FIT1008-FIT1054 Introduction to Computer Science (FIT2085: for Engineers)

Code Review Prac 1: Programming in Python

Semester 2, 2019

Objectives of this prac

- To get to know your demonstrator.
- To introduce you to some of the software you will be using.
- To get accustomed to expectations for prac, including commenting code and coding style.

Important

In order to answer the questions in this prac you will need some basic knowledge of Python. If you do not have it, please go through the online material pointed out in Week 0, particularly the JS-to-Python video. If you get stuck, you can always use Philip Guo's excellent code visualization tool Python Tutor at `http://www.pythontutor.com/visualize.html`. Remember to select Python 3.6, and you may leave the rest of the default settings. After editing your code and clicking on `Visualize execution`, you can step your code line by line pressing `Forward` or go straight to the end using `Last`. **You *must* use Python 3.3 or higher; some of the code we will use does not work in earlier versions.**

Before you do this prac you must read the documents downloadable from the Moodle links "PracGuide" in and "Lecture-Tute-Prac Expectations" in Week 0. This will ensure you have understood how code review pracs are run in this unit and what to expect from them (first document), and what to expect in general from your demonstrators and, importantly, what is expected from you (second document).

In particular, you need to be aware that all pracs in this unit might be longer and more complicated than those you have seen in previous units. If you do not make a start on your program before your class, you will soon realise that you **do not have enough time** to complete it. Code Review Prac 1 is shorter and easier, mainly because there were no lectures in Week 0 and, thus, this prac does not require any preparation. However, they will soon get longer and tougher. You may also benefit from watching the video guide to installing Python tools, available in Week 0 via Moodle.

Remember to submit your code via Moodle and to hand in the code review before you leave the lab. Otherwise you will not be able to get a mark for the code review.

Introduction to Pydoc (for FIT2085 students)

In Python, a *module* refers to any python file (suffix .py) with Python code in it to define functions, classes and/or variables. The documentation for a Python module is written alongside the code itself. The `pydoc` module allows for documentation to be automatically generated, whenever the programmer adheres to the pydoc conventions (which were used in the Week 0 document "Examples of good documentation").

For example, the documentation about the Python *list* type is automatically generated by `pydoc` and can be displayed on-screen in several different ways, including:

- In the Python interactive prompt by typing `help(list)`.
- At your operating system's command prompt (without going through the interactive prompt), by typing `pydoc list`.
- In a browser, either by generating a static HTML file for a particular module, or by starting a live server with all available documentation.

The documentation in `pydoc` works by turning specially-formatted Python strings (called `docstrings`) into professional-looking HTML documentation. The core developers for the Python standard library use `pydoc` to create the documentation for it. As you use the official Python documentation for the language at `http://docs.python.org/3/reference/` and for the library at `http://docs.python.org/3/library/` you'll become accustomed to the default `pydoc` style.

You will be required to document every module you write in this unit, and you are required to use docstrings to do so. The example downloadable from Moodle for Week 0 gives you an idea of the amount of required documentation. Yes, it might indeed look like an overkill at first, but when you start creating longer and more complex code, you will see how useful it can become (particularly, when the code you are using was written by somebody else in the company is not yours, or you wrote it years ago...).

Exercise 1

Using *PyCharm*, code and run the following four Python modules, where red indicates a keyword, blue a string, and green a comment. The four Python modules can also be downloaded from Moodle and the Appendix at the end of this document provides some insight into the snippets that might be useful (particularly for FIT2085 students).

hello.py

```
name = input("Enter␣name␣(max␣60␣chars):␣")
print("Hello␣" + name + ".␣Welcome")
```

convert_temperature.py

```
temperature_Fahrenheit = int(input("Enter␣temperature␣in␣Fahrenheit␣"))
temperature_Celsius = int(5*(temperature_Fahrenheit-32)/9)
print("Temperature␣in␣Celsius␣is␣" + str(temperature_Celsius))
```

leap_year.py

```
year = int(input('Enter␣year:␣'))
leap_year = False

if year %4 != 0:
    leap_year = False
elif year % 100 != 0:
    leap_year = True
elif year % 400 != 0:
    leap_year = False
else:
    leap_year = True

if leap_year:
    string = ""
else:
    string = "␣not"
# a simpler form would be: string = "" if leap_year  else " not"

print(year, "is" + string, "a␣leap␣year")
```

construct_list.py

```
size = int(input("Enter␣number␣of␣values:␣"))
the_list = []

for i in range(size):
    the_list.append(int(input("Value:␣")))

print(the_list)
```

- Run these modules in interactive mode as well.
- Run the same code using a Jupyter Notebook.
- Write down the differences you have noticed between running the code in different ways.

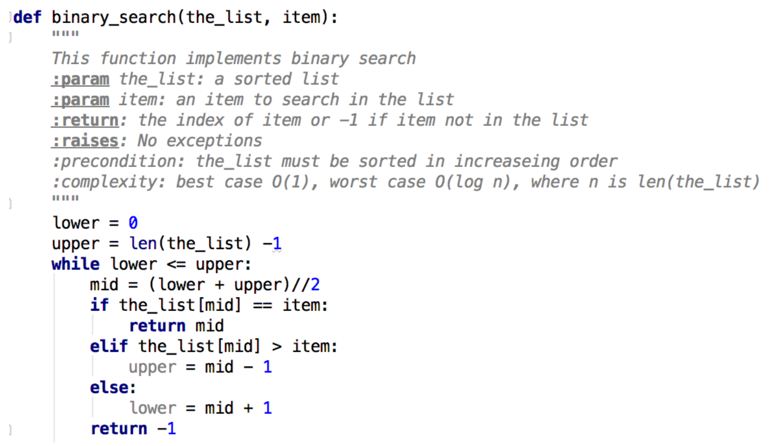
Note that one could have coded the functionality provided by these files in many other ways, some clearer than those provided. For example, `leap_year.py` was coded like that to illustrate the use of cascading if-then-elses, but the following code would be considered clearer and more modular (as it encapsulates the conditions required for a year to be a leap year):

```
year = int(input('Enter␣year:␣'))

if is_a_leap_year(year):
    print(year, 'is␣a␣leap␣year')
else:
    print(year, 'is␣NOT␣a␣leap␣year')

def is_a_leap_year(year):
    return ((year % 4 == 0) and (year % 100 != 0)) or (year % 400 == 0)
```

Exercise 2

As mentioned before, and unless explicitly stated, you must provide documentation for each program you write in this unit[a]. The documentation must be written with the code, not in a separate file. Later in the semester, we will introduce testing and you will need to provide that as well. To the right you will find one example with all the required items.

For the second `leap_year.py` Python module given in Exercise 1, add documentation using docstrings[b].



```python
def binary_search(the_list, item):
    """
    This function implements binary search
    :param the_list: a sorted list
    :param item: an item to search in the list
    :return: the index of item or -1 if item not in the list
    :raises: No exceptions
    :precondition: the_list must be sorted in increaseing order
    :complexity: best case O(1), worst case O(log n), where n is len(the_list)
    """
    lower = 0
    upper = len(the_list) -1
    while lower <= upper:
        mid = (lower + upper)//2
        if the_list[mid] == item:
            return mid
        elif the_list[mid] > item:
            upper = mid - 1
        else:
            lower = mid + 1
    return -1
```

[a]See Guidelines here: `http://bit.ly/style_python`

[b]**PyCharm**: triple quote ⏎ , then add best/worst case complexity (if you know it), exceptions and pre-conditions

Exercise 3

Later in the unit you will implement various data types and you might want to use a menu to access their functionality. Consider the following (glaringly uncommented) code, which uses a menu to append items to a list, print it, reverse it and quit.

```python
def print_menu():
    print('\nMenu:')
    print('1. append')
    print('2. reverse')
    print('3. print')
    print('4. quit')

def reverse(my_list):
    length = len(my_list)
    for i in range(length//2):
        temp = my_list[i]
        my_list[i] = my_list[length-i-1]
        my_list[length-i-1]=temp

my_list = []
quit = False
input_line = None

while not quit:
    print_menu()
    command = int(input("\nEnter command: "))

    if command == 1:
        item = input("Item? ")
        my_list.append(item)
    elif command == 2:
        reverse(my_list)
    elif command == 3:
        print(my_list)
    elif command == 4:
        quit = True
```

Extend the above code so that a user can perform the following commands on `my_list` using the menu:

- `pop`: removes the last item of the list, and prints it.
- `count`: prints the number of times a given value appears in the list.

Make sure you use good code layout, with consistent and meaningful variable names, and you modularise if appropriate. Commenting the methods you create to implement the above functionality is not required for this exercise, but if you want to do it, even better.

IMPORTANT: as discussed in the first lecture, this unit is not about Python but about the basics of computer science, like data structures and algorithms. Thus, we want you to exercise those basics (like loops, traversing arrays backwards and forwards, etc), rather than use already-made libraries and Python-only features that hide the basics. That is, we want you to code like we did for the `reverse` command above (which defines a method which uses a loop to reverse), rather than simply calling the methods provided by Python for lists. Sometimes this will be OK (as we, for example, did for the `append` method), because coding it by yourself would either be too difficult/time consuming at this point (for example, or coding append would require you to know how to increase the size of the list), or does not help the learning of the prac. But in general, we will like you to make your own loops (rather than calling a library function) and avoid using things like list slicing, map, negative indices for accessing the list elements, etc, unless you are in the advanced questions (in those ones you are free to do whatever you want!). The prac sheets will usually make it clear what you are allowed to use or not, but when in doubt: use what is easiest for you and send a query to the forums. When you get the answer, change your code if needed. That way, you will not waste time coding something that is not needed.

For the above exercise, you are not allowed to use negative indices or slicing. Do your own loops :-)

Exercise 4

This is the code review task and the code here should be appropriately commented

If you are in FIT2085, you have already worked during Tute 1 on how to convert a number **n** in decimal format to hexadecimal. If you are not in FIT1008, and you do not remember how to do this, you can have a look at the tutorial solutions for FIT2085 in Moodle.

In any case, the method to convert **n** to binary is very similar: divide **n** by 2 (integer division), take the reminder and start again until the result of the division is 0. The binary value is he sequence of reminders from last to first. For example, assume you want to convert the decimal integer 18 into binary representation: first divide 18 by 2, obtaining 9 with remainder 0; then divide 9 by 2, obtaining 4 with reminder 1; then divide 4 by 2, obtaining 2 with reminder 0; then divide 2 by 2, obtaining 1 with reminder 0; then divide 1 by 2, obtaining 0 with reminder 1. Since we obtained 0, the loop finishes, returning the string ”10010” as the binary representation of decimal number 17.

Implement a program that provides a menu to convert numbers from decimal representation into hexadecimal and binary. Your menu should use 1 for converting a number in decimal representation to binary, 2 for converting to hexadecimal, and 3 for quitting. For options 1 and 2 it should also print the result.

As usual, make sure you use good code layout, with consistent and meaningful variable names, and you modularise if appropriate. This time you are required to provide accurate and informative comments.

Note: the method to transform a decimal number to octal (base 8) is the same as for hexadecimal (base 16) and binary (base 2). The only difference is the base use for the division (8 for octal, 16 for hexadecimal and 2 for binary), and the fact symbols other than digits need to be used when the reminder exceeds 9 (that is, the base is 10 or higher, as is the case for hexadecimal). If you have time (not required for getting the marks) you might want to implement a method that converts any decimal number into any base, both provided as input.

Advanced Question 1

For those who have been coding for a while and like a bit of a challenge, here are some advanced questions. These questions do not need to be completed to get marks for the code review (all the above questions do need to be completed to get a mark).

Consider a puzzle composed of sliding blocks, as shown in the figure. Any block adjacent (*left*, *right*, *above* or *below*) the blank spot can be moved into the position of the blank spot. No diagonal moves are allowed.

The objective of the puzzle is to be able to rearrange the numbers into various patterns. For example, can you put the numbers in reverse order or if the numbers have been jumbled up and you put them back into the standard configuration? (see figure below)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Assume the puzzle is represented in terms of a lists of list, and value `None` is used to mark the blank spot.

```
puzzle = [[ 1,  2,  3,  4],
          [ 5,  6,  7,  8],
          [ 9, 10, 11, 12],
          [13, 14, 15, None]]
```

Write a Python program that allows the user to move the blank square up, down, left, and right. At each step the program should display the current state of the puzzle. If the movement is not possible, print a message for the user.

Advanced Question 2

For those who have the time and inclination, modify the above code to do the following:

- Present the user with a randomly initialised configuration
- Modify the menu so that only the allowed movements are shown
- Detect when the user has won (the configuration is the sorted one) and congratulate them.

Advanced Question 3

For those who have even more time and inclination, modify the above code to allow users to undo as many movements as they want. You will need to keep the list of movements (in what should be a stack data type, you will learn about this in the future).

Appendix (mainly for FIT2085 students)

Some things to note about the code snippets above:

- The first line of module **`hello.py`** does two things: it calls a function named **`input`** and stores its output in a variable called **`name`**.
- Function **`input`** is what's called a "built-in" function. It's part of Python proper and available for all and every program. There are other ways for you to acquire functions (you may import them from the Python standard library, you may install additional packages, or you may write them yourself).
- Calls to **`input()`** are usually followed by a call to string method **`strip()`**. We have not done this here for simplicity (it should have been **`name = input(Enter name (max 60 chars): ").strip()`**), but you should in the future. This is how it works: **`input()`** returns a string which may have leading and trailing spaces. We don't want those, and the method **`str.strip()`** strips off whitespace on either end of a string. So a Python line like this: **`a = input("Something:  ").strip()`** asks for input with the prompt "Something", and then it cleans off spaces on either side of the input string *before* it references that string with the variable name **`a`**.
- We could have made '**`hello.py`** more robust by adding a test after reading the name as follows:

```
name = input("Enter␣name␣(max␣60␣chars):␣")
if not name:
    name = "Anonymous␣Student"
print("Hello␣" + name + ".␣Welcome")
```

  Note how Python evaluates the expression **`not name`** as **`True`** for the empty string, and **`False`** for any other string. This is because, in Python as in JavaScript, any empty string will take the **`False`** value when used in a boolean context such as a conditional. The expression **`if not name`** is equivalent to **`if not name == ""`** (that is: if not(name equals empty string))[1].
- Note also how we use **`not`** (rather than !) for negating the boolean value of an expression in, for instance, a conditional.
- The code that assigns the value "Anonymous Student" to the **`name`** only executes if the expression in the **`if`** clause is **`True`**. Note that the line starting with **`if`** ends in a colon, and that the next line is indented. As mentioned in the lecture, a series of indented lines is called a block, and they are used

[1]Among Python programmers the expressions `if name` and `if not name` are said to be more "pythonic".

in creating conditionals, loops, functions and classes. Python requires neither brackets nor coloured shapes for marking blocks. Indentation does it.
- The call to `int(x)` function converts the number or string `x` to an integer. It is also a builtin, and one of the many available for type conversion.