Python Programming

INFO.2820.061

Instructor: Andy Garside

Loops, Selection Statements, and Text Formatting

Introduction

All of the code we have looked at so far has executed in a linear manner, where Python statements are executed in order and each statement is executed only once. However, to solve many problems, a program needs to be able to repeatedly or conditionally perform one or more statements. In this lecture we will discuss Python **looping** and **selection** statements (collectively called **control statements**) which give us this capability.

As we look at Python looping statements it will be useful to have knowledge of a few new things including **augmented assignments, random numbers** and generating a **range** of numbers**.**

Augmented Assignments, Random numbers and range

Augmented Assignments

Assignment statements such as x = x + 1 occur very frequently in programming. Because of this, many languages, including Python, include a shortened form of the assignment statement that incorporates an operator along with the equals sign (=). These shortened assignment forms are called **augmented assignment operations** and are used frequently in code, including loops.

The table below shows a number of Python augmented assignment operations and their meanings.

Augmented Assignment	Meaning	Examples
`<variable> += <value>`	`<variable> = <variable> + <value>`	`x += 1, s += "boat"`
`<variable> -= <value>`	`<variable> = <variable> - <value>`	`x -= 1`
`<variable> *= <value>`	`<variable> = <variable> * <value>`	`x *= 1`
`<variable> /= <value>`	`<variable> = <variable> / <value>`	`x /= 1`
`<variable> %= <value>`	`<variable> = <variable> % <value>`	`x %= 1`

Random numbers

Most programs are designed to be predictable. Given a certain set of input values the output is expected to be identical each time. However, in some applications like computer games, it's useful to be able to generate random outcomes. To do this in Python, we can make use of the **random** module.

The **random** module contains a collection of resources for creating random numbers. One of the most commonly used functions from the **random** module is the **randint** function. The **randint** function takes two arguments, *start* and stop. When called, **randint** generates and returns a random integer between the *start* and *stop* values, inclusive.

Below are four invocations of the **randint** function. Notice that the **random** module must fist been imported. Each of the `random.randint(1,6)` calls below generates a random number between 1 and 6, inclusive.

```
>>> import random
>>> random.randint(1,6)
4
>>> random.randint(1,6)
2
>>> random.randint(1,6)
5
>>> random.randint(1,6)
2
```

Notice that the number 2 is generated twice. This may appear non-random but it's actually quite common to see a repeated number when selecting numbers randomly from a small set.

Note: true random numbers are not available in computing languages. Instead, computers generate **pseudo-random** numbers using mathematical algorithms.

The `range` built-in

Like the **round** function we learned about in our last lecture, **range** is built into Python and is always available (That is, you don't have to **import** a module to use it). The purpose of **range** is to generate and return a range of numbers. How it works varies depending on what form you use.

Simple ranges starting at 0

In its simplest form, **range** takes one argument, *stop*, and generates a sequence of integer values from 0 to *stop* - *1*. For example, **range(4)** generates the number 0, 1, 2, 3.

To view the results of range we will make use of the Python built-in, **list**. **list** takes the results of a sequence, like the output or **range**, and puts each value into a separate element in a **list**. The resulting **list** can then be displayed. For example,

```
>>> list(range(4))
[0, 1, 2, 3]
```

Starting a range somewhere other than 0

At times, it is useful to generate a range of numbers that begins somewhere other than 0. To do this, there is a second form of the **range** built-in that takes the *start* position as the first argument and the *stop* position as the second. This form of **range** generates numbers from *start* to *stop* – *1*. For example,

```
>>> list(range(1,4))
[1, 2, 3]
```

Notice in the example above that 4 is not in the list of results. A common Python programming error is to use the **range** function and expect the returned values to include the *stop* value, when it ends at *stop* – *1*. This is known as an **off-by-one** error, and it's something to be looked at when you're proof reading your code.

Specifying the Steps in a Range

To this point, every sequence of numbers we've seen are such that each number is one more than its predecessor. There are times when it's useful to produce a sequence where this difference, also called the **step value**, is something other than 1.

There is another form of **range** that adds a third argument called *step*. This form generates numbers from the *start* to *stop -1* where each number is generated by adding *step* to the previous number. For example, we can generate the list of odd integers 1, 3, 5 by starting the range at 1 and using a step of 2, as follows.

```
>>> list(range(1,6,2))
[1, 3, 5]
```

You need to be careful when examining a **range** statement. In the **range** statement below the *stop* value is set to 7 and the *step* is set to 3. If you look quickly at this statement it's easy to think that the generated list of numbers will be 1, 4, 7. However, since **range** stops generating numbers at *stop -1*, this actually generates the sequence 1, 4. This may seem obvious but it's easy to overlook.

```
>>> list(range(1,7,3))
[1, 4]
```

"for" Loops

In programming, a **loop** is a control statement that executes a sequence of statements repeatedly (in a "loop"). Each repetition of the loop is known as a **pass** or **iteration** and the loop stops when one or more conditions are met. One of the most common forms of loop is the **for** loop.

The definite iteration, count-controlled "for" loop

Many loops are designed to run a specific number of iterations. Such loops are called **definite iteration** loops. For example, a loop might display a sequence of 10 numbers, or display a sequence of strings. Loops the repeat based on a count value are called **count-controlled** loops and are a specific instance of a **definite iteration** loop. Below is an example of a Python count-controlled **for**

loop that uses the **range** function to generate a sequence of five odd numbers from 1 to 9

```
>>> for num in range(1,10,2):
        print(num)
1
3
5
7
9
```

This form of **for** loop begins with the keyword **for** followed by a **variable name**, *num*, which will be repeatedly assigned the values generated by the **range** function. The initial line is called the **loop header** and it must ***end with a colon.***

Following the **loop header** is the loop body. The **loop body** is a sequence of one or more statements that are executed each pass through the loop. In the example above, the **loop body** consists of the single statement **print(***num***)** which displays the new value of *num* each time through the loop.

Notice that the statement in the **loop body** is **indented**. Unlike many programming language, **indentation** is significant in Python. In the case of a loop, the statements in the **loop body** must ***be indented and aligned in the same location.***

Improper indentation is the cause of many Python bugs!

*Statement **indentation** is significant in Python. In the case of a **for** loop, Python uses **indentation** to determine which statements are inside the **loop body** and which are not.*

*Consider the following two program snippets (the second snippet is on the next page). The two snippets are identical except for the last print statement, **print(num + 1)**. Notice how the output differs for each loop.*

*In this first snippet the print statement **print(num + 1)** is **indented**, so Python considers it part of the **loop body** and it executes the statement each time through the loop. The results of executing this code is the sequence 1, 2, 3, 4.*

```
for num in range(1,5,2):
    print(num)
    print(num + 1)
1
2
3
4
```

*In this second snippet, the last print statement is **not indented,** so Python sees it as **outside** the **loop body**. As a result, it executes only the **print(num)** in the loop. When the loop is done, it then executes the **print(num +1)**. The results of executing this code is the sequence 1,3,4.*

```
for num in range(1,5,2):
    print(num)
print(num + 1)
1
3
4
```

*To show just how significant **indentation** is in Python, consider the following. This snippet is identical to the first snippet only the **print(num + 1)** statement is indented by **one more space** than the **print(num)** statement above it.*

```
for num in range(1,5,2):
    print(num)
     print(num + 1)
```

*When you attempt to run this snippet of code, Python will produce an "**Unexpected indent error**"!*

Indentation and the IDLE Shell. *When you write a loop in the IDLE Shell, IDLE will automatically indent the body statements for you. When you reach the last statement you want to put inside the loop body, you can hit the **backspace** key to tell IDLE to unindent the statement so it is outside the loop body.*

*Note: personally, I don't find it convenient to create loops inside the interactive shell. When I have anything more than a few lines of code I put it in a script so it can be saved, edited and reused. However, the interactive shell **is** nice for trying out small snippets of code statements to verify they do what you expect.*

Loops That Count Down

The **step value** passed to **range** can be **negative**. This is useful for generating sequences of numbers that count down rather than up.

For example, the code below generates a sequence of numbers from 10 down to 1 (when the **step value** is **negative**, **range** stops generating values at the value *stop* **+** *1*, rather than *stop* **-** *1*). The **print** statement prints the value of count and then a space. All of the numbers appear on one line because of the argument *end* **=** *“”*.

```
for count in range(10, 0, -1):    # Loop for count = 10…1
    print(count, end = " ")       # print count and a space
```

```
10 9 8 7 6 5 4 3 2 1
```

Augmented Assignment & range in a count-controlled “for” loop

Augmented assignment and **range** are often used in **for** loops. Below is a **for** loop that calculates the product of the number 1 through 4. That is, 1 * 2 * 3 * 4.

```
product = 1               # initialize the product to 1
for num in range(1,5):    # loop for num = 1, 2, 3, 4.
    product *= num        # product = product * num
print(product)            # print the result (outside the for loop)
```

When this code is run it displays the number 24.

Random numbers in a count-controlled “for” loop

Below, we combine a number of the things we've learned and generate a sequence of random integers between 1 and 6 using a count-controlled **for** loop. The **for** loop uses the **range** builit-in to cause the loop to execute 10 times. Each iteration of the loop executes a **print** statement that displays the value of two expressions. The first expression, `random.randint(1,6)` , generates a random integer between 1 and 6, inclusive. The second statement, `end = " "`, overrides the default **print** behavior to tell it to display a string consisting of a single space at the end of the line rather than a newline (\n).

Notice that the loop variable, *num*, is not used within the loop. There are instances where a programmer wants to make something occur a number of times but they don't need the value generated by the loop header. In this case, the loop variable *num* is not needed because the function `random.randint(1,6)` doesn't need it. It only needs the fixed *start* and *stop* values 1 and 6. It's critical, however, that a variable name appear in the **for** loop header in order for the **for** loop syntax to be correct. Without it, we would see a syntax error.

```
import random            # import the random module to access randint
for num in range(10):    # loop 10 time. We don't use the variable num
    print(random.randint(1,6), end = " ")   # display random number
```

```
4 4 1 2 6 5 3 3 6 2
```

Formatting output

> Note: For readers who are familiar with the **printf** function in the **C** programming language, the following will be somewhat familiar.

Until this point, all the code we have written that displays output has done so using a simple form of the **print** statement where each expression is a simple string or arithmetic expression. For example,

```
print("Three + five is:", 3 + 5)
```

```
Three + five is: 8
```

There are frequently times when it is useful to have more control of the formatting of the output. For example, you may want to align the output into neat columns, or display floating-point numbers with a specific precision.

The following **for** loop uses the **range** built-in to loop through the integers 6 to 12 and display each number along with the value 2 raised to the power of that number. The **print** statement before the loop displays a simple header indicating what is in each column. Notice that the numerical output doesn't line up well with the header string, "num 2^num". This can make the output difficult to interpret.

```
print("num  2^num")
for num in range(6,12):
    print(num, 2 ** num)
```

```
num  2^num
6 64
7 128
8 256
9 512
10 1024
11 2048
```

We can improve this output by making use of the special formatting capabilities of the **print** function. To do this, we will use statements of the following form where <*format string*> contains a set of format characters that specify **minimum** field widths for each <*expression*> that follow the %.

print("<*format string*>" % (<*expression*>,..))

Note: Strictly speaking, the parenthesis, (), surrounding the comma-separated <*expression*> sequence are optional if there is only one expression. They are required when there are two or more <*expression*> values.

The <*format string*> can be any valid string but it must contain one format specifier like those in the table below for each expression in <*expression*>.

Below, <*field width*> is an integer value indicating the **minimum** width of the field being displayed. A positive integer indicates the output should be **right-justified** and a negative integer indicates **left-justification**.

Field specifier	Examples	Meaning
%<field width>a	%10a	Display an **integer**, **right-justified**, with a **minimum** field width of **10** characters
	%-12a	Display an **integer**, **left-justified**, with a **minimum** field width of **12** characters
%<field width>s	%20s	Display a **string**, **right-justified**, with a **minimum** field width of **20** characters
	%-30s	Display a **string**, **left-justified**, with a **minimum** field width of **30** characters
%<field width>.<precision>f	%6.3f	Display a **floating-point** number, **right-justified**, with a **minimum** field width of **6** characters (including the decimal point and fractional digits) and a **precision** of **3** digits after the decimal point (trailing zeros are added if necessary)
	%-10.2f	Display a **floating-point** number, **left-justified**, with a **minimum** field width of **10** characters (including the decimal point and fractional digits)and a **precision** of **2** digits after the decimal point (trailing zeros are added if necessary)

Using the information in the table above, we can modify the **print** statement from our previous example to better format the output as shown below

```
print("num  2^num")
for num in range(6,12):
    print("%2d%7d" % (num, 2 ** num))
```

```
num  2^num
 6     64
 7    128
 8    256
 9    512
10   1024
11   2048
```

In the loop's **print** statement, the **%2d** of the format string indicates the first expression, *num*, should be displayed as an **integer** with a **minimum** field width of 2 characters and **right-justified** in the field.

The **%7d** indicates the second expression, `2 ** num`, should be displayed as an **integer** with a **minimum** field width of 7 characters and **right-justified** in the field. Notice that the largest value of `2 ** num` is only 4 characters wide. We used the value 7 in the format string to force 3 additional spaces between the values in the two columns. We could have achieved a similar thing by inserting 3 spaces between the field specifiers and changing the 7 to a 4, as follows

```
print("%2d   %4d" % (num, 2 ** num))
```

The output is much more readable now that we've set the field width for each value. However, most people prefer to read numbers in a **left-justified** format. We can make one last change to our loop and add dash (**-**) before each field width making the integer format **negative** and causing the output to be **left-justified** as follows

```
print("%-2d   %-4d" % (num, 2 ** num))
```

```
num  2^num
6    64
7    128
8    256
9    512
10   1024
11   2048
```

As one more example, consider the following loop which calculates 2.5^num for the numbers 6 to 12. Because the calculated value is a floating-point number,

we have to change the field specifier to the character **f**. If we want to show the floating-point result to 2 digits after the decimal point and we want to **left-justify** the result, we can use **%-1.2** as the field format.

> Note: Technically, we could use %-.2 instead of %-1.2, making the minimum field width 0 characters rather than 1, since we're **left-justifying** the floating-point number and there are no characters following the floating-point number. We used %-1.2 in this case just for visual consistency.

Notice that all characters in the format string other than the **field width** formatters are displayed just as they would be with any other string, including the literal characters "2.5**" and " = ".

```
for num in range(6,12):
    print("2.5**%-2d = %-1.2f" % (num, 2.5 ** num))
```

```
2.5**6  = 244.14
2.5**7  = 610.35
2.5**8  = 1525.88
2.5**9  = 3814.70
2.5**10 = 9536.74
2.5**11 = 23841.86
```

Selection Statements: `if`, `if-else`, and `if-elif-else`

Sometimes it is desirable to **conditionally** execute one or more statements. That is, instead of always executing a set of statements, the statements are executed only if one or more conditions are met. For example, if one number is larger than another, or only if the number is not zero, etcetera. In the following section we'll discuss several types of **selection statements** (also called **control statements**) and show examples of their uses.

Logical Operators and Compound Boolean Expressions

With **selection statements**, statements are executed when one or more **conditions** are met. Conditions can take many forms, but they all have one thing in common, they evaluate to either **True** or **False**.

We have, so far, discussed three data types: **strings**, **integers** and **floating-point** numbers. There is a fourth commonly used data type called **Boolean** (**bool**). **Boolean** values can evaluate to only two values, **True** or **False** (Note that **True** and **False** are keywords in Python and they begin with **upper-case** letters.)

Below, the variable *answer* is created and assigned the value **True**. When we examine the **data type** of *answer* using the **type** function we see it is of type **bool** (Boolean).

```
>>> answer = True
>>> type(answer)
<class 'bool'>
```

Boolean values appear many places in Python. As we've seen, you can assign a **variable** a Boolean value, and it is common for a **function** to return a Boolean value. Python also supports a set of **comparison operators** that evaluate to a Boolean value, operators that can be used to create **conditions**.

The table below lists a number of Python **comparison operators** and their meanings.

Comparision Operator	Meaning	Examples
==	Equals (comparison)	num1 == num2
!=	Not equals	x != y
<	Less than	speed_sound < speed_light
>	Greater than	omega > alpha
<=	Less than or equal	savings <= price
>=	Greater than or equal	time >= 60

The IDLE Shell output below shows two assignments followed by some examples of comparison operators and their uses.

```
>>> x = 4     # assignment
>>> y = 5     # assignment
>>> x == 7    # comparison
False
```

```
>>> x < y
True
>>> y >= x
True
>>> 4 > 5
False
```

The **precedence** of comparison operators is less than arithmetic operators but greater than an assignment. That is, **arithmetic operators** are evaluated first, then **comparison operators** are evaluated, and then any **assignment** happens last.

In the expression below, 3 + 5 is evaluated to 8 before it is checked if it is greater than 9. Since 8 > 9 is **False**, **False** is assigned to the variable *question*.

```
>>> question = (3 + 5 > 9)
>>> question
False
```

Comparison Operators and strings

The comparison operators we've discussed can also be used with characters and **strings** (remember, in Python, a character is just a **string** with only one character). When using **comparison operators** with strings, Python evaluates each character in the string one by one, from **left** to **right**. As soon as two different characters are found, Python can determine which is greater.

With integers and floating-point numbers, the meaning of comparisons are fairly obvious. However, with strings, is "A" > "Z" or "fish" < "fox"? What about "02" > "1"?

Python, like other programming languages, translates characters into binary character sets. The internationally recognized character set for languages is called **Unicode** (https://home.unicode.org/) and it contains over 128,000 characters. Within the Unicode character set is a subset of Latin characters belonging to a smaller character set called **ASCII** (American Standard Code for Information Interchange). For most uses, what's important to know is that the characters '0' through '9' evaluate as less than the upper-case characters 'A' to 'Z', and the upper-case characters 'A' to 'Z', evaluate as less than the lower-case characters 'a' to 'z'. For example, on the line below, all of the specified characters are shown from left to right in order of increasing value.

```
0..9    A..Z    a..z
```

Note: there are additional characters that fall in-between 0..9, A..Z and a..z. For more information, there is a complete table of the Unicode Latin character set here https://en.wikipedia.org/wiki/List_of_Unicode_characters#Basic_Latin which includes the decimal value for each character.

Knowing this character order, we can evaluate various comparison operators using strings.

```
>>> "a" > "A"    # Lower case chars are > than upper-case chars
True
>>> "z" > "A"    # Lower case chars are ALWAYS > than upper-case chars
True
```

Remember that strings are evaluated **left** to **right**, **character** by **character** until a difference is found. In the strings below, the letter 'F' is less than the letter 'f' so "Fox" > "fox" is **False**.

```
>>> "Fox" > "fox"
False
```

Notice below that the string "2" is greater than the string "128". This is because the first character of the string "128" is '1' which is less than the first character of the string "2". This is counter intuitive since, numerically, 2 < 128.

```
>>> "2" > "128"
True
```

If, If-else, and If-elif-else Statements

Now that we have a knowledge of **comparison operators**, we can use them in a variety of **selection statements**. One of the most common and most useful is the simple **`if`** selection statement.

The **`if`** selection statement tests if one or more conditional expressions are **True** and, if so, the source code contained within the body of the **`if`** is executed. Otherwise, the body is skipped and code execution continues at the first statement following the body. The simplest form of the **`if`** statement is

```
If <condition>:
    <sequence of statements>    # the "body"
```

For example

```
if 5 > 2:
    print("Five is greater than two!")
```

```
Five is greater than two!
```

As with the "**for**" loops, **indentation** is significant with the "**if**" statement. For a statement to be considered inside the body of the **if**, it must be indented to the same level as the other body statements. If it's indented differently, you will receive an **indentation error** when you run the program. If you forget to indent it all together, it will be outside the body, meaning it will always be executed regardless of whether the **if** condition is **True** or **False**.

Having done it many times myself, it's easy to forget to add the colon, **:**, to the end of the `if` header. If you do, you'll get an **invalid syntax** error when you run the program.

Sometimes it is desirable to perform one set of statements if a condition is **True** and another set if the condition is **False**. We can accomplish this with the **if-else** statement (also known as the **two-way selection statement**). With an **if-else** statement, the body under the **if** condition is executed if the condition is **True**. Otherwise, the body under the **else** is executed. The general syntax of the **if-else** statement is as follows. (Notice that there is a colon, **:**, following the keyword `else`.)

```
If <condition>:
    <sequence of statements>    # executed if <condition> is True
else:
    <sequence of statements>    # executed if <condition> is False
```

In the example below, the user is prompted to enter two integers. The first integer is assigned to the variable `first` and the second integer is assigned to the variable `second`. The if-else statement tests whether the condition `first > second` is **True** and, if so, it prints the text `"The first integer is greater!"`. Otherwise, the condition is False and the text "The second integer is greater!" is displayed.

```
first = int(input("Enter an integer: "))
second = int(input("Enter another integer: "))
if first > second:
    print("The first integer is greater!")
else:
    print("The second integer is greater!")
```

```
Enter an integer: 6
Enter another integer: 7
The second integer is greater!
```

There are times when it is desirable to perform different actions (statements) depending on the value of a condition. For example, consider a grade calculator program that maps a set of numeric student grades to equivalent letter grades. For example, students with grades at or above 90 get a letter grade of 'A'. Students at or above 80 but below 90 receive a 'B'. Students at or above 70 but below 80 receive a 'C'. And all other students receive an 'F'.

> Note: Grades in this class are not calculated this way. For a numeric mapping of grades for this course, see your syllabus. ☺

To accomplish the number grade to letter grade conversion we can use a Python **multi-way if** statement. The **multi-way if** statement adds the keyword **elif** (short for else-if) to an **if** in order to evaluate additional conditions that result in different sets of executed statements. For example, the code below implements our number grade to letter grade converter.

```
grade = float(input("Enter a numeric grade: "))
if grade >= 90:
    letter = 'A'
elif grade >= 80:
    letter = 'B'
elif grade >= 70:
    letter = 'C'
else:
    letter = 'F'
print("The letter grade is", letter)
```

```
Enter a numeric grade: 74.6
The letter grade is C
```

The code first prompts the user for a numeric letter grade, converts it to a float, and then assigns it to the variable **grade**.

The **if** condition then checks if the condition **grade >= 90** is **True**. If so, it assigns the string **'A'** to the variable **letter** and processing continues at the **print** statement at the end of the **if-elif-else** statement, resulting in the text **`The letter grade is A`** being printed.

If the condition is **False**, the condition in the first **elif** is evaluated. If **grade >= 80** is **True**, the string **'B'** is assigned to the variable **letter** and processing continues at the **print** statement at the end of the **if-elif-else** statement and the text **`The letter grade is B`** is printed.

A similar process occurs for the letter grade 'C'.

Finally, if none of the **elif** conditions are **True**, the statement following the **else** condition is executed and **letter** is assigned the value 'F'. As with the other cases, processing then continues at the **print** statement after the **if-elif-else** statement and the text **`The letter grade is F`** is printed.

Logical Operators and Compound Boolean Expressions

The grade calculator program has a noticeable flaw. If the user enters an invalid grade, like -10.7, the program will still produce a letter grade and the user will not be notified of their mistake.

Ideally, the program should check the user's input and print an error message if the grade entered is less than 0 or greater than 100. In this case, we want to check if either of two different conditions are **True**. If either is **True**, we want to print an error message. Otherwise, we want to perform the calculations from our earlier version of the grade calculator.

To perform two or more Boolean comparisons, also known as **compound Boolean expressions**, Python includes a set of logical operators including **or**, **and** and **not**. These logical operators are keywords in Python and have the following meanings.

Logical Operator	Examples	Meaning
or	A or B	If either A is **True or** B is **True**, return **True**.
and	A and B	If A is **True and** B is **True**, return **True**.
not	not A	If A is **False**, return **True.** If A is **True,** return **False.**

Using the logical operator **or**, we can update our grade calculator to include an **if** statement with a compound expression. In "*new line 1*", the first expression evaluates to **True** if the **grade < 0** and the second expression evaluates to **True** if the **grade > 100**. If either expression is **True**, the body of the **if** is executed (*new line 2*) and the error message **"Error: The grade must be between 0 and 100"** is printed. If neither expression is **True**, the **else** path is taken (*new line 3*) and the original **if-elif-else** that was used to previously calculate the grade is executed.

```
grade = float(input("Enter a numeric grade: "))
if grade < 0 or grade > 100:                                    # new line 1
    print("Error: The grade must be between 0 and 100")         # new line 2
else:                                                           # new line 3
    if grade >= 90:
        letter = 'A'
    elif grade >= 80:
        letter = 'B'
    elif grade >= 70:
        letter = 'C'
    else:
        letter = 'F'
    print("The letter grade is", letter)
```

```
Enter a numeric grade: -10.7
Error: The grade must be between 0 and 100
```

Notice the **indentation**. *new line 2* and all the statements from the original grade calculation are indented to the same level. This is because all of these statements are part of the bodies of the original **if-else** statement. In the same way, all of the assignments to the variable **letter** are at the same **indentation** level since they are all bodies of the **if-elif-else** statement.

Importantly, the statement `print("The letter grade is", letter)` is indented to the same level as the **if-elif-else**. If this statement were not indented, it would always be executed, regardless of how the compound expression *new line 1* was resolved. In this case, it would result in an error, because the variable **letter** would have never been defined.

Parenthesis and Precedence.

It's important to consider the precedence of operators when creating expressions. For example, is the expression *5 + 2 ** 2 > 10* **True** or **False**? To add clarity, Python allows the use of parenthesis within and around any expression to force operations to happen in any desired order, or to simply make it clear to the reader what the order of operations is. For example, *5 + 2 ** 2 > 10* is **False**, but *(5 + 2) ** 2 > 10* is **True**.

Below is a table summarizing the operators we have learned and their precedence relative to one another.

Operator Type	Operator Symbol
Exponentiation	**
Arithmetic negation	-
Multiplication, division, remainder	*,/,%
Addition, subtraction	+,-
Comparison	==,!=,<,>,<=,>=
Logical negation	**not**
Logical conjunction	**and**
Logical disjunction	**or**
Assignment	=

Short-Circuit Evaluation

In a compound Boolean expression it is not always necessary to evaluate every expression. Take the following expression, for example

x > 5 **or** x < 10

If the expression x > 5 evaluates to **True**, it is unnecessary to evaluate the expression x < 10. This behavior is called **short-circuit evaluation** and, besides reducing the number of statements that are executed (thus, improving performance), it can have practical uses.

Consider the following **if-else**. It takes the total sum of a set of numbers as input (the **total**) along with the number of items (the **count)**. If the count is greater than 0 and the average is greater than 10, it prints the statement, "**`The average is > 10`**" otherwise, it prints the statement, "**`The average is <= 10`**".

Without short-circuit evaluation, if the count entered is 0, the expression **total // count** would be evaluated resulting in a division by zero error. Because Python

employs short-circuit evaluation, the first expression, **count > 0**, evaluates to **False** and the expression **total // count** is never evaluated.

```
total = int(input("Enter the sum: "))
count = int(input("Enter the number of items: "))
if count > 0 and total // count > 10:
    print("The average is > 10")
else:
    print("The average is <= 10")
```

Conditional Iteration: The `While` loop

Earlier we discussed the **for** loop which executes a counted number of times. For example, the count may be specified by the programmer or it may be a value entered by the user. Either way, the **for** loop executes a set number of times.

There are times in programming when you want statements to be executed an indefinite number of times, only to be terminated when the user decides they are done. In this scenario, the user typically enters a special value, a so called **sentinel** value, to indicate they are done using the program. In Python, as in many other languages, we can use a **while** loop for this purpose.

A **while** loop has the following general format.

```
while <condition>:
        <sequence of statements>      # The “body”
```

The <*condition*> of the **while** loop is evaluated and, if **True**, the body of the while loop is executed. When done, processing returns to the top of the **while** loop and the <*condition*> is evaluated again. If the <*condition*> is still **True**, the body is executed again. This process continues until the <*condition*> evaluates to **False**.

Consider the grade calculator we created earlier. It would be nice if the calculator could calculate as many grades as the user wanted and only terminate when the user enters a **sentinel** value, say a negative number. Below is an updated example of the grade calculator. This version prompts the user for an initial grade. The **while** loop then evaluates the expression **grade >= 0**. When the expression is **True**, the body of the **while** loop is executed and a letter grade is calculated and displayed. The variable **grade** is called a **loop control variable** as its value determines whether the loop continues, thus controlling execution of the loop.

The last statement in the body of the **while** loop is a repeat of the prompt asking the user for a new grade. This allows the grade to change on each pass of the **while** loop, allowing the user to continuously calculate new grades. When the user is done, they can enter a grade less than zero. This will cause the expression **grade >= 0** to be **False**, causing the **while** loop body to be skipped and execution to continue at the first line after the **while** loop body. The last statement executed is the statement `print("All done!")` and then the program terminates.

```
grade = float(input("Enter a numeric grade: "))
while grade >= 0:
    if grade >= 90:
        letter = 'A'
    elif grade >= 80:
        letter = 'B'
    elif grade >= 70:
        letter = 'C'
    else:
        letter = 'F'
    print("The letter grade is", letter)
    grade = float(input("Enter a numeric grade: "))
print("All done!")
```

An example of the output follows

```
Enter a numeric grade: 93.2
The letter grade is A
Enter a numeric grade: 72.9
The letter grade is C
Enter a numeric grade: -1
All done!
```

Infinite `while` loops

In our most recent grade calculator, the **while** loop terminates when the user enters a grade less than zero. This gives the user control of how long the program runs. There are times when a program, whether accidentally or on purpose, contains a **while** loop where the condition never evaluates to **False**. Consider the following example

```
while True:
    print("Stop yelling at me!")
```

In the **while** loop above, the condition is simply the value **True**. This is called a **while true** loop. As a result, the body of the while loop will always be executed and the program will never terminate. The output will be a never-ending set of strings as follows

```
Stop yelling at me!
Stop yelling at me!
.
.
```

This is an example of a so called **infinite loop**. Any **while** loop whose condition never evaluates to **False** is an **infinite loop**.

An infinite loop will continue indefinitely until the user does something to interrupt it. Typically, the best way to do this is by pressing **Ctrl-c**, which sends a **keyboard interrupt** to the program. Note that you may have to press **Ctrl-c** a number of times to get the program to catch the interrupt and stop.

Count Controlled `while` loops

Recall the following for loop we discussed earlier. This is an example of a **count-controlled for** loop.

```
for count in range(10, 0, -1):    # Loop for count = 10…1
    print(count, end = " ")       # print count and a space
```

```
10 9 8 7 6 5 4 3 2 1
```

Any **count-controlled for loop** can also be written as a **count-controlled while loop.** For example, the loop above can be written with a **while** loop as shown below.

The **count** is **initialized** to the value 10 before the **while** loop. The **while** loop condition, **count > 0**, is **True** as long as count remains greater than 0.

Within the loop, the print statement displays the value of **count** followed by a **space** (note that a newline is not printed). The value of **count** is then decremented by 1 and execution continues at the top of the **while** loop. Since the count begins at 10 and is decremented by 1 each time through the loop, the loop is guaranteed to terminate in exactly 10 passes.

```
count = 10
while count > 0:
    print(count, end = " ")
    count -= 1
```

```
10 9 8 7 6 5 4 3 2 1
```

The `break` statement

A **while True** loop doesn't always have to result in infinite processing. Many languages, including Python, provide a **break** statement which when executed, breaks out of the body of a loop and continues execution at the first statement following the loop.

The following program calculates the sum (total) of a set of numbers entered by the user. The total is initialized to 0 and then the program enters a **while True** loop. At the top of the loop the user is prompted to enter a number. The **input** function gathers this value and returns it as a **string** in the variable **data**.

The **if** statement compares the input value, **data**, to the empty string, **""**, and when this is True, execution breaks from the loop and continues at the print statement at the end of the program. When the statement is False, **data** is converted to a floating-point value using the **float** function and assigned to the variable **num**. The variable **num** is then added to the **total**.

The last line of the program prints the string "**`The sum is`**" followed by the total.

```
total = 0.0
while True:
    data = input("Enter a number or press Enter to quit: ")
    if data == "":
        break
    num = float(data)
    total += num
print("The sum is", total)
```

```
Enter a number or press Enter to quit: 3.4
Enter a number or press Enter to quit: 4.2
Enter a number or press Enter to quit:
The sum is 7.6
```

The **break** statement provides a convenient way to exit a loop without having to execute all the statements in the loop body.

Conclusion

In this lecture we've looked at **loops** and **selection statements**, including **for** loops, **while** loops, **if**, **if-else** and **if-elif-else** statements. We've also looked **augmented assignments**, **random** numbers, and the **range** built-in. With these tools we can create far more flexible programs than we could with the otherwise straight-line code execution we saw in previous lectures.