## Assignment #6

For this assignment you will be writing 3 programs which should be saved independently as their own “.py” file. The filename you should use for each program is outlined in the sections below. When you’re finished you should submit your programs to the Assignment #6 category inside of NYU Classes.

[Problem 1 |](https://emilydidthis.github.io/CSCI-UA.0002-Fall22/assignments/06.html#problem-1-simple-functions)[Problem 2 |](https://emilydidthis.github.io/CSCI-UA.0002-Fall22/assignments/06.html#problem-2)[Problem 3](https://emilydidthis.github.io/CSCI-UA.0002-Fall22/assignments/06.html#problem-3)

### Problem 1: Simple Functions

For this part you will be completing a series of short programming challenges that use various function concepts. Each challenge should be saved as its own file.

### Part 1a

Copy the following program into a new file. Then replace the underlined line of code with the correct statement. Make sure to follow the directions provided!

# these are the basic arithmetic functions you will be using for this challenge

# function: add

# input: two integers/floats

# processing: adds the two supplied values

# output: returns the sum (integer/float)

def add(a,b):

return a+b

# function: sub

# input: two integers/floats

# processing: subtracts the two supplied values

# output: returns the difference (integer/float)

def sub(a,b):

return a-b

# function: mult

# input: two integers/floats

# processing: multiplies the two supplied values

# output: returns the product (integer/float)

def mult(a,b):

return a\*b

# function: sqrt

# input: one integer/float

# processing: computes the square root of the supplied value

# output: returns the square root (float)

def sqrt(a):

return a\*\*0.5

# function: square

# input: one integer/float

# processing: raises the supplied integer/float to the 2nd power

# output: returns the square (integer/float)

def square(a):

return a\*\*2

# these are the two points you will be using

# point 1

x1 = 0

y1 = 0

# point 2

x2 = 100

y2 = 100

# compute the distance between the two points above using the distance formula.

# you may ONLY use the functions above to do this - no math operators are allowed!

# your calculation must also be done on a single line.

distance = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

print (distance) # answer should be 141.4213562373095

Recall that this is the [distance formula](https://andymath.com/wp-content/uploads/2019/08/distance-formula.jpg).

### Part 1b

Write two functions called ‘maximum’ and ‘minimum’ - these function should accept two arguments and return the larger/smaller of the two supplied values. For the purpose of this program you can always assume that the arguments being supplied are numeric. Describe your function using IPO notation. Your program should work perfectly with the following code:

a = 5

b = 10

c = 15

d = 20

ans1 = maximum(a,b)

ans2 = maximum(a,c)

ans3 = maximum(a,d)

print (ans1,ans2,ans3) # 10 15 20

ans4 = minimum(d,c)

ans5 = minimum(d,b)

ans6 = minimum(d,a)

print (ans4,ans5,ans6) # 15 10 5

ans7 = maximum( maximum(a,b), maximum(c,d) )

print ("The biggest is:", ans7)

### Part 1c

Write a function called ‘valid\_date’ which accepts two arguments - a month and a day (both integers, will always be integers). Test to see if the date being described is valid or not. If the date is valid your function should return a True value, and if it is not it should return a False value. For the purpose of this program you can assume February has 28 days. Describe your function using IPO notation. Your program should work perfectly with the following code:

print (valid\_date(99,1)) # False

print (valid\_date(1,99)) # False

print (valid\_date(99,99)) # False

print (valid\_date(-99,1)) # False

print (valid\_date(1,-99)) # False

print (valid\_date(-99,-99)) # False

print (valid\_date(9,25)) #True

print (valid\_date(10,15)) # True

print (valid\_date(11,31)) # False

print (valid\_date(2,28)) # True

print (valid\_date(2,29)) # False

### Part 1d

# write a function called 'simple\_sort\_version1' that accepts two values. you can assume

# that your three values will always be the same data type (all ints, all floats or all strings).

# sort these two values in ascending order and return them in that order.

# you may use any function that you've written so far in this assignment if you'd like to (maximum, minimum, etc)

# your function should work perfectly with the following lines of code

a,b = simple\_sort\_version1(10,20)

print (a,b) # 10 20

a,b = simple\_sort\_version1(20,10)

print (a,b) # 10 20

a,b = simple\_sort\_version1(30,30)

print (a,b) # 30 30

### Part 1e

# next, write a new function called 'simple\_sort\_version2' that accepts three values. you can assume

# that your three values will always be the same data type (all ints, all floats or all strings).

# sort these values in ascending order and return them.

# you may use any function that you've written so far in this assignment if you'd like to (simple\_sort\_version1, maximum, minimum, etc)

# your function should work perfectly with the following lines of code

a,b,c = simple\_sort\_version2(10,20,30)

print (a,b,c) # 10 20 30

a,b,c = simple\_sort\_version2(10,30,20)

print (a,b,c) # 10 20 30

a,b,c = simple\_sort\_version2(30,20,10)

print (a,b,c) # 10 20 30

a,b,c = simple\_sort\_version2(30,20,20)

print (a,b,c) # 20 20 30

### Problem 2

For part 2 of this assignment you will be writing a series of programs that build on one another, so it’s important to progress through this part in order. You only need to submit the final version of your code (no need to submit the intermediate steps that you took in order to reach the final program). When you’re finished you should submit your program to the Assignment #6 category inside of NYU Classes.

### Part 2a: Number Analyzer

Write a series of five functions that determine if a given positive integer is EVEN, ODD, PRIME, PERFECT and ABUNDANT. Each of these functions should accept a single integer as an argument and return whether the integer meets the criteria for that classification. Here’s some IPO notation to get you started:

# function: is\_even

# input: a positive integer

# processing: determines if the supplied number is even

# output: boolean

# function: is\_odd

# input: a positive integer

# processing: determines if the supplied number is odd

# output: boolean

# function: is\_prime

# input: a positive integer

# processing: determines if the supplied number is prime. a prime number is

# a number that only has two factors - the number 1 and itself.

# for example, 17 is prime because it only has two factors (1 and 17).

# in order to determine this you might need to count the # of factors

# between 1 and the number you are testing. a note on efficiency: if you are

# testing a number (say, 15) and you find a factor (say, 3) - do you need to even continue

# to find additional factors?

# output: boolean

# function: is\_perfect

# input: a positive integer

# processing: determines if the supplied number is perfect. a perfect number

# is a number that is equal to the sum of its factors (i.e. the

# number 6 is perfect because 6 = 1 + 2 + 3)

# output: boolean

# function: is\_abundant

# input: a positive integer

# processing: determines if the supplied number is abundant. an abundant number

# is a number that is less than the sum of its factors (i.e. the

# number 12 is abundant because 12 < 1 + 2 + 3 + 4 + 6)

# output: boolean

Test your code before continuing!!! Make sure that your functions work as desired and then move onto the next step. Here’s some sample code that will help you identify whether your functions are working correctly:

a1 = is\_even(5)

a2 = is\_even(6)

print (a1, a2) # False True

b1 = is\_odd(5)

b2 = is\_odd(6)

print (b1, b2) # True False

c1 = is\_prime(5)

c2 = is\_prime(17)

c3 = is\_prime(21)

print (c1,c2,c3) # True True False

d1 = is\_perfect(6)

d2 = is\_perfect(7)

d3 = is\_perfect(10)

print (d1,d2,d3) # True False False

e1 = is\_abundant(12)

e2 = is\_abundant(13)

e3 = is\_abundant(14)

print (e1,e2,e3) # True False False

Once you’re happy with your functions you should write a short program to prompt the user for a starting and ending number (both integers) - ensure that the first number is positive, and that the second number is greater than the first number. Then use your functions to analyze all numbers in this range to and generate output similar to the following:

Enter starting number: -5

Invalid, try again

Enter starting number: 5

Enter ending number: 0

Invalid, try again

Enter ending number: 30

5 is ... odd prime

6 is ... even perfect

7 is ... odd prime

8 is ... even

9 is ... odd

10 is ... even

11 is ... odd prime

12 is ... even abundant

13 is ... odd prime

14 is ... even

15 is ... odd

16 is ... even

17 is ... odd prime

18 is ... even abundant

19 is ... odd prime

20 is ... even abundant

21 is ... odd

22 is ... even

23 is ... odd prime

24 is ... even abundant

25 is ... odd

26 is ... even

27 is ... odd

28 is ... even perfect

29 is ... odd prime

30 is ... even abundant

### Part 2b: Number Analyzer

Next, write a program that prompts the allows the user to analyze numbers within a given range for numbers that fit the above criteria. The program should continue to execute as long as the user wishes to keep going. Your program should rely on the functions you wrote for part 2a - you shouldn’t have to re-write the routines that you worked out to compute whether a number is prime, abundant, perfect, odd or even. Here’s a sample running of the program:

Main Menu

1 - Find all prime numbers within a given range

2 - Find all perfect numbers within a given range

3 - Find all abundant numbers within a given range

4 - Chart all even, odd, prime, perfect and abundant numbers within a given range

5 - Quit

Your choice: apple

I don't understand that ...

Main Menu

1 - Find all prime numbers within a given range

2 - Find all perfect numbers within a given range

3 - Find all abundant numbers within a given range

4 - Chart all even, odd, prime, perfect and abundant numbers within a given range

5 - Quit

Your choice: 1

Enter starting number (positive only): -5

Invalid, try again

Enter starting number (positive only): 5

Enter ending number: 3

Invalid, try again

Enter ending number: 20

All prime numbers between 5 and 20

##############

5

7

11

13

17

19

##############

Main Menu

1 - Find all prime numbers within a given range

2 - Find all perfect numbers within a given range

3 - Find all abundant numbers within a given range

4 - Chart all even, odd, prime, perfect and abundant numbers within a given range

5 - Quit

Your choice: 2

Enter starting number (positive only): 1

Enter ending number: 1000

All perfect numbers between 1 and 1000

##############

6

28

496

##############

Main Menu

1 - Find all prime numbers within a given range

2 - Find all perfect numbers within a given range

3 - Find all abundant numbers within a given range

4 - Chart all even, odd, prime, perfect and abundant numbers within a given range

5 - Quit

Your choice: 3

Enter starting number (positive only): 1

Enter ending number: 100

All abundant numbers between 1 and 100

##############

12

18

20

24

30

36

40

42

48

54

56

60

66

70

72

78

80

84

88

90

96

100

##############

Main Menu

1 - Find all prime numbers within a given range

2 - Find all perfect numbers within a given range

3 - Find all abundant numbers within a given range

4 - Chart all even, odd, prime, perfect and abundant numbers within a given range

5 - Quit

Your choice: 4

Enter starting number (positive only): 1

Enter ending number: 100

Number Even Odd Prime Perfect Abundant

1 x

2 x x

3 x x

4 x

5 x x

6 x x

7 x x

8 x

9 x

10 x

11 x x

12 x x

13 x x

14 x

15 x

16 x

17 x x

18 x x

19 x x

20 x x

21 x

22 x

23 x x

24 x x

25 x

26 x

27 x

28 x x

29 x x

30 x x

31 x x

32 x

33 x

34 x

35 x

36 x x

37 x x

38 x

39 x

40 x x

41 x x

42 x x

43 x x

44 x

45 x

46 x

47 x x

48 x x

49 x

50 x

51 x

52 x

53 x x

54 x x

55 x

56 x x

57 x

58 x

59 x x

60 x x

61 x x

62 x

63 x

64 x

65 x

66 x x

67 x x

68 x

69 x

70 x x

71 x x

72 x x

73 x x

74 x

75 x

76 x

77 x

78 x x

79 x x

80 x x

81 x

82 x

83 x x

84 x x

85 x

86 x

87 x

88 x x

89 x x

90 x x

91 x

92 x

93 x

94 x

95 x

96 x x

97 x x

98 x

99 x

100 x x

Main Menu

1 - Find all prime numbers within a given range

2 - Find all perfect numbers within a given range

3 - Find all abundant numbers within a given range

4 - Chart all even, odd, prime, perfect and abundant numbers within a given range

5 - Quit

Your choice: 5

Goodbye!

Note that you might want to write some additional “helper” functions - for example, the options above all require the user to supply a range of numbers to analyze. You can probably write a function that does this so you don’t have to continually copy and paste the same code over and over again.

### Problem 3

For part 3 of this assignment you will be writing a series of programs that build on one another, so it’s important to progress through this part in order. You only need to submit the final version of your code (no need to submit the intermediate steps that you took in order to reach the final program) as well as the “module” that you create for Part 3a. When you’re finished you should submit your program to the Assignment #6 category inside of NYU Classes.

### Part 3a

For this step you are given three functions - go ahead and copy these functions into a new file called “digitalclock.py”:

# function: horizontal\_line

# input: a width value (integer) and a single character (string)

# processing: generates a single horizontal line of the desired size

# output: returns the generated pattern (string)

def horizontal\_line(width,char):

return width\*char + "\n"

# function: vertical\_line

# input: a shift value and a height value (both integers) and a single character (string)

# processing: generates a single vertical line of the desired height. the line is

# offset from the left side of the screen using the shift value

# output: returns the generated pattern (string)

def vertical\_line(shift,height,char):

pattern = ""

for i in range(height):

pattern += shift\*" " + char + "\n"

return pattern

# function: two\_vertical\_lines

# input: a width value and a height value (both integers) and a single character (string)

# processing: generates two vertical lines. the first line is along the left side of

# the screen. the second line is offset using the "width" value supplied

# output: returns the generated pattern (string)

def two\_vertical\_lines(width,height,char):

pattern = ""

for i in range(height):

pattern += char + " "\*(width-2) + char + "\n"

return pattern

Next, create a file called ‘LastName\_FirstName\_Assign6\_part3.py’ - make sure this file is in the same folder as your newly created ‘digitalclock.py’ file. Import your module and run the following code - you should be able to see a series of graphical patterns that match the output below:

import digitalclock

print ("Horizontal line, width = 5:")

temp = digitalclock.horizontal\_line(5, '\*')

print (temp)

print ()

print ("Horizontal line, width = 10:")

temp = digitalclock.horizontal\_line(10, '+')

print (temp)

print ()

print ("Horizontal line, width = 15:")

temp = digitalclock.horizontal\_line(15, 'z')

print (temp)

print ()

print ("Vertical Line, shift=0; height=3:")

temp = digitalclock.vertical\_line(0, 3, '!')

print (temp)

print ()

print ("Vertical Line, shift=3; height=3:")

temp = digitalclock.vertical\_line(3, 3, '&')

print (temp)

print ()

print ("Vertical Line, shift=6; height=5:")

temp = digitalclock.vertical\_line(6, 5, '$')

print (temp)

print ()

print ("Two Vertical Lines, width=3; height=3:")

temp = digitalclock.two\_vertical\_lines(3, 3, '^')

print (temp)

print ()

print ("Two Vertical Lines, width=4; height=5:")

temp = digitalclock.two\_vertical\_lines(4, 5, '@')

print (temp)

print ()

print ("Two Vertical Lines, width=5; height=2:")

temp = digitalclock.two\_vertical\_lines(5, 2, '#')

print (temp)

print ()

Expected Output:

Horizontal line, width = 5:

\*\*\*\*\*

Horizontal line, width = 10:

++++++++++

Horizontal line, width = 15:

zzzzzzzzzzzzzzz

Vertical Line, shift=0; height=3:

!

!

!

Vertical Line, shift=3; height=3:

&

&

&

Vertical Line, shift=6; height=5:

$

$

$

$

$

Two Vertical Lines, width=3; height=3:

^ ^

^ ^

^ ^

Two Vertical Lines, width=4; height=5:

@ @

@ @

@ @

@ @

@ @

Two Vertical Lines, width=5; height=2:

# #

# #

### Part 3b

As you can see, you have three “primitive” functions for generating simple shapes (horizontal lines, vertical lines and parallel vertical lines). Your next task is to write 10 new functions that generate the digits 0-9 using your three line functions. These functions should be stored in your ‘digitalclock.py’ module. The goal here is to render the digits as they would appear on a digital display:

黑色的钟表

描述已自动生成

Each function should accept a “width” argument to control how wide the number should be as well as a single character. You can assume numbers will always be printed with a height of 5. For example, here is the function for the number 1:

# function: number\_1

# input: a width value (integer) and a single character (string)

# processing: generates the number 1 as it would appear on a digital display

# using the supplied width value

# output: returns the generated pattern (string)

def number\_1(width, character):

pattern = vertical\_line(width-1, 5, character)

return pattern

And here’s a sample program that calls the function a few times (test this in your main program, not in your module):

print ("Number 1, width=5: ")

temp = digitalclock.number\_1(5, '\*')

print(temp)

print()

print ("Number 1, width=10: ")

temp = digitalclock.number\_1(10, '\*')

print(temp)

print()

print ("Number 1, width=2: ")

temp = digitalclock.number\_1(2, '\*')

print(temp)

print()

And here’s the expected output:

Number 1, width=5:

\*

\*

\*

\*

\*

Number 1, width=10:

\*

\*

\*

\*

\*

Number 1, width=2:

\*

\*

\*

\*

\*

Here’s a sample program that prints all of the numbers (0-9).

temp = digitalclock.number\_0(5, '\*')

print(temp)

print()

temp = digitalclock.number\_1(5, '\*')

print(temp)

print()

temp = digitalclock.number\_2(5, '\*')

print(temp)

print()

temp = digitalclock.number\_3(5, '\*')

print(temp)

print()

temp = digitalclock.number\_4(5, '\*')

print(temp)

print()

temp = digitalclock.number\_5(5, '\*')

print(temp)

print()

temp = digitalclock.number\_6(5, '\*')

print(temp)

print()

temp = digitalclock.number\_7(5, '\*')

print(temp)

print()

temp = digitalclock.number\_8(5, '\*')

print(temp)

print()

temp = digitalclock.number\_9(5, '\*')

print(temp)

print()

And here’s the expected output:

\*\*\*\*\*

\* \*

\* \*

\* \*

\*\*\*\*\*

\*

\*

\*

\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\* \*

\* \*

\*\*\*\*\*

\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*\*\*\*\*

\*

\*\*\*\*\*

\* \*

\*\*\*\*\*

\*\*\*\*\*

\*

\*

\*

\*

\*\*\*\*\*

\* \*

\*\*\*\*\*

\* \*

\*\*\*\*\*

\*\*\*\*\*

\* \*

\*\*\*\*\*

\*

\*

### Part 3c

Write a function called ‘print\_number’ that prints out any desired number to the screen. This function should also be placed in your ‘digitalclock.py’ module. Here’s the IPO for this function:

# function: print\_number

# input: a number to print (integer), a width value (integer) and a single character (string)

# processing: prints the desired number to the screen using the supplied width value

# output: does not return anything

And here’s some sample code that you can use to test your function:

digitalclock.print\_number(0, 5, '\*')

digitalclock.print\_number(1, 6, '\*')

digitalclock.print\_number(2, 7, '\*')

digitalclock.print\_number(3, 8, '\*')

digitalclock.print\_number(4, 9, '\*')

digitalclock.print\_number(5, 10, '\*')

digitalclock.print\_number(6, 11, '\*')

digitalclock.print\_number(7, 12, '\*')

digitalclock.print\_number(8, 13, '\*')

digitalclock.print\_number(9, 14, '\*')

And here’s the expected output:

\*\*\*\*\*

\* \*

\* \*

\* \*

\*\*\*\*\*

\*

\*

\*

\*

\*

\*\*\*\*\*\*\*

\*

\*\*\*\*\*\*\*

\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

\*

\*\*\*\*\*\*\*\*

\*

\*\*\*\*\*\*\*\*

\* \*

\* \*

\*\*\*\*\*\*\*\*\*

\*

\*

\*\*\*\*\*\*\*\*\*\*

\*

\*\*\*\*\*\*\*\*\*\*

\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*

\*\*\*\*\*\*\*\*\*\*\*

\* \*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*

\*

\*

\*

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\*

### Part 3d

Write two new functions that simulate the addition and subtraction operators. Each of these functions should accept a width value as an argument (integer) and a single character (string) – the function should then return the generated pattern. You can assume the operators will always be 5 units high. Again, these functions should be placed in your ‘digitalclock.py’ module. Here’s some sample code:

temp = digitalclock.plus(5, '\*')

print(temp)

print()

temp = digitalclock.minus(5, '\*')

print(temp)

Which will generate ...

\*

\*

\*\*\*\*\*

\*

\*

\*\*\*\*\*

Note that your “plus” sign may look odd if it is rendered using an even size value - for example:

# rendered using a width of 6

\*

\*

\*\*\*\*\*\*

\*

\*

To fix this you should double up the vertical line in the center for even sizes, like this:

# rendered using a width of 6

\*\*

\*\*

\*\*\*\*\*\*

\*\*

\*\*

### Part 3e

Write a function called “check\_answer” which will determine if a given addition or subtraction problem was solved correctly. This function should be inside of your “digitalclock.py” module. Here’s the IPO notation for the function:

# function: check\_answer

# input: two numbers (number1 & number2, both integers); an answer (an integer)

# and an operator (+ or -, expressed as a String)

# processing: determines if the supplied expression is correct. for example, if the operator

# is "+", number1 = 1, number2 = 2 and answer = 3 then the expression is correct

# (1 + 2 = 3).

# output: returns True if the expression is correct, False if it is not correct

Here’s a sample program that you can use to test your function:

answer1 = digitalclock.check\_answer(1, 2, 3, "+")

print (answer1)

answer2 = digitalclock.check\_answer(1, 2, -1, "-")

print (answer2)

answer3 = digitalclock.check\_answer(9, 5, 3, "+")

print (answer3)

answer4 = digitalclock.check\_answer(8, 2, 4, "-")

print (answer4)

And here’s the expected output:

True

True

False

False

### Part 3f

Finally, put everything together and write a program that lets the user practice a series of random addition and subtraction problems. Begin by asking the user for a number of problems (only accept positive values) and a size for their numbers (only accept numbers between 5 and 10). Also prompt them for a single character to be used to generate their patterns - only accept single character strings (i.e. ‘a’ is OK, but ‘apple’ is not). The generate a series of random addition and subtraction problems - display the numbers to the user with your digital display functions. Then prompt the user for an answer and check the answer using your check\_answer function. Your program should also keep track of how many correct questions the user answered during their game. Here’s a sample running of the program:

How many problems would you like to attempt? -5

Invalid number, try again

How many problems would you like to attempt? 5

How wide do you want your digits to be? 5-10: 3

Invalid width, try again

How wide do you want your digits to be? 5-10: 5

What character would you like to use? foo

String too long, try again

What character would you like to use? \*

Here we go!

What is .....

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*

\*\*\*\*\*

\*

\*

\*

\*

\*

\*

\*

= 4

Correct!

What is .....

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*

\*

\*

\*

= -5

Correct!

What is .....

\*

\*

\*

\*

\*

\*\*\*\*\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

= 0

Sorry, that's not correct.

What is .....

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*

\*\*\*\*\*

\*

\*

\*

\*

\*

\*

\*

= 3

Correct!

What is .....

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*

\*\*\*\*\*

\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

\*

\*\*\*\*\*

= 4

Correct!

You got 4 out of 5 correct!