Attach the assignment to your HW/Lab ENGR 102 Section **Lab 6b**
Date: _____ **DUE DATE: 10/03/22**
You name ____Team # (table)____________

ENGR 102 Sect Lab 6b-ind

22 points =100%

Reading assignment:

Lecture Slides: 2 presentations	**L06**
zyBook	**Loops**

Attention!!

Individual submission.

Deliverables:

There are three deliverables for this individual assignment. **Please submit these files to zyBooks:**

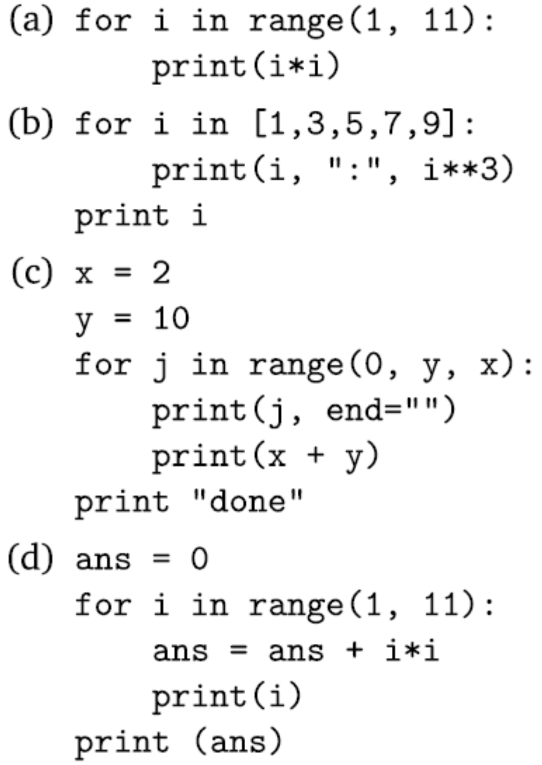
- howdy_whoop.py
- juggler_sequence.py
- balancing_numbers.py

Practice problems (no submission)

Practice Problem. What is the sequence of numbers that would be generated by each of the following statement

a) range(5)
b) range(3,10)
c) range(4, 13, 3)
d) range(15, 5, -2)

Practice Problem 2 What is the output that would be generated by each of the following programs



```
(a) for i in range(1, 11):
        print(i*i)
(b) for i in [1,3,5,7,9]:
        print(i, ":", i**3)
    print i
(c) x = 2
    y = 10
    for j in range(0, y, x):
        print(j, end="")
        print(x + y)
    print "done"
(d) ans = 0
    for i in range(1, 11):
        ans = ans + i*i
        print(i)
    print (ans)
```

Practice Problem 3 Theory:

For the most part, operations on floats produce floats, and operations on *ints* produce *ints*. Most of the time, we don't even worry about what type of operation is being performed; for example, integer addition produces pretty much the same result as floating point addition, and we can rely on Python to do the right thing.

In the case of division, however, things get a bit more interesting. As the table shows, Python (as of version 3.0) provides two different operators for division. The usual symbol *I* is used for "regular" division and a double slash *I I* is used to indicate integer division. The best way to get a handle on the difference between these two is to try them out.

Notice that the *I* operator always returns a float. Regular division often produces a fractional result, even though the operands may *be ints*. Remember, floating point values are always approximations. To get a division that returns an integer result, you can use the integer division operation //. Integer division always produces an integer. Think of integer division as "gozinta." The expression, 10 / 3 produces 3 because three gozinta (goes into) ten three times (with a remainder of one). While the result of integer division is always an integer; the data type of the result depends on the data type of the operands. A float integer-divided by a float produces a float with a 0 fractional component. The last is the remainder operation *%*. The remainder of integer-dividing 10 by 3 is 1. Notice again that the data type of the result depends on the type of the operands.

Depending on your math background, you may not have used the integer division or remainder operations before. The thing to keep in mind is that these two operations are closely related. Integer division tells you how many times one number goes into another, and the remainder tells you how much is left over: Mathematically you could write the idea like this: *a= (a/ /b)(b) +(a%b).*

As an example application, suppose we calculated the value of our loose change in cents (rather than dollars). If I have 383 cents, then I can find the number of whole dollars by computing *383//100 = 3*, and the remaining change is *383%100 = 83*. Thus, I must have a total of three dollars and 83 cents in change.

By the way, although Python, as of version 3.0, treats regular division and integer division as two separate operators, many other computer languages (and earlier Python versions) just use / to signify both. When the operands are *ints*, */* means *integer division*, and when they are *floats*, it signifies regular division. This is a common source of *errors*. For example, in our temperature conversion program the formula
*915 * celsius + 32* would not compute the proper result, since *9/5* would evaluate to 1 using integer division. In these "old-fashioned" languages, you need to be careful to write this expression as
*9. 015.0 * celsius + 32* so that the proper form of division is used, yielding a fractional result.

Practice Problem 3
What do you think will happen when the operands to the integer division or remainder operations are negative? Consider each of the following cases and try to predict the result. Then try them out in Python. Explain why.

(a) -10 / / 3
(b) -10 % 3
(c) 10 / / -3
(d) 10 % -3
(e) -10 / / -3

Theory. Rounding

You already know that combining an int with an int (usually) produces an *int*, and combining a *float* with a *float* creates another *float*. But what happens if we write an expression that mixes an *int* with a *float*? For example, what should the value of x be after this assignment statement?
*X = 5.0 * 2*

If this is floating point multiplication, then the result should be the *float* value *10.0*. If an *int* multiplication is performed, the result is *10*

In order to make sense of the expression *5. 0 * 2,* Python must either change *5. 0 to 5* and perform an *int* operation or convert *2 to 2.0* and perform floating point operation. In general, converting a *float* to an int is a **dangerous step**, because some information (the fractional part) **will be lost**. On the other hand, an *int* can be safely turned into a *float* just by adding a fractional part of .0. So, in mixed-typed expressions, Python will automatically convert *ints* to *float*s and perform **floating point** operations to produce a *float* result.

Sometimes we may want to perform a type conversion ourselves. This is called an **explicit type** conversion. Python provides the built-in functions int and float for these occasions. Here are some interactive examples that illustrate their behavior.

```
»> int(4.5)
4
»> int (3. 9)
3
»> float (4)
4.0
```

```
»> float (4. 5)
4.5
>>> float(int(3.3))
3.0
>>> int(float(3.3))
3
>>> int(float(3))
3
```

Converting to an *int* simply discards the fractional part of a *float*; the value is truncated, not rounded. If you want a rounded result, you could add *0.5* to the value before using *int ()*, assuming the value is positive.

A more general way of rounding off numbers is to use the built-in round function which rounds a number to the nearest whole value.

```
»> round(3.14)
3
»> round(3.5)
4
```

Notice that calling round like this results in an *int* value. So, a simple call to round is an alternative way of converting a *float to an int*. If you want to round a float into another float value, you can do that by supplying a second parameter that specifies the number of digits you want after the decimal point. Here's a little interaction playing around with the value of pi from the math library:

```
>>> import math
»> math.pi
3.1415926535897931
>>> round(math.pi, 2)
3.1400000000000001
>>> round(math.pi,3)
3.1419999999999999
>>> print(round(math.pi, 2))
3.14
>>> print(round(math.pi,3))
3.142
```

Notice that when we round pi to two or three decimal places, **we do not get exactly two or three** decimal places in the result. Remember, **floats are always approximations**; we get something that's very close to what we requested. However, the last two interactions show something interesting about the Python print statement. Python is smart enough to know that we probably don't want to see all the digits in something like 3.140000000000001, so it actually prints out the rounded form. That means if you write a program that rounds off a value to two decimal places and you print out the value, you'll end up seeing two decimal places, **just like you expect**.

Practice Problem 7

What do you think will happen if you use a negative number as the second parameter in the round function? For example, what should be the result of round(314.159265, -1). Explain the rationale for your answer. After you've written your answer; consult the Python documentation or try out some examples to see what Python actually does in this case. Put your explanation and examples here.

Deliverables:

There are three deliverables for this individual assignment. **Please submit these files to zyBooks:**

- `howdy_whoop.py`
- `juggler_sequence.py`
- `balancing_numbers.py`

Activity #1: Howdy Whoop – individual

Write a program named howdy_whoop.py that takes as input from the user two positive integers. Output the numbers 1 to 100, each on its own line, unless the number is evenly divisible by one or both of the integers entered by the user. If the number is evenly divisible by the first integer, print `Howdy`. If it's evenly divisible by the second integer, print `Whoop`. If it's evenly divisible by both, print `Howdy Whoop`. Format your output as shown below. User input is shown in bold and red text.

Example output (using inputs **2** and **3**):

```
Enter an integer: 2
Enter another integer: 3
1
Howdy
Whoop
...
Whoop
Howdy
```

Activity #2: Juggler sequence – individual

The Juggler sequence (https://en.wikipedia.org/wiki/Juggler_sequence) produces a sequence of integers using the following procedure. Given a number n, if n is even then the next number is the floor of the square root of n. If n is odd, then the next number is the floor of $n^{3/2}$. The Juggler sequence ends when n reaches 1.

As an example, if you start with the number 7, then the terms of the sequence will be: 7, 18, 4, 2, 1.

Write a program named juggler_sequence.py that takes in a positive integer from the user, and computes the Juggler sequence, printing out all the numbers in the sequence, followed by a line stating how many iterations it took to reach the value 1. Format your output as shown below. User input is shown in bold and red text.

Example output (using input **7**):

```
Enter a positive integer: 7
The Juggler sequence starting at 7 is:
7, 18, 4, 2, 1
It took 4 iterations to reach 1
```

<u>Activity #3</u>: Balancing numbers – individual

A positive number n is a balancing number if the sum of numbers from 1 to $(n-1)$ is equal to the sum of numbers $(n+1)$ to $(n+r)$ where r is a positive integer. Check out the first few minutes of this video for an explanation and examples: https://www.youtube.com/watch?v=jMfZ9jRsHSI.

For example, 6 is a balancing number with r of 2. That means the numbers 1 through 5 sum to the same amount as the next two integers after 6 (7 and 8). In other words,

$$\underbrace{1+2+3+4+5}_{15} = \underbrace{7+8}_{15}$$

Write a program named balancing_numbers.py that takes in an integer value n from the user and determines if it is a balancing number. If n is a balancing number, output the corresponding value of r. Do **NOT** use any containers like lists, tuples, sets, and dictionaries, or the `sum()` function. Format your output as shown below.

Example output (using input **6**):

```
Enter a value for n: 6
6 is a balancing number with r=2
```

Example output (using input **102**):

```
Enter a value for n: 102
102 is not a balancing number
```