 1s (tails). We can use the `random` module for this, which has the following two functions we could use for this (you need to pick one):
    - `random()`: returns a random value between 0 and 1

    - `randint(a,b)`: returns a random integer from the interval [`a,b`]

2. Flip a coin 100 times to generate a random sequence of heads and tails

3. After each flip print out whether it is a head or tail, and also how many times heads and tails have been flipped so far
4. Modify this code so that we have a biased coin; the user should be able to set the level of bias. Previously, the chance of heads or tails were both 50%.
5. Run the code again using a probability of getting heads 20% of the time, and print what bias you actually got. Repeat this for 1,000 and 10,000 coin flips.

Further Resources:¶

- Lists - This is a great, comprehensive page, and overall a good resource to look at
- Dictionaries - Same resource as above

- Flow control - Same resource as above

- Official Python page on these topics