EE2703 - Week 1

Niranjan A. Kartha <ee21b095@smail.iitm.ac.in>

February 4, 2023

1 Applied Programming Lab: Week 1 Submission

by Niranjan A. Kartha (EE21B095)

---

2 Please use the notebook!

I have used the `ipynb` file to run and test code, as well as add documentation and explanations for it. So please use the Jupyter Notebook file as it provides a more streamlined experience.

---

3 Running the notebook

I have not used any extra libraries in this submission, so any default installation of Jupyter Notebook should be able to run the code cells in this notebook.

Some of the cells are designed to throw errors on purpose, so running all cells from top to bottom **will not work**.

4 My environment setup

I have used a local installation of Jupyter Notebook on Linux to create my submission. I have also installed the jupyter vim binding extension for Jupyter Notebook.

4.1 LaTeX

I am using a local install of TeX Live (from the `texlive-most` group on the Arch Repos) and pandoc to allow Jupyter Notebook to render my notebook into a PDF.

---

5 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.

We can edit the document metadata by accessing the `Edit -> Edit Notebook Metadata` menu in Jupyter Notebook:

This lets us edit the metadata as JSON. We can edit the `"authors"` property and edit the name of the author.

```
{
  "authors": [
    {
      /* I have edited the line below */
      "name": "Niranjan A. Kartha <ee21b095@smail.iitm.ac.in>"
    }
  ],
  "kernelspec": {
    "name": "python3",
    "display_name": "Python 3 (ipykernel)",
    "language": "python"
  },
  "language_info": {
    "name": "python",
    "version": "3.10.9",
    "mimetype": "text/x-python",
    "codemirror_mode": {
      "name": "ipython",
      "version": 3
    },
    "pygments_lexer": "ipython3",
    "nbconvert_exporter": "python",
    "file_extension": ".py"
  },
  "name": "Week1Prob.ipynb",
  "title": "EE2703 - Week 1",
  "toc-autonumbering": true,
  "vscode": {
    "interpreter": {
      "hash": "b0fa6594d8f4cbf19f97940f81e996739fb7646882a419484c72d19e05852a7e"
    }
  }
}
```

---

6 Basic Data Types

Problem statement:

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).

6.1 Numerical types

6.1.1 Division

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

```
2.4
```

Python3 treats the division operator `/` as floating point division, and gives us a floating point answer. Since $\frac{12}{5} = 2.4$, Python outputs `2.4` as the answer.

6.1.2 Integer division

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

```
2
```

`//` is the integer division operator in Python. This corresponds to dividing numbers and then taking the `floor` of it. `floor(2.4) = 2` and so the above code outputs `2`.

6.1.3 Floating points

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

```
10 10 1.0
```

We can notice that the two `10`s printed do not have a decimal point, while the `1.0` does.

Since `/` represents floating point division, the `10`s are converted into `floats` before the division operation is performed. Since floating point division outputs `floats`, the `1.0` that we get is a `float`.

We can confirm this by printing the types of the values involved:

[4]:
```
print("a:", type(a))
print("b:", type(b))
print("a/b:", type(a / b))
```

```
a: <class 'int'>
b: <class 'int'>
a/b: <class 'float'>
```

6.2 Strings and related operations

6.2.1 Printing a string

[5]:
```python
a = "Hello "
print(a)
```

```
Hello
```

We can notice that `a` was already declared as `10` in the previous section of this file. Since python is a dynamically-typed language, we are allowed to change the datatypes of variables while assigning values to them, and so the above code executes even though we are assigning a `str` to a variable that was previously an `int`.

6.2.2 Concatenation

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 print(a+b)  # Output should contain "Hello 10"

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

The above code throws a `TypeError` because the string concatenation operator `+` expects both operands to be `str`s. We can fix this by converting `b` to a `str`, as follows:

[7]:
```python
print(a + str(b)) # fixed code
```

```
Hello 10
```

6.2.3 Repetition

Problem statement:

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 '*-*-*-'

[8]:
```python
print("-" * 40)
print("42".rjust(40))
print("*-" * 20)
```

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

The * operator can be used to repeat strings. We use this to repeat - 40 times and *- 20 times.

The rjust function right-justifies a string to the desired length by adding characters to its left. We use this to pad 42 with spaces on the left upto a length of 40.

6.2.4 F-strings

[9]:
```
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
```

F-strings let us insert python expressions into strings by enclosing them in {curly braces}. These expressions are executed and then replaced with the value that they evaluate to.

We can add format specifiers after python expressions by following it with a colon (:). >10 is right-align to a width of 10. This is similar to the rjust function used above.

6.2.5 Dictionaries

Problem statement:

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
```

[10]:
```
courses = [
    {
        "id": "EE2703",
        "name": "Applied Programming Lab"
    },

    {
        "id": "EE2003",
        "name": "Computer Organization"
    },

    {
        "id": "EE5311",
        "name": "Digital IC Design"
    }
]

for course in courses:
    print(f"""{course["id"]:<10}{course["name"]:>40}""")
```

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

The list `courses` contains 3 objects, each of which is a `dict`, and each of those `dicts` contains two properties.

Lists in Python can be iterated through with `for` loops as above. We iterate through the elements of `courses` and print out the values in the desired format, by using format strings.

I have used triple quotes to enclose the string inside the `print` statement since I am using single quotes to index the dictionary.

The following code which does not use triple quotes results in a syntax error:

[11]:
```
print(f"{courses[0]["id"]}") # will throw a syntax error
```

```
  Cell In[11], line 1
    print(f"{courses[0]["id"]}") # will throw a syntax error
                         ^
SyntaxError: f-string: unmatched '['
```

I have also made use of F-strings and format specifiers as in the previous block. `<10` left-justifies to a width of `10`, and `>40` right-justifies to a width of `40`.

7 Functions for general manipulation

Problem statement:

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.

```
[10]: def twosc(x, N=16):
          print(format(x & ~(-1 << N), f"0>{N}b"))
```

The line above prints the two's complement representation of an integer `x` up to `N` binary digits. I have added the explanation for the above code in the coming section.

```
[2]: twosc(-20)
```

```
1111111111101100
```

7.1 Explanation

The above implementation utilizes the fact that integers are already stored in two's complement in the computer's memory.

It first generates a bitmask of `N` `1`s, by first right shfiting `-1` (which is just a sequence of `1`s in binary) by `N` and then taking a one's complement.

Here is an example for generating a bitmask of size `16`:

```
[3]: print(bin(
         ~(-1 << 16)
     ))
```

```
0b1111111111111111
```

It then takes this bitmask and then bitwise-ANDs it with the input number. This isolates the last `N` bits of that number.

```
[4]: print(bin(
         20 &
         ~(-1 << 16)
     ))
```

```
0b10100
```

Now it formats the number, converting it to binary and left-padding it with zeroes until it is is the correct length. If it is a negative number, there will be no padding added since it is already sign-extended to the left with `1`s by Python.

```
[11]: print(format(
          20 &
          ~(-1 << 16),
          f"0>16b"
      ))
```

```
0000000000010100
```

The above implementation also works with large numbers and large values of `N` because of how Python deals with big numbers. The implementation within the language is as if there were an infinite number of 1s to the left of a negative number. So whatever the value of N is, Python will sign-extend x with ones to make our algorithm work.

If N is less than the actual number of bits used for the representation of x, the most signifcant bits are discarded, and the least significant N bits of x are printed.

---

8 List comprehensions and decorators

8.1 Basic list comprehension

Problem statement:

Explain the output you see below

[17]:
```
[x*x for x in range(10) if x%2 == 0]
```

[17]:
```
[0, 4, 16, 36, 64]
```

The above code is a list comprehension that generates a list from a range. It looks at all the integers from 0 to 9 (inclusive), and for every integer in that range which is divisible by 2, it adds the square of that integer to the list.

8.2 Nested list comprehension

Problem statement:

Explain the output you see below

[18]:
```
matrix = [[1,2,3], [4,5,6], [7,8,9]]
[v for row in matrix for v in row]
```

[18]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The list comprehension above first iterates through the rows in the matrix, and then iterates through all the elements v in the row. So the above code flattens the matrix in row-major order.

8.3 Prime numbers

Problem statement:

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

[13]:
```
from math import sqrt

# utilizes the fact that all primes above 3 are of the form 6n +/- 1
# performs a primality test with O(sqrt(n)/3) modulo operations
def is_prime(x):
    # deal with the smaller cases
    if x == 1:
        return False
    if x == 2 or x == 3:
        return True

    # get rid of all the multiples of 2 and 3
    if x % 2 == 0 or x % 3 == 0:
        return False

    # Now we check if every prime above 3 divides x. Since we do not have
    # access to a list of primes, we iterate through a larger list which
    # includes all these primes. Here, all numbers of the form 6n +/- 1.

    # we check if every number of the form 6n +/- 1 less than or equal
    # to sqrt(x) divides x
    high = int(sqrt(x)) + 2

    # loop through all the multiples of 6 up to and including
    # floor(sqrt(x)) + 1
    for i in range(6, high, 6):
        if x % (i - 1) == 0 or x % (i + 1) == 0:
            return False

    return True
```

The above function is a more optimized version of the usual way to test for primality. It utilizes the fact that all prime numbers above 3 are of the form $6k \pm 1$. It also uses the fact that if $ab = n$ for $a, b, n \in \mathbb{Z}$ where $a \leq b$, then $a \leq \sqrt{n}$.

So we only need to check divisibility by 2, 3, and numbers of the form $6k \pm 1$ which are less than or equal to $\sqrt{n}$, since if n is composite, there is guaranteed to be a number in that list which divides n.

[14]:
```
print([x for x in range(1, 100) 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]
```

8.4 First class functions

Problem statement:

Explain the output below

[21]:
```
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
```

Functions in Python are first-class citizens, meaning we can pass functions as arguments to other functions.

`f2` takes a function `f` and returns another function that calls `f` and adds `"Hello "` and `" world"` around its output.

Applying this to `f1`, the resulting function is essentially a modified version of `f1`. This is stored in `f3`. Calling this modified function results in a net behaviour of surrounding the input with `"Hello happy "` and `" world"`.

8.5 Decorators

Problem statement:

Explain the output below

[22]:
```
@f2
def f4(x):
    return "nappy " + x

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

```
Hello nappy flappy world
```

Function decorators in Python are a syntax construct that is functionally identical to the code in the previous section. It applies the function `f2` to `f4` to modify it, and the modified function is called each time instead of `f4`.

Each time `f4(...)` is called, it is like calling `f2(f4)(...)` (where `f4` in the second case here is the original function in the definition).

---

9 File IO

Problem statement:

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.

[15]:
```
def write_primes(N, filename):
    file = open(filename, 'w')
    file.write( # write to the file
        ", ".join( # join the array by comma-space
            [str(x) for x in range(1, 100) if is_prime(x)] # array of prime␣
↪numbers represented as strings
        )
    )
```

The above code opens a file in write mode and writes the array to it. The array is formatted so that the numbers are separated by a comma-space.

We can invoke the above function like this:

[24]:
```
write_primes(100, 'primes.txt')
```

The above code block writes 100 primes to a text file called `primes.txt`.

---

10 Exceptions

Problem statement:

> 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.

[16]:
```
def check_prime(x):
    try:
        x = int(x)
        if is_prime(x):
            print(f"{x} is prime")
        else:
            print(f"{x} is not prime")
    except ValueError:
        print(f"{x} is not an integer.")
x = input('Enter a number: ')
check_prime(x)
```

```
Enter a number: asdf
asdf is not an integer.
```

We use a `try` statement to catch any exceptions that are thrown in the execution of a function. the `int(...)` function throws a `ValueError` when the argument passed to it is not an integer. So, on passing any input that raises a `ValueError`, the error handler under `except ValueError:` is executed.