EE2703 - Week 1

Rishi Nandha V EE21B111

February 4, 2023

1 Document metadata

The notebook's metadata can be accessed in the JupyterLab lab server using the cog-wheels icon at the right-top corner of the interface. The lines shown had to be changed from the follow:

```
"authors": [
        {
            "name": "Nitin Chandrachoodan <nitin@ee.iitm.ac.in>"
        }
    ]
```

To the following:

```
"authors": [
        {
            "name": "Rishi Nandha V EE21B111"
        }
    ]
```

Thus the document's metadata was modified.

Additionally, for final formatting and changing fonts I am exporting the notebook as a `.tex` and using Overleaf instead of directly exporting into a `.pdf`. The following was added to the preamble to change the teletype in the markdown cells, change the overall font and the cover page.

2 Basic Data Types

2.1 Numerical types

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

```
2.4
```

`2.4` is printed here because `/` is the float division operator. It divides the left literal with the right literal and returns the answer with decimal points as a float.

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

```
2
```

`2` is printed here because `//` is the integer division operator. `12 // 5` returns the integer $\lfloor \frac{12}{5} \rfloor$ as the answer.

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

```
10 10 1.0
```

Here `10/10` is a float division as explained above. Thus the answer obtained, `1.0`, is a float denoted with a trailing zero to signify that it's a float.

This is also used to implicitly typecast and the below cell has been added to demonstrate how an integer and a float are differentiated using the trailing decimal point.

[4]:
```
print(10/2, type(10/2), sep = '\t- ')
print(10//2, type(10//2), sep = '\t- ')
```

```
5.0     - <class 'float'>
5       - <class 'int'>
```

2.2 Strings and related operations

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

```
Hello
```

Here `Hello \n` is printed into the console because default ending attached to all print statements is `\n`

[6]:
```
print(a+str(b))
```

```
Hello 10
```

`print(a+b)` gives an TypeError since we are trying to add an integer to a string. Our intended output can be obtained by typecasting the integer which is the change that has been done above.

[7]:
```
print("-"*40)

print("-"*38+"42")

print("*-"*20)
```

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

The `*` token when used with a string and an integer acts as a string repetition operator.

[8]:
```
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
```

The prefix of `f` makes a string an `f-string` which directs the compiler to take the expressions within braces and replace them with the evaluated values as expressions. Here `{a}` is being replaced by `Hello` and `{b:>10}` is getting replaced with `10` but right aligned by 10 white spaces as manipulated by the operator `:>`

[9]:
```
n=int(input('Number of Entries: '))
print()
```

```
l=[]
for i in range(n):
    d={}
    value=input(f'Entry no. {i+1} in the format ABxxxx -> Name of the␣
 ↪course: ')
    d['id']= value[:6]
    d['name']=value[10:].strip()
    l.append(d)
print()
for k in l:
    print(f'{k["id"]}    {k["name"]:>40}')
```

```
Number of Entries:  3


Entry no. 1 in the format ABxxxx -> Name of the course:  EE2703 ->␣
 ↪Applied Programming Lab
Entry no. 2 in the format ABxxxx -> Name of the course:  EE2003 ->␣
 ↪Computer Organization
Entry no. 3 in the format ABxxxx -> Name of the course:  EE5131 ->␣
 ↪Digital IC Design


EE2703                      Applied Programming Lab
EE2003                        Computer Organization
EE5131                            Digital IC Design
```

Here a loop is made to create the dictionary that gets into the list. Note that here if the dictionary `d={}` is created outside the loop as a global dummy variable, appending the list must be done with `l.append(d.copy())` since otherwise shallow copies of the dictionary will be made into the list.

For formatting the final string, we use an f-string again and the `:>` to perform the right aligning.

3 Functions for general manipulation

[10]:
```
def twosc(x, N=16):
    ans = ''
    for i in range(N):
        ans = str(x%2)+ans
        x=x//2
    print(ans)

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

```
0000000000001010
1111111111110110
11101100
```

The algorithm used for converting positive decimal integers into binary numbers is fairly trivial here. For the negative numbers, note that when the expression `a//b` is used with a negative dividend `a`, it finds the largest quotient `q` such that `q*b<=a` and `a%b` just evaluates to `a - (a//b)*b`. Proof that this gives the two's compliment can be found below:

Let $(a)_2$ be a binary numbers of N bits. Convention used to represent negative numbers is $(-a)_2 = (b)_2$ such that by definition of 2s-compliment $(a + b) = 2^N$.

$$\lfloor \frac{(2^N - a)}{2^i} \rfloor \mod 2 = \text{i-th bit (0-indexed)}$$

$$\implies \lfloor \frac{(-a)}{2^i} \rfloor \mod 2 = \text{i-th bit (0-indexed)}$$

Thus the same algorithm works for both positive and negative integers when following the 2s-compliment convention

4 List comprehensions and decorators

[11]:
```
[x*x for x in range(10) if x%2 == 0]
```

[11]:
```
[0, 4, 16, 36, 64]
```

`x` is taken from the `range(0,10)` and then it is checked if it is even, if it is even then it is squared and becomes a part of the list. Note here that this is in contrast to appending an entry and recreating the list every time as in a `for` -loop since the list is created in one-go here. This kind of list comprehension is unique to python and infact faster than running a `for` -loop.

[12]:
```
# 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]
```

[12]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

For or If Statements within a list comprehension are nested right to left. Thus here the `for row in matrix` is the larger loop and `for v in row` is the smaller loop

[13]:
```
def is_prime(x):
    c=0
    for i in range(2,(x//2)+1):
        if x%i==0:
            c=c+1
    if c==0:
        return(True)
    else:
        return(False)

print([i for i in range(2,100) if is_prime(i)==True])
```

```
[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]
```

Nested statements within list comprehension has been used here again like above

[14]:
```
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 a function declaration, `*` preceeding a formal parameter denotes that the function can get a variable number of arguments and similarly `**` for a variable number of keyword-arguments. Here `args` will become a list of all the non-keyword arguments and `kwargs` a dictionary similarly. `*args` converts the list a comma seperated set of arguments again in `f(*args, **kwargs)` and similarly for `**kwargs`

But again note here that when `f1(*args, *kwargs)` is called we should note that f1() was not given an arbitrary length for arguments, hence this will throw an error unless `f3()` is given only one string as input

`f3=f2(f1)` is a functional assignment which is telling giving the function `f1` as a function object to `f2` and also `f2(f1)` as a composite function object to `f3`

[15]:
```
@f2
def f4(x):
    return "nappy " + x

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

```
Hello nappy flappy world
```

`@` is known as the decorator. The above code is equivalent to the code in the previous cell. Calling `@f2` and then defining `f4` is the same as defining `f4` and then doing a functional assignment `f4 = f2(f4)`

(References used: https://builtin.com/software-engineering-perspectives/python-symbol)

5 File IO

[16]:
```
def write_primes(N, filename):
    with open(filename, 'wb') as f:
        f.write(bytes([i for i in range(2,N+1) if is_prime(i)==True]))
        f.flush()

# Demonstration
```

```
write_primes(100, 'data.bin')
with open('data.bin','rb') as f:
    primes = list(f.read())
    print(primes)
```

```
[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]
```

A binary file is opened within the code block `with open(filename, 'wb') as f:` and the list of primes generated is saved into the file after encoding it into bytes. This can be retrieved back as shown below by using `list()` on the bytes data.

6 Exceptions

[17]:
```
def check_prime(x):
    if (x==1 or x==0):
        print(f'{x} is neither a prime nor a composite')
        return None
    try:
        if is_prime(x):
            print(f'{x} is a prime number')
        else:
            print(f'{x} is not a prime')
    except TypeError:
        print(f"Error: {x} is not an integer")
# Demo

check_prime(0.1)
check_prime(1)
check_prime(5)
```

```
Error: 0.1 is not an integer
1 is neither a prime nor a composite
5 is a prime number
```

Try and Except are used for exception handling to display a user-friendly error message and handle the error appropriately. The `try:` codeblock is first run and if it raises the error mentions in the `except` statement the code under this block is run