2ND CLASS

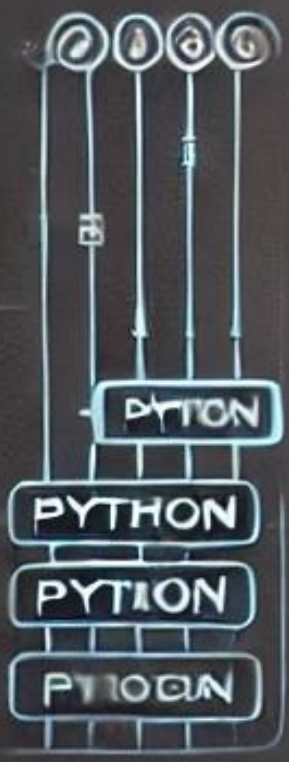
PYTHON LANGUAGE

PYTHON LANGUAGE

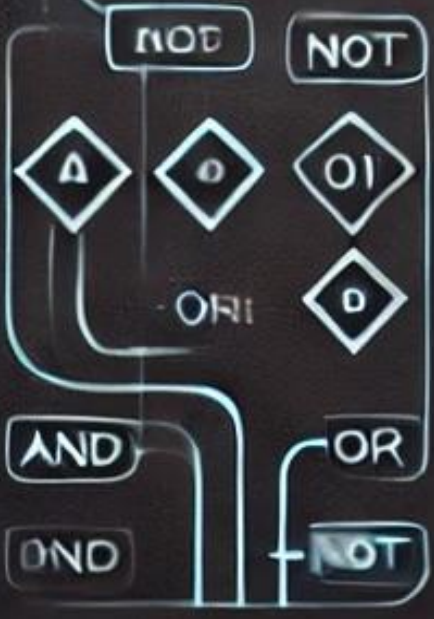
PROGRAM FLAW & BOOLEAN LOGIC

ZIED O. AHMED

2ND CLASS



2ND CLASS



PROGRAM FLAW: BOGEAN & LOGIC

ZIED O. AHMED

LECTURE THREE: PROGRAM FLOW & BOOLEAN LOGIC

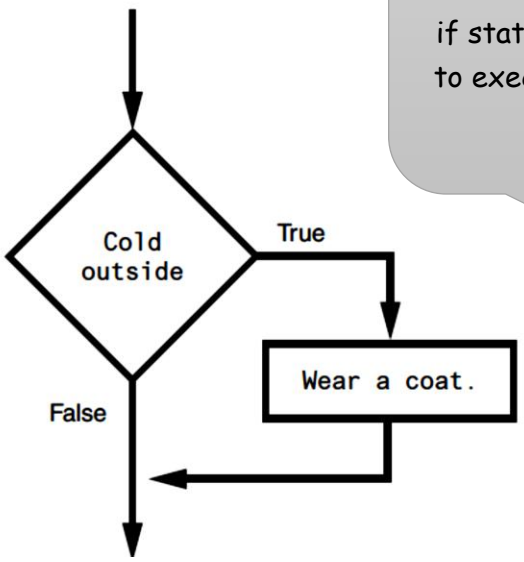
3.1 The if Statement

A *control structure* is a logical design that controls the order in which a set of statements execute.

A *sequence structure* is a set of statements that execute in the order in which they appear. For example, the following code is a sequence structure because the statements execute from top to bottom:

```
name = input('What is your name? ')
age = int(input('What is your age? '))
print('Here is the data you entered:')
print('Name:', name)
print('Age:', age)
```

CONCEPT: The if statement is used to create a decision structure, which allows a program to have more than one path of execution. The if statement causes one or more statements to execute only when a Boolean expression is true.

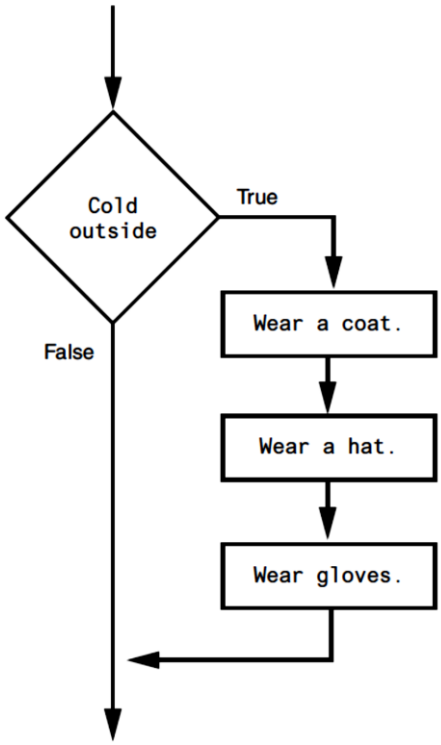


In the flowchart, the diamond symbol indicates some condition that must be tested. In this case, we are determining whether the condition *Cold outside* is **true** or **false**. If this condition is **true**, the action *Wear a coat* is performed. If the condition is **false**, the action is skipped. The action is *conditionally executed* because it is performed only when a certain condition is true.

Programmers call the type of decision structure *a single alternative decision structure.* This is because it provides only one alternative path of execution.

In Python, we use the if statement to write a single alternative decision structure. Here is the general format of the if statement:

```
if condition:
    statement
    statement
    etc.
```



For simplicity, we will refer to the first line as the *if clause*. The if clause begins with the word *if*, followed by a *condition*, which is an expression that will be evaluated as either **true** or **false**. A colon appears after the condition. Beginning at the next line is a block of statements.

A *block* is simply a set of statements that belong together as a group. Notice in the general format that all of the statements in the block are indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.

3.2 Boolean Expressions and Relational Operators

The expressions that are tested by the if statement is called *Boolean expressions*.

A *relational operator* determines whether a specific relationship exists between two values. For example, the greater than operator (>) determines whether value is greater than another. The equal to operator (==) determines whether two values are equal. The Table below lists the relational operators that are available in Python.one

Boolean expressions named in honor of the English mathematician George Boole. In the 1800s, Boole invented a system of mathematics in which the abstract concepts of true and false can be used in computations.

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

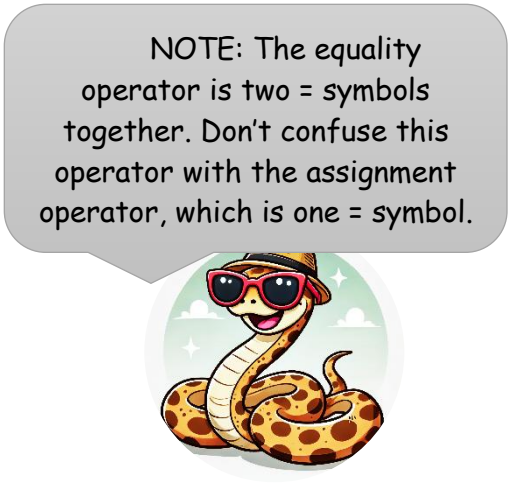
Table below shows examples of several *Boolean expressions* that compare the variables x and y.

Expression	Meaning
x > y	Is x greater than y?
x < y	Is x less than y?
x >= y	Is x greater than or equal to y?
x <= y	Is x less than or equal to y?
x == y	Is x equal to y?
x != y	Is x not equal to y?

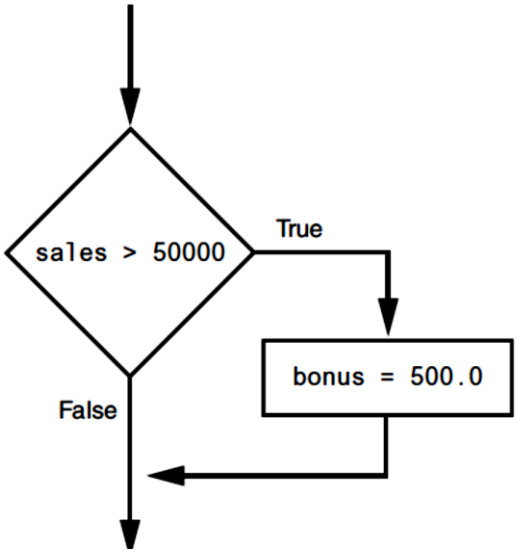
Let's look at the following example of the if statement:

```
if sales > 50000:
    bonus = 500.0
```

This statement uses the > operator to determine whether sales is greater than 50,000. If the expression sales > 50000 is true, the variable bonus is assigned 500.0.

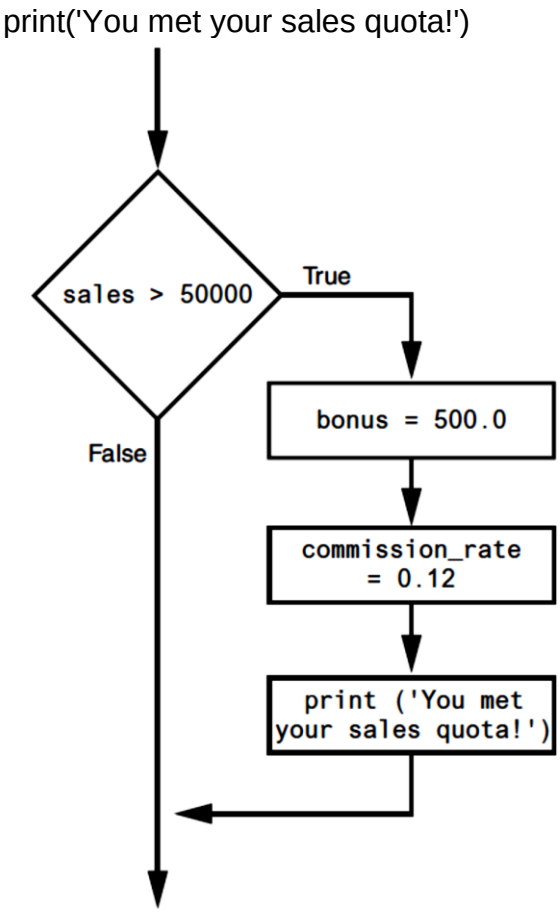


If the expression is false, however, the assignment statement is skipped. Figure below shows a flowchart for this section of code.



The following example conditionally executes a block containing three statements. Figure below shows a flowchart for this section of code:

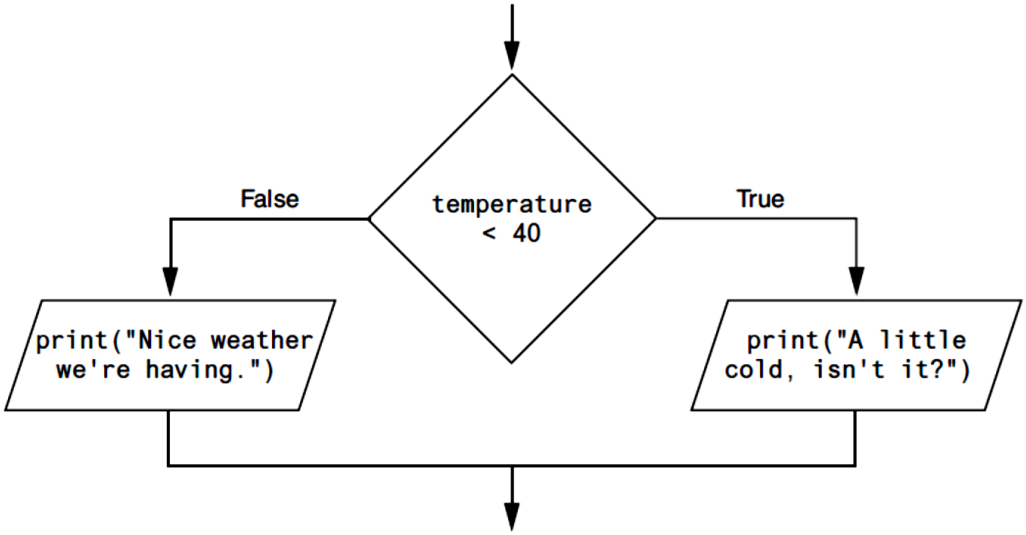
```
if sales > 50000:
    bonus = 500.0
    commission_rate = 0.12
```



3.3 The if-else Statement

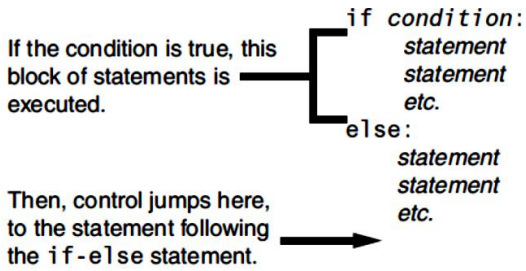
Now, we will look at the *dual alternative decision structure*, which has two possible paths of execution—one path is taken if a condition is **true**, and the other path is taken if the condition is **false**. Figure below shows a flowchart for a dual alternative decision structure.

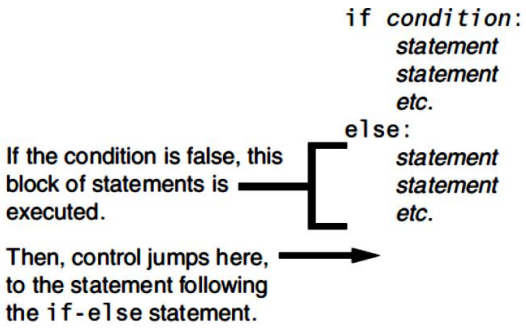
CONCEPT: An if-else statement will execute one block of statements if its condition is true, or another block if its condition is false.



In code, we write a dual alternative decision structure as an if-else statement. Here is the general format of the if-else statement:

```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
```





The following code shows an example of an if-else statement. This code matches the flowchart that was shown in Figure above.

```
if temperature < 40:
    print("A little cold, isn't it?")
else:
    print("Nice weather we're having.")
```

When you write an if-else statement, follow these guidelines for indentation:
- Make sure the if clause and the else clause are aligned.
- The if clause and the else clause are each followed by a block of statements.

Make sure the statements in the blocks are consistently indented.

```
if temperature < 40:
    print("A little cold, isn't it?")
    print("Turn up the heat!")
else:
    print("Nice weather we're having.")
    print("Pass the sunscreen.")
```

Align the if and else clauses.

The statements in each block must be indented consistently.

3.4 Comparing Strings

You can also compare strings. For example, look at the following code:

```
name1 = 'Mary'
name2 = 'Mark'
if name1 == name2:
    print('The names are the same.')
else:
    print('The names are NOT the same.')
```

CONCEPT: Python allows you to compare strings. This allows you to create decision structures that test the value of a string.

The == operator compares name1 and name2 to determine whether they are equal. Because the strings 'Mary' and 'Mark' are not equal, the else clause will display the message 'The names are NOT the same.'

Let's look at another example. Assume the month variable references a string. The following code uses the != operator to determine whether the value referenced by month is not equal to 'October':

```
if month != 'October':
    print('This is the wrong time for Octoberfest!')
```

Program below is a complete program demonstrating how two strings can be compared. The program prompts the user to enter a password, then determines whether the string entered is equal to 'prospero'.

```
1  # This program compares two strings.
2  # Get a password from the user.
3  password = input('Enter the password: ')
4
5  # Determine whether the correct password
6  # was entered.
7  if password == 'prospero':
8      print('Password accepted.')
9  else:
10     print('Sorry, that is the wrong password.')
```

Program Output (with input shown in bold)

```
Enter the password: ferdinand Enter
Sorry, that is the wrong password.
```

Program Output (with input shown in bold)

```
Enter the password: prospero Enter
Password accepted.
```

String comparisons are case sensitive. For example, the strings 'saturday' and 'Saturday' are not equal because the "s" is lowercase in the first string, but uppercase in the second string.

Program Output (with input shown in bold)

```
Enter the password: Prospero Enter
Sorry, that is the wrong password.
```

3.5 Other String Comparisons

In addition to determining whether strings are equal or not equal, you can also determine whether one string is greater than or less than another string. This is a useful capability because programmers commonly need to design programs that sort strings in some order.

Computers do not actually store characters, such as A, B, C, and so on, in memory. Instead, they store numeric codes that represent the characters. ASCII (the American Standard Code for Information Interchange) is a commonly used character coding system. here are some facts about it:

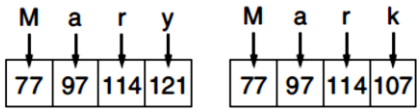
- The uppercase characters A through Z are represented by the numbers 65 through 90.
- The lowercase characters a through z are represented by the numbers 97 through 122.
- When the digits 0 through 9 are stored in memory as characters, they are represented by the numbers 48 through 57. (For example, the string 'abc123' would be stored in memory as the codes 97, 98, 99, 49, 50, and 51.)
- A blank space is represented by the number 32.

When a program compares characters, it actually compares the codes for the characters. For example, look at the following if statement:

```
if 'a' < 'b':
    print('The letter a is less than the letter b.')
```

This code determines whether the ASCII code for the character 'a' is less than the ASCII code for the character 'b'. The expression 'a' < 'b' is true because the code for 'a' is less than the code for 'b'. So, if this were part of an actual program it would display the message 'The letter a is less than the letter b.'

Let's look at how strings containing more than one character are typically compared. Suppose a program uses the strings 'Mary' and 'Mark' as follows:



When you use relational operators to compare these strings, the strings are compared character-by-character. For example, look at the following code:

```
name1 = 'Mary'
name2 = 'Mark'
if name1 > name2:
    print('Mary is greater than Mark')
else:
    print('Mary is not greater than Mark')
```

Program below shows a simple demonstration of how two strings can be compared with the < operator. The user is prompted to enter two names, and the program displays those two names in alphabetical order.

```
1  # This program compares strings with the < operator.
2  # Get two names from the user.
3  name1 = input('Enter a name (last name first): ')
4  name2 = input('Enter another name (last name first): ')
5
6  # Display the names in alphabetical order.
7  print('Here are the names, listed alphabetically.')
8
9  if name1 < name2:
10     print(name1)
11     print(name2)
12 else:
13     print(name2)
14     print(name1)
```

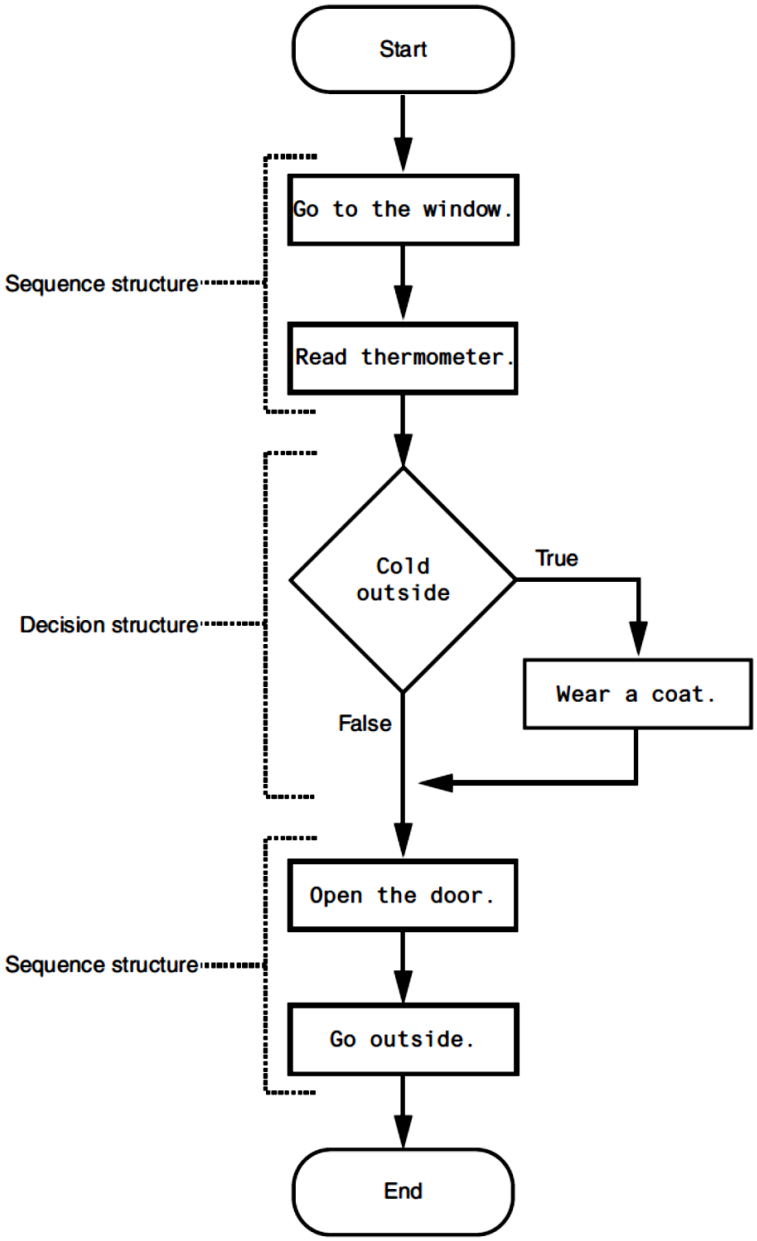
Program Output (with input shown in bold)

```
Enter a name (last name first): Jones, Richard Enter
Enter another name (last name first) Costa, Joan Enter
Here are the names, listed alphabetically:
Costa, Joan
Jones, Richard
```

3.6 Nested Decision Structures and the if-elif-else Statement

Programs are usually designed as combinations of different control structures. For example, Figure below shows a flowchart that combines a decision structure with two sequence structures.

CONCEPT: To test more than one condition, a decision structure can be nested inside another decision structure.

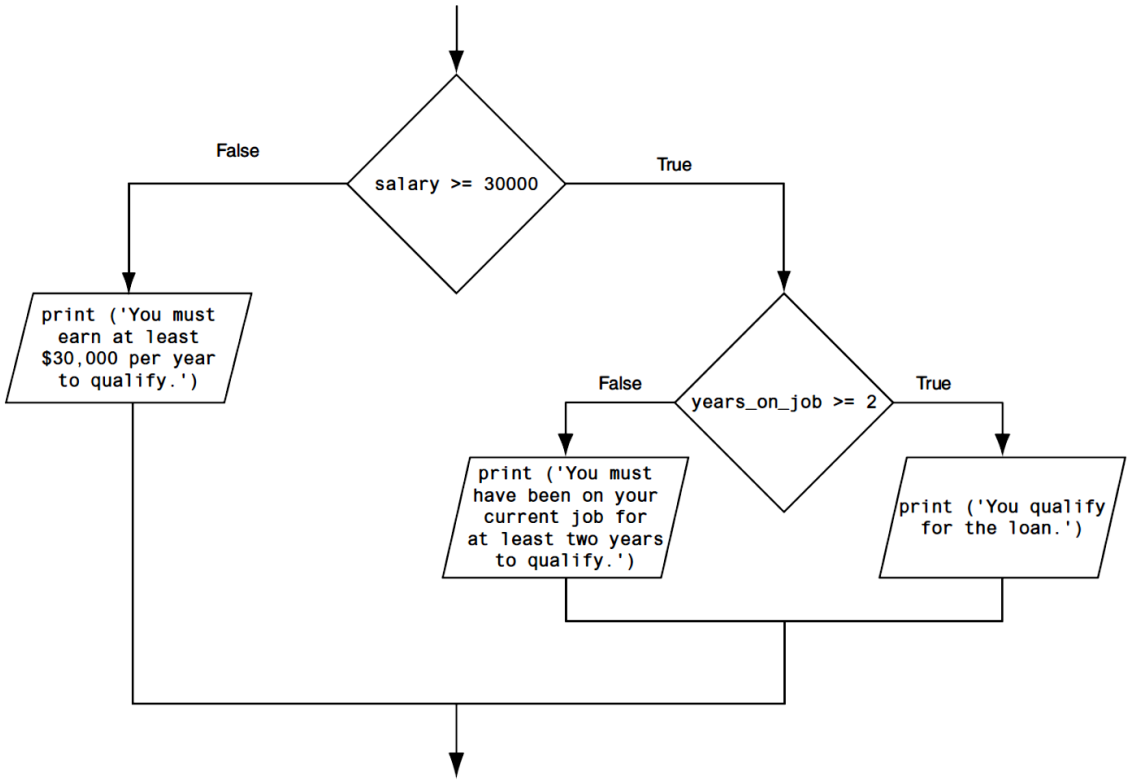


You can also nest decision structures inside other decision structures. For example, consider a program that determines whether a bank customer qualifies for a loan. To qualify, two conditions must exist:

(1) the customer must earn at least $30,000 per year.

(2) the customer must have been employed for at least two years.

Figure below shows a flowchart for an algorithm that could be used in such a program. Assume the salary variable is assigned the customer's annual salary, and the years_on_job variable is assigned the number of years that the customer has worked on his or her current job.




```
1  # This program determines whether a bank customer
2  # qualifies for a loan.
3
4  MIN_SALARY = 30000.0 # The minimum annual salary
5  MIN_YEARS = 2        # The minimum years on the job
6
7  # Get the customer's annual salary.
8  salary = float(input('Enter your annual salary: '))
9
10 # Get the number of years on the current job.
```

```
11  years_on_job = int(input('Enter the number of' +
12                           'years employed: '))
13
14  # Determine whether the customer qualifies.
15  if salary >= MIN_SALARY:
16      if years_on_job >= MIN_YEARS:
17          print('You qualify for the loan.')
18      else:
19          print('You must have been employed',
20                'for at least', MIN_YEARS,
21                'years to qualify.')
22  else:
23      print('You must earn at least $',
24            format(MIN_SALARY, ',.2f'),
25            ' per year to qualify.', sep='')
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 Enter
Enter the number of years employed: 1 Enter
You must have been employed for at least 2 years to qualify.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 25000 Enter
Enter the number of years employed: 5 Enter
You must earn at least $30,000.00 per year to qualify.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 Enter
Enter the number of years employed: 5 Enter
You qualify for the loan.
```

```
if salary >= MIN_SALARY:
    if years_on_job >= MIN_YEARS:
        print('You qualify for the loan.')
    else:
        print('You must have been employed',
              'for at least', MIN_YEARS,
              'years to qualify.')
else:
    print('You must earn at least $',
          format(MIN_SALARY, ',.2f'),
          ' per year to qualify.', sep='')
```

3.7 The if-elif-else Statement

Here is the general format of the if-elif-else statement:

```
if condition_1:
    statement
    statement
    etc.
elif condition_2:
    statement
    statement
    etc.
```

Insert as many `elif` *clauses as necessary . . .*

```
else:
    statement
    statement
    etc.
```

The following is an example of the if-elif-else statement

```
if score >= A_SCORE:
    print('Your grade is A.')
elif score >= B_SCORE:
    print('Your grade is B.')
elif score >= C_SCORE:
    print('Your grade is C.')
elif score >= D_SCORE:
    print('Your grade is D.')
else:
    print('Your grade is F.')
```

3.8 logical Operators

Python provides a set of operators known as *logical operators*, which you can use to create complex Boolean expressions. Table below describes these operators.

Operator	Meaning
`and`	The `and` operator connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.
`or`	The `or` operator connects two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which.
`not`	The `not` operator is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The `not` operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

CONCEPT: The logical and operator and the logical or operator allow you to Connect multiple Boolean expressions to create a compound expression. The logical not operator reverses the truth of a Boolean expression.

Table below shows examples of several compound Boolean expressions that use logical operators

Expression	Meaning
`x > y and a < b`	Is x greater than y AND is a less than b?
`x == y or x == z`	Is x equal to y OR is x equal to z?
`not (x > y)`	Is the expression x > y NOT true?

The and Operator

The **and** operator takes two Boolean expressions as operands and creates a compound Boolean expression that is **true** only when both subexpressions are **true**. The following is an example of an *if* statement that uses the **and** operator:

```
if temperature < 20 and minutes > 12:
    print('The temperature is in the danger zone.')
```

Table below shows a truth table for the **and** operator. The truth table lists expressions showing all the possible combinations of **true** and **false** connected with the **and** operator. The resulting values of the expressions are also shown.

Expression	Value of the Expression
true and false	false
false and true	false
false and false	false
true and true	true

The or Operator

The **or** operator takes two Boolean expressions as operands and creates a compound Boolean expression that is **true** when either of the subexpressions is **true**. The following is an example of an *if* statement that uses the **or** operator:

```
if temperature < 20 or temperature > 100:
    print('The temperature is too extreme')
```

The table shows a truth table for the or operator.

Expression	Value of the Expression
true or false	true
false or true	true
false or false	false
true or true	true

The not Operator

The **not** operator is a unary operator that takes a Boolean expression as its operand and reverses its logical value. In other words, if the expression is **true**, the not operator returns **false**, and if the expression is **false**, the not operator returns **true**. The following is an *if* statement using the **not** operator:

```
if not(temperature > 100):
    print('This is below the maximum temperature.')
```

The table shows a truth table for the not operator

Expression	Value of the Expression
not true	false
not false	true

NOTE: In this example, we have put parentheses around the expression temperature > 100. This is to make it clear that we are applying the not operator to the value of the expression temperature > 100, not just to the temperature variable.

3.9 Boolean Variables

So far in this lecture, we have worked with int, float, and str (string) variables. In addition to these data types, Python also provides a bool data type. The bool data type allows you to create variables that may reference one of two possible values: True or False. Here are examples of how we assign values to a bool variable:

```
hungry = True
sleepy = False
```

CONCEPT: A Boolean variable can reference one of two values: True or False. Boolean variables are commonly used as flags, which indicate whether specific conditions exist.

For example, suppose a salesperson has a quota of $50,000. Assuming sales references the amount that the salesperson has sold, the following code determines whether the quota has been met:

```
if sales >= 50000.0:
    sales_quota_met = True
else:
    sales_quota_met = False
```

This code is equivalent to the following:

```
if sales_quota_met == True:
    print('You have met your sales quota!')
```