EE2703 - Week 1

Srikar Babu Gadipudi EE21B138

February 4, 2023

1 Prerequisites to run the notebook

Any platform to run jupyter notebooks and a working python script should be enough to run the notebook. Libraries like numpy would be expected.

2 Document metadata

> *Problem statement: modify this document so that the author name reflects your name and roll number. Explain the changes you needed to make here. If you use other approaches such as LaTeX to generate the PDF, explain the differences between the notebook approach and what you have used.*

2.1 Explanation

We change the author name in the raw json file to name and roll number. We can also change the title of the document in a similar manner. Apart from changing the author we can also rename the notebook to whatever we want from the rename interface in the jupyter lab environment.

When using LaTeX to generate the PDF, we can similarly see a author section that can be changed as per requirement.

3 Basic Data Types

Here we have a series of small problems involving various basic data types in Python. You are required to complete the code where required, and give *brief* explanations of your answers. Remember that the documentation and explanation is as important as the answer.

For each of the following cells, first execute them, and then give a brief explanation of why the answer comes out to be the way it does. If there is an error during execution of the cell, explain how you fixed it. **Add a new cell of type Markdown with the explanation** after the corresponding cell. If you are using plain Python, add suitable comments after each line and explain this in the documentation (clearly you would be better off using Notebooks here).

3.1 Numerical types

[1]:
```
print(12 / 5)
```

```
2.4
```

3.1.1 Explanation

This is a basic division operation, where two `int` datatype numbers are divided to genearte a `float` datatype value.

[2]:
```
print(12 // 5)
```

```
2
```

3.1.2 Explanation

This is floor division operation. Two `int` datatype numbers are divided and the floor of the resulting value is returned. The output of this operation is always an `int`.

[3]:
```
a=b=10
print(a,b,a/b)
```

```
10 10 1.0
```

3.1.3 Explanation

a and b are assigned to the value 10, which is then printed in a single line as a, b, a/b. As there is a division operation being performed, the divided value reults in a `float` datatype.

3.2 Strings and related operations

[4]:
```
a = "Hello "
print(a)
```

```
Hello
```

3.2.1 Explanation

The variable a is assigned to the `string` "Hello".

[5]:
```
print(a+b)  # Output should contain "Hello 10"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 print(a+b)  # Output should contain "Hello 10"

TypeError: can only concatenate str (not "int") to str
```

3.2.2 Explanation

An error is observed. This is because of the `int` datatype of variable b which cannot be concatenated to the string a to print the resulting string. We cannot add an `int` datatype variable to a `string` datatype variable.

[6]:
```
print(a + str(b))
```

```
Hello 10
```

This cell is the fix to the error thrown above. We use typecasting to convert `int` to `string` datatype, using the function `str()`. The `str()` converts any datatype to a string.

[7]:
```
# Print out a line of 40 '-' signs (to look like one long line)
# Then print the number 42 so that it is right justified to the end of
# the above line
# Then print one more line of length 40, but with the pattern '*-*-*-'
print("-"*40)
print(f"{42:>40}")
print("*-"*20)
```

```
----------------------------------------
                                      42
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

3.2.3 Explanation

Use a simple multiply string to generate the required number of `-` and print *42* in the next line by right justifying it using formatting techniques. The formatting is done using `{:}`, for left justified text, we use `<` and for right justified text, we use `>`. The number of spaces to be filled in is given by the number after `</>` symbols. Here we right justified the *42* by 40.

Then we print the pattern `*-` 20 times to fill up the line of lenght *40*.

[8]:
```
print("-"*40)
print(f"{42:>42}")
print("*-"*20)
```

```
----------------------------------------
                                        42
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

This cell justifies *42* such that it is right justified to the end of the line drawn above it.

[9]:
```
print("-"*40,42, sep='')
print("*-"*20)
```

```
----------------------------------------42
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

This cell prints 42 at the end of the series in the same line without any space.

[10]:
```
print(f"The variable 'a' has the value {a} and 'b' has the value {b:>10}")
```

```
The variable 'a' has the value Hello  and 'b' has the value         10
```

3.2.4 Explanation

The `{}` notation is used to add in values of variables and `:<>` is used to add in spaces for left or right justification of text.

[11]:
```
# Create a list of dictionaries where each entry in the list has two keys:
# - id: this will be the ID number of a course, for example 'EE2703'
# - name: this will be the name, for example 'Applied Programming Lab'
# Add 3 entries:
# EE2703 -> Applied Programming Lab
# EE2003 -> Computer Organization
# EE5311 -> Digital IC Design
# Then print out the entries in a neatly formatted table where the
# ID number is left justified
# to 10 spaces and the name is right justified to 40 spaces.
# That is it should look like:

# EE2703                       Applied Programming Lab
# EE2003                         Computer Organization
# EE5131                             Digital IC Design

lis = [{"id" : "EE2703", "name" : "Applied Programming Lab"},
       {"id" : "EE2003", "name" : "Computer Organization"},
       {"id" : "EE5131", "name" : "Digital IC Design"}]

for i in range(len(lis)):
    print(f"{lis[i]['id']:<10}{lis[i]['name']:>40}")
```

```
EE2703                       Applied Programming Lab
EE2003                         Computer Organization
EE5131                             Digital IC Design
```

3.2.5 Explanation

Create a list of dictionaries with two keys. And print these values in a tabular format using `{}` formatting.

The dictionary has two keys - *id* and *name*. The *id* is the course number which is printed with left justification of *10*, and *name* is the name of the course which is printed with right justification of *40*.

4 Functions for general manipulation

[12]:
```
# Write a function with name 'twosc' that will take a single integer
# as input, and print out the binary representation of the number
# as output.  The function should take one other optional parameter N
# which represents the number of bits.  The final result should always
# contain N characters as output (either 0 or 1) and should use
# two's complement to represent the number if it is negative.
```

```
# Examples:
# twosc(10): 0000000000001010
# twosc(-10): 1111111111110110
# twosc(-20, 8): 11101100
#
# Use only functions from the Python standard library to do this.
def twosc(x, N=16):
    #check whether x is positive or negative
    if x >= 0:
        #convert to binary
        rep = bin(x)
        #slicing the string
        bin_rep = rep[2:]
        #adding '0's to get the required length
        if len(bin_rep) <= N:
            bin_rep = bin_rep.zfill(N)
    else:
        #convert to binary
        rep = bin(-x)
        #string slicing
        bin_rep = rep[2:]
        #adding '0's to get the required length
        if len(bin_rep)<= N:
            bin_rep = bin_rep.zfill(N)
        #taking 2's complement of the positive value to get the representation␣
 ↪of negative number
        arr = []
        #1's complement
        for i in range(len(bin_rep)):
            arr.append(int(bin_rep[i]))
            if arr[i] == 1:
                arr[i] = 0
            else:
                arr[i] = 1
        #adding 1 to get 2's complement
        for i in range(len(bin_rep)):
            if arr[len(bin_rep) - 1 - i] == 0:
                arr[len(bin_rep) - 1 - i] = 1
                break
            else:
                arr[len(bin_rep) - 1 - i] = 0
        #to convert array into a string using join function
        bin_rep = ''.join(str(i) for i in arr)

    #returning the string of required length
    return bin_rep[len(bin_rep) - N:]
```

[13]:
```
print(twosc(10))
```

```
0000000000001010
```

[14]:
```
print(twosc(-10))
```

```
1111111111110110
```

[15]:
```
print(twosc(-20, 8))
```

```
11101100
```

4.0.1 Explanation

The above code displays any `int` number in its binary form of specified number of bits. It also represents negative values by its 2's complement representation.

With the number and the number of bits as the arguments to the function. The code first checks whether the number is positive or negative. If found positive, it is converted to the binary form using the function `bin()` which returns a string with *0b* in the beginning. It is filtered out using string manipulation, then it is filled with zeros using the `zfill()` function to match the number of bits given in the argument. If the number found is negative, it represented as the binary form of its positive number. And the 2's complement is taken by taking 1's complement and then adding 1 to this representation.

This gives the binary representation of the given number.

5 List comprehensions and decorators

[16]:
```
# Explain the output you see below
[x*x for x in range(10) if x%2 == 0]
```

[16]:
```
[0, 4, 16, 36, 64]
```

5.0.1 Explanation

A list of squares of even numbers till 10 (10 not included) is generated. This condensed representation of for loop running through `range(10)` and the condition `x%2==0` (even number) to append to the list as `x*x`. This is an example of list comprehension.

[17]:
```
# Explain the output you see below
matrix = [[1,2,3], [4,5,6], [7,8,9]]
[v for row in matrix for v in row]
```

[17]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5.0.2 Explanation

A 1D list is generated by appending through rows one after the other. The *row* vaiable runs through every row in the matrix (individual 1D list) and the *v* variable runs through these rows.

[18]:
```
# Define a function `is_prime(x)` that will return True if a number
# is prime, or False otherwise.
# Use it to write a one-line statement that will print all
# prime numbers between 1 and 100
def is_prime(x):
    for i in range(2, int(x/2) + 1):
        if (x%i) == 0:
            return False
    return True
```

5.0.3 Exaplanation

The above function returns boolean value whether the given number is a prime number or not. It is designed using a basic *for* loop through the required range.

[19]:
```
print([i for i in range(1, 100) if is_prime(i)])
```

```
[1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
73, 79, 83, 89, 97]
```

5.0.4 Explanation

List of all prime numbers under 100 are generated as a list using list comprehension. Here the previously defined `is_prime` function is used, to check whether a number is prime or not.

[20]:
```
# Explain the output below
def f1(x):
    return "happy " + x
def f2(f):
    def wrapper(*args, **kwargs):
        return "Hello " + f(*args, **kwargs) + " world"
    return wrapper
f3 = f2(f1)
print(f3("flappy"))
```

```
Hello happy flappy world
```

5.0.5 Explanation

The wrapper function in f2 takes f, another function as its argument. when f3 called as f2(f1) with argument “flappy”, f2 runs the wrapper function returning “Hello” and calling f1 function with argument “flappy” that returns “happy flappy” and finally f2 adds in ” world“. In conclusion, the function f3 called in such a manner returns”Hello happy flappy world”.

[21]:
```
# Explain the output below
@f2
def f4(x):
    return "nappy " + x
```

```
print(f4("flappy"))
```

```
Hello nappy flappy world
```

5.0.6 Explanation

A decorator is used to call function f2 with f4 as its argument. When f4 is passed in the argument "flappy" it generates "nappy flappy" which is then passed into function f2, that generated the string, inside wrapper function, "Hello nappy flappy world".

6 File IO

[22]:
```
# Write a function to generate prime numbers from 1 to N (input)
# and write them to a file (second argument).  You can reuse the prime
# detection function written earlier.
def write_primes(N, filename):
    f = open(filename, "w")
    for i in range(2, N+1):
        if is_prime(i):
            f.write(str(i)+ " ")

    f.close()
    return
```

6.1 Explanation

A new file is generated in write mode where all the prime numbers till N are written into. The `is_prime` function is used to check whether number is prime or not.

[23]:
```
write_primes(20, "some.txt")
```

A test cell to check the working of the above function.

7 Exceptions

[25]:
```
# Write a function that takes in a number as input, and prints out
# whether it is a prime or not.  If the input is not an integer,
# print an appropriate error message.  Use exceptions to detect problems.
def check_prime(x):
    try:
        y = int(x)
        if is_prime(y) and y > 0:
            print("%i is prime number" %y)
        elif y<=0:
            print("%i is a non-positive number" %y)
        else:
            print("%i is not prime" %y)
    except:
```

```
        print("%s is not an int" %x)
x = input('Enter a number: ')
check_prime(x)
```

```
Enter a number:  13
```

```
13 is prime number
```

7.1 Explanation

We use exception handling to detect whether the given input is a prime number or not. We try to typecast to `int`, if it fails then the except block is run to prompt the user that the input is not an integer. The `try` and `except` commands are used to check whether an error is raised or not. Here if the enetered input is not an `int` then the user is prompted that the input is not a valid. If the number is negative, then the user is prompted accordingly. Only when the user enters a positive integer, the fucntion checks whether this number is prime or not using `is_prime` function.