COM661 Full Stack Strategies and Development

Practical A1: Introducing Python

Aims

- To introduce Python as a high-level programming language
- To demonstrate the Python command line interpreter
- To introduce program files and demonstrate their invocation
- To introduce Python variables, strings and simple list structures
- To present basic Python loop and selection structures
- To develop a first interactive application in Python

Contents

A1.1 Python Basics

Python is an object-oriented programming language that is suitable for a wide range of software development tasks. In recent years, it has become popular for web applications development, due to its strong support for integration with other languages and tools, and its extensive libraries.

Python is a simple language to learn, with relatively few keywords and data types – but it is extremely powerful and flexible. Python code is also among the most readable of all high-level languages.

Note:
If you are installing Python on your own computer, please use **version 3.7** to be compatible with the material in this section. There is still a large legacy community that uses the older Python 2.7, but official support for this will soon be withdrawn. Version 3.x has some important differences from earlier versions

The first program attempted in any programming language is traditionally one to display the text "Hello, World" on the display. The Python version is typically simple.

```
>>> print("Hello, World")
Hello, World
```

Do it now!
Open a terminal window (command prompt) by clicking the Windows "***Start***" button, entering "***cmd***" in the search box, and clicking "***cmd.exe***' from the list of options provided. Now begin the Python interpreter by entering the command **`python`** in the terminal window. At the **>>>** prompt, enter the Python statement

```
print("Hello, World")
```

and press the *enter* key. The Python interpreter executes the command and displays the requested text on the next line.

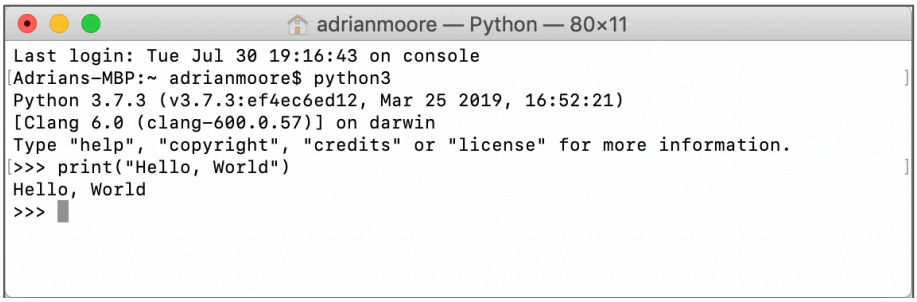

Fig A1.1 Hello World program

Note:
Note that Figure A1.1 uses the command **`python3`** instead of **`python`** to invoke the interpreter. This is sometimes the case when both Python v3 and Python v2 are available on the same computer. The version number that appears in the message generated in response to your command will confirm that you are running the correct version on your machine. In the labs, the command **`python`** should be used.

Python supports all of the standard mathematical operations as illustrated below – addition, subtraction, multiplication, raising to a power, division and modulus (remainder).

Do it now!
Try the simple mathematical functions below in the Python interpreter. Pay particular attention to the division examples and note that Python differentiates between floating point division (the `/` operator) and integer division (`//`).

```
>>> 1 + 2
3
>>> 5 - 3
2
>>> 5 * 2
10
>>> 5 ** 2
25
>>> 21 / 3
7.0
>>> 23 / 3
7.666666666666667
>>> 23//3
7
>>> 49 % 10
9
```

Python variables and assignments also operate in the manner that we would expect. See the example below, where we assign integer values to a pair of variables **`first`** and **`second`** and calculate the result of a 3rd variable **`third`**. See also that when we perform the assignment **`first=second`** and then subsequently change the value of **`second`**, the value of **`first`** is not affected. In other words, assigning one variable to the value of another creates a new copy of the value.

Python has no explicit variable declarations. Variables come into existence when they are assigned a value and are automatically destroyed when they go out of scope.

```
>>> first = 10
>>> second = 20
>>> third = first + second
>>> print(first, second, third)
10 20 30
>>> first = second
>>> second = 40
>>> print(first, second)
20 40
```

Do it now!
Try the variable assignments above and verify their operation

String values in Python are enclosed in either single or double quote characters and are concatenated by the **+** operator.

Python also supports a powerful **slice** operator that can be applied to strings. The slice operator is invoked by **`string[start:end]`** which extracts all characters from position **`start`** to position **`end-1`**. If we omit either **`start`** or **`end`**, then the slice is assumed to be either from the beginning or to the end of the string. A negative value denotes character positions counting from the end of the string so that **`string[-1]`** refers to the last character, **`string[-2]`** refers to the next-to-last character, and so on.

```
>>> name="Adrian"
>>> print(name)
Adrian
>>> print(name[1])
d
>>> print(name[2:4])
ri
>>> print(name[:4])
Adri
>>> print(name[3:])
ian
>>> print(name[-1])
n
>>> print(name[1] + name[4:])
dan
```

Do it now!
Try the string manipulation operations from the example above and verify their operation. Experiment with other examples of your own until you are comfortable with the operation of the string slice operator.

IMPORTANT: Strings in Python are **immutable** (i.e. once created, they **cannot** be modified).

Do it now!
Try modifying the value of the name variable, by issuing the command **`name[0]='X'`**. Observe the error message as displayed in Figure B1.2

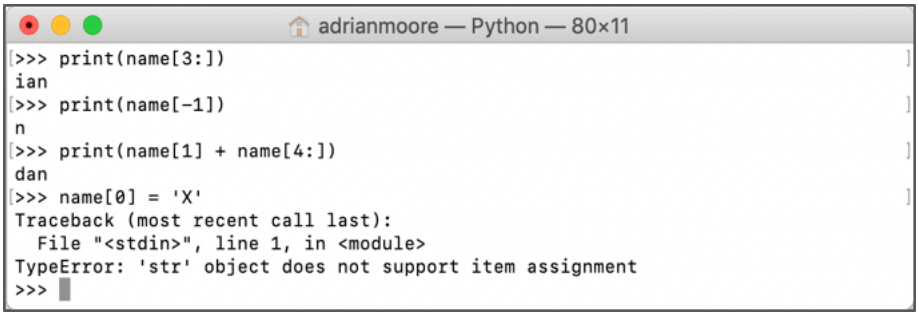

Fig B1.2. Immutable Strings

Try it now!
Given that strings are immutable, try and devise a command that has the same effect that you tried to achieve in the previous example. i.e. the value of **`name`** should be 'X' followed by the remaining characters from the original value of **`name`**

Arrays in Python are implemented as **lists**. Lists are actually much more powerful than standard arrays, but at their basic level can be treated as an equivalent structure.

Lists are created by presenting a collection of values separated by commas and enclosed in a set of square brackets. Lists can be comprised of objects of the same type, or can be mixed. List elements can also themselves be lists. Each list element is identified by an index value, beginning at 0 for the first element.

Note that lists are **mutable** (i.e. values of individual elements can be changed).

Note also that the slice operators can equally be applied to lists

```
>>> list1 = [12,23,34,45,56]
>>> list2 = ["apples","pears","oranges"]
>>> list3 = [1,"speedboat",True,[12,13,14],99]
>>> print(list1)
[12, 23, 34, 45, 56]
>>> list1[2] = 100
>>> print(list1)
[12, 23, 100, 45, 56]
>>> print(list2[:2])
["apples", "pears"]
```

Do it now!
Try the examples above and verify their operation. Experiment with other arrays, slice operators and print statements.

A1.2 Python Program Files

The Python command line interpreter is a useful tool for trying out short sequences of instructions, but for most applications we will organise our code within source files.

Examine the source code of *countEven.py*, which prints the numbers from 0 to 10 and indicates which are even numbers.

File*: A1/countEven.py*

```
print("Numbers from 1 to 10")
for number in range(1,11):
      if number % 2 == 0:
            print(str(number) + " (even number)")
      else:
            print(number)
print("-----")
```

Note in particular the following elements of a Python program

i) The structure of the program is controlled by the indentation. See how the code that is affected by the **`for`** and **`if`** statements is obviously apparent by the pattern of indentation. You can indent using **either** tabs or spaces – but be careful not to mix them as this **will** confuse the Python interpreter.

ii) All Python control structure statements (e.g. **`for`**, **`if`**, **`else`**, etc.) are terminated by a colon character

iii) The **`range()`** function returns a set of integers from 0 to one fewer than the value provided. To generate a list of numbers starting at a value other than zero, we pass both the lower and upper bounds. E.g. **`range(1,11)`** would generate the numbers from 1-10.

iv) In the **`if`** clause, we need to convert the number to a string object before we concatenate it with the other string.

Do it now!

Exit the command line interface by the command **`exit()`**. Now run the program by issuing the command **`python countEven.py`**

Note: Before running the program, you need to navigate to the drive and directory in which your Python file is located. For example, if it is in **`P:\COM661\python`** you would select the drive by entering **`P:`** (then press return) and move to the desired directory by entering **`cd \COM661\python`**. On Windows machines, an easier way is to use a Windows Explorer to navigate to the directory you want and then select *File* | *Open Command Prompt* from the menu.

Top Tip:
Drag the entire folder containing your files to the ***Visual Studio Code*** icon on the desktop. This will launch the Visual Studio Code editor in project mode – allowing you to easily move between your files. You can also open a Visual Studio Code Terminal window to run your code from within the editor.

Visual Studio Code is available as a free download from http://code.visualstudio.com

Do it now!
Adjust the program structure by moving the final statement into the **`else`** clause. (Hint: change the indentation so that the final **`print()`** statement is immediately below that in the **`else`** clause) Run the program again and see how the output generated is as illustrated in Figure A1.3.

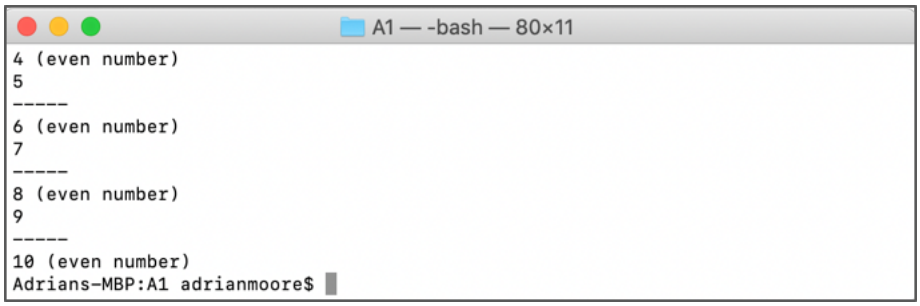

Figure A1.3. Finding Even Numbers

Try it now!
Modify the program to print out ONLY the multiples of 3 in the range 12-25 inclusive.

Usually, it is beneficial to arrange our application as a collection of modules such as classes, methods or functions. Here, we demonstrate how Python functions are defined with the **def** keyword. Once again, the indentation of the code is used to indicate the extent of the function.

File*: A1/countEvenFunction.py*

```
def countEven(start,limit):
   print("Numbers from {} to {}".format(start, limit-1))
   for number in range(start,limit):
      if number % 2==0:
         print(str(number) + " (even number)")
      else:
         print(number)
   print("-----")


countEven(12, 21)
```

Note here how we have used a slightly different form of the **`print()`** statement to embed variable values within the output. We will further explore the **`print()`** format options later.

Do it now!
Run *countEvenFunction.py* and verify its operation. Be sure that you understand the effect of each line of code and that you can trace its operation.

Try it now!
The Fibonacci series of numbers begins 0,1,1,2,3,5,8… and continues such that each new number is the sum of the previous two. Write a function **`fibonacci()`** that accepts a single integer parameter **`n`**, and prints out the first **`n`** values in the Fibonacci sequence. For example, calling the function with **`fibonacci(10)`** will generate the output illustrated in Figure A1.4.

NOTE: This can be achieved using only the Python code elements introduced so far (i.e. do not implement a recursive solution).

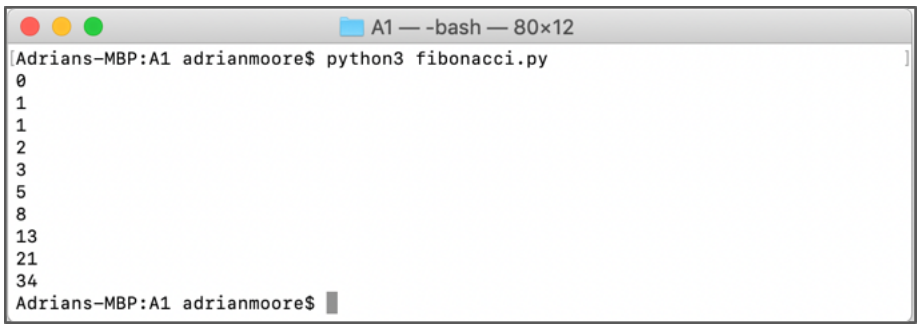

Figure B1.4. Fibonacci Series

A1.3 A First Python Application – A Battleships Game

As a first "proper" Python application, we will create a simple version of the popular guessing game "Battleships" (Ref: https://en.wikipedia.org/wiki/Battleship_(game))

Do it now!
Create a new file *battleships.py*. All code that we introduce in this section should be added to this file when instructed.

First, define **board** as an empty list (array), and **board_size** as the dimension of our board. Then we define the board to be a 2-dimensional collection of elements, where each element consists of a 'O' character.

File*: A1/battleships.py*

```
board = []
board_size = 5

for x in range(0, board_size):
   board.append(["O"] * board_size)

print("Let's play Battleships!")
print(board)
```

The Python method **append()** is applied to list objects to add an element to the end of the list. Here we specify the element to be added as **["O"]*5** which is equivalent to specifying **["O","O","O","O","O"]**. Repeating this operation 5 times (in the **for** loop) results in the creation of our 5x5 playing area, as seen in the output generated by the **print()** statement.

Do it now!
Enter the code above into *battleships.py*. Save the file and run it by entering **python battleships.py** at the command line.

When the board is printed, we can see the 5x5 structure, but it does not display in a particularly pleasing fashion. Replace the **print(board)** command with a call to a new function **print_board()** and supply the code for this function as below.

File*: A1/battleships.py*

```
def print_board(board):
   for row in board:
      print(row)

board = []
board_size = 5

for x in range(0, board_size):
   board.append(["O"] * board_size)

print("Let's play Battleships!")
print_board(board)
```

Do it now!
Amend the code of *battleships.py* as shown above. Save the file and run it again.

This produces a slightly better visual effect, but we can improve it even further by removing all of the brackets, quote characters and commas from the display as below

File*: A1/battleships.py*

```
def print_board(board):
   for row in board:
      print(" ".join(row))

...
```

Here, we use the **`join()`** method to create a string consisting of all of the elements in the original list (**`row`**). The subject of the method (the single space string " ") is used to separate the individual elements.

Do it now!
Replace the previous **`print_board()`** function with the new version. Save and run *battleships.py* to see the improved display.

We are now ready to randomly generate the position of the enemy battleship that the player is trying to locate. In this simple version of the game, the battleship will occupy only

a single cell, so its position can be represented by a simple **x** and **y** value – each in the range 0-4 (the dimensions of the board).

In order to generate the random battleship positions, we import the **`random`** object and call its **`randint()`** method as shown below. For debugging purposes, we also display the randomly generated battleship location – although we will remove this in the final version of the game.

File*: A1/battleships.py*

```
...

import random

def random_pos():
    return random.randint(0, board_size-1)

ship_row = random_pos()
ship_col = random_pos()
print("Battleship at {},{}".format(ship_row, ship_col))
```

Note the **`format()`** method applied to the string being printed, which substitutes the variables for the **`{}`** characters in the order in which they are presented.

Do it now!
Make the additions to your code and save and run the program. You should get output such as that illustrated in Figure A1.5 below.

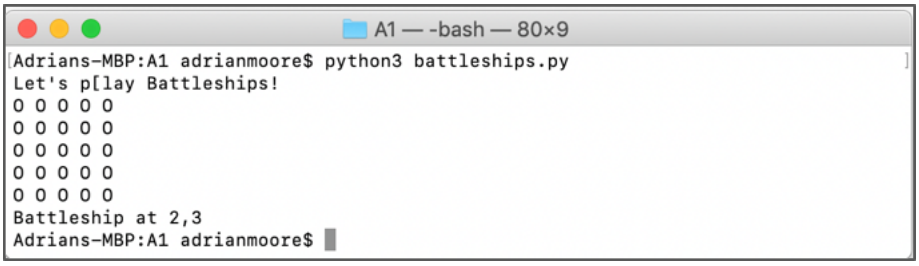

Figure B1.5. Batttleships v1

Note:
The file *battleships_v1.py* contains all the code so far (if required). The code in this file may be in a slightly different order to yours, but should be functionally identical. Normally, we perform any **import** operations at the top of the file, followed by any function definitions and finally the main body of the code.

Now we will invite the user to guess a row and column position and compare to the **`ship_row`** and **`ship_col`** location of the battleship. If the user guesses correctly, we provide a "success" message. If the user guesses incorrectly, we provide a "failure" message and record the position attempted in the board array. Finally, we print out the new board.

Note that we convert the result of the keyboard **`input()`** from string to integer using the **`int()`** function, so that we can use it in the comparison and array access operations.

File*: A1/battleships.py*

```
...

guess_row = int(input("Guess Row: "))
guess_col = int(input("Guess Col: "))

if guess_row == ship_row and guess_col == ship_col:
   print("Congratulations! You sunk my battleship!")
else:
   print("You missed my battleship!")
   board[guess_row][guess_col] = "X"

print_board(board)
```

Do it now!
Add the new code to the end of your existing *battleships.py*. Save and run the application, testing for both correct and incorrect guesses.

The game is beginning to take shape, but the user only gets one attempt to guess the position of the battleship. Now, we will extend it so that a number of attempts are allowed. In this implementation, we will allow a maximum of 5 guesses.

We implement multiple turns by enclosing the previous code in a **`for`** loop with a maximum of 5 iterations. If the user makes a correct guess, we use the **break** command to exit the loop immediately after the "success" message. In addition, we add messages to report the number of attempts made so far, and to confirm the end of the game.

File*: A1/battleships.py*

```
...

for turn in range(1,6):
   guess_row = int(input("Guess Row: "))
   guess_col = int(input("Guess Col: "))

   if guess_row == ship_row and guess_col == ship_col:
      print "Congratulations! You sunk my battleship!"
      break
   else:
      print "You missed my battleship!"
      board[guess_row][guess_col] = "X"

   print "After guess {} of 5".format(turn)
   print_board(board)

print "Game over"
```

Do it now!
Make the modifications indicated to *battleships.py*. Save the code and run the application to verify the effect of the new code.

The final enhancement is to provide additional information to the user when an unsuccessful guess has been made. First, we check for out-of-range input, and then we test for a guess that has previously been attempted.

The new modifications are highlighted overleaf. See how we use the **`elif`** statement as a contraction of **`else if`**. In Python, we use the **`elif`** option when possible as each successive **`else if`** would generate an increased level of indentation. As **`elif`** statements are on the same indentation level, it makes for more readable code.

Also, note the **** character which is used in Python to divide a long line of code. The **** option over-rides the indentation rules, so we are free to indent lines immediately following **** as we like.

Do it now!
Make the highlighted changes to *battleships.py* and test the application.

File*: A1/battleships.py*

```
...

for turn in range(1,6):
    guess_row = int(input("Guess Row :"))
    guess_col = int(input("Guess Col:") )

    if guess_row < 0 or guess_row >= board_size or \
       guess_col < 0 or guess_col >= board_size:
        print("Oops, that's not even in the ocean.")

    elif guess_row == ship_row and guess_col == ship_col:
        print "Congratulations! You sunk my battleship!"
        break

    elif(board[guess_row][guess_col] == "X"):
        print("You guessed that one already.")

    else:
        print "You missed my battleship!"
        board[guess_row][guess_col] = "X"
        if turn == 5:
            print("Game over!")

    print("This was turn " + str(turn))
    print_board(board)

...
```

Try it now!

Try extending *battleships.py* as follows:

i) Generate 3 random battleships for the user to find. You will also have to extend the number of attempts allowed.

ii) Extend the game further so that one of the battleships occupies 3 cells, one occupies 2 cells and one occupies 1 cell. All multi-cell battleships lie either vertically or horizontally on the board. When a user makes a successful guess, the character 'S' should be displayed in that cell on the board.

iii) Implement a 2-player game where players take turns to guess locations and the winner is the player with the most correct guesses. A player's turn should continue for as long as his guesses are successful.

A1.4 Further Information

- http://docs.python.org/tutorial/introduction.html
  An Informal Introduction to Python

- http://en.wikibooks.org/wiki/Python_Programming/Variables_and_Strings
  Python variables and strings

- http://www.developer.com/lang/other/article.php/626321/Learn-to-Program-Using-Python-Variables-and-Identifiers.htm
  Developer.com – Python variables and identifiers

- http://www.youtube.com/watch?v=Yltwd5Br6XY
  YouTube tutorial – Python variables

- https://www.learnpython.org
  The learnpython.org interactive tutorial

- http://docs.python.org/tutorial/
  The Python Tutorial

- http://www.python.org/about/gettingstarted/
  Getting started with Python

- http://www.python.org/getit/
  Download and install Python