EE2703 - Week 1

ANIRUDH B S (EE21B019)

February 1, 2023

1 Document metadata

The changes that needed to be made to modify the author name firstly included to generate the TeX file using Jupyter Notebook. Go to File, Download As and select LaTeX. In the TeX file, change the author name to ANIRUDH B S (EE21B019). I would prefer this approach of generating the TeX file first and then generate the PDF as I would get another chance to look over any changes which I could make to the TeX file to generate the PDF. This notebook consists of almost everything that has been covered in the PDF.

2 Basic Data Types

2.1 Numerical types

[1]:
```
print(12 / 5) # Simple Division
```

```
2.4
```

2.1.1 Reason

Python is a language that interprets data types differently. It can go about converting from integer to float very easily. Actual division of 12 by 5 gives answer 2.4 and python being a language that juggles around with data types converts 12 (int) and 5 (int) to give 2.4 (float). Languages like C or C++ would have given a result of 2 which is quite intuitive as both the numbers are integers and output also needs to be an integer.

[2]:
```
print(12 // 5) #Floor Division
```

```
2
```

2.1.2 Reason

The // operator is essentially used to do floor division. That is, if we need an integer output after division rounded down then we make use of the // operator. For example, 4//3 is 1 and not 1.3333.

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

```
10 10 1.0
```

2.1.3 Reason

- a and b are initialised as integers and thus they remained as integers as they are not hampered with in the problem. Thus a is 10 and b is also 10
- As the case with Cell No.1, the result of division is a floating number. Thus 10/10 gave 1.0 as against 1, which we definitely can get by casting it to int. The result of division is always a **floating point number in Python**.

2.2 Strings and related operations

[4]:
```
a = "Hello "
print(a)  #Changing of data type from int to str
```

```
Hello
```

2.2.1 Reason

Python is a very robust and versatile language. A variable once used as int can again be used as a str or list or any other data type. Thus this conversion is permitted in python as against orthodox OOPs like C or C++ which do not permit such a conversion.

[5]:
```
b = '10' #Changing data type from int to str for concatenation
```

2.2.2 This cell has been added.

The cell has been added by me because we need the output to be "Hello 10". Python cannot concatenate a str with an int and both the inputs to be concatenated needs to of str data type. Note that python doesn't use anything called string. String as in conventional C or C++ is a clearly described data type called str in python.

[6]:
```
print(a+b)  # Output contains "Hello 10"
```

```
Hello 10
```

2.2.3 The original block had a bug !

The bug was because b was not a str variable. However, adding the cell b= '10' will result in the output being printed as "Hello 10".

This is a nice example of operator overloading. The '+' operator can be used for concatenation of two str variables or addition of two ints or floats. As mentioned earlier, a cell has been added as we cannot concatenate a str and int object. We need both str and this has been done in the preceding cell.

[7]:
```
for i in range(40):
    print('-', end = "") #Print 40 minus signs in a row.
print (42) #Now we print 42 to right justify the number as asked in the question
for j in range(40):
    if j%2 == 0 :
        print ('*', end = "") #Prints * and - alternatively as described
    else :
```

```
        print('-', end = "")
```

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

[2]:
```
for i in range(40):
    print('-', end = "") #Print 40 minus signs in a row.
print()
for k in range(40):
    print(' ', end = "")#Print 40 blank spaces
print (42) #Now we print 42 to right justify the number as asked in the question
for j in range(40):
    if j%2 == 0 :
        print ('*', end = "") #Prints * and - alternatively as described
    else :
        print('-', end = "")
```

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

2.2.4 Why two cells ?

There was slight ambiguity with reference to the question. The right justify of 42 was with respect to the line of 40 '-'s or a new line of 40' ' was not clear. Thus, I have printed both the possibilities.

The idea for this has been taken from the documentation of python. The print function has following parameters according to the documentation. - objects - Object that needs to be printed - sep - Indicates the separator between words. Generally it is white space - end - Indicates how the line should end. Typically it ends with ''

The format of print is (objects, sep = ' ', end =")

[8]:
```
print(f"The variable 'a' has the value {a} and 'b' has the value {b:>10}") #Use␣
↪of f strings
```

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

2.2.5 Explanation : F strings to format text

F strings stand for formatted strings. Python allows us to use f strings which allows us to plug in values directly into the output stream. This is unique to python, we could also use concatenation of strings to generate the ouptut, but this is a really new and elegant way of writing code. Also from this, we are able to conclude that after 10 spaces b is printed.

[9]:
```
l = []  #Creation of empty list
b = {}
c = {}
d = {}  #Creation of empty dictionaries
```

```
b["id"] = "EE2703"                          #Adds id = EE2703 and corresponding␣
 ↪course name to d.
b["name"] = "Applied Programming Lab"
l.append(b)
c["id"] = "EE2003"                          #Adds id = EE2003 and corresponding␣
 ↪course name to d.
c["name"] = "Computer Organization"
l.append(c)
d["id"] = "EE5311"                          #Adds id = EE5311 and corresponding␣
 ↪course name to d.
d["name"] = "Digital IC Design"
l.append(d)
for elem in l:
    print(elem["id"], end='') #Print course ID
    for j in range((10-len(elem["id"]))+(40-len(elem["name"]))): #Print spaces
        print(' ', end = '')
    print(elem["name"]) #Print course name
```

```
EE2703                          Applied Programming Lab
EE2003                            Computer Organization
EE5311                                Digital IC Design
```

2.2.6 Reason

I have used separate dictionaries to store each id and corresponding name since if I use only one dictionary, the list will contain only EE5311 and Digital IC Design as its three elements. This is because Python maintains a reference to the address of the dictionary, so once after modification of the dictionary even after appending to the list would result in that value getting altered.

The above method of printing has been adopted as it looks more friendly to the reader. As apparent from the comments in the code, we have utilised exactly 50 characters including the id, name and blank space. As mentioned earlier, the documentation of Python provides us with an opportunity to change how a sentence printed on stdout could end in using end = ' ' as done above.

The complexity of the code is O(k) where k is the number of keys present in the dictionary. This is true because we print exactly 50 letters each time. So the constant 50 can be dropped out in determining the time complexity. In my opinion, this is a good time complexity and thus the method.

3 Functions for general manipulation

[10]:
```
def twosc(x, N=16):
    l = []
    if x >= 0 and x < pow(2, N-1): #Prints binary representation for positive␣
 ↪number
        while x >= 1 :
            l.append(x%2)          #We store remainder in a list. This list has␣
 ↪the binary representation in reverse order
```

```
            x = int(x/2)
        for _ in range(N-len(l)) :
            print(0, end='')        #Print 0s to required precision.
        for v in reversed(l):
            print(v, end='')
    elif x>= -pow(2,N-1) and x<0: #Prints binary representation for negative␣
↪number
        y = -x                      #Find binary representation of abs(x)
        count = 0
        while y >= 1 :
            if count == 0 :
                l.append(y%2)     #Before occurance of first 1, 0s are 0s.
                if y%2 == 1 :
                    count  = 1    #Find occurance of first 1, post this␣
↪complement everything.
            elif count == 1 and y%2 == 0 :
                l.append(1)       #After occurance of first 1, 0s become 1 and␣
↪1s become 0s
            else :
                l.append(0)
            y = int(y/2)
        for _ in range(N-len(l)) :
            print(1, end='')
        for v in reversed(l):
            print(v, end = '')    #Print 0s to required precision.
    else : #Prints error message for numbers having bits not in range
        print("x needs more than N bits to be represented fully.")
    print()
```

The overall time complexity of the above algorithm is O(log (x) + N) which seems to be a pretty good complexity. Thus, I have chosen to implement the above function. The gist of the function is as follows : We first check if the number is positive or negative - - If x is positive, we recursively compute the remainder, append it to a list and divide by 2 until the number is less than 1. This will give the binary representation in reverse order. Now for N- len(list) times I need to insert 0s in the begining and then print out the binary representation. - If x is negative, I will take the absolute value and carry out the same. However here I will check for the instance of the first 1 in the list what I obtain (this is in reverse order of binary represetation), I will keep 0s as it is and the first 1 as it is and then complement every bit after reaching first 1. This is how we obtain the two's complement of a number. Similarly we print out the number to N bits.

Assumption - I have considered N to be always big enough to fit x in the above cases. In the code however, I have added a check too. The condition for x to be fitted in N bits is -2^(N-1) <= x < 2^(N-1) as we are using signed two's complement representation.

[11]:
```
twosc(10)
twosc(-10)
twosc(-20, 8)
```

```
0000000000001010
1111111111110110
11101100
```

The above box is just a box to check whether the code works fine or not.

4 List comprehensions and decorators

[12]:
```
[x*x for x in range(10) if x%2 == 0]
```

[12]:
```
[0, 4, 16, 36, 64]
```

4.0.1 Explanation for the output

Python being a very versatile language converts data types on the fly. The above code block essentially tells the computer to store the square of even numbers in range 1 to 10 in a list.

4.0.2 How did we deduce this ?

x * x denotes the normal squaring operation. The for x in range (10) tells us that for all x in 1 to 10, store x * x in a list if x%2 is zero, implying that it is even.

[13]:
```
matrix = [[1,2,3], [4,5,6], [7,8,9]]
[v for row in matrix for v in row]
```

[13]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.0.3 Explanation for the output

The row in matrix tells us that for each row in the matrix, i.e. [1,2,3], [4,5,6], and [7,8,9], iterate over all v in that row and put them in a list.

The crux of this code block is that you store every a_{ij} of matrix in a list starting from (1,1) (1,2) ... (3,3) elements of the matrix.

[14]:
```
def is_prime(x):  #is_prime(x) will return True if a number is prime or False␣
↪otherwise
    if x <= 0 :
        return False
    if x == 1:
        return False
    for i in range (2, x):
        if x%i == 0 :
            return False
    return True
```

4.0.4 Reason for choosing the above approach

Since we were told to use only available basic functions in the Python Library, I have come up with a O(n) algorithm that scans through all the elements (2,3,4...x-1) checks if any of the numbers

divides x or not and then decides whether x is prime or not.

[15]:
```
for k in range(1, 101): #Prints prime numbers from 1 to 100 on a line
    if(is_prime(k)) : #Checks if number is prime and then prints it
        print (k, end = '')
        print (" ", end = '')
```

```
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
```

We are just iterating over the numbers from 1 to 100 and printing out all the primes from 1 to 100 on a same line using the documentation of python as explained earlier.

4.1 Alternate Approach (Use of Numpy)

[16]:
```
from numpy import sqrt
def is_primeagain(x):
    if x <=0 :
        return False
    if x == 1:
        return False
    for i in range (2, int(sqrt(x))+1): #This is permitted only if numpy is␣
 ↪imported
        if x%i == 0 :
            return False
    return True
```

[17]:
```
is_primeagain(4)
```

[17]:
```
False
```

4.1.1 This cell has been added

The purpose of this cell is to convey the message that a faster algorithm exists for finding whether a number is prime or not. The complexity of the above algorithm is O($\sqrt{n}$) as against O(n) as described earlier. However in getting this, we employ the sqrt function in math library of python.

4.1.2 Intuition behind the algorithm

We know a number is prime iff its divisors are only 1 and itself. To check this, we only need to check if (2,3,4... $\sqrt{N}$) divide N or not. In case a number divides, we are done, its not prime. If any number does not satisfy also we are done as proved qualitatively here. Suppose you get two divisors of N, both greater than $\sqrt{N}$, then by definition, the product of the divisors must be N. However, it is quite clear that the product exceeds N and thus, if any divisor exists, it exists in (2, $\sqrt{N}$)

[5]:
```
def f1(x):
    return "happy " + x
def f2(f):
    def wrapper(*args, **kwargs):  #Functional programming in python
        return "Hello " + f(*args, **kwargs) + " world"
```

```
    return wrapper
f3 = f2(f1)
print(f3("flappy"))
```

```
Hello happy flappy world
```

4.1.3 Explanation

- – args - Represents the fact that the programmer does not know how many arguments are going to be passed. Also termed as arbitrary arguments.
- ** kwargs - Represents the fact that the programmer does not know how many key words are going to be passed. Also termed as arbitrary key word argument. Python uses a dictionary to implement the kwargs functionality. Python allows functional programming, that is we can pass functions as parameters to a function and also return functions.

In this cell, we define a function f1(x) and then pass it to f2(). In f2(), we define a wrapper function that concatenates the strings. The statement ("Hello"+ f(* args, ** kwargs) + " world") is equivalent to writing"Hello " + "happy" + x + " world". Thus when we declare f3 to be f2(f1) and pass"flappy", we are essentially doing the following - f2(f1("flappy")), which as from above results in"Hello happy flappy world"

[6]:
```
@f2  #Use of Wrapper Functions
def f4(x):
    return "nappy " + x

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

```
Hello nappy flappy world
```

4.1.4 Explanation - Decorators in Python

- The wrapper function or the decorators in Python take in a function and give out a function.
- These aspects of Object Oriented Programming make Python a very unique, versatile and robust programming language. The wrapper functions promote data encapsulation. This form of functional programming is used to validate input data in many applications.

Here, we first tell the interpreter that we are wrapping the function f2() around f4() by using @ . So, f4 is the function that is passed on as parameter to function f2. The difference being here that we would not need to call f2. Since it sets the value of f2, it is automatically called. So, the first three lines of code in the block essentially translates to f2(f4(x)) which is "Happy" + "nappy" + x + "world" which is "Happy nappy" + x + "world". Thus when we call f4("flappy") the output is "Happy nappy flappy world" as it should be from the explanation given above.

5 File IO

[20]:
```
def write_primes(N, filename):
    f = open(filename, "w")   #Opens the file passed as parameter.
    for i in range (1, N+1) :
```

```
        if is_prime(i) :   #Prints the prime numbers on separate lines in the␣
↪mentioned file.
            f.write(str(i)+'\n')
    f.close() #We close the file after the operation is complete
```

5.0.1 Description of the code

We open the file in which we write the primes in. Firstly, it is assumed that the file is blank. In case it is not and contains some other details which must not be erased (since opening file in "w" mode erases everything) we can open in append ("a") mode. We then iterate over all the numbers from 1 to N and print out the primes in the given file.

[21]:
```
write_primes(100, 'prime.txt') #Call the function to print
```

5.0.2 This cell has been added

This has been added so as to generate a text file composed of primes in the range 1 to 100. This serves as a check as to whether our function is written correctly as we have already determined the primes in 1 to 100.

6 Exceptions

[22]:
```
def check_prime(x):
    try :
        y = int(x) #Checks if x is int, else throws exception
        print(is_prime(y))
    except ValueError:
        print("Enter an integer")
x = input('Enter a number: ') #When input is taken, x is always in str.
check_prime(x)
```

```
Enter a number: 7
True
```

6.0.1 Error Handling in Python

We use the try-except method of exception handling in Python. We first try to figure out any error that could take place in the try block and appropriately act depending on the type of exception raised using the except block. In the instance of an Exception (more technically, ValueError in this case), the except block exits with the statement "Enter an integer". However, if the input is a valid integer, we then proceed to check if it is a prime number or not.