MATH6005 Python Lab 2

January 21, 2019

2 Conditionals and Loops

This week we are going to learn about conditional logic and iteration: `if` statements to control programme flow and `for` and `while` loops to repeat blocks of code. There is also a *debug* challenge and a *'do something we haven't taught you how to do'* challenge at the end of the lab.

The lab assumes you are using `Spyder 3` and Python 3.x

Solutions to exercises will be available from blackboard from next week.

2.1 Using conditional logic to control programme flow

- Most code makes extensive use of **if then** statements to control execution.
- This makes use of the `if`, `elif`, and `else` keywords

```
if <Boolean Operation 1>:
    do something
elif <Boolean Operation 2>:
    do something different
else:
    take default action
```

A boolean operation is an operation that returns a True or False value (a boolean). These are summarised in the table below.

- **Tip**: Remember that the = operator is for **assignment**; use == when you want to **compare** equality E.g.

```
x = 3 # assigns the value 3 to the variable 'x'
x == 3 # returns the value True if x has the value 3 and False otherwise
```

Operation	Description	Operation	Description
a == b	a equal to b	a != b	a not equal to b
a <b	a less then b	a >b	a greater than b
a <= b	a less than or equal to b	a >= b	a greater than or equal to b

2.1.1 Example 1: Animal noises!

We are going to write a function that displays a different noise to the user depending on the type of animal specified.

The function `animal_call()` requires a single keyword argument `animal` (default == 'mouse')

- If the animal is a mouse then print out 'squeak'
- Else if the animal is a cat then print out 'meow'
- Else if the animal is a dog then print out 'woof'
- Else if the animal is a cow then print out 'moo'
- Else print out 'Sorry, I do not know what noise that animal makes'

```
In [1]: def animal_call(animal='mouse'):
            '''
            Print out a animal noise dependent on the parameter passed in.

            Keyword arguments:
            animal -- the animal whose noise you would like to make.
                Options: mouse (default), cat, dog and cow.
            '''

            if animal == 'mouse' :
                print('squeak!')
            elif animal == 'cat' :
                print('meow!')
            elif animal == 'dog' :
                print('woof!')
            elif animal == 'cow' :
                print('moo!')
            else:
                print("Sorry, I don't know what noise this animal makes.")

        animal_call()
        animal_call('cow')
        animal_call('fish')
        animal_call('cat')
        animal_call('MouSe')
```

```
squeek!
moo!
Sorry, I don't know what noise this animal makes.
meow!
Sorry, I don't know what noise this animal makes.
```

Points to remember

- The function `animal_call` has a default value for animal (mouse). So if you execute `animal_call()` it prints 'squeak'.

- `elif` is the python command for **else if**
- `else` is a 'catch all' that executes if `animal` does not equal any of the prespecified .
- At the end of each `if`, `elif` and `else` you need to have a colon `:`
- Remember Python whitespace rules and indent underneath an if statement.
- Python string comparisons are case sensitive. Therefore the if statement does not recognise 'MouSe' as 'mouse'.
- To solve the case sensitive problem you could modify the code to use the `str.lower()` function.

- This converts animal to all lower case i.e.

```
if animal.lower() == 'mouse':
    ...
```

2.1.2 Example 2: Purchasing chocolate bars from a vending machine

We are going to write some code to represent a vending machine.

A vending machine has a stock list, i.e. a list of items it sells, and keeps track of the number of items remaining for each type of stock. The inventory count has been implemented as a list of numbers who indexes match the corresponding item in the stock list.

```
stock_list = ['mars', 'twister', 'whisper', 'twix']
inv_list = [0, 1, 0, 1]
```

When a person asks the machine to `vend`, they make a `choice` about what to purchase.
Two checks need to be done to confirm this is possible:

- Is the choice a valid choice from the stock list
- Is there sufficient inventory to meet the customers requirement.

The system needs to report information back to the user

- If the user makes an invalid choice then the system should inform them
- If the user makes a valid choice, but there is currently no stock the system should inform them
- If the user makes a valid choice that is in stock
- the system should dispense the item and tell the user it has done so
- the system should decrement the appropraite inventory.

To implement the logic above we are going to use a nested if statement.

The first if statement is going to check if the users `choice` is contained in the `stock_list`. We learnt how to do this in Week 1. We need to use the `in` statement. This results a `bool` (True/False) indicating if the value of choice is contained in the list.

```
if choice in stock_list:
    ...
```

The nested if then checks if there is remaining inventory to vend.

```
if inv_remaining[stock_list.index(choice)] > 0:
    ...
```

This second if statement is considered 'nested' because it sits inside the first if statement

```
if choice in stock_list:
    if inv_remaining[stock_list.index(choice)] > 0:
        ...
    else:
        ...
else:
    ...
```

We will need to find the index of a `choice` in the `stock_list`. For example, if we wanted to find the index of 'twix' in `stock_list` we would use the following code:

```
choice = 'twix'
stock_list.index(choice)
```

Example code and test cases are given below.

```
In [33]: def vend(stock_list, inv_remaining, choice='mars'):
             """
             A simplified vending machine.

             A user's choice must be a valid item from the stocklist
             There must also be inventory remaining.

             Prints the outcome of the vending process.

             Keyword arguments:
             stock_list -- a list of strs: item names available for purchase
             inv_remaining -- a list of ints: no. of each item left in machine
             choice -- a str representing the users choice (default 'mars')
             """
             if choice.lower() in stock_list:

                 if inv_remaining[stock_list.index(choice)] > 0:

                     inv_remaining[stock_list.index(choice)] -= 1
                     print('Stock of {0} reduced by 1. Remaining: {1}'.format(choice, inv_list))

                 else:
                     print('{0} out of stock'.format(choice))
             else:
                 print('Incorrect choice. Valid options = {0}'.format(stock_list))
```

```
stock_list = ['mars', 'twister', 'whisper', 'twix']
inv_remaining = [0, 1, 0, 1]

vend(stock_list, inv_remaining)  # mars out of stock
vend(stock_list, inv_remaining, 'twister') # successful stock reduced to zero
vend(stock_list, inv_remaining, 'twister') #out of stock
vend(stock_list, inv_remaining, 'banana') #do not stock that item
vend(stock_list, inv_remaining, 'twix') #successful stock reduced to zero
```

```
mars out of stock
Stock of twister reduced by 1. Remaining: [0, 0, 0, 0]
twister out of stock
Incorrect choice. Valid options = ['mars', 'twister', 'whisper', 'twix']
Stock of twix reduced by 1. Remaining: [0, 0, 0, 0]
```

2.1.3 Exercise 1: Booking Tickets to the Cinema

You are writing the server side code for a website that allows people to book seats at a cinema.

Task:

- Write three functions to book tickets, refreshments and the total cost of the booking. The functions are:
- **tickets:** Returns the costs of tickets (i.e one or more) purchased. Normal tickets cost £10.99. If the booking is for a Wednesday the price of each ticket is reduced £2.00. If premium seating is requested an books cost extra £1.50 per person regardless of the day.
- **refreshments:** Returns the cost of refreshments. A user could buy 'popcorn' for £2.00 or 'fizzy pop' for £3.50
- **cinema_trip:** Adds the cost of tickets and refreshments together.

Hints:

- We have provided a skeleton of the code below along with 2 test cases. Open `cinema_exercise.py`
- A pre-written skeleton of the code for you to use in Spyder can be found in cinema.py
- The functions `tickets` and `refreshments` need to be written and return the correct cost.
- Each of the functions requires you to use `if` statements.
- The `tickets` function requires you to check if the day of the booking warrents a discount and if premium seating charges apply.

```
In [41]: def tickets(number, day, premium_seating):
             """
             The cost of the cinema ticket.
             Normal ticket cost is $10.99
             Wednesdays reduce the cost by $2.00
```

```
    Premium seating adds an extra $1.50 regardless of the day

    Keyword arguments:
    number -- integer value representing the number of seats to book
    day -- day of the week to book (1 = Monday ... 7 = Sunday)
    premium_seating -- boolean True/False.  Are premium seats required.
    """
    #fill in your code here.
    return 0.0


def refreshment(choice ='popcorn'):
    """
    The cost of refrehments.
    Choices are popcorn or fizzy pop

    Keyword arguments:
    choice The users choice of refreshment (default = 'popcorn')
    """
    #fill in your code here
    return 0.0


def cinema_trip(persons, day, premium_seating, treat):
    """
    The total cost of going to the cinema

    Keyword arguments:
    persons -- number of people who need a ticket
    day -- day of the week to book (1 = Monday, 7 = Sunday)
    preimum_seating -- boolean True/False if premium seats are required
    treat -- string value representing a choice of refreshment
    """
    #fill in your code here
    return tickets(persons, day, premium_seating) + refreshment(treat)


persons = 2
day = 1
premium_seating = True
treat = "popcorn"

msg = "today a trip to the cineman will cost you £{:.2f}"

print(msg.format(cinema_trip(persons, day, premium_seating, treat)))
#expected answer = £26.98

persons = 3
```

```
        day = 3
        premium_seating = True
        treat = "fizzy pop"

        print(msg.format(cinema_trip(persons, day, premium_seating, treat)))
        #expected answer = £34.97
```

```
today a trip to the cineman will cost you £0.00
today a trip to the cineman will cost you £0.00
```

2.1.4 Exercise 2: The FizzBuzz game

Task

- Write a function that checks if a number is a multiple of 3, 5 or both.
- If the number is a multiple of 3 return "FIZZ"
- Else If the number is a multiple of 5 return "BUZZ"
- Else If the number is a multiple of 3 AND 5 return "FIZZBUZZ"
- Else return the number (cast as a string)

Input data to test: 1, 3, 5, 15, 23

Expected Output: 1, "FIZZ", "BUZZ", "FIZZBUZZ", 23

Hints

- When creating a fizzbuzz function remember that a function has a **single responsibility**.
- That means you should try to seperate function and the printing (display) of the result
- A function definition might look like the below:

```
def fizzbuzz(n):
    ...
```

- You will need to use the mod operator to check if a number is a multiple of 3, 5 or both. e.g.

```
number = 6
number % 3 #( = True)
number % 5 #( = False)
```

- This function will requires one `if` statement two `elif` statements and one `else` statement
- Look at Lecture 2 for an example implementation

2.2 Using loops to repeat blocks of code

Loops are an essential part of coding, but the concept is often difficult to learn the first time it is encountered. We are going to work with several examples of looping to build up your experience.

- There are two types of loop in Python
- `for` loops and `while` loops
- We generally use `while` if we **do not know** the number of iterations in advance
- We generally use `for` if we **know** the number of iterations in advance

2.2.1 Loops Example 1:

The code below is a `for` loop to `print` the integer numbers 0 to 4

```
In [17]: for number in range(5):
             print(number)
```

```
0
1
2
3
4
```

- To create a **for** loop you need the following:
- `for` keyword
- a variable -- whose value will change on each loop iteration. In the case above it is called `number`
- the `in` keyword
- the `range()` function - which is an built-in function in the Python library to create a sequence of numbers
- `range()` takes up to three keyword arguments: set the start (inclusive, default = 0), end (exclusive) and step (default = 1)
- `range(start, stop, step)`
- To learn more about the range function run the code below:

```
help(range)
```

2.2.2 Loops Example 2:

The code below uses a `for` loop to iterate across a `List` of integers. It appends the square of the list item to a new list.

```
In [24]: def square_list_items(to_square):
             """
             Returns a list of numeric value that are
             the square on an input list

             Keyword arguments:
             to_square -- the list of numeric values to square
```

```
        """
        result_list = []

        for item in to_square:
            result_list.append(item**2)

        return result_list

    my_list = [1, 2, 3, 4, 5]
    result = square_list_items(my_list)
    print(result)
```

```
[1, 4, 9, 16, 25]
```

- Notice that `to_square` can be iterated over directly. We did not use `range` here
- Each time the loop iterates the value of `item` changes.

- The first time the loop executes `item` is equal to 1; in the second 2; in the third 3 and so on until all items in `to_square` have been used.
- An **alternative** implementation using `range` is below

```
In [ ]: def square_list_items(to_square):
            """
            Returns a list of numeric value that are
            the square on an input list

            Keyword arguments:
            to_square -- the list of numeric values to square
            """
            result_list = []

            for index in range(len(to_square)):
                result_list.append(to_square[index]**2)

            return result_list

        my_list = [1, 2, 3, 4, 5]
        result = square_list_items(my_list)
        print(result)
```

- In order to iterate (loop) the correct number of time we use `range(len(to_square))`
- The function `len(to_square)` returns the **length** of the list i.e. the number of items contained.
- The variable `index` value starts at 0 and increments by 1 each time until it reaches 4
- The variable `index` is used to access the correct list item in each loop.

- E.g. in loop 0 we want item `to_square[0]` and in loop 2 we want `to_square[2]`

2.2.3 Loops example 3: A guessing game using a while loop

We generally use `while` loops if we **do not know** the number of iterations in advance. A classic use of a `while` loop is a main loop of a computer game. The code below implements a guess the number game.

- There are 2 functions: `guessed_correctly` and `play_game`
- `guessed_correctly` is simple and just checks a user specified correct answer against a guess.
- `play_game` implements the a loop that calls `guess_correctly`
- We don't know how many times we need to call `guessed_correctly`, as the user may repeatedly guess incorrectly or may guess correctly first time. So we use a `while` loop
- In the loop we continually reprompt the user for a guess.
- The built in function `input` prompts the user to enter a guess.
- We convert the guess (of type string) to an integer by calling `int`

```
In [39]: def guessed_correctly(guess, correct_answer):
             '''
             Returns True or False to indicate if user has guessed correctly.

             Keyword arguments:
             guess -- user's guess
             correct_answer -- the answer that needs to be guessed!
             '''
             return guess == correct_answer


         def play_game(correct_answer):
             '''
             A game to guess a integer number (between 1 and 10).
             Re-prompts user for guess until correct one is supplied.

             Keyword arguments:
             correct_answer -- specified by user
             '''

             guess = -1

             while not guessed_correctly(guess, correct_answer):
                 guess = int(input("Please guess a whole number between 1 and 10: >>"))

             print("Well done you guessed correctly!")


         #a test: this returns false because the user guessed incorrectly
         result = guessed_correctly(guess=5, correct_answer=10)
         print(result)

         #a test: this returns true because the user guessed correctly
         result2 = guessed_correctly(guess=10, correct_answer=10)
```

```
        print(result2)

        #rather than repeatedly calling guessed_correctly we have created a loop.
        #while guessed_correctly returns false
        #we continue to re-prompt the user for another guess.
        play_game(10)
```

```
False
True
```

```
Please guess a whole number between 1 and 10: >> 1
Please guess a whole number between 1 and 10: >> 3
Please guess a whole number between 1 and 10: >> 10
```

```
Well done you guessed correctly!
```

- The guessing game we have implemented could go on forever!
- We can impose a upper limit on the number of iterations the while loop
- In this case we have specified a keyword argument `lives`: the maximum number of guesses allowed.
- In the loop below we use

```
while lives > 0:
    ...  # game logic
    lives -= 1
else:
    ...
```

The loop now iterates until all player lives are lost. We can also break the loop at any time by calling the `break` statement

The loop implements the following logic:

- Prompts for a user guess
- If it is the correct guess then break the loop
- Else decrement the number of lives by 1
- If all lives (guesses) have been used then end the loop and execute the `else` statement

The latter part demonstrates that while loops can have an `else` statement. This code will **not be excuted** if a `break` statement is called.

```
In [36]: def play_game(correct_answer, lives=3):
             '''
             A game to guess a integer number (between 1 and 10).
```

```
            Re-prompts user for guess until correct one is supplied.

            Keyword arguments:
            correct_answer -- specified by user
            lives -- the number of guesses a user is allowed to
            make before they lose the game (default = 3)
            '''

            guess = -1

            while lives > 0:

                guess = int(input("Please guess a whole number between 1 and 10: >>"))

                if guessed_correctly(guess, correct_answer):
                    print("Well done you guessed correctly!")
                    break # you win! no need to continue! break the loop

                else:
                    lives -= 1

            else:
                #code executed becasue lives > 0 = False
                print('Sorry no lives left - you lose!')


        play_game(10)
```

```
Please guess a whole number between 1 and 10: >> 1
Please guess a whole number between 1 and 10: >> 2
Please guess a whole number between 1 and 10: >> 10


Well done you guessed correctly!
```

2.2.4 Loops Exercise 1: Find the maximum value in a list

- Write a function that accepts a `List` of integers as a parameter and then finds and **returns** the maximum value in the list.
- Print the result to the screen.

Input data:

- Create a `List` of integer values

```
to_search = [0, 1000, 2, 999, 5, 100, 54]
```

Expected Output

- 1000

Hints:

- Write a function with the name `find_max` that requires a `list` as a parameter e.g.

```
def find_max(to_search):
    ...
```

- There are many ways to find the maximum value in a `List`. A simple solution is to iterate (with a for loop) over all items in the `List` and keep track of the largest value found. e.g.

```
for item in to_search:
    ...
```

- You will need a variable to keep track of the maximum value found e.g. `current_max`
- Don't call your variable `max` -- this is a reserved name in Python.
- On each iteration you will need to compare the `current_max` to the current value of the `List` being searched.

```
item > current_max
```

- **If** `item` is greater than the `current_max` then you need to update `current_max`.
- Look at Week 1 lectures and lab notes if you cannot remember how to create a function.
- The function requires both a `for` loop and an `if` statement.

2.3 Loops Exercise 2: Fizzbuzz game (again!)

Write a programme that prints the numbers 1 to 100. But, for multiple of three print "Fizz". For multiples of five print "Buzz" and for multiples of three and five print "FizzBuzz".

Hints:

- We have already written a fizzbuzz function. This is the advantage of functions - you can reuse them!
- When creating a fizzbuzz function remember that a function has a **single responsibility**. That means you should try to seperate the iteration (loop) from fizzbuzz function. (e.g. call the function from within a loop
- Take a look a the Week 2 Lecture notes for some help.

2.3.1 Loops Exercise 3: Net Present Value

In simple terms Net Present Value (NPV) is the sum of the Present Values (PV) of future expected cash flows.

For example, you invest £2000 for 3 years and receive £100 per year along with a final payment of £2500. Using an interest rate of 10% what is the investments Net Present Value?

So your cashflows in years are: `[-2000, 100, 100, 2600]`

From last week we know the formula for the Present Value of an a future expected cashflow is: PV = FV / (1 + r)^n

If we convert each of our cashflows into PVs we get:

- Year 0: PV = -2000 / (1.1)^0 = -2000.00
- Year 1: PV = 100 / (1.1) ^1= 90.91
- Year 2: PV = 100 / (1.1)^2 = 82.64
- Year 3: PV = 2600 / (1.1)^3 = 1953.42

Our Net Present Value is therefore:
NPV = -2000 + 90.91 = 82.64 + 1953.42 = 126.97
Task

- Create a function that can calculate the NPV of any investment decision.

- Test the function with the above data.
- How would your decision change if the interest rate was 15%?

Hints:

- You have already created a PV function last week. You can reuse this function within the calculation of the net present value. The code is included below.
- The future cashflows can be represented as a `List`

```
cashflows = [-2000, 100, 100, 2600]
```

- There are two variables in this problem - the cashflows and the interest rate. A function definition might look like:

```
def npv(cashflows, rate):
    ...
```

- A 'for loop' might come in handy to iterate over each of the cashflows and call the `pv` function.

- Remember that the `pv` function has 3 keyword arguments - the future (cash) value, the interest rate and the time into the future (years)

- You will need to keep track of the year in your loop. Look back at the `for` loop examples if you need help.

- The NPV is a sum of the present value of the cash flows. The calculation takes place in a loop. Therefore you need to keep track of the running total each time you loop

2.3.2 Week 2: Debug Challenge

Each laboratory will have a debug challenge. You will be given a pre-existing script containing Python code. The catch is that the code doesn't run!

Your challenge is to find and correct the errors so that the script correctly executes.

The challenges are based around common problems students have when writing code. If you do the exercises it will help you debug your own code and maybe even avoid the mistakes in the first place!

Challenge 1: This weeks debug challenges loops and nested loops.

A classic task in programming is to sort a list. Python makes this very simple by including a number of ways to sort a list.

Under the hood the sorting routines are all variations on loops where values in the array are swapped until it is in ascending order. We are going to look at a famous sorting algorithm called **Insertion Sort**.

Instructions:

- open `wk2_debug_challenge.py` in `spyder 3`
- Attempt to run the code.
- The code will raise errors.
- Fix the bugs!

Hints:

- Read the Python interpreter output.
- The errors reported can look confusing at first, but read them carefully and they will point you to the lines of code with problems.
- The `Spyder` IDE may give you some hints about formatting errors
- It can be useful to use `print()` to display intermediate calculations and variable values.
- Remember that `Spyder` has a variable viewer where you can look at the value of all variables created.
- There might be multiple bugs! When you fix one and try to run the code you might find another!

2.3.3 Week 2: Do something we haven't taught you how to do challenge

Each laboratory will challenge you to do something that we haven't taught you before.

No course can teach you everything you need for all programming problems. Being a competent Python programmer means that you need to learn how to find solutions to problems yourself. These challenges are designed to help you begin to use internet resources in order to solve your problem.

Challenge 1: You have been given a file that contains 4 columns of comma seperated values. When data are stored like this the file is called a CSV (comma seperated value) file. The filename is **moviedb.csv**

Your challenge is to read the data from the .csv file into a Python `List`.

Note: *If you have never seen a CSV file before then we recommend you take a quick look at it.* You can use MS Excel to open CSV files or you can view it using a text editor. You could also perform an internet search to research them.

Challenge 2: To be completed after you have completed challenge 1

Your challenge is to read the data from the moviedb.csv file into a Python `List` *but store the row of headers (the first row in the file) in a seperate python* `List`

Challenge 3: To be completed after you have completed challenge 1 and 2

If we take another look at moviedb.csv we can see that there are different types of data contained within it

- Column headers (strings)-- e.g. Id, title, budget
- ID (int) -- a number that acts as a unique identifer for the movie
- Budget (float) -- a budget (in millions of $)
- Box_office (float) -- the movie revenue
- Year (int) -- the year the movie was released
- Meta_Critic (int) -- a numeric review score (out of 100) from metacritic.com

At the moment the data that you read into a Python `List` is all of type `str` (string). Your challenge is read the data into a python `List` and convert it to the correct data type given the above information. Note: there are several ways to complete this challenge! But sometimes the best solution is one that works!