LOOPS

Loops are an extremely useful feature in any programming language. They allow you to direct the computer to execute certain statements more than once. In Python, there are two kinds of loops: `while` loops and `for` loops. The `while` loop is easier to learn because it resembles the `if` statement, so we will look at it first. However, in the long run, the `for` loop is much more commonly used in real programs.

Repetition using the `while` statement

"Stir the soup until it boils." The concept of doing something repeatedly is second nature to us. Now let's teach the computer how to repeat an action.

In Python, we use a `while` statement to allow some statements to be performed several times. What is really nice is that the format of a while statement is very similar to if:

```
while <condition>:
    # steps that will be repeated
```

This group of statements that starts with the while statement and includes the body is called a *loop*. In particular, we would call this a "while loop" since it began with the word while. There is another type of loop called a "for-loop" that we will see later. Doing the body of the loop each time is called one *iteration* of the loop.

Here is how a while loop works: Python will first test the condition. If it is already false, then the entire loop is skipped and we go on to the next piece of the program. But if the condition is true, then we enter the loop and perform the statements in the body. When the body is finished, Python will re-evaluate the condition. If it is still true, the body is performed again! The process repeats itself. At some point, the condition needs to become false, so that we can eventually exit the loop.

The most basic kind of loop we need to write is how to teach the computer how to count. The following code will print the numbers from 1 to 5.

```
print("I'm going to count to 5:")

n = 1
while n <= 5:
    print(n)
    n += 1

print("I'm finished counting.")
```

Many of the while loops we will write will have the same basic structure of this example. And once you have written one loop, it is easy to adapt it to similar other problems. For instance, if we can count to 5, then we can easily modify the above code so that it can count to 1000 instead of 5. How would we make this change?

Another simple modification we can make to the above loop is to count by fives:

```
print("Let’s count by fives from 0 to 100.")

n = 0
while n <= 100:
    print(n)
    n += 5
```

Questions:

1. How would we modify the code so that it also prints out the sum of the numbers that it prints? In other words we want the sum of the multiples of 5 from 0 to 100 inclusive.

2. How would we modify the above code so that it prints the numbers in reverse order, i.e. from 100 down to 0?

3. Try this experiment: What errors result if your program is careless about indentation?

4. What numbers are printed by this Python code?
```
n = 8
while n < 12:
    print(n)
    n += 1
```

5. What is the output of this program?
```
n = 20
while n >= 5:
    print(n)
    n -= 1
print("After the loop, n equals", n)
```

Important observation: With a loop there is very often some “counter” variable that keeps changing on every iteration. In the above examples, it was this variable n. Notice that before the loop, n received some *initial value*, such as zero. Then, when we introduced the while loop, we established some *limit* on the value of n, such as 100. Finally, at the end of the loop body, we *increment* the counter variable by some amount. Almost always this increment is 1, but in the above example we saw that it could be something else like 5.

So, when you write loops, you should remember: *initial value, limit value, increment*

Since all the while loops we have seen have had the same basic structure, it is useful to keep in mind a basic pattern. Suppose you want a loop to do something exactly 20 times. Here is how you would do it using a while loop:

```
n = 0
while n < 20:
    # do whatever you need the loop to do on each iteration
    n += 1
```

The way the above code is written, it is guaranteed to perform the body exactly 20 times. If you wanted to do it 30 times, just change the 20 to 30.

<u>Application: Prime numbers</u>

A prime number is a positive integer that has exactly 2 factors, namely 1 and itself. Examples of prime numbers are 2, 3, 5, 7, 11, 13, etc.

How can we tell if a number <u>n</u> is prime? Here is an algorithm to do it.

- For each possible <u>divisor</u> from 1 to n, see if it divides evenly into n.
- We need to <u>count</u> how many times we find a divisor that divides evenly into n. Store this in another variable.
- When finished, see if the count is 2.

Let's illustrate the algorithm with some examples.

Is 5 a prime number?

- The possible divisors to test are 1, 2, 3, 4, 5.
- Do they divide evenly into 5? 1 → yes, 2 → no, 3 → no, 4 → no, 5 → yes
- We counted 2 divisors. It is prime.

Is 6 a prime number?

- The possible divisors to test are 1, 2, 3, 4, 5, 6.
- Do they divide evenly into 6? 1 → yes, 2 → yes, 3 → yes, 4 → no, 5 → no, 6 → yes
- We counted 4 divisors. It is NOT prime.

Here is the Python program to determine if a number is prime. It follows exactly from the algorithm just described above.

```
n = int(input("Enter a positive integer:  "))

# The divisor goes from 1 to n, inclusive, just like we are able
# to count from 1 to n.  Also, start count at 0.
count = 0
divisor = 1
while divisor <= n:
    if n % divisor == 0:
        count += 1
    divisor += 1

if count == 2:
    print("Your number is prime.")
else:
    print("Your number is not prime.")
```

Discussion:

1. Where is the variable `count` used in this program?

2. Where is the variable `divisor` used in this program?

3. What is the difference between the variables `divisor` and `n`?

4. If the value of `n` is 9, then what are the values of `count` and `divisor` at the end of the program?

<u>More about loops</u>

Earlier we wrote a loop to some simple counting and adding. Now, let's create a more useful loop: let's add five input numbers. We want to allow the user to enter 5 numbers, and calculate their sum. We'll need a loop, and inside the loop we need to do the following:

- Ask the user for the <u>next number</u>.
- Continually add this next number to the <u>sum</u>.
- Keep track of <u>how many</u> numbers so far have been read.

The information that we need to keep track of tells us what variables we want.

Important observation: If you want to add or count something, make sure your variable starts at zero.

Here is the program.

```
# add5.py – Let’s ask the user for 5 numbers, and then
# their sum.  It’s critical to think about the
# variables we need.

count = 0
sum = 0
while count < 5:
    nextNumber = int(input("Enter a number:  "))
    sum += nextNumber
    count += 1

print("The sum of your numbers is ", sum)
```

Discussion: What does this code accomplish?

```
count = 1
sum = 0
while count <= 100:
    sum += count
    count += 1
print(sum)
```

<u>Common loop mistakes</u>

We all make mistakes. Be patient and forgiving with yourself and others. When composing programs, we need to be careful and methodical, because one small mistake can have a profound effect on the output. There are three easy ways to mess up a loop, and I’ll illustrate them here with the task of trying to print the numbers from 1 to 10 inclusive.

In each example, see if you can determine exactly why the code is in error, and how you would fix it.

Mistake #1 – Condition initially false. In this case, the loop will not execute at all. It gets skipped.

```
num = 1
while num > 10:
    print(num)
    num = num + 1
```

Mistake #2 – Infinite loop. Here, the condition is always true; never has a chance to become false.

```
num = 1
while num <= 10:
    print(num)
```

Mistake #3 – Off by one error. This means we have one iteration too many or too few. I will illustrate both ways of making this error.

```
num = 1
while num < 10:
    print(num)
    num = num + 1
```

```
num = 1
while num <= 10:
    num = num + 1
    print(num)
```

Question: What is wrong with the following loops?

```
# Let’s add up the numbers from 1 to max, inclusive.
sum = 0
count = 1
while count <= max:
    sum += max
    count += 1
print(sum)

# --------------------------------------------
# Let’s count from 1-100 but odd numbers only.
count = 1
while count != 100:
    print(count)
    count += 2
```

Error checking of input

Now that we know how to write loops and if-statements, there is a lot we can do. As you know, very often in our programs, we need the user to enter some input. But what if the input doesn’t make any sense? The easiest thing is for the program to just accept the invalid input and ultimately spit out some worthless answer. But it would be more useful if the program can warn the user of a problem with the input, and then give the user another chance. This is called error checking.

Let’s say we want the user to enter a positive number. If the user enters zero or a negative value, we need to loop until the user supplies us with a positive number.

The key to error checking is to use a Boolean variable. I like to have a variable called `needInput`. This variable is initially set to `True`, because we definitely need input from the user. If the user supplies us with good input, then the value of `needInput` can be set to False, and this will be our signal that we don't need to ask again for the input. But if the input is invalid, we need to print some kind of error message and have the user try again.

Here is the code to do error checking. You will find it very useful in this class and in the future!

```
needInput = True
while needInput:
    value = int(input("Enter positive number:  "))
    if value > 0:
        needInput = False
    else:
        print("Sorry, try again.")
```

Exercise:

1. How would you perform error checking if we wanted the user to enter an odd number? In other words, rewrite the above code to accomplish this, making the appropriate changes, even though the overall structure will be the same.

2. How would you perform error checking if we wanted the user to enter just a single digit from 0 to 9?

3. Let's write a program that asks the user to enter numbers one at a time. The program will find the sum of all the numbers entered. When the user enters –1, this is a signal that the program is finished, and should print the total (not including the –1). We use the word *sentinel* to refer to an input value that signals a program to stop reading input.

The `for` loop

Python provides a second way to write a loop, using the keyword `for` instead of `while`. Here is the format of a `for` loop:

```
for <variable> in <list>:
    # steps to perform on each iteration
```

The most common way to write a `for` loop is to count, like this example:

```
for i in range(5, 15):
    print(i)
```

Python has a very useful built-in function called range that creates a list of numbers that we can incorporate into our `for` loop. The range function takes 2 parameters. The general meaning of `range(a, b)` is to return us a list of numbers starting at a and going as high as b – 1, inclusive.

For example, `range(3, 7)` returns this list: 3 4 5 6

So, if we want to print the numbers from 1 to 10 inclusive, we need to call the range function and make sure that it will return us a list containing exactly these numbers. If we said `range(1, 10)`, this will give us the list 1 2 3 4 5 6 7 8 9. The number 10 would be omitted. We must instead say `range(1, 11)`.

General observation: Very often in Python you will notice that whenever we use some interval of numbers, such as from 1 to 10 in some built-in function like range, you should typically expect the range to begin with the first number, and then end with the last number minus 1.

So, to print 1-10 using a `for` loop, we would say:

```
for i in range(1, 11):
    print(i)
```

One convenient feature of the `for` loop is that the increment statement for our counter variable is omitted. If we had used a while loop instead of a `for` loop, we would have to explicitly include a statement "i = i + 1" at the end of the body of the loop.

It is generally a good practice to use the variable i to be the counter of a `for` loop. The letter i stands for "index".

While we are on the subject of the range function, there is a variant of the range function that accepts three parameters, like this: range(0, 100, 5). The third parameter is the stride. It would allow us to count by increments other than 1. The stride can even be negative to allow us to count backwards. However, we won't need to do this very often.

Printing a row instead of a column

Sometimes we prefer to print several values horizontally rather than vertically. To do this, we need to slightly modify the way we use the print() function. We need to add a second argument when we call this function. The second argument will say `end = " "`. For example, we would change

```
print(value)
```

into

```
print(value, end = " ")
```

The second argument tells the print() function to print a single space after it prints the value. When we omit the second argument, print() by default will print a newline character after what it was told to print. In other words,

```
print(value)
```

is the same thing as saying

```
print(value, end = "\n")
```

In fact, you could tell the print() function to use some other character as its delimiter, such as a comma. Let's observe how print() can help us print a row of numbers.

Loop example	Output

`for i in range(1, 10):`
`    print(i, end = "")`
	`123456789`

`for i in range(1, 10):`
`    print(i, end = ",")`
	`1,2,3,4,5,6,7,8,9,`

`for i in range(1, 10):`
`    print(i, end = " ")`
	`1 2 3 4 5 6 7 8 9`

`for i in range(1, 10):`
`    print(i, end = "   ")`
	`1   2   3   4   5   6   7   8   9`

`for i in range(1, 3):`
`    print(i, end = " Mississippi ")`
	`1 Mississippi 2 Mississippi`

After printing your horizontal list, you should say `print()` again to start future printing on a new line.

Questions:

1. The `for` loop is generally preferred over the `while` loop when we know how many loop iterations we want. Looking back over the earlier exercises with `while` loops, find one where we number of loop iterations was known in advance (e.g. a constant number). Show how the `while` loop can be rewritten using a `for` loop.

2. What does this code accomplish?
```
sum = 0
for i in range(2, 7):
    print(i)
    sum = sum + i
print(sum)
```

3. What does this code accomplish?
```
sum = 0
for i in range(1, 6):
    sum = sum + i
    print(sum)
```

4. What does this code accomplish?
```
for i in range(3, 8):
    print(i ** 2)
```

5. What does this code accomplish?
```
count = 0
for i in range(1, 100):
    if i % 7 == 0:
        count += 1
print(count)
```

6. Write code containing a loop that will print the multiples of 4 from 4 to 48 inclusive.

7. Write code containing a loop that will count how many positive integers are evenly divisible into the number 50.

8. Write code containing a loop that will print all 3-digit numbers that end in 7. When you are done, you might notice that there is a lot of output. So, modify your program so that it prints only 12 numbers per line.

9. Ask the user to enter a digit. Then, print all 2-digit numbers that end with this digit.

10. The range function returns a list, and does not have to be associated with a for-loop. For instance, assuming that n is an integer, the if-statement
```
if n in range(5, 15):
```

    is equivalent to this if-statement
```
if _______ <= n <= _______
```

    Fill in the blanks.

11. Let's write a loop that illustrates compound interest. Suppose you have $5000, and you invest in an account that pays 5% per year. Show how much money you have at the end of each of the next 10 years. Format the figures to 2 decimal places. Also make sure that your formatting allows all of the decimal points to be vertically aligned.

12. What is the output of this loop?
```
for i in range(25, 5, -5):
    print(i)
```

13. Write two loops that produce the following output:
```
   1    2    3    4    5    6    7    8    9   10
   1    4    9   16   25   36   49   64   81  100
```
Each loop should print a row of numbers. The second row shows the squares of the numbers in the first row. You will need to use formatted output get the rows to line up. Also, don't forget to print a newline character after the first loop.

Nested loop

A nested loop is simply a loop within a loop. These are actually quite common in real computer programs because they allow the computer to do a lot of work with a small number of program statements.

A nested loop is useful whenever we need to process multi-dimensional information. In this course, we will not encounter this very often. But here are some general examples:

- An image consists of rows and columns of pixels.
- A video is essentially a list of images, so this adds a third dimension of time to our images.
- Games often have a 2-D board.
- Airline and hotel reservations. For example, a seat on an airplane is given by a row number and which seat in that row.

In real life, our brains process nested loops all the time! Here are typical examples:

- For each week… for each day of the week… for each hour of the day
- For each building… for each room in the building

I need to caution you that we should not confuse a nested loop with consecutive loops. Just because you see two loops does not automatically mean that one loop must be inside the other. Consider the following examples. In the first case, the two loops are simply consecutive. In the second case, the two loops indeed nest.

```
# Example #1
count = 0
for i in range(0, 10):
    count += 1
for j in range(0, 20):
    count += 1
print(count)
# -------------------------
# Example #2
count = 0
for i in range(0, 10):
    for j in range(0, 20):
        count += 1
print(count)
```

What is the output of each loop situation here?

As another example, let's suppose we wanted to print a field of star/asterisk (*) characters having 10 rows and 20 stars per row. Here is how it would be done:

```
for i in range(0, 10):
    for j in range(0, 20):
        print("*", end = "")
    print()
```

Notice that the nested loop is set up exactly the same way as in Example #2 above. This is because we want 10 outer iterations and 20 inner iterations. We want to print 10*20 = 200 stars. Since we wish to print the stars right next to each other within the line, we add a second argument to print: `end = ""`. This overrides the default behavior of print, which would have printed a newline character after the star. In this case, “” means print nothing at all between the stars.

Also notice the second print statement in the code. It is positioned immediately after the inner loop, but it is located within the outer loop. This makes sure that we print the newline after each row of stars. It is critically important that we indent the code correctly in this example.

Discussion: For questions 1-3, make the following modifications to the above source code that printed the field of stars. Describe what effect each change has on the output.

1. Change `end=""` to `end = "|"`. (Change it back before continuing to the remaining questions.)

2. Indent the final `print()` statement further to the right so that it is directly underneath the first print statement.

3. Unindent the final `print()` statement so that it vertically lines up with the word `for`.

4. Modify the program so that it prints out a triangular arrangements of stars. The first line should print one star. The second line should print two stars. And so on until the 10th line shows 10

stars. This can be accomplished using a nested `for` loop similar to the above example. Do not include an `if` statement.

5. What is the output of each of these nested loops?

```
for i in range(1, 6):
    for j in range(1, 6):
        print(j, end = " ")
    print()
# ---------------------------
for i in range(1, 6):
    for j in range(1, 6):
        print(i, end = " ")
    print()
# ---------------------------
for i in range(1, 6):
    for j in range(1, i):
        print(j, end = " ")
    print()
# ---------------------------
for i in range(1, 6):
    for j in range(i, 6):
        print(j, end = " ")
    print()
```

Break and continue

The Python keywords `break` and `continue` are special statements that allow you to interrupt the normal flow of a loop. They are powerful statements, and most loops that we will write will not need to use them. A common mistake for novices is to use `break` or `continue` when it is not necessary.

The purpose of the `break` statement is to immediately abandon the loop. We would do this when we realize that we have completely finished doing the work that the loop needed to do, and that further iterations are not necessary.

The purpose of the `continue` statement is to abandon or skip just the current iteration of the loop. Here is an analogy. Let's say you want to play all of the songs on a CD or record. That is a loop. But let's suppose that you don't like the third song. You want to skip it. When you get to the third song, you would want the mechanism to immediately "continue" to the next song.

You can think of the difference between `break` and `continue` like this -- one morning you become suddenly sick at work, and you ask the boss if you can leave work early and come back tomorrow. That is a "continue" situation. On the other hand, you might decide to quit your job. That would be "break." Break has a stronger effect than continue.

Here is a simple example of using the `continue` statement. We can print the numbers 1-10 but skip the number 3:

```
for i in range(1, 11):
    if i == 3:
        continue
    print(i)
```

If we had used the word `break` instead, the code would only have printed the numbers 1 and 2. We use `break` more often than `continue`. The `break` statement is often used when we want to search for something. Suppose you left your umbrella inside your friend's house, but you can't remember which room of the house you left it. You begin a methodical search of the house, going from room to room. Suppose your friend's house has 10 rooms. Will you have to search every room? Probably not. You would stop searching once you find the umbrella. The pseudocode would look like this:

```
for each room in my friend's house:
    Look for the umbrella in this room
    if I see the umbrella,
        Grab it
        break
```

Without the `break` statement, we would keep searching for the umbrella all over the house even after having found it! So, we see that using the `break` statement can make our code more efficient.

As with any programming construct, the `continue` and `break` statements can be misused or overused (i.e. used when they are not necessary). It turns out that we can always write a loop without the words `continue` and `break`. In fact, many programmers tend to avoid these two kinds of statements. Here is how the umbrella-finding example would be written in pseudocode without making use of the `break` statement:

```
found = False
while found == False and there are still rooms to search in:
    Go into the next room
    if I see the umbrella,
        Grab it
        found = True
```

To summarize, the `continue` statement allows us to exit the current iteration of the loop and go on the next iteration (if any). The `break` statement is stronger, in that it exits the loop that it's contained in, and goes on to the statement that follows the loop. To be sure we understand the difference between `break` and `continue`, let's look at some examples that use each. In the following Python program, the first loop contains a `break` statement. The second loop is the same as the first, except break has been replaced with `continue`. Study the output of each loop.

```
# breakContinue.py - Illustrate the difference between break and continue.

for i in range(1, 6):
    print(i)
    print("  before break")
    if i == 3:
        break
    print("  after break")

# print a blank line to separate the results
print()

for i in range(1, 6):
    print(i)
    print("  before continue")
    if i == 3:
        continue
    print("  after continue")


# Here is the output when we run the program:
'''
1
  before break
  after break
2
  before break
  after break
3
  before break

1
  before continue
  after continue
2
  before continue
  after continue
3
  before continue
4
  before continue
  after continue
5
  before continue
  after continue
'''
```

One syntax note about the above program: I used a *triple-quoted string* in order to show the output of the program. A triple-quoted string can be used to hold a very long comment to avoid having to type # at the beginning of each line, which would get tedious if the comment is very long. This technique is also helpful to "comment out" a large chunk of code.

Here is another example that compares the `break` and `continue` statements. This time we have nested loops. The purpose of this example is to show you that when we use a `break` statement to break out of a loop, we only break out of the (inner) loop that the `break` statement is contained in. In other words, if we see a `break` statement in an inner loop, we only exit the inner loop. We are still inside the outer loop. Breaking out of an inner loop has the effect of immediately starting the next iteration of the outer loop, if any. Study the output of this example to be sure you understand.

```
# breakContinueNested.py - This example uses nested loops.

for i in range(1, 6):
    for j in range(1, 6):
        if j == 3:
            break
        print("{0:d}{1:d} ".format(i, j), end = "")
    print()

# Now let's try continue instead of break
print()

for i in range(1, 6):
    for j in range(1, 6):
        if j == 3:
            continue
        print("{0:d}{1:d} ".format(i, j), end = "")
    print()

# Here is the output:
'''
11 12
21 22
31 32
41 42
51 52

11 12 14 15
21 22 24 25
31 32 34 35
41 42 44 45
51 52 54 55
'''
```

Discussion:

1. In the umbrella example above, why do we need the last statement `found = True` ?

2. Earlier, you saw the example of a loop that uses a `continue` statement to skip the number 3 while printing the numbers 1-10. Rewrite this loop (without changing the output, of course) so that it does not use the `continue` statement.

3. Let's revisit the prime number program we saw earlier. Make the following changes to improve its efficiency.
   a. Once we discover that a number has a divisor other than itself and 1, we should break from the loop because the number cannot be prime.
   b. Other than the number 2, we should only test odd number divisors.
   c. When testing divisors, we should stop when we reach the square root of the number in question.

<u>Algorithms with loops</u>

Example #1: The Euclidean algorithm. Given two numbers, it finds their greatest common divisor.

Algorithm:

1. Let m = the larger number, and let n = the smaller number.
2. Let r = the remainder after dividing m/n.
3. If r = 0, then our answer is n, and we are done. But if r ≠ 0, let m = n, n = r, and go to step 2.

Let's try out this algorithm. Do you understand the steps? Will the procedure work? The algorithm here is rather tersely stated. It said nothing about I/O, and if we ever wanted to write this into a computer program, we would have to fill in those details.

Exercise: Work out the Euclidean algorithm above with the numbers 32 and 84. Does the algorithm give the correct answer?

Example #2: Allow the user to play a simple guessing game.

For example, we could have the user guess a number between 1 and 100. We need to set the maximum number of guesses to be $\log_2 100$, which is about 7. The secret number can be hard-coded in the program, or better yet, we can use a built-in function to select a random number. However, it may be easier to test our algorithm if we hard code the correct answer.

Algorithm:

```
1. Let the secret number be 42.
2. Let the maximum number of allowed guesses to be 7.
3. Initialize the number of actual guesses accrued to be 0.
4. While the number of guesses < maximum guesses, loop:
     a. Ask the user for a guess
     b. Increment the number of guesses made
     c. If the guess matches the secret number:
        Win!  Congratulate the user.
        Break from the loop, as the game is over.
     d. Otherwise, at this point, we know the guess is wrong.
        If the guess > secret number,
           Tell the user the guess was too high.
        Otherwise,
           Tell the user the guess was too low.
5. If we get this far, that means the user has run out of guesses and has lost the game.
```

Discussion:

How does the above algorithm determine that the player has won or lost the game?

<u>Practice with loops</u>

1. The Fibonacci sequence is this list of numbers: 1, 1, 2, 3, 5, 8, 13, 21, etc. The first two numbers are 1 and 1, and each subsequent number in the list is the sum of the previous two. Print the first 25 Fibonacci numbers.

2. Design a program that asks the user to repeatedly enter integers, and then stops when the user has entered the same number twice in a row.

3. Output the first twenty powers of 2. (2, 4, 8, 16, … , 2^{20}) Also find the sum of these numbers.

4. Ask the user to enter a number. Print all the divisors of this number. For example, if the user chooses 10, then print out the numbers 1, 2, 5 and 10. Also tell the user how many divisors.

5. Print out all the prime numbers between 1 and 1000.

6. Given a positive integer, how would we use a loop to
   a. figure out how many digits it has?
   b. find the sum of the digits?
   c. calculate its binary representation?