ASSIGNMENT_1

Annangi Shashank babu (EE21B021)

February 4, 2023

1 Document metadata

The notebook's metadata can be modified in the *Property Inspector*, under the section *Notebook Metadata*.

1.1 Numerical types

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

```
2.4
```

In python the division of two numbers returns the answer of type *float* for this operator.

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

```
2
```

The operator // represents the floor division and returns a number of type *int* in python.

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

```
10 10 1.0
```

Since python is dynamically typed language, We don't have to declare the type of variables **a** and **b** are while assigning a value to them. Python states **a** and **b** to be of appropriate type during the run-time of the programme. Even though **a** is divisible by **b**, the return type of **a/b** is still *float*.

[4]:
```
#explanation
c = a/b
d = a // b
def print_type(**kwargs):
    for key, value in kwargs.items():
        print(key + ' --> ' + str(type(value)))

print_type(a = a, b = b, c = c, d = d)
```

```
a --> <class 'int'>
b --> <class 'int'>
c --> <class 'float'>
d --> <class 'int'>
```

1.2 Strings and related operations

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

print_type(a = a)
```

```
Hello
a --> <class 'str'>
```

Now python Dynamically (during runtime itself) converted the type of **a** from *int* to *string*.

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

```
a --> <class 'str'>
b --> <class 'int'>
```

Python can't concatenate a integer to a string so printing(a + b) will raise a TypeError, to get a output "Hello 10" we have to type-cast the integer (**b**) to the type *string* first, and then concatenate as shown below.

[7]:
```
#explanation
b = str(b)
print_type(a = a, b = b)
print(a + b)
```

```
a --> <class 'str'>
b --> <class 'str'>
Hello 10
```

[8]:
```
# 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 '*-*-*-'
length = 40
e = "-"
d = "*"
c = 42
for i in range (length):
    print(e,end="")
print()
for i in range (length - 2):
    print (" ",end="")
print(c)
for i in range (length // 2):
    print(d + e,end="")
print();
```

```
----------------------------------------
                                      42
```

*-

[9]:
```python
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 Python, an f-string is a literal string, prefixed with ‘f’, which contains expressions inside braces. The expressions are replaced with their values. The syntax {b:>10} adds 9 white-space characters to the string and then prints the value of **b**.

[10]:
```python
# 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

course1 = { 'id' : 'EE2703', 'name' : 'Applied Programming Lab'}
course2 = { 'id' : 'EE2003', 'name' : 'Computer Organization'}
course3 = { 'id' : 'EE5131', 'name' : 'Digital IC Design'}
courses = [course1,course2,course3]
for course in courses:
    print(course['id'],end='')
    for i in range (50 - len(course['name'] + course['id'])):
        print('',end=' ')
    print(course['name'])
```

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

Using the user code for padding and formatting the strings can prove to be cumbersome at time, this is where f-strings comes in handy. F-strings are a great way to format strings and make the source code elegent.

[11]:
```python
#easier way for the above problem statement
for course in courses:
    print(f"{course['id'] : <10}{course['name'] : >40}")
```

```
EE2703                    Applied Programming Lab
EE2003                      Computer Organization
```

2 Functions for general manipulation

[12]:
```
# 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, flag = 'true'):
    if x < 0:
        flag = 'false'
        twosc(2**(N) + x,N,flag)  # unsigned representation of 2**16 - 10B ==␣
  ↪signed representation of -10
    elif (x > (2**(N-1) - 1)) and (flag == 'true'):
        print('error,number of bits not enough')
    else :
        print(str(bin(x)[2:]).rjust(N,'0'))
    pass




twosc(10,16)
twosc(-10)
twosc(2**16 - 10)
twosc(2**16 - 10,17)
twosc(-20, 8)
```

```
0000000000001010
1111111111110110
error,number of bits not enough
01111111111110110
11101100
```

3 List comprehensions and decorators

[13]:
```
# Explain the output you see below
[x*x for x in range(10) if x%2 == 0]
```

[13]:
```
[0, 4, 16, 36, 64]
```

The above code is equivalent to the below one

[14]:
```
#explanation
c = []
for x in range(10) :
    if x % 2 == 0:
        c.append(x*x)
c
```

[14]:
```
[0, 4, 16, 36, 64]
```

The output is a list of squares of all the numbers in between [0,10)

[15]:
```
# 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]
```

[15]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The above code is equivalent to the below one

[16]:
```
#explanation
d = []
row = 0
v = 0
for row in matrix :
    for v in row :
        d.append(v)
d
```

[16]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[17]:
```
#explanation
print_type(matrix = matrix, row = row, v = v ,d = d, c= c)
```

```
matrix --> <class 'list'>
row --> <class 'list'>
v --> <class 'int'>
d --> <class 'list'>
c --> <class 'list'>
```

The variable *row* is a list and the variable *matrix* is a list of list's.

[18]:
```python
# 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 == 0 or x == 1 :
        return 0
    i = 2
    while i*i <= x :
        if x % (i) == 0:
            return 0
        i += 1
    return 1
for i in range(0,101):
    if is_prime(i):
        print(i,end=' ')
print()
```

```
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 above function has a time complexity of O(n^1.5). The function checks if there are any factors of a number *x* in range of [2,sqrt(x)], if it finds any factors then it returns 0, else returns 1 because there also cant be any factor of x in range (sqrt(x),x) if there is no factor in range [2,sqrt(x)].

[19]:
```python
# 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
```

args represents non-keyword arguments and kwargs represent keyword arguments. args and kwargs are to be used when we want a function to take *variable* length of aurguments. *args receives arguments as a tuple. A keyword argument is where you provide a name to the variable as you pass it into the function. kwargs receives arguments as a dictionary.

f2 is a function which takes another function as a argument and makes a new *wrapper* function (wrapper can take variable length of arguments both keyword and non-keyword) and returns the newly created function (this functions also can take variable length is aurgument as it is just a modified copy of wrapper function). f3 has the same defination as f5 shown below to get a better clarity on what the defination of f3 function is.

```
[20]: #explanation

def f5(*args, **kwargs):
        return "Hello " + f1(*args, **kwargs) + " world"
print(f5('flappy'))
#equivalent to f5 = f2(f1)
```

```
Hello happy flappy world
```

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

```
Hello nappy flappy world
```

The defination of f4 is equivalent to the defination of f6 shown below to get a better idea on what f4 function's defination is.

```
[22]: #explanation
def f6(x):
    return "nappy " + x
f6 = f2(f6)
print(f6("flappy"))
# is equivalent to
#@f2
#def f6()
#    return "nappy" + x
# print(f6("flappy"))
```

```
Hello nappy flappy world
```

```
[23]: # File IO
```

```
[24]: # 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 seive(N) :
    numbers = [1 for i in range(0 , N+1)]
    numbers[0] = 0
    numbers[1] = 0
    for j in range(2 , N+1):
        i = j * j;
        while (i < N + 1) :
            if numbers[i] == 1:
                numbers[i] = 0
            i += j
    return numbers
```

```
def write_primes(N, filename):
    seive(N)
    with open(filename,'w') as file_object:
        numbers = seive(N)
        for i in range(N+1):
            if numbers[i] == 1:
                file_object.write(str(i) + '\n')
write_primes(200,'prime.txt')
```

A more optimized algorithm (sieve of Eratosthenes) which has a time complexity of O(nlog(logn)) is used, which conveys the information of whether a number in range [1,N] is prime or otherwise, then this information is used to write the prime numbers into a file 'prime.txt' present in same directory as source code.

[25]:
```
# Exceptions
```

[26]:
```
# 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 seive(N) :
    numbers = [1 for i in range(0 , N+1)]
    numbers[0] = 0
    numbers[1] = 0
    for j in range(2 , N+1):
        i = j * j;
        while (i < N + 1) :
            if numbers[i] == 1:
                numbers[i] = 0
            i += j
    return numbers
def check_prime(x):
    try:
        x = int(x)
        if x < 0:
            print("error : input must be a positive integer")
            return;
    except ValueError :
        print('error : The input given is not a integer')
    else :
        if (x) > 2:
            numbers = seive((x))
            if numbers[x] == 1:
                print(str(x) + '  is a prime number')
            else : print(str(x) + ' is NOT a prime number')
        else : print(str(x) + ' is NOT a prime number')
x = input('Please enter a positive integer: ')
```

```
check_prime(x)
```

```
Please enter a positive integer:  93
```

```
93 is NOT a prime number
```

Error message is displayed if the number is less than 0 or the input given is not a integer. if the number is valid then seive algorithm is applied to find whether it is prime or not.