40 PYTHON PRACTICE QUESTIONS

FOR BEGINNERS

SELECTED FROM 1000+ HOURS OF TUTORING PYTHON

SOLUTIONS, EXPLANATIONS & RUN-THROUGHS INCLUDE

BY LIU ZUO LIN

Introduction

Hello there! My name is Liu Zuo Lin, and I'm a Software Engineer based in Singapore. I've been tutoring Python as a side hustle since early 2021, and have provided guidance to more than 70 unique students so far in a variety of areas:

- Basic Python concepts
- Object-Oriented Programming in Python
- Data Structures & Algorithms
- Machine Learning & Data Science
- Etc

After a while, I realised a couple of things about tutoring:

1. I was explaining the same concepts over and over again
2. I was explaining these concepts in the same way over and over again
3. I was reusing the same practice questions over and over again

So I figured – why not gather a whole repertoire of questions that I've used and explained hundreds of times, arrange them by topic, and document their solution in a giant PDF file? And thus this series of books were born!

How This Book Is Structured

This book is a collection of 40 basic Python practice questions that I often use with my beginner-level students, and aims to strengthen your fundamental Python concepts through practice and repetition. Without further ado, let's get started!

Questions

40 Python practice questions are numbered and arranged by topic:

- Data Types, Variables & Functions (Questions 1 to 5)
- If-Else Statements & Logical Operators (Questions 6 to 10)
- Loops - For & While Loop (Questions 11 to 15)
- Accumulation Using A Loop (Questions 16 to 20)
- Strings (Questions 21 to 25)
- Lists (Questions 26 to 30)
- Dictionaries (Questions 31 to 35)
- File Reading & Writing (Questions 36 to 40)

Answers

After the questions come the answers, and for each question, 1 answer will be provided. Each answer contains 1) the full code solution 2) detailed explanation of the code logic and 3) a detailed run through of the code

Data Types, Variables & Functions

1) What Year Was I Born In?

Write a Python program that prompts the user for his/her age, and prints the year that the user was born in.

Note – user inputs are underlined

```
Enter your age: 21
My age is 21, and I was born in 2001
```

```
Enter your age: 36
My age is 36, and I was born in 1986
```

```
Enter your age: 100
My age is 100, and I was born in 1922
```

2) Sum of Input Numbers

Write a Python program that prompts the user for 2 numbers, and prints the sum of the 2 numbers. Assume that the user will enter integer numbers.

```
Enter the first number: 4
Enter the second number: 5
Sum of 2 numbers is 9
```

```
# Case 2:
Enter the first number: 10
Enter the second number: 7
Sum of 2 numbers is 17
```

3) Repeating Pattern

Write a Python program that prompts the user for 1) a string pattern and 2) a number *n*. The program then prints the pattern repeated *n* times on the same line.

```
Enter your pattern: *
Enter a number: 5
*****
```

```
Enter your pattern: a
Enter a number: 4
aaaa
```

```
Enter your pattern: apple
Enter a number: 3
appleappleapple
```

4) Area of A Circle

Write a function *area(radius)* that takes in a radius, and returns the area of the circle.

- The area of a circle is (pi x radius^2)
- Assume pi = 3.14

```
area(1) # 3.14
area(2) # 12.56
area(3) # 28.26
```

5) Adding Numbers

Write a function *add(a, b, c)* that takes in 3 numbers - a, b and c. The function then returns the sum of all 3 numbers.

```
add(1,2,3) # 6
add(2,4,6) # 12
add(1,1,9) # 11
```

If-Else Statements & Logical Operators

6) Larger Than 10

Write a function *larger10(number)* that takes in an integer *number*, and returns *True* if *number* is larger than 10 and *False* otherwise

```
larger10(5)    # False
larger10(10)   # False
larger10(10.1) # True
larger10(12)   # True
```

7) Area or Circumference Of A Circle

Write a function *calculate(mode, radius)* that takes in a mode and a radius.

- If mode = “area”, the function returns the area of the circle
- If mode = “circumference”, the function returns the circumference of the circle
- If mode is anything else, the function returns 0
- area of circle = pi x radius^2 (assume pi = 3.14)
- circumference = 2 x pi x radius

```
calculate("area", 1)           # 3.14
calculate("circumference", 1)  # 6.28

calculate("area", 2)           # 12.56
calculate("circumference", 2)  # 12.56

calculate("area", 3)           # 28.26
calculate("circumference, 3)   # 18.84

calculate("stuff", 2)          # 0
```

8) Larger & Largest

Write a function *larger(a, b)* that takes in 2 numbers *a* and *b*, and returns the larger number. You are not allowed to use the built-in *max* function.

```
larger(4, 5)     # 5
larger(5, 4)     # 5
larger(10, 15)   # 15
larger(15, 10)   # 15
```

Next, write a function *largest(a, b, c)* that takes in 3 numbers *a, b & c*, and returns the largest number. Hint: you can use the *larger(a, b)* function above inside your *largest(a, b, c)* function.

```
larger(4, 5)     # 5
larger(5, 4)     # 5
larger(10, 15)   # 15
larger(15, 10)   # 15
```

9) Scholarship Eligibility

A student takes 3 subjects - English, Math & Science. A student is eligible for a scholarship if all his scores are more than 80. Write a function *eligible(english, math, science)* that takes in 3 scores (*english, math & science*) and returns *True* if the student is eligible for a scholarship and *False* otherwise.

```
eligible(80, 81, 81)     # False
eligible(81, 81, 81)     # True

eligible(100, 100, 79)   # False
eligible(90, 90, 90)     # True
```

10) Scholarship Eligibility 2

A student takes 3 subjects - English, Math, Science. He is eligible for a scholarship if:

- all his scores are more than 80
- his average score is more than 85
- his conduct is “excellent”

Write a function *eligible(english, math, science, conduct)* that takes in his english, math & science scores, as well as his conduct, and returns *True* if the student is eligible for a scholarship and *False* otherwise.

```
eligible(81, 81, 81, "excellent")   # False
eligible(99, 99, 99, "ok")          # False
eligible(79, 99, 99, "excellent")   # False

eligible(81, 90, 90, "excellent")   # True
```

Loops - For & While Loop

11) Printing Odd Numbers

Write a function *odd(start, end)* that takes in 2 integers *start & end*, and prints all odd numbers between *start & end* (inclusive). Assume that *start* will always be smaller than *end.*

```
odd(1,5)

1
3
5
```

```
odd(3,11)

3
5
7
9
11
```

```
odd(12,20)

13
15
17
19
```

```
odd(20,27)

21
23
25
27
```

Write 2 versions of this function - one using a for loop and the other using a while loop

12) Printing Even Numbers Backwards

Write a function *even_backwards(start, end)* that takes in 2 integers *start & end*, and prints all even numbers between *start & end* (inclusive), but in descending order.

```
even(1, 6)

6
4
2
```

```
even(3, 11)

10
8
6
4
```

```
even(12, 20)

20
18
16
14
12
```

```
even(22, 31)

30
28
26
24
22
```

Write 2 versions of this function, one using a for loop and the other using a while loop.

13) Pyramid Of Stars

Write a function *pyramid(n)* that takes in an integer *n*, and prints the following pattern

```
pyramid(3)

*
**
***
```

```
pyramid(4)

*
**
***
****
```

```
pyramid(5)

*
**
***
****
*****
```

```
pyramid(6)

*
**
***
****
*****
******
```

Write 2 versions of this function, one using a for loop and the other using a while loop.

14) Thousands

Write a function *thousands(n)* that takes in an integer n, and prints all the thousands from 1000 to n x 1000 (inclusive)

```
thousands(5)

1000
2000
3000
4000
5000
```

Write 2 versions of this function, one using a for loop and the other using a while loop.

15) Squares & Cubes

Write a function *squares_cubes(start, end)* that takes in 2 integers, start & end, and prints the number itself, squares and cubes of all numbers between start and end (inclusive).

```
squares_cubes(2, 7)

2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
```

```
squares_cubes(5, 10)

5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

Write 2 versions of this function, one using a for loop and the other using a while loop.

Accumulation Using A Loop

16) Summation From 1

Write a function *summation(n)* that takes in an integer *n*, and returns the sum of all numbers from 1 to n. Assume *n* is a positive integer.

```
summation(4)   # 10 (1+2+3+4)
summation(5)   # 15 (1+2+3+4+5)
summation(6)   # 21 (1+2+3+4+5+6)
```

17) Factorial

Write a function *factorial(n)* that takes in an integer *n*, and returns the factorial of *n*. The factorial of a number is 1 x 2 x 3 x … x n

```
factorial(4)   # 24 (1x2x3x4)
factorial(5)   # 120 (1x2x3x4x5)
factorial(6)   # 720 (1x2x3x4x5x6)
```

18) Summation Of Odd Numbers

Write a function *sum_odd(start, end)* that takes in 2 integers start & end, and returns the sum of all *odd* numbers between start and end (inclusive)

```
sum_odd(1, 8)    # 16 (1+3+5+7)
sum_odd(1, 9)    # 25 (1+3+5+7+9)
sum_odd(1, 12)   # 36 (1+3+5+7+9+11)
```

19) Printing Triangular Numbers

Triangular numbers are derived by the summing all integers from 1 to n. Here are the first 6 triangular numbers:

```
1st triangular number: 1
2nd triangular number: 3  (1+2)
3rd triangular number: 6  (1+2+3)
4th triangular number: 10 (1+2+3+4)
5th triangular number: 15 (1+2+3+4+5)
6th triangular number: 21 (1+2+3+4+5+6)
```

Write a function *triangular(n)* that takes in an integer n, and prints the first *n* triangular numbers.

```
triangular(3)

1
3
6
```

```
triangular(6)

1
3
6
10
15
21
```

20) Sum To 100000

Write a Python script to find 1 + 2 + 3 + … + 99,998 + 99,999 + 100,000. Do this using a for or while loop, and try not to hardcode this. You should get:

```
5000050000
```

Strings

21) Summation Of String Of Numbers

Write a function *summation(numbers_string)* that takes in a string of numbers *numbers_string* separated by spaces, and returns the sum of the numbers

```
summation("1 2 3 4") -> 10
summation("1 3 4 5") -> 13
summation("-2 20 4 6") -> 28
```

22) Printing Words In A Sentence

Write a function *print_words(sentence)* that takes in a string *sentence*, and prints individual words within the sentence line by line. Here, we can assume that individual words in a sentence are separated by a space character.

```
print_words("I am happy")
I
am
happy
```

```
print_words("I have a dog")
I
have
a
dog
```

23) Removing Vowels

Write a function *remove_vowels(string)* that takes in a string, and returns a version of the original string but with vowels removed. Vowels refer to a, e, i, o or u.

```
remove_vowels("apple")       # "ppl"
remove_vowels("orange")      # "rng"
remove_vowels("pear")        # "pr"
remove_vowels("pineapple")   # "pnppl"
remove_vowels("durian")      # "drn"
```

24) String Pyramid

Write a function *pyramid(string)* that takes in a string, and prints this pattern:

```
pyramid("abcde")

a
ab
abc
abcd
abcde
```

```
pyramid("pineapple")

p
pi
pin
pine
pinea
pineap
pineapp
pineappl
pineapple
```

25) Alternate Letters

Write a function *alternate_letters(string)* that takes in a string, and returns a new string with only the characters at the odd positions - 1st, 3rd, 5th etc letters

```
alternate_letters("apple")       # "ape"
alternate_letters("orange")      # "oag"
alternate_letters("pear")        # "pa"
alternate_letters("pineapple")   # "pnape"
alternate_letters("durian")      # "dra"
```

Lists

26) Sum/Product Of List Of Integers

Write a function *sum_list(lis)* that takes in a list of integers *lis,* and returns the sum of all integers inside the list.

```
sum_list([1,2,3,4])    # 10 (1+2+3+4)
sum_list([1,5,6,7])    # 19 (1+5+6+7)
```

Using the same logic, write another function *product_list(lis)* that takes in a list of integers *lis,* and returns the product of all integers inside the list.

```
product_list([1,2,3,4])    # 24 (1x2x3x4)
product_list([1,5,6,7])    # 210 (1x5x6x7)
```

27) Converting List Of Strings To List Of Integers

Write a function *convert_int(lis)* that takes in a list of number strings *lis*, and returns a new list with all the number strings converted into integers.

```
convert_int(["1", "2", "3"])          # [1, 2, 3]
convert_int(["-1", "10", "3.14"])     # [-1, 10, 3]
```

28) Arithmetic Mean

Write a function *mean(lis)* that takes in a list of numbers, and returns the arithmetic mean of the numbers – (sum of all numbers / number of numbers)

```
mean([4,5,6])      # 5.0 as (4+5+6)/3
mean([1,2,20,3])   # 6.0 as (1+2+20+3)/4
```

29) Median

Write a function *median(lis)* that takes in a list of numbers (int or float) *lis,* and returns the median of the numbers

- To find the median, first sort the list from smallest to largest
- If the list has an odd number of elements, the median is the number in the middle
- If the list has an even number of elements, the median is the average of the 2 middle numbers

```
median([1,3,2])            # 2
median([1,5,4,2,3])        # 3
median([1,3,2,4])          # 2.5 as (2+3)/2
median([1,5,4,2,3,100])    # 3.5 as (3+4)/2
```

30) Filtering Numbers Divisible By 3

Write a function *filter3(lis)* that takes in a list of integers, and returns a new list containing only numbers from the original list that are divisible by 3

```
filter3([1,2,3])          # [3] as only 3 is divisible by 3
filter3([2,3,4,5,6,7])    # [3, 6] as only 3 and 6 are divisible by 3
filter3([2,4,5,7,8,10])   # [] as nothing in list is divisible by 3
```

Dictionaries

31) Finding Square Of Numbers

Write a function *squares(lis)* that takes in a list of numbers *lis*, and returns a dictionary where keys are the numbers in the list, and values are the squares of the numbers in the list.

```
squares([1,2,3])     # {1:1, 2:4, 3:9}
squares([4,5,6,7])   # {4:16, 5:25, 6:36, 7:49}
```

32) Finding Letter Counts In A Word

Write a function letter_counts(word) that takes in a word, counts the number of times each character appears, and returns a dictionary where keys = letters in the word and values = the number of times they appear in the word.

```
letter_counts("hello")      # {"h":1, "e":1, "l":2, "o":1}
letter_counts("apple")      # {"a":1, "p":2, "l":1, "e":1}
letter_counts("pineapple")  # {"p":3, "i":1, "n":1, "e":2, "a":1, "l":1}
```

33) Combining 2 Dictionaries

Write a function *combine(d1, d2)* that takes in 2 different dictionaries and combines them into 1 larger dictionary.

- key-value pairs that exist in only 1 dictionary will simply get added
- key-value pairs that exist in both dictionaries will use the sum of values from both dictionaries as the new value

```
combine({"a":4, "b":5}, {"c":6})          # {"a":4, "b":5, "c":6}
combine({"a":4, "b":5}, {"a":6, "c":6})   # {"a":10, "b":5, "c":5}
combine({"a":2, "b":7}, {"a":10, "b":6})  # {"a":12, "b":13}
```

34) Reversing Keys & Values In A Dictionary With Unique Values

Write a function *reverse(d)* that takes in a dictionary *d*, and reverses the keys and values. In other words, keys become values, and values become keys. Assume that all values are unique.

```
d1 = {
    "apple": "pie",
    "orange": "juice",
    "pear": "cake"
}

reverse(d1)   # returns
{
    "pie": "apple",
    "juice": "orange",
    "cake": "pear"
}
```

35) Reversing Keys & Values In A Dictionary With Non-Unique Values

Write a function *reverse(d)* that takes in a dictionary *d*, and returns a new dictionary where:

- Each key is an value from the previous dictionary
- Each value is a list of keys from the previous dictionary pointing to the key

```
d = {
      "apple": 4,
      "orange": 5,
      "pear": 6
      "pineapple": 4,
      "durian": 5
}

reverse(d)   # returns
{
      4: ["apple", "pineapple"],
      5: ["orange", "durian"],
      6: ["pear"]
}
```

File Reading & Writing

36) Reading From A File Into A List

You are given a text file containing a list of fruits, fruits.txt

```
apple
orange
pear
pineapple
durian
```

Write a Python script that reads fruits.txt, and places all the fruits inside a list.

```
["apple", "orange", "pear", "pineapple", "durian"]
```

37) Counting Fruits From A File

You are given a text file, fruits.txt, containing a list of fruits, as well as their counts (separated by a comma). There could be duplicate entries of fruits.

```
apple,4
orange,5
pear,6
pineapple,3
durian,10
apple,20,
orange,16
```

Write a Python script to read from fruits.txt, count the total number of fruits, and place the information into a dictionary.

- keys being each unique fruit
- values being the sum of counts for each fruit

```
{"apple":24, "orange":21, "pear":6, "pineapple":3, "durian":10}
```

38) Finding Average Scores For Each Subject

You are given a text file scores.txt containing the scores of some students for 3 subjects - English, Math & Science. Each row contains 4 columns - name, english score, math score and science score, and are separated by commas.

```
Name,English,Math,Science
alice,85,81,82
bob,90,71,77
charlie,76,50,33
```

Write a Python script to find the average score for English, Math and Science, and place the information in a dictionary.

```
{
      "english": 83.667,    # (85+90+76)/3 = 83.667
      "math": 67.333,       # (81+71+50)/3 = 67.333
      "science": 64.0       # (82+77+33)/3 = 64
}
```

39) Writing A Dictionary To A File

You are given a dictionary containing some fruits and their quantities in a shop.

```
fruits = {
      "apple": 4,
      "orange": 5,
      "pear": 6
}
```

Write a Python script to write to a text file out.txt in this format:

```
fruit,quantity
apple,4
orange,5
pear,6
```

40) Writing A Nested Dictionary To A File

You are given a nested dictionary containing some fruits and their information.

```
fruits = {
      "apple": {"price": 2.5, "quantity":10, "origin": "sg"},
      "orange": {"price": 1.5, "quantity":6, "origin": "sg"},
      "pear": {"price": 3.5, "quantity":4, "origin": "my"},
}
```

The values in the *fruits* dictionary are smaller dictionaries containing more information regarding each fruit. Write a Python script to write this information to a text file in this format:

```
fruit,price,quantity,origin
apple,2.5,10,sg
orange,1.5,6,sg
pear,3.5,4,my
```

Answers

1) What Year Was I Born In?

```
age = input("Enter your age :")  # asking for age input
age = int(year)                  # converting age to int type
year_born = 2022 - age           # computing year user was born
print(f"My age is {age}, and I was born in {year_born}")
```

1. We ask our user for their age using the *input* function. Note that the input function *always* returns a string value.
2. Since *age* is a string value, we need to convert it into an integer value as we need to do integer subtraction later.
3. We compute *year_born* by subtracting *age* from the current year

2) Sum Of Input Numbers

```
num1 = input("Enter the first number: ")      # prompt for num1
num2 = input("Enter the second number: ")     # prompt for num2

num1 = int(num1)                              # convert num1 to int
num2 = int(num2)                              # convert num2 to int
total = num1 + num2                           # adding num1 and num2

print(f"Sum of 2 numbers is {total}")
```

1. We first ask the user to enter 2 numbers using the *input* function. Remember that the 2 numbers will be strings as the *input* function always returns strings.
2. We convert both *num1* and *num2* into integer type as we need to do integer addition later.
3. We add *num1 & num2* together and assign it to *total.*

3) Repeating Pattern

```
char = input("Enter your pattern: ")    # prompt for character
number = input("Enter a number: ")      # prompt for number

number = int(number)                    # convert number to integer
pattern = char * number                 # string multiplication

print(pattern)
```

1. Using the *input* function, we first prompt the user for *character* (the character that gets multiplied) and then *number* (the number that *character* gets multiplied by)
2. As *number* will originally be a string, we need to convert it into an integer.
3. We then multiply *character & string* using the * operator

4) Area of a circle

```
def area(radius):
    return 3.14 * radius * radius
```

1. We first define our function *area* to take in 1 parameter *radius*
2. We apply the circle area formula, and multiple 3.14 with *radius^2*
3. The *return* keyword specifies the output of the function, so we return the area of the circle.

```
area(1)   # returns 3.14
area(2)   # returns 12.56
area(3)   # returns 28.26
```

5) Adding Numbers

```
def add(a, b, c):
    return a + b + c
```

1. We first define our function *add* to take in 3 parameters *a, b & c*
2. We simply add *a, b & c* using the + operator, and return it

```
add(1,2,3)   # returns 6 (1+2+3)
add(2,4,6)   # returns 12 (2+4+6)
add(1,1,9)   # returns 11 (1+1+9)
```

6) Larger Than 10

```
def larger10(number):
    if number > 10:
        return True     # happens if number is larger than 10
    else:
        return False    # happens if number is NOT larger than 10
```

Here, we return True (a boolean value) if our input number is larger than 10 (the if statement), and return False (boolean value) if our input number is NOT larger than 10 (the else statement). A more concise way to write this statement as you level up your Python:

```
def larger10(number):
    return number > 10
```

The statement *number > 10* itself evaluates to either True or False. Instead of using if-else statements, we can simply return *number > 10*.

```
larger10(5)     # False, as 5 is not larger than 10
larger10(10)    # False, as 10 is NOT larger than 10 (they're equal)
larger10(10.1)  # True, as 10.1 is larger than 10
larger10(12)    # True, as 12 is larger than 10
```

7) Area or Circumference Of A Circle

```
def calculate(mode, radius):
    if mode == "area":
        return 3.14 * radius * radius     # if "area" passed in

    elif mode == "circumference":
        return 2 * 3.14 * radius          # "circumference"

    else:
        return 0    # neither "area" nor "circumference"
```

1. Here, we need to write if-else statements to check what *mode* is.
2. If *mode* is *"area"*, we return the radius of the circle (pi * radius ^2)
3. If *mode* is *"circumference"*, we return the circumference (2 * pi * radius)
4. The else loop runs if *mode* is anything other than area or circumference, and it automatically returns 0.

```
calculate("area", 1)            # 3.14 (3.14 * 1^2)
calculate("circumference", 1)   # 6.28 (2 * 3.14 * 1)

calculate("area", 2)            # 12.56 (3.14 * 2^2)
calculate("circumference", 2)   # 12.56 (2 * 3.14 * 2)

calculate("area", 3)            # 28.26 (3.14 * 3^2)
calculate("circumference", 3)    # 18.84 (2 * 3.14 * 3)

calculate("stuff", 2)           # automatically returns 0
```

8) Larger & Largest

```
def larger(a, b):
    if a > b:
        return a    # returns a if a is larger than b
    return b        # returns b otherwise
```

Note - when a function returns something, the function *ends* and *nothing* happens afterwards. This is why there is no need for an *else* loop for the *return b* statement.

- If a is larger than b, the '*return a*' statement runs, and nothing executes in the function afterwards.
- If a is not larger than b, we don't reach the '*return a*' statement at all. Instead, the '*return b*' statement is executed straightaway

```
def largest(a, b, c):
    return larger(larger(a, b), c)

# larger(a, b) returns the larger of a and b
# The larger(larger(a,b), c) will be the largest number out of a,b,c
```

In the *largest(a, b, c)* function, we reuse our *larger(a, b)* function we have written previously.

- The expression *larger(a, b)* returns the larger number between a & b.
- The expression larger(larger(a, b), c) returns the larger number between 1) the larger number between a and b and 2) c. Which also happens to be the largest number between a, b and c.

```
larger(4, 5)       # 5
larger(5, 4)       # 5
largest(4, 5, 6)   # 6
largest(4, 6, 5)   # 6
```

9) Scholarship Eligibility

```
def eligible(english, math, science):
    if english > 80 and math > 80 and science > 80:
        return True
    else:
        return False
```

A student needs his/her english, math & science scores to *all* be above 80 to be eligible for the scholarship. As such, we can use the *and* logical operator in our if-else statement here.

```
# A more concise way to write this function
def eligible(english, math, science):
    return english > 80 and math > 80 and science > 80
```

Here, the statement *'english > 80 and math > 80 and science > 80'* itself evaluates to a boolean value. Instead of using the boolean value in an if-else statement to return another boolean value, we can simply return the boolean value.

```
eligible(80, 81, 81)      # False
eligible(81, 81, 81)      # True
eligible(100, 100, 79)    # False
eligible(90, 90, 90)      # True
```

10) Scholarship Eligibility 2

```
def eligible(english, math, science, conduct):
    all_above_80 = english > 80 and math > 80 and science > 80
    avg_above_85 = (english+math+science)/3 > 85

    return all_above_80 and avg_above_85 and conduct=="excellent"
```

Here, a student needs 3 conditions to be eligible for the scholarship.

1. All his scores must be more than 80
2. His average score must be more than 85
3. His conduct (string value) must be “excellent”

We hence create these 3 boolean values

1. *all_above_80* – True if all scores are more than 80, False otherwise
2. *avg_above_85* – True if average score exceeds 85, False otherwise
3. *conduct == “excellent”* – True if conduct is “excellent”, False otherwise

join them using the *and* logical operator, and return it.

11) Printing Odd Numbers

```
# for loop version
def odd(start, end):
    for number in range(start, end+1):
        if number % 2 == 1:    # True if number is odd, False otherwise
            print(number)
```

Remember that the 2nd argument in the *range* function (that specifies the *end* of the range) is exclusive, meaning that it won't be counted in the range. If we want *end* to be included, we need to add 1 to the second argument in the *range* function.

- range(1, 5) generates the numbers 1, 2, 3, 4 (5 itself is excluded)
- range(1, 6) generates the numbers 1, 2, 3, 4, 5 (6 itself is excluded)
- range(1, 7) generates the numbers 1, 2, 3, 4, 5, 6 (7 itself is excluded)

```
# while loop version
def odd(start, end):
    number = start
    while number <= end:       # runs forever as long as number <= end
        if number % 2 == 1:    # True if number is odd, False otherwise
            print(number)
        number += 1            # adds 1 to number
```

Our while loop keeps running as long as *number* does not exceed *end.* As such, we need to increase number by 1 at each iteration (the *'number += 1'* statement) to iteratively bring number closer and closer to the breaking condition (number > end), and prevent our program from going into an infinite loop.

12) Printing Even Numbers Backwards

```
# for loop version
def even_backwards(start, end):
    for number in range(end, start-1, -1):
        if number % 2 == 0:   # True if number is even, False otherwise
            print(number)
```

Here, we need to print the numbers in decreasing order, so we need to use a negative step (the 3rd argument) in our *range* function. Once again, the 2nd argument is exclusive, so we need to use *start-1* instead of *start*.

```
# while loop version
def even_backwards(start, end):
    number = end
    while number >= start:    # runs as long as number >= start
        if number % 2 == 0:   # True if number is even, False otherwise
            print(number)
        number -= 1           # subtracts 1 from number
```

As we need to print the numbers in decreasing order, we need to flip our while condition. Here, our while loop continues to run as long as *number* is more than *start*. Our program hence subtracts 1 from *number* at each step, iteratively bringing *number* closer to the breaking condition (number < start), preventing an infinite loop.

13) Pyramid Of Stars

```
# for loop version
def pyramid(n):
    for number in range(1, n+1):
        print("*" * number)
```

```
# while loop version
def pyramid(n):
    number = 1
    while number <= n:
        print("*" * number)
        number += 1
```

To print this pattern (given that n=3):

```
*
**
***
```

We first need to be able to generate these numbers:

```
1
2
3
```

Once we are able to generate these numbers using a for/while loop, we can simply multiply these numbers with the * character to print the pattern.

```
*     # "*" * 1
**    # "*" * 2
***   # "*" * 3
```

14) Thousands

```
# for loop version - incrementing by 1000 using the step argument.
def thousands(n):
    for i in range(1000, n*1000+1, 1000):
        print(i)
```

```
# for loop version 2 - step=1, but we multiply numbers by 1000
def thousands2(n):
    for i in range(1, n+1):
        print(i * 1000)
```

```
# while loop version - incrementing by 1000
def thousands(n):
    i = 1000
    while i <= n*1000:
        print(i)
        i += 1000
```

```
# while loop version 2 - incrementing by 1, but multiplying it by 1000
def thousands2(n):
    i = 1
    while i <= n:
        print(i*1000)
    i += 1
```

Here, we need to increase our number by 1000 at each iteration. There are 2 ways we can do this:

1. Actually increasing the number by 1000 at each iteration
2. Increasing the number by 1, but multiplying each number by 1000

Both get the job done - which to use is completely up to you.

15) Squares & Cubes

```
# for loop version
def squares_cubes(start, end):
    for i in range(start, end+1):
        print(i, i**2, i**3)     # printing i itself, i^2 and i^3
```

```
# while loop version
def squares_cubes(start, end):
    i = start
    while i <= end:
        print(i, i**2, i**3)     # printing i itself, i^2 and i^3
        i += 1
```

We first start off by being able to print these numbers

```
1
2
3
4
```

After we have verified that we can print these numbers, we simply also print the squares and cubes of each number.

```
1 1 1
2 4 8
3 9 27
4 16 64
```

16) Summation From 1

```
def summation(n):
    output = 0
    for i in range(1, n+1):
        output += i

    return output
```

Here, we want to compute 1+2+…+n. As we want to end up with an integer, we need to start with an *empty* integer, which is 0 in this case.

```
summation(5)
# range(1,5+1) generates the numbers 1,2,3,4 & 5
# output starts as 0

       action              output
i=1    add 1 to output     1
i=2    add 2 to output     3
i=3    add 3 to output     6
i=4    add 4 to output     10
i=5    add 5 to output     15

15 (1+2+3+4+5) is thus returned
```

```
summation(7)
# range(1, 7+1) generates numbers 1,2,3,4,5,6,7
# output starts as 0

       action              output
i=1    add 1 to output     1
i=2    add 2 to output     3
i=3    add 3 to output     6
i=4    add 4 to output     10
i=5    add 5 to output     15
i=6    add 6 to output     21
i=7    add 7 to output     28

28 (1+2+3+4+5+6+7) is thus returned
```

17) Factorial

```
def factorial(n):
    output = 1
    for i in range(1, n+1):
        output *= i

    return output
```

Here, we want to compute 1* 2 * …* n. Similarly, as we want to end up with an integer, we need to start with an *empty* integer, which is 1 in this case as we are doing multiplication. (We cannot use 0 as anything multiplied with 0 is still 0)

```
factorial(5)
# range(1,6) generates numbers 1,2,3,4,5
# output starts as 1

      action                 output
i=1   multiply output by 1   1
i=2   multiply output by 2   2
i=3   multiply output by 3   6
i=4   multiply output by 4   24
i=5   multiply output by 5   120

120 (1x2x3x4x5) is thus returned
```

```
factorial(7)
# range(1,7+1) generates numbers 1,2,3,4,5,6,7
# output starts as 1

      action                 output
i=1   multiply output by 1   1
i=2   multiply output by 2   2
i=3   multiply output by 3   6
i=4   multiply output by 4   24
i=5   multiply output by 5   120
i=6   multiply output by 6   720
i=7   multiply output by 7   5040

5040 (1x2x3x4x5x6x7) is thus returned
```

18) Summation Of Odd Numbers

```
def sum_odd(start, end):
    output = 0
    for i in range(start, end+1):
        if i % 2 == 1:      # True if i is odd, False otherwise
            output += i     # add i to output only if i is odd

    return output
```

Here, we wish to compute the sum of *only odd* numbers from *start* to *end.* As we want to end up with an integer, we need to start with an *empty* integer.

```
sum_odd(1,5)
# range(1,5+1) generates the numbers 1,2,3,4,5
# output starts as 0

       i%2==1    action            output
i=1    True      add 1 to output   1
i=2    False     no                1
i=3    True      add 3 to output   4
i=4    False     no                4
i=5    True      add 5 to output   9
```

```
sum_odd(2,8)
# range(2,8+1) generates the numbers 2,3,4,5,6,7,8
# output starts as 0

       i%2==1    action            output
i=2    False     no                0
i=3    True      add 3 to output   3
i=4    False     no                3
i=5    True      add 5 to output   8
i=6    False     no                8
i=7    True      add 7 to output   15
i=8    False     no                15

15 (3+5+7) is thus returned
```

19) Printing Triangular Numbers

```
def triangular(n):
    total = 0
    for i in range(1, n+1):
        total += i
        print(total)
```

We need to accumulate numbers using a for loop into a variable, and also print it at every iteration.

```
triangular(5)
# total starts at 0

       total       action
i=1    0+1 = 1     print 1
i=2    1+2 = 3     print 3
i=3    3+3 = 6     print 6
i=4    6+4 = 10    print 10
i=5    10+5 = 15   print 15
```

20) Sum To 100000

```
total = 0
for i in range(1, 100001):    # generates numbers from 1 to 100000
            total += i        # add i to total

print("sum from 1 to 100,000 is", total)
```

Don't be intimidated by the large number - we simply need to tweak our for loop to generate numbers from 1 to 100000 and this can be easily done using the *range* function. Remember to use 100001 instead of 100000, as the *end* argument in the *range* function is exclusive.

21) Summation Of String Of Numbers

```
def summation(numbers_string):
    output = 0
    for i in numbers_string.split():
        output += int(i)

    return output
```

Here, we can use the built-in .split() function to separate the string by whitespace into a list of smaller strings. We then loop through the list, convert each string into integer, and accumulate them into output.

```
"1 2 3 4".split()  ->  ["1", "2", "3", "4"]

        int(i)   output
i="1"   1        1   (0+1)
i="2"   2        3   (1+2)
i="3"   3        6   (3+3)
i="4"   4        10  (6+4)

10 (1+2+3+4) is thus returned
```

22) Printing Words In A Sentence

```
def print_words(sentence):
    for word in sentence.split():
        print(word)
```

To separate a string by whitespace, we can use the built-in *.split()* function to turn the string into a list of smaller strings, and then print the strings one by one.

```
"I have a dog".split()  ->  ["I", "have", "a", "dog"]

word   action
I      print "I"
have   print "have"
a      print "a"
dog    print "dog"
```

23) Removing Vowels

```
def remove_vowels(string):
    output = ""
    for letter in string:
        if letter not in "aeiou":
            output += letter

    return output
```

In this case, we want to end up with a string, so we need to start with an empty string. At each iteration, we check if a letter is a vowel by seeing if it exists inside a string of vowels. If we meet a vowel, we take no action. Else, we add it to output.

```
remove_vowels("apple")
# output starts as ""

letter  is_vowel   action            output
a       True       no
p       False      add p to output   p
p       False      add p to output   pp
l       False      add p to output   ppl
e       True       no                ppl

"ppl" is thus returned
```

```
remove_vowels("pineapple")
# output starts as ""

letter  is_vowel   action             output
p       False      add p to output    p
i       True       no                 p
n       False      add n to output    pn
e       True       no                 pn
a       True       no                 pn
p       False      add p to output    pnp
p       False      add p to output    pnpp
l       False      add l to output    pnppl
e       True       no                 pnppl

"pnppl" is thus returned
```

24) String Pyramid

```
def pyramid(string):
    for i in range(len(string)):
        print(string[:i+1])
```

Let's first break down the pattern we need to print:

```
# string = "pineapple"

p           # string[:1]
pi          # string[:2]
pin         # string[:3]
pine        # string[:4]
pinea       # string[:5]
pineap      # string[:6]
pineapp     # string[:7]
pineappl    # string[:8]
pineapple   # string[:9]
```

If we are able to print the numbers from 1 to 9 (with respect to the input *string*), we can print the pattern. To get these numbers, we can simply use a for loop.

```
pyramid("pineapple")
# len(string) is 9
# range(len(string)) generates the numbers 0,1,2,3,4,5,6,7,8

      i+1   string[:i+1]   prints
i=0   1     string[:1]     p
i=1   2     string[:2]     pi
i=2   3     string[:3]     pin
i=3   4     string[:4]     pine
i=4   5     string[:5]     pinea
i=5   6     string[:6]     pineap
i=6   7     string[:7]     pineapp
i=7   8     string[:8]     pineappl
i=8   9     string[:9]     pineapple
```

25) Alternate Letters

As we want our function to return a string, we need to start off with an empty string. We thus add only letters whose indexes are even numbers (0, 2, 4, ...)

```
def alternate_letters(string):
    out = ""
    for i in range(len(string)):
        if i % 2 == 0:
            out += string[i]
    return out
```

```
# alternative solution using string slicing (with step=2)
def alternate_letters(string):
    return string[::2]
```

26) Sum/Product Of List Of Integers

```
def sum_list(lis):
    output = 0
    for number in lis:
        output += number
    return output
```

As we want our function to return a number, we need to start with an empty number 0.

```
sum_list([1,5,6,7])

number   output
1        1  (0+1)
5        6  (1+5)
6        12 (6+6)
7        19 (12+7)

19 (0+1+5+6+7) is thus returned
```

```
#alternative solution using built-in sum function
def sum_list(lis):
    return sum(lis)
```

27) Converting List Of Strings To List Of Integers

```
def convert_int(lis):
    output = []
    for number in lis:
        output.append(int(number))

    return output
```

We want our function to return a list, so we need to start off with an empty list.

```
convert_int(["1", "2", "3"])
# output starts off as an empty list []

number   int(number)   action              output
"1"      1            1 added to output   [1]
"2"      2            2 added to output   [1,2]
"3"      3            3 added to output   [1,2,3]

[1,2,3] is thus returned
```

28) Arithmetic Mean

```
def mean(lis):
    return sum(lis) / len(lis)
```

Arithmetic mean is the sum of all elements divided by the number of elements. We derive the total sum of elements using *sum(lis)*, and the number of elements using *len(lis)*. We then divide *sum(lis)* by *len(lis)* to get the arithmetic mean (also known as average).

```
mean([4,5,6])

sum([4,5,6])   -> 12
len([4,5,6])   -> 3

4 (12/3) is thus returned
```

29) Median

```
def median(lis):
    lis = sorted(lis)              # returns sorted copy of lis

    if len(lis) % 2 == 0:          # if lis has even number of elements
        left = lis[len(lis)//2-1]
        right = lis[len(lis)//2]
        return (left+right)/2

    else:                          # if lis has odd number of elements
        return lis[len(lis)//2]
```

To find the median, we first need to sort the list.

- *sorted(lis)* returns a sorted copy of the list instead of sorting the original list itself.
- If our list has an even number of elements, we need to return the average of the 2 middle elements.
- If our list has an odd number of elements, we simply need to return the middle element.

```
median([1,5,2,4,3])

sorted(lis)   -> [1,2,3,4,5]
len(lis)      -> 5
len(lis)//2   -> 2
lis[2]        -> 3

3 is thus returned
```

```
median([1,5,2,4,3,6])

sorted(lis)     -> [1,2,3,4,5,6]
len(lis)        -> 6
left            -> 3
right           -> 4
(left+right)/2  -> 3.5

3.5 is thus returned
```

30) Filtering Numbers Divisible By 3

```
def filter3(lis):
    output = []
    for number in lis:
        if number % 3 == 0:   # True if number is divisible by 3
            output.append(number)
    return output
```

We only want to keep numbers that are divisible by 3.

- We can check if a number is divisible by 3 using *number % 3 == 0*
- We start with an empty list *output*, iterate through *lis*, and add numbers to *output* only if they are divisible by 3

```
filter3([1,2,3,4,5])

number   number%3   divisible_by_3   action          output
1        2          False            ignore          []
2        2          False            ignore          []
3        0          True             add to output   [3]
4        1          False            ignore          [3]
5        2          False            ignore          [3]

[3] is thus returned
```

```
filter3([2,3,4,5,6,7])

number   number%3   divisible_by_3   action          output
2        2          False            ignore          []
3        0          True             add to output   [3]
4        1          False            ignore          [3]
5        2          False            ignore          [3]
6        0          True             add to output   [3,6]
7        1          False            ignore          [3,6]

[3,6] is thus returned
```

31) Finding Square Of Numbers

```
def square(lis):
    out = {}
    for number in lis:
        out[number] = number**2
    return out
```

We want our function to return a dictionary, so we start with an empty dictionary {}. We then iterate through the numbers in the list, generate the square of the number, and assign out[number] = number**2

```
square([4,5,6,7])

number   number**2   action      output
4        16          out[4]=16   {4:16}
5        25          out[5]=25   {4:16, 5:25}
6        36          out[6]=36   {4:16, 5:25, 6:36}
7        49          out[7]=49   {4:16, 5:25, 6:36, 7:49}

{4:16, 5:25, 6:36, 7:49} is thus returned
```

```
square([9,5,10,8])

number   number**2   action      output
9        81          out[9]=81   {9:81}
5        25          out[5]=25   {9:81, 5:25}
10       100         out[10]=100 {9:81, 5:25, 10:100}
8        64          out[8]=64   {9:81, 5:25, 10:100, 8:64}

{9:81, 5:25, 10:100, 8:64} is thus returned
```

32) Finding Letter Counts In A Word

```
def letter_counts(word):
    out = {}
    for letter in word:
        if letter not in out:  # if letter doesn't exist in out
            out[letter] = 1    # create new key-value pair

        else:                  # if letter already exists in out
            out[letter] += 1   # increment value by 1
    return out
```

When finding letter counts, we might meet repeated letters that has been seen before

- If we meet a new letter that does not exist in *out*, we set its count to 1
- If we meet a letter that is *already* in *out*, we simply increase its count by 1 (instead of setting its count to 1)

```
letter_counts("pineapple")

letter   action      out
"p"      new pair    {"p":1}
"i"      new pair    {"p":1, "i":1}
"n"      new pair    {"p":1, "i":1, "n":1}
"e"      new pair    {"p":1, "i":1, "n":1, "e":1}
"a"      new pair    {"p":1, "i":1, "n":1, "e":1, "a":1}
"p"      increment   {"p":2, "i":1, "n":1, "e":1, "a":1}
"p"      increment   {"p":3, "i":1, "n":1, "e":1, "a":1}
"l"      new pair    {"p":3, "i":1, "n":1, "e":1, "a":1, "l":1}
"e"      increment   {"p":3, "i":1, "n":1, "e":2, "a":1, "l":1}

{"p":3, "i":1, "n":1, "e":2, "a":1, "l":1} is thus returned
```

33) Combining 2 Dictionaries

```
def combine(d1, d2):
    out = d1.copy()                 # creating exact copy of d1
    for key, value in d2.items():   # generates keys + values
        if key in out:
            out[key] += value
        else:
            out[key] = value

    return out
```

Here, we want our function to combine both dictionaries d1 & d2.

- We first create an exact copy of d1 (so we don't mess up the original dictionary)
- We then add the stuff from d2 into our d1 copy
- If a key does not exist, we create a new key-value pair
- If a key already exists, we add the values together

```
d1 = {"a":4, "b":5, "c":6, "d":7}
d2 = {"c":1, "d":2, "e":3, "f":4}
combine(d1, d2)

out  -> {"a":4, "b":5, "c":7, "d":7}
    - .copy() creates an exact copy
    - changing out won't accidentally mess up d1

key  value  out
c    1      {"a":4, "b":5, "c":7, "d":7}   # c's value increased by 1
d    2      {"a":4, "b":5, "c":7, "d":9}   # d's value increased by 2
e    3      {"a":4, "b":5, "c":7, "d":9, "e":3}         # "e":3 added
f    4      {"a":4, "b":5, "c":7, "d":9, "e":3, "f":4}  # "f":4 added

{"a":4, "b":5, "c":7, "d":9, "e":3, "f":4} is then returned
```

34) Reversing Keys & Values In A Dictionary With Unique Values

```
def reverse(d):
    out = {}
    for key, value in d.items():
        out[value] = key

    return out
```

Remember that in a dictionary, keys must be unique. In this context, values are unique, so we can simply iterate through all key-value pairs, and set d[value] = key.

```
d = {"apple":"pie", "orange":"juice", "pear":"cake"}
reverse(d)

key       value     out
apple     pie       {"pie":"apple"}
orange    juice     {"pie":"apple", "juice":"orange"}
pear      cake      {"pie":"apple", "juice":"orange", "cake":"pear"}

{"pie":"apple", "juice":"orange", "cake":"pear"} is hence returned
```

35) Reversing Keys & Values In A Dictionary With Non-Unique Values

```
def reverse(d):
    out = {}
    for key, value in d.items():
        if value not in out:
            out[value] = [key]      # if value does not exist
        else:
            out[value].append(key)  # if value already exists

    return out
```

In this context, values are *not* unique. In our final dictionary, each key has 1 or more values, so we need to use a list to store the values.

- If we meet a new value, we create a new key-value pair (the value is a list)
- Else if a value already exists, we simply append to the existing list.

```
d = {"a":4, "b":5, "c":6, "d":4, "e":5}

key  value  out
a    4      {4:["a"]}
b    5      {4:["a"], 5:["b"]}
c    6      {4:["a"], 5:["b"], 6:["c"]}
d    4      {4:["a", "d"], 5:["b"], 6:["c"]}
e    5      {4:["a", "d"], 5:["b", "e"], 6:["c"]}

{4:["a", "d"], 5:["b", "e"], 6:["c"]} is hence returned
```

36) Reading From A File Into A List

```
lis = []
with open("fruits.txt") as f:
    for line in f:            # iterating fruits.txt line by line
        line = line.strip()   # removing newline character
        lis.append(line)      # adding line to lis
```

- '*for line in f*' allows us to iterate through the file line by line
- by default, *line* has a newline character at the end
- we need to remove the newline character using the .strip() function
- We then add each *line* to lis

37) Counting Fruits From A File

```
d = {}
with open("fruits.txt") as f:
    for line in f:
        line = line.strip()       # removing newline character
        line = line.split(",")    # separating line by comma
        fruit = line[0]           # extract fruit
        count = int(line[1])      # extract count & convert to int

        if fruit not in d:
            d[fruit] = count
        else:
            d[fruit] += count
```

Here, the fruit and count (in each line) are separated by commas, so we need to do some preprocessing before we add them into out output dictionary

- The *.strip()* function removes the newline (\n) character
- The *.split()* function separates the line be a comma, separating fruit from count
- We need to convert count from a string to an integer
- If *fruit* does not exist inside our output dictionary *d*, we create a new key-value pair
- If *fruit* already exists inside *d*, we simply increment its value

```
line        line.split()       d
apple,4     ["apple", "4"]     {"apple":4}
orange,5    ["orange", "5"]    {"apple":4, "orange":5}
pear,6      ["pear", "6"]      {"apple":4, "orange":5, "pear":6}
apple,20    ["apple", "20"]    {"apple":24, "orange":5, "pear":6}
orange,16   ["orange", "16"]   {"apple":24, "orange":21, "pear":6}

{"apple":24, "orange":21, "pear":6} is thus returned
```

38) Finding Average Scores For Each Subject

```
english = []
math = []
science = []
with open("scores.txt") as f:
    first = True            # created to ignore first line
    for line in f:
        if first:           # this only runs onces
            first = False   # ignoring first line (header)
        else:
            line = line.strip()        # removing newline character
            line = line.split(",")     # separating by comma
            eng_score = int(line[1])   # extracting english score
            math_score = int(line[2])  # extracting math score
            sci_score = int(line[3])   # extracting science score

            english.append(eng_score)  # adding eng_score to english
            math.append(math_score)    # adding math_score to math
            science.append(sci_score)  # adding sci_score to science

d = {
    "english": sum(english)/len(english),  # average english score
    "math": sum(math)/len(math),           # average math score
    "science": sum(science)/len(science)   # average science score
}
```

- We create 3 lists - *english, math & science* – to keep track of the respective scores
- We need to ignore the first line of the text file (the header)
- Subsequent lines contain 4 values separated by commas - student name, english score, math score and science score.
- We use the *line.split(",")* method to separate the values by comma.
- We then add the scores to their respective lists
- After iterating through all lines, we compute the average score of each list, and put it in the dictionary d.

39) Writing A Dictionary To A File

```
fruits = {"apple": 4, "orange": 5, "pear": 6}

with open("out.txt", "w") as f:
    f.write("fruit,quantity\n")         # writing header

    for fruit, qty in fruits.items():
        f.write(f"{fruit},{qty}\n")     # writing each line
```

- We first write the header (remember to include the newline character \n)
- For each key-value pair, we write a formatted string to our output file - *fruit* & *qty* are separated by comma (once again, remember to include the newline character \n)

40) Writing A Nested Dictionary To A File

```
fruits = {
    "apple": {"price": 2.5, "quantity":10, "origin": "sg"},
    "orange": {"price": 1.5, "quantity":6, "origin": "sg"},
    "pear": {"price": 3.5, "quantity":4, "origin": "my"},
}

with open("out.txt", "w") as f:
    f.write("fruit,price,quantity,origin\n")   # writing header

    for fruit, info in fruits.items():
        price = info["price"]                  # extracting price
        qty = info["quantity"]                 # extracting quantity
        origin = info["origin"]                # extracting origin

        f.write(f"{fruit},{price},{qty},{origin}\n")
```

- Similarly, we first need to write the header first
- For each fruit, we need to write 4 values - fruit, price, qty & origin
- We extract them from the *info* dictionary, and write it to our file using a formatted string.

Conclusion

And there we have it – 40 Python practice questions for beginner-level Python learners. Hopefully these questions were not too easy, and were useful in strengthening your Python fundamentals.

Thanks for reading! It has been fun compiling these 40 questions from several Google drives that I use with my students, and I sincerely hope that this book has in some way made you a better Python programmer than before.

Yours truly,
Liu Zuo Lin