EE2703 - Week 1

Keshaw Choudhary <ee21b069@smail.iitm.ac.in>

February 4, 2023

0.1 Numerical types

[17]:
```
print(12 / 5)
```

```
2.4
```

The "/" operator divides floating-point values, hence it always returns a float value. In this case, the division return 2.4

[18]:
```
print(12 // 5)
```

```
2
```

The "//" operator performs integer division, which means it produces a whole number while discarding the division's remainder. In this case, the division result is 2.

[19]:
```
a=b=10
print(a,b,a/b)
```

```
10 10 1.0
```

In the given code first assigns the value "10" to the variable "b", and then assigns the same value to variable "a". The "=" operator is used to assign a value to a variable. The "print()" function is then used to output the values of "a", "b", and the division of "a/b". The "/" operator divides floating-point numbers and returns a decimal value. In this case, the division value is 1.0.

0.2 Strings and related operations

[20]:
```
a = "Hello "
print(a)
```

```
Hello
```

The "=" operator is used to assign a value to the "a" variable. The string "Hello" is given to the variable "a" on the first line. The "print()" function is used to print the value of a, which is "Hello," on the screen as a output.

[21]:
```
print(a+b)  # Output should contain "Hello 10"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[21], line 1
```

```
----> 1 print(a+b)  # Output should contain "Hello 10"

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

The "+" operator is used to concatenate strings. In this case, the value of variable "a" is "Hello," and the value of variable "b" is "10," which is a number. When we try to concatenate a string and a number, we get a TypeError since we can't concatenate a string and number directly.

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

```
Hello 10
```

The value of variable "a" is "Hello" and the value of variable "b" is "10," which is a number, we will get a TypeError since we cannot concatenate text and number directly. So we use the "str()" function to convert the value of b to a string. So the line a + "str(b)" will concatenate the string "Hello" with the string "10" to form a new string "Hello 10".

[ ]:
```
# 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(" "*38 + "42")
print("*-*-*-*-" * 5)
```

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

The first line use the "*" operator to repeat the "-" character 40 times, resulting in a line of 40 "-" signs. The "print()" function is then used to display this line on the screen as a output. The second line use the "*" operator to repeat the " " character 38 times to concatenate the number 42 to it. The "print()" function is then used to display this line on the screen as an output. The third line use the "*" operator to repeat the "--" pattern 5 times, yielding a line of 40 characters. The "print()" method is then used to display this line on the screen as an output.

[ ]:
```
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
```

This code use an f-string (also known as a "formatted string literal") to output a string containing the values of the variables "a" and "b" as well as some text. The letter f or F is followed by a double-quote (") or single-quote (') character in the f-string. Curly brackets {} are used within the string to indicate where variable values should be put. In this case, the f-string has the text"The variable 'a' has the value ", followed by the value of the variable"a" (in this case "Hello"), and then the text "and 'b' has the value", followed by the value of the variable "b" (in this case 10), right justified with 10 spaces.

```
[ ]: # 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
```

```
[ ]: courses = [    {"id": "EE2703", "name": "Applied Programming Lab"},
                    {"id": "EE2003", "name": "Computer Organization"},
                    {"id": "EE5311", "name": "Digital IC Design"}]

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

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

The list "courses" has three dictionaries, each of which represents a "course" with an ID number and a name. The "for loop" iterates through the list, formatting the result with "format()". The format specifier "{:<10}" left-justifies the "course["id"]" string to ten spaces. The format specifier "{:>40}" right-justifies the string "course["name"]" to 40 spaces.

1 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):
```

```
    pass
```

[ ]:
```
def twosc(x, N=16):
    if x >= 0:
        binary = bin(x)[2:].zfill(N)
    else:
        binary = bin(2**N + x)[2:]
        binary = binary[-N:]
    print(binary)
twosc(10)
twosc(-10)
twosc(-20, 8)
```

```
0000000000001010
1111111111110110
11101100
```

The function accepts an integer "x" as well as an optional argument "N" representing the number of bits. "N" has the default value of 16. If "x" is positive, it uses the "bin()" function to convert it to binary and eliminates the leading "0b" prefix. The resulting binary string is then padded on the left with "0s" using "zfill(N)" until it is the desired length "N". If "x" is negative, it calculates its two's complement by adding "2**N" and converting the result to binary, removing the leading "0b" prefix. The resultant binary string is then sliced so that just the final N characters are kept. The binary string is then printed as output.

2 List comprehensions and decorators

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

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

This code provides a list comprehension that generates a list of squares of even values between 0 to 9. The "range(10)" function returns a number sequence range from 0 to 9. If " x%2 == 0", the if clause filters the numbers in the sequence such that only even numbers are used in list comprehension. Each even integer is squared in the expression "x*x". Finally, the output is a list of squares of even values between 0 and 9.

[ ]:
```
# 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]
```

To compress a 2-dimensional list into a 1-dimensional list, this code use layered list comprehension. Here's what's going on. The outer list comprehension "[v for row in matrix for v in row]" loops across the "rows" in the matrix. The inner list comprehension for "v" in "row" iterates over each "row" values "V". As a result, a 1-dimensional list of all the values in the "matrix" [1, 2, 3, 4, 5, 6, 7, 8, 9] is a output of the code.

[ ]:
```
# 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
    for i in range(2, int(x**0.5)+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]
```

The function takes a single integer “x” as input. Because 2 is the lowest prime number, if “x” is smaller than 2, it returns False. It then loops over all numbers from 2 to the square root of “x + 1” (since a number cannot have a factor bigger than its square root) and uses the modulo operator “%” to check if “x” is divisible by any of them. If a factor is found, the function returns False; otherwise, it returns True.

[ ]:
```
# 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
```

This code declares two functions, “f1” and “f2”, and then creates a third function, “f3”, by calling f2 with f1 as an argument. ” f1“ is a simple function that accepts a single parameter,” x“, and returns the string”happy” + x. ” f“2” is a higher-order function that takes an argument from another function“ f” and returns a new function wrapper. Wrapper is a closure that can accept any number of arguments *”args“ and keyword arguments”**kwargs**“. It returns the string”Hello” + f(*args, **kwargs) + “world” , where “f” is the function passed as an argument to “f2”. The result of calling “f2” with “f1” as an argument is given to “f3”. The output of executing “f3” with the input “flappy” is “Hello happy flappy world”.

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

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

```
Hello nappy flappy world
```

3 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):
    pass
```

[23]:
```
def write_primes(N, filename):
    with open(filename, 'w') as f:
        for i in range(1, N+1):
            if is_prime(i):
                f.write(str(i) + '\n')
N=int(input("Enter a number"))
write_primes(N,"prime.txt")
```

The "write_primes" function accepts two parameters, "N" and "filename," where "N" is the upper limit for the number range to check for primality and "filename" is the name of the file to which the prime numbers will be written. The function opens the file supplied by "filename" in write mode and iterates over the integers "1" to "N"(inclusive) using a for a loop. The function uses the "is_prime" function to determine whether an integer is a prime or Not. If the integer is prime, the function saves it as a string followed by a newline character to the file. This process is repeated until all prime numbers ranging from "1" to "N" have been written to the file.

4 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):
    pass
x = input('Enter a number: ')
check_prime(x)
```

[ ]:
```
def check_prime(x):
    try:
        if(isinstance(x,str)):
            x=int(x)
        if (isinstance(x, int)) == True:
            if x <= 1:
                print(f'{x} is not a prime number')
            else:
                for i in range(2, int(x**0.5) + 1):
```

```
                    if x % i == 0:
                        c=0
                        break
                    else:
                        c=1
                if c==0:
                    print(f'{x} is not a prime number')
                else:
                    print(f'{x} is a prime number')
        else:
            print("Invalid input: Please enter an integer")

    except ValueError:
        print("Invalid input: Please enter an integer")
x = input('Enter a Number')
check_prime(x)
```

```
5 is a prime number
```

The "try-except" block is used to handle exceptions (error situations) that may arise during code execution. It enables the code to continue operating in the event of an error. The "try" block in the code includes the code that might throw an exception, and the "except" block contains the code that will be performed if an exception is thrown. If a non-integer input is entered, a "ValueError" is thrown and the message "Invalid input: Please provide an integer" is printed. This prevents the application from crashing and allows it to continue running even if an error occurs.

A given number is a prime number or not. It first verifies whether the input is an integer. If the input is not an integer, the error message "Invalid input: Please enter an integer" is printed.

If the input is an integer, it first checks if the number is less than or equal to "1". If the answer is yes, the number is not a prime number. If the input number is bigger than "1," a for loop is used to determine if the number is divisible by any number between "2" and its square root of the number "x" rounded up to the close integer. If the number is divisible by any of these values, then "c" become "0" and break it (if a number have any factor then it's not a prime) otherwise it's become "1". And if "c" is equal to "0" the message "not a prime number" is printed. If the "c" is "1", then it is printed as a prime number.

If the input is not an integer and cannot be transformed to one, a "ValueError" will be raised. The code catches this issue with a "try-except" block and produces the proper error message in that situation.