EE2703 - Week 1

Nitin Chandrachoodan <nitin@ee.iitm.ac.in>

February 4, 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.

1.1 Numerical types

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

```
2.4
```

Normal division between intigers give you a *float* data type, which in this case is 2.4.

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

```
2
```

This code calculaets the intiger division of 12 by 5. This means that the reault is rounded to the closest intiget , and the datatype *int* is used

[2]:
```
a = b = 10
print(a,b,a/b)
```

```
10 10 1.0
```

This code prints the intigers stored in q and b and then their float division result.

The result contains 10,10,1.0 which means that the first two are *int* and the third is *float*

1.2 Strings and related operations

[3]:
```
a = "Hello "
print(a)
```

```
Hello
```

Printing a simple Character string

[4]:
```
#Using an f-string to print the result
print(f"{a}{b}")
```

```
Hello 10
```

Using **f-string** to print an interpolated string using only one line of code. Noramlly a variable won't be able to be represented with a string in the print funciton . This is fixed by using f-string

[14]:
```
# Ending the print function with a null character , replacing the newline.
for i in range(0,40):
    print('-',end="")
    print(42)

# For loop alternating - and * to make the pattern
for i in range(0,20):
    print("*",end="")
    print('-',end="")
```

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

Using for loops to print iterations of patterns to give the desired result

[15]:
```
# f-string to print an interpolated string with a single print function
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
```

Using **f-string** to print an interpolated string using only one line of code. Noramlly a variable won't be able to be represented with a string in the print funciton . This is fixed by using f-string

One more thing that can be noted here is that the variable b is formatted to the right by 10.

[37]:
```
# Initialising the list of dictionaries containing the Course ID and Course Names
dict_list =[{'id': 'EE2703', 'name': 'Applied Programming Lab'},
    {'id': 'EE2003', 'name': 'Computer Organization'},
    {'id': 'EE5311', 'name': 'Digital IC Design'}]

# Iterating over the list and printing its element with the right formatting
for i in dict_list:
    print(f"{i['id']:<10}{i['name']:>40}")
```

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

A list of dictionaries is created to hold the courses corresponding their course IDs The this list if iterated through a for loop to print with proper formatting and order with methods same as those used in the previous code cell

[ ]:
```

```

2 Functions for general manipulation

[8]:
```
# Defining a function to find the binary represenation of numbers given
def twosc(number, N=16):
    if number >= 0:
        # Positive numbers: Find binary representation of number and fill with 0s
        binary_representation = bin(number)[2:].zfill(N)
    else:
        # Negative numbers: Find binary representation of 2^N + number and fill␣
 ↪with 0s
        binary_representation = bin(2**N + number)[2:].zfill(N)
    return binary_representation

# Calling the function to test it
twosc(-20,8)
```

[8]:
```
'11101100'
```

3 List comprehensions and decorators

[4]:
```
[x*x for x in range(10) if x%2 == 0]
```

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

The output is a **list comprehension**

The list comprehension is composed of three parts:

- The expression `x*x` specifies the operation to be performed on each element in the list.
- The `for x in range(10)` part specifies the source of the elements in the list, in this case the numbers from 0 to 9.
- The if `x%2 == 0` part specifies a filter that only allows elements that satisfy this condition to be included in the list.

Therefore the result generated is a list of aquares of the even numbers from 0 to 9, that is, `[0, 4, 16, 36, 64]`.

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

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

This is againa **List Comprehension** example.

The list is "comprehending" `v` in this case, as given by the first element

The looping itself is composed of two parts:

1) The expression `for row in matrix` specifies that `row` is the variable that will iterate over the rows of the matrix `matrix`.

2) The inner `for v in row` part specifies that `v` is the variable that will iterate over the values in each row.

Therefore, the list comprehension `[v for row in matrix for v in row]` generates a list of all the values in the matrix by iterating over each row and then over each value in each row. The resulting list is [1, 2, 3, 4, 5, 6, 7, 8, 9], which is the flattened representation of the original matrix.

[2]:
```
# Defining a function to find if a number is prime
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

# Using a list comprehension to print the prime numbers between 1 and 100 using␣
 ↪a if statement on is_prime() to make sure that the number is prime
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]
```

First part is the defenition of a function `is_prime()` to find if the number supplied is a prime or not - This is done by checking its divisibility by numbers from 2 to $\sqrt{N} + 1$ - If no such divisor is found then the function returns `True` and `False` otherwise

Second part is essentially printing a list comprehension that iterates from 1 to 100 and printing the primes in that range using `is_prime()`

[6]:
```
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 defines two functions f1(x) and f2(f).

f1(x) takes a single argument x and returns the string “happy” concatenated with x.

f2(f) takes a single argument f which is a function and returns a new function wrapper which takes any number of arguments *args and any number of keyword arguments **kwargs. The wrapper function returns the string “Hello” concatenated with the result of calling f with the arguments *args and **kwargs, which is then concatenated with the string ” world”.

Finally, the code assigns f3 to the result of calling f2 with the argument f1, effectively wrapping f1 with the wrapper function defined by f2.

The code then calls f3 with the argument “flappy” and prints the result, which is the string “Hello happy flappy world”.

[7]:
```
# Explain the output below
@f2
def f4(x):
    return "nappy " + x

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

```
Hello nappy flappy world
```

The output of the code is Hello nappy flappy world.

This is because f4 is **decorated** with the f2 decorator. The f2 decorator returns a closure wrapper that adds the string “Hello” to the start of the result of a call to the decorated function and adds ” world” to the end.

So when f4 is decorated with f2, it becomes wrapper and the call f4(“flappy”) will run the closure and add “Hello” to the start of the string returned by f4 and add ” world” to the end.

This is how the code works:

- f2 is a decorator that takes a function as input.
- f2 returns a closure (the wrapper function) that takes the original function’s result and adds the string “Hello” to the start and ” world” to the end.
- f4 is a function that returns the string “nappy” + x
- f4 is decorated with f2 using the @f2 syntax, so it becomes the closure wrapper.
- The code calls f4(“flappy”) which is the same as calling the closure wrapper with argument “flappy”.
- The closure wrapper takes the string returned by f4 (“nappy flappy”) and adds the strings “Hello” to the start and ” world” to the end to give “Hello nappy flappy world”.

4 File IO

[3]:
```
# Defining a function to print the primes till N to a text file trial.txt
def write_primes(N, filename):
    f=open(filename,'w')
    f.write(str([x for x in range(1,N+1) if is_prime(x)]))
    f.close()

# Calling the function to test it out
write_primes(200,'trial.txt')
```

This cell defines the `write_primes()` function to write primes till a given number into a text file - This involves opening the file in `w` mode - Then the `write` function is used with the predefined function `is_prime()`

Then the function is called to test it

5 Exceptions

[16]:
```python
def check_prime(x):
    # Convert x to integer and handle exceptions in case it's not possible
    try:
        x = int(x)
    except ValueError:
        print("Error: input must be an integer")
        return

    # Check if x is less than 2 and return an error message if true
    if x < 2:
        print("Error: input must be greater than 1")
        return

    if is_prime(x):
        print(f"{x} is a prime number")
        return
    else:
        print(f"{x} is not a prime number")

# Prompt user to enter a number
x = input('Enter a number: ')
# Call the check_prime function with the user input
check_prime(x)
```

```
2 is a prime number
```

A function `check_prime()` is defined - The function checks for two kinds of errors, not an intiger error and less than 2 error - If any of the above error flags are triggered the function returns immideatly - Otherwise it checks for prime numbers and non-prime numbers using the `is_prime()` function that was defined earlier and then prints a result accordingly