EE2703 - Week 1

Santhosh S P ee21b119

January 29, 2023

1 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 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).

2.1 Numerical types

2.1.1 Division operation

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

```
2.4
```

The above line prints the result of the division operation between the integers 12 and 5. The output is implicitly converted into a float to prevent data loss.

2.1.2 Floor division operation

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

```
2
```

The `//` operator used in the above line is the floor division operator. It rounds down the result of the division operation between the two operands to the nearest integer.

The above line prints the result of floor division of 12 by 5. The result of the normal division is 2.4, which is rounded down to 2. Therefore, the output obtained is 2.

2.1.3 The assignment operator

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

```
10 10 1.0
```

The operator `=` is the assignment operator in python. It assigns the object containing the value, which is found to the right of the operator to the variable to its left.

The first line of the above cell assigns the same value `10` to multiple variables `a` and `b`. The value `10` is first assigned to the variable `b`, then the value stored in variable `b` (`10` here) is assigned to the variable `a`.

The line `print(a,b,a/b)` prints the the value stored in variables `a`, `b` and the result of the division between `a` and `b` in a single line, seperated by spaces. In this case it displays `10 10 1.0`, as the values stored in `a` and `b` are 10 and the result of `a/b` is `1.0` because of implicit type conversion of the result into float from integer.

2.2 Strings and related operations

2.2.1 Printing strings

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

```
Hello
```

The variable `a` which earlier referred to an object containing an integer `10`, now points to an object containing a string `Hello` after the reassignment in the first line of the cell.

The second line prints the value stored in the object that the variable `a` points to, (`Hello` in this case).

2.2.2 Addition of strings

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

```
Hello 10
```

The original code given in the above cell returns a `TypeError` because concatenation is only possible on strings, whereas the variable `b` refers to an integer object containing the value `10`.

This error can be resolved and the required output `Hello 10` can be obtained if we reassign the variable `b` to an object of type `string` containing the value `10` by the statement `b="10"` before the line `print(a+b)`.

2.2.3 Multiplying strings by positive integers

[ ]:
```
# Print out a line of 40 '-' signs (to look like one long line)
print(40*"-")

# Then print the number 42 so that it is right justified to the end of
# the above line
print(f"{42:>40}")

# Then print one more line of length 40, but with the pattern '*-*-*-'
print(20*"*-")
```

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

Multiplication of a string by an integer (say `n`), results in repeated concatenation of the string by `n` times.

In the line `print(40*"-")` concatentates the given string "-" 40 times, to produce the resultant string (a line of 40 '-' signs), and prints it.

The line `print(f"{42:>40}")`, assigns a space of 40 characters to print the result (`42` in this case) aligned to the right. It prints the integer `42` right justified to the end of the previous line.

The line `print(20*"*-")` concatenates the string `*-` 20 times, to produce a alternating pattern of `*-` of total length 40 and prints it on the next line.

2.2.4 f-strings

[ ]:
```
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
```

In the above block of code, f-strings are used to format the string. These strings have a placeholder to directly insert the value of the variable in the string. In place of `{a}`, the actual object whose value is referred to by the variable name `a` (which is `Hello` in this case), is inserted in the f-string.

Similarly, in place of `{b:>10}`, the actual value of the object referred by the variable `b` (10 in this case), is inserted with right-alignment taking a total of 10 spaces in the f-string.

2.2.5 Pretty printing contents of a list of dictionaries

[ ]:
```
# 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




dict_keys = ["id", "name"]
dict_values = [["EE2703", "Applied Programming Lab"], ["EE2003", "Computer␣
 ↪Organization"], ["EE5311", "Digital IC Design"]]
dict_list = []
for value in dict_values:
    dict_list.append(dict(zip(dict_keys, value)))


for i in dict_list:
    print(f"{i['id']:<10}",end="")
    print(f"{i['name']:>40}")
```

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

In the above cell, we create a list of three dictionaries, each with two keys, `id` and `name`.

We create a list `dict_keys` containing the common keys (`id` and `name`), and a nested list `dict_values` containing three lists, each containing the `id` and `name` of a course. We also initialize an empty list `dict_list` to store them as a list of dictionaries.

In the `for` loop, we select each element of `dict_list` one at a time (say `value` in this case), and call `zip(dict_keys, value)`. It creates an iterator of tuples containing one element of each list. On applying `dict()` on it, a dictionary with keys `id` and `name` is generated. We append each one of the three dictionaries formed into `dict_list` to create a list of dictionaries.

To print the contents of the list in a neat way, we choose one element of the list (a dictionary) at once. The line `print(f"{i['id']:<10}",end="")` prints the `id` value of a single dictionary, left justified to 10 spaces. The `end=""` part makes sure that the cursor position does not change after it's done printing.

The next line `print(f"{i['name']:>40}")` prints the `name` value of a single dictionary, right justified to 40 spaces. Here, by default `end="\n"` therfore after printing the cursor automatically moves to the next line.

In a similar way, the contents of the three dictionaries are printed in a neat, table-like format.

3 Functions for general manipulation

[ ]:
```
# 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):
        if x>=0:
            return f"{x:0{N}b}"
        else:
            return f"{2**N-abs(x):0{N}b}"

print(twosc(10))
print(twosc(-10))
print(twosc(-20, 8))
```

```
0000000000001010
1111111111110110
11101100
```

In the above cell, we define a function `twosc(x,N)` to return the 2's complement of an input integer (`x`) in a given number of bits (`N`, which by default is 16).

If the input $x \geq 0$, the f-string `f"{x:0{N}b}"` is returned, which is the binary representation of x with `N` bits. If x is negative, the f-string `f"{2**N-abs(x):0{N}b}"` is returned. It is the 2's complement representation of x with `N` bits.

It is derived using the fact that,

x + 2's complement of $x = 2^N$

Also, in this case there is no need to check if the number x lies in the range of numbers that can be represented by N-bits, because it automatically adds zeroes in the front in case of extra places and in case of a lesser value of N than the required value, it returns the number with minimum possible bits in binary.

4 List comprehensions and decorators

4.1 List of square of even numbers

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

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

This list comprehension creates a list of square of even numbers in the range 0 to 9. The `range(10)` creates an immutable sequence of numbers from 0 to 9, and `x%2==0` checks if each of the numbers is even. If it's even, the square of x is calculated and added to the list.

Finally, on running the block, the required result is printed.

4.2 List comprehension on a 2-D list

[ ]:
```
# 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]
```

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

`matrix` is a 2-D list of three rows and three columns. `for row in matrix` chooses each row of the matrix, one at a time (in this case, one among [1,2,3] or [4,5,6] or [7,8,9]). `for v in row` chooses a single element of the selected row, which is added to the list according to the given list comprehension.

4.3 Printing all prime numbers between 1 and 100

[ ]:
```
# 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):
    if x<2:
        return False
    else:
        for i in range(2,x//2 +1):
            if x%i==0:
                return False
        return True

print([x for x in range(1,101) if is_prime(x)])
```

```
[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]
```

In the above cell, we define a function called `is_prime(x)` which checks if `x` is a prime number. It first checks if the number x is less than 2. If yes, it returns `False` as there are no prime numbers below 2. If the number $x \geq 2$, we define a for loop that checks if the number is divisible by any of the numbers from 2 to `x//2` (both included), to check if the number has any factors other than 1 and the number itself.

If it's divisible by any of the numbers in the given range, the function returns `False`, implying that the number is not prime. If not, the condition returns `True` at the end of the loop, implying that the number is prime.

The last line uses a list comprehension to print all the primes between 1 and 100. It picks a number from the list of integers from 1 to 100 (both included), checks if it's prime. If yes, it adds it into the list. The `print()` over the list comprehension, prints the list of the prime numbers between 1 and 100.

4.4 Function decorators

[ ]:
```
# 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
```

In the above block we define a function `f1(x)` to return the concatenated output of the given input string with the string `happy`.

We define an other function `f2(f)`, which in turn has an other function `wrapper`. The `wrapper` function returns a concatenated string containing the strings `Hello`, the output of the function f, with the same arguments provided to the `wrapper` function, and the string `world`. The function `f2` returns the function object `wrapper`.

`f3=f2(f1)` results in the assignment of the `wrapper` function object corresponding to the function `f1`, to `f3`.

The final line calls the function `f3` with the argument `flappy` which results in a call to `f2` with an argument `f1` with the argument `flappy`, which returns the string `Hello happy flappy world`.

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

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

```
Hello nappy flappy world
```

The line `@f2` before `def f4(x)`, is same as the statement `f4=f2(f4)`. This results in the assignment of the `wrapper` function object corresponding to function `f4` to `f4`.

The final line calls `f4` with the argument `flappy`, which results in a call to `f2` with an argument `f4` with the argument `flappy`, resulting in the return of the string `Hello nappy flappy world`.

5 File IO

[ ]:
```
# 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):
    with open(filename,"w") as f:
        for i in [x for x in range(1,N+1) if is_prime(x)]:
            f.write(f"{i}\n")


write_primes(200,"primes.txt")
```

The function `write_primes(N,filename)` generates prime numbers from 1 to N (both included) and writes them to a file `filename`.

It opens the given `filename` in `write` mode using a file handle named `f`. `[x for x in range(1,N+1) if is_prime(x)]` is a list comprehension to generate a list of prime numbers from 1 to N (both included). Using a for loop, we choose each element one by one and write them in a the file as a f-string along with `\n` character at the end of each line.

In the above cell, we write all the prime numbers in the range 1 to 200, to a file named `primes.txt` in the same directory as the notebook.

6 Exceptions

[ ]:
```
# 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:
        n=int(x)
        if is_prime(n):
            print(f"{n} is a prime number.")
        else:
            print(f"{n} is not a prime number.")

    except ValueError:
        print("Please enter an integer.")
```

```
x = input('Enter a number: ')
check_prime(x)
```

```
Enter a number: 23.2
Please enter an integer.
```

This cell contains a function `check_prime(x)` which checks if the given input x is a prime number using the `is_prime(n)` function defined earlier.

The line `n=int(x)` converts the string input to an integer, because by default inputs received from the user are in the form of strings. We then use the `is_prime()` function on `n` to check and print if the number is prime or not prime.

The previous three lines, throw a `ValueError` if the input can't be converted to an integer (like `x="12.3"` or `x="demostring"`), which can be caught by a `try-except` block and a required help message can be printed instead of letting the program crash due to the error.