Python Programming
Module 5: `if` Statements

Learning objectives

1. Writing conditional tests, which always evaluate to True or False.
2. Writing simple if statements, if-else chains, and if-elif-else chains.
3. Using if, if-else and if-elif-else structures to identify particular conditions you needed to test, and to know when those conditions have been met in your programs.
4. You learned to handle certain items in a list differently than all other items while continuing to utilize the efficiency of a for loop.

If statements

A simple example

The following short example shows how if tests let you respond to special situations correctly. Imagine you have a list of cars and you want to print out the name of each car. Car names are proper names, so the names of most cars should be printed in title case. However, the value `'bmw'` should be printed in all uppercase. The following code loops through a list of car names and looks for the value `'bmw'`. Whenever the value is `'bmw'`, it's printed in uppercase instead of title case:

```
  cars = ['audi', 'bmw', 'subaru', 'toyota']

  for car in cars:
1     if car == 'bmw':
            print(car.upper())
      else:
            print(car.title())
```

The loop in this example first checks if the current value of car is `'bmw'` at `1`. If it is, the value is printed in uppercase. If the value of car is anything other than `'bmw'`, it's printed in title case:

```
  Audi
  BMW
  Subaru
  Toyota
```

Conditional tests

At the heart of every `if` statement is an expression that can be evaluated as `True` or `False` and is called a conditional test. Python uses the values `True` and `False` to decide whether the code in an if statement should be executed. If a conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the if statement.

Checking for Equality

The simplest conditional test checks whether the value of a variable is equal to the value of interest:

```
1 >>> car = 'bmw'
2 >>> car == 'bmw'
  True
```

The line at `1` sets the value of car to `'bmw'` using a single equal sign, as you've seen many times already. The line at `2` checks whether the value of car is `'bmw'` using a double equal sign (= =). This equality operator returns `True` if the values on the left and right side of the operator match, and `False` if they don't match. The values in this example match, so Python returns `True`.
When the value of car is anything other than `'bmw'`, this test returns `False`:

```
1 >>> car = 'audi'
2 >>> car == 'bmw'
```

```
  False
```

A single equal sign is really a statement; you might read the code at 1 as "Set the value of car equal to 'audi'." On the other hand, a double equal sign, like the one at 2, asks a question: "Is the value of car equal to 'bmw'?" Most programming languages use equal signs in this way.

Ignoring Case When Checking for Equality

Testing for equality is case sensitive in Python. For example, two values with different capitalization are not considered equal:

```
  >>> car = 'Audi'
  >>> car == 'audi'
  False
```

If case matters, this behaviour is advantageous. But if case doesn't matter and instead you just want to test the value of a variable, you can convert the variable's value to lowercase before doing the comparison:

```
  >>> car = 'Audi'
  >>> car.lower() == 'audi'
  True
```

This test would return `True` no matter how the value `'Audi'` is formatted because the test is now case insensitive. The `lower()` function doesn't change the value that was originally stored in car, so you can do this kind of comparison without affecting the original variable:

```
1 >>> car = 'Audi'
2 >>> car.lower() == 'audi'
  True
3 >>> car
  'Audi'
```

At 1, we store the capitalized string `'Audi'` in the variable car. At 2, we convert the value of car to lowercase and compare the lowercase value to the string `'audi'`. The two strings match, so Python returns `True`. At 3, we can see that the value stored in car has not been affected by the conditional test.

Checking for Inequality

When you want to determine whether two values are not equal, you can combine an exclamation point and an equal sign `(!=)`. The exclamation point represents not, as it does in many programming languages.

For example,

```
  requested_topping = 'mushrooms'

1 if requested_topping != 'anchovies':
  print("Hold the anchovies!")
```

The line at 1 compares the value of requested_topping to the value 'anchovies'. If these two values do not match, Python returns True and executes the code following the if statement. If the two values match, Python returns False and does not run the code following the if statement.
Because the value of requested_topping is not 'anchovies', the print statement is executed:

```
Hold the anchovies!
```

Most of the conditional expressions you write will test for equality, but sometimes you'll find it more efficient to test for inequality.

Numerical Comparisons

Testing numerical values is pretty straight forward. For example, you can test to see if two numbers are not equal:

```
answer = 17

1 if answer != 42:
    print("That is not the correct answer. Please try again!")
```

The conditional test at 1 passes, because the value of answer (17) is not equal to 42. Because the test passes, the indented code block is executed:

```
That is not the correct answer. Please try again!
```

You can include various mathematical comparisons in your conditional statements as well, such as less than, less than or equal to, greater than, and greater than or equal to:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking Multiple Conditions

You may want to check multiple conditions at the same time. For example, sometimes you might need two conditions to be True to take an action. Other times you might be satisfied with just one condition being True. The keywords and and or can help you in these situations.

Using and to Check Multiple Conditions

To check whether two conditions are both `True` simultaneously, use the keyword `and` to combine the two conditional tests; if each test passes, the overall expression evaluates to `True`. If either test fails or if both tests fail, the expression evaluates to `False`.

For example, you can check whether two people are both over 21 using the following test:

```
1 >>> age_0 = 22
  >>> age_1 = 18
2 >>> age_0 >= 21 and age_1 >= 21
  False
3 >>> age_1 = 22
  >>> age_0 >= 21 and age_1 >= 21
  True
```

At `1`, we define two ages, `age_0` and `age_1`. At `2`, we check whether both ages are 21 or older. The test on the left passes, but the test on the right fails, so the overall conditional expression evaluates to `False`. At `3`, we change `age_1` to 22. The value of `age_1` is now greater than 21, so both individual tests pass, causing the overall conditional expression to evaluate as `True`.

Using `or` to Check Multiple Conditions

The keyword `or` allows you to check multiple conditions as well, but it passes when either or both of the individual tests pass. An or expression fails only when both individual tests fail.

Let's consider two ages again, but this time we'll look for only one person to be over 21:

```
1 >>> age_0 = 22
  >>> age_1 = 18
2 >>> age_0 >= 21 or age_1 >= 21
  True
3 >>> age_0 = 18
  >>> age_0 >= 21 or age_1 >= 21
  False
```

We start with two age variables again at `1`. Because the test for `age_0` at `2` passes, the overall expression evaluates to `True`. We then lower `age_0` to 18. In the test at `3`, both tests now fail and the overall expression evaluates to `False`.

Checking Whether a Value Is in a List

To find out whether a particular value is already in a list, use the keyword `in`. Let's consider some code you might write for a pizzeria. We'll make a list of toppings a customer has requested for a pizza and then check whether certain toppings are in the list.

```
  >>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
1 >>> 'mushrooms' in requested_toppings
  True
```

```
2 >>> 'pepperoni' in requested_toppings
  False
```

At `1` and `2`, the keyword in tells Python to check for the existence of `'mushrooms'` and `'pepperoni'` in the list `requested_toppings`. This technique is quite powerful because you can create a list of essential values, and then easily check whether the value you're testing matches one of the values in the list.

Checking Whether a Value Is Not in a List

You can use the keyword `not in` this situation. For example, consider a list of users who are banned from commenting in a forum. You can check whether a user has been banned before allowing that person to submit a comment:

```
  banned_users = ['andrew', 'carolina', 'david']
  user = 'marie'

1 if user not in banned_users:
      print(user.title() + ", you can post a response if you wish.")
```

The line at `1` reads quite clearly. If the value of user is not in the list `banned_users`, Python returns `True` and executes the indented line.

The user `'marie'` is not in the list `banned_users`, so she sees a message inviting her to post a response:

```
  Marie, you can post a response if you wish.
```

Boolean Expressions

A Boolean expression is just another name for a conditional test. A Boolean value is either `True` or `False`, just like the value of a conditional expression after it has been evaluated.

Boolean values are often used to keep track of certain conditions, such as whether a game is running or whether a user can edit certain content on a website:

```
  game_active = True
  can_edit = False
```

Boolean values provide an efficient way to track the state of a program or a particular condition that is important in your program.

If statements

Simple if Statements

The simplest kind of if statement has one test and one action:

```
  if conditional_test:
      do something.
```

You can put any conditional test in the first line and just about any action in the indented block following the test. Let's say we have a variable representing a person's age, and we want to know if that person is old enough to vote. The following code tests whether the person can vote:

```
  age = 19
1 if age >= 18:
2     print("You are old enough to vote!")
```

At `1`, Python checks to see whether the value in age is greater than or equal to 18. It is, so Python executes the intended print statement at `2`:

```
  You are old enough to vote!
```

You can have as many lines of code as you want in the block following the if statement. Let's add another line of output if the person is old enough to vote, asking if the individual has registered to vote yet:

```
  age = 19
  if age >= 18:
     print("You are old enough to vote!")
     print("Have you registered to vote yet?")
```

The conditional test passes, and both print statements are indented, so both lines are printed:

```
  You are old enough to vote!
  Have you registered to vote yet?
```

If the value of age is less than 18, this program would produce no output.

if-else Statements

Often, you'll want to take one action when a conditional test passes and a different action in all other cases. We'll display the same message we had previously if the person is old enough to vote, but this time we'll add a message for anyone who is not old enough to vote:

```
  age = 17
1 if age >= 18:
      print("You are old enough to vote!")
      print("Have you registered to vote yet?")
2 else:
      print("Sorry, you are too young to vote.")
      print("Please register to vote as soon as you turn 18!")
```

If the conditional test at `1` passes, the first block of indented print statements is executed. If the test evaluates to `False`, the else block at `2` is executed. Because age is less than 18 this time, the conditional test fails and the code in the else block is executed:

```
  Sorry, you are too young to vote.
  Please register to vote as soon as you turn 18!
```

This code works because it has only two possible situations to evaluate: a person is either old enough to vote or not old enough to vote. The if-else structure works well in situations in which you want Python to always execute one of two possible actions. In a simple if-else chain like this, one of the two actions will always be executed.

The if-elif-else Chain

Often, you'll need to test more than two possible situations, and to evaluate these you can use Python's `if-elif-else` syntax. Python executes only one block in an `if-elif-else` chain. It runs each conditional test in order until one passes. When a test passes, the code following that test is executed and Python skips the rest of the tests.

Many real-world situations involve more than two possible conditions. For example, consider an amusement park that charges different rates for different age groups:

```
Admission for anyone under age 4 is free.
Admission for anyone between the ages of 4 and 18 is $5.
Admission for anyone age 18 or older is $10.
```

How can we use an if statement to determine a person's admission rate? The following code tests for the age group of a person and then prints an admission price message:

```
age = 12

1 if age < 4:
      print("Your admission cost is $0.")
2 elif age < 18:
      print("Your admission cost is $5.")
3 else:
      print("Your admission cost is $10.")
```

The if test at `1` tests whether a person is under 4 years old. If the test passes, an appropriate message is printed and Python skips the rest of the tests. The elif line at `2` is really another if test, which runs only if the previous test failed. At this point in the chain, we know the person is at least 4 years old because the first test failed. If the person is less than 18, an appropriate message is printed and Python skips the else block. If both the if and elif tests fail, Python runs the code in the else block at `3`.

In this example, the test at `1` evaluates to `False`, so its code block is not executed. However, the second test evaluates to `True` (12 is less than 18) so its code is executed. The output is one sentence, informing the user of the admission cost:

```
Your admission cost is $5.
```

Any age greater than 17 would cause the first two tests to fail. In these situations, the else block would be executed and the admission price would be $10.

Rather than printing the admission price within the if-elif-else block, it would be more concise to set just the price inside the if-elif-else chain and then have a simple print statement that runs after the chain has been evaluated:

```
  age = 12
  if age < 4:
1     price = 0
  elif age < 18:
2     price = 5
  else:
3     price = 10

4 print("Your admission cost is $" + str(price) + ".")
```

The lines at `1`, `2`, and `3` set the value of price according to the person's age, as in the previous example. After the price is set by the if-elif-else chain, a separate unindented print statement `4` uses this value to display a message reporting the person's admission price.

This code produces the same output as the previous example, but the purpose of the if-elif-else chain is narrower. Instead of determining a price and displaying a message, it simply determines the admission price. In addition to being more efficient, this revised code is easier to modify than the original approach. To change the text of the output message, you would need to change only one print statement rather than three separate print statements.

Using Multiple `elif` *Blocks*

You can use as many elif blocks in your code as you like. For example, if the amusement park were to implement a discount for seniors, you could add one more conditional test to the code to determine whether someone qualified for the senior discount. Let's say that anyone 65 or older pays half the regular admission, or $5:

```
  age = 12

  if age < 4:
     price = 0
  elif age < 18:
     price = 5
1 elif age < 65:
     price = 10
2 else:
     price = 5

  print("Your admission cost is $" + str(price) + ".")
```

Most of this code is unchanged. The second `elif` block at `1` now checks to make sure a person is less than age 65 before assigning them the full admission rate of $10. Notice that the value assigned in the else

block at 2 needs to be changed to $5, because the only ages that make it to this block are people 65 or older.

Omitting the else Block

Python does not require an else block at the end of an if-elif chain. Sometimes an else block is useful; sometimes it is clearer to use an additional elif statement that catches the specific condition of interest:

```
  age = 12

  if age < 4:
      price = 0
  elif age < 18:
      price = 5
  elif age < 65:
      price = 10
1 elif age >= 65:
      price = 5

  print("Your admission cost is $" + str(price) + ".")
```

The extra elif block at 1 assigns a price of $5 when the person is 65 or older, which is a bit clearer than the general else block. With this change, every block of code must pass a specific test in order to be executed.

The else block is a catch all statement. It matches any condition that wasn't matched by a specific if or elif test, and that can sometimes include invalid or even malicious data. If you have a specific final condition you are testing for, consider using a final elif block and omit the else block. As a result, you'll gain extra confidence that your code will run only under the correct conditions.

Testing Multiple Conditions

The if-elif-else chain is powerful, but it's only appropriate to use when you just need one test to pass. As soon as Python finds one test that passes, it skips the rest of the tests. This behaviour is beneficial, because it's efficient and allows you to test for one specific condition.

However, sometimes it's important to check all of the conditions of interest. In this case, you should use a series of simple if statements with no elif or else blocks. This technique makes sense when more than one condition could be `True` and you want to act on every condition that is `True`.

Let's reconsider the pizzeria example. If someone requests a two-topping pizza, you'll need to be sure to include both toppings on their pizza:

```
1 requested_toppings = ['mushrooms', 'extra cheese']

2 if 'mushrooms' in requested_toppings:
      print("Adding mushrooms.")
3 if 'pepperoni' in requested_toppings:
      print("Adding pepperoni.")
4 if 'extra cheese' in requested_toppings:
```

```
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

Because every condition in this example is evaluated, both mushrooms and extra cheese are added to the pizza:

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

Checking for Special Items

Let's continue with the pizzeria example. The pizzeria displays a message whenever a topping is added to your pizza, as it's being made. The code for this action can be written very efficiently by making a list of toppings the customer has requested and using a loop to announce each topping as it's added to the pizza:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print("Adding " + requested_topping + ".")

print("\nFinished making your pizza!")
```

The output is straightforward because this code is just a simple for loop:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

But what if the pizzeria runs out of green peppers? An `if` statement inside the `for` loop can handle this situation appropriately:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
1 if requested_topping == 'green peppers':
    print("Sorry, we are out of green peppers right now.")
2 else:
    print("Adding " + requested_topping + ".")
print("\nFinished making your pizza!")
```

This time we check each requested item before adding it to the pizza. The code at `1` checks to see if the person requested green peppers. If so, we display a message informing them why they can't have green peppers. The else block at `2` ensures that all other toppings will be added to the pizza.
The output shows that each requested topping is handled appropriately:

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

Checking That a List Is Not Empty

As an example, let's check whether the list of requested toppings is empty before building the pizza. If the list is empty, we'll prompt the user and make sure they want a plain pizza. If the list is not empty, we'll build the pizza just as we did in the previous examples:

```
1 requested_toppings = []

2 if requested_toppings:
      for requested_topping in requested_toppings:
            print("Adding " + requested_topping + ".")
      print("\nFinished making your pizza!")
3 else:
      print("Are you sure you want a plain pizza?")
```

This time we start out with an empty list of requested toppings at `1`. Instead of jumping right into a for loop, we do a quick check at `2`. When the name of a list is used in an `if` statement, Python returns `True` if the list contains at least one item; an empty list evaluates to `False`. If `requested_toppings` passes the conditional test, we run the same for loop we used in the previous example. If the conditional test fails, we print a message asking the customer if they really want a plain pizza with no toppings `3`.
The list is empty in this case, so the output asks if the user really wants a plain pizza:

```
Are you sure you want a plain pizza?
```

If the list is not empty, the output will show each requested topping being added to the pizza.

Using Multiple Lists

Let's watch out for unusual topping requests before we build a pizza. The following example defines two lists. The first is a list of available toppings at the pizzeria, and the second is the list of toppings that the user has requested. This time, each item in `requested_toppings` is checked against the list of available toppings before it's added to the pizza:

```
1 available_toppings = ['mushrooms', 'olives', 'green peppers',
                        'pepperoni', 'pineapple', 'extra cheese']

2 requested_toppings = ['mushrooms', 'french fries', 'extra cheese']
```

```
3 for requested_topping in requested_toppings:
4     if requested_topping in available_toppings:
          print("Adding " + requested_topping + ".")
5     else:
          print("Sorry, we don't have " + requested_topping + ".")

  print("\nFinished making your pizza!")
```

At `1`, we define a list of available toppings at this pizzeria. Note that this could be a tuple if the pizzeria has a stable selection of toppings. At `2`, we make a list of toppings that a customer has requested. Note the unusual request, `'french fries'`. At `3`, we loop through the list of requested toppings. Inside the loop, we first check to see if each requested topping is actually in the list of available toppings `4`. If it is, we add that topping to the pizza. If the requested topping is not in the list of available toppings, the else block will run `5`. The else block prints a message telling the user which toppings are unavailable.
This code syntax produces clean, informative output:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

In just a few lines of code, we've managed a real-world situation pretty effectively!

Exercises

1. Write a Python program to find those numbers which are divisible by 7 and multiple of 5, between 1500 and 2700 (both included).

2. Write a Python program to convert temperatures to and from celsius, fahrenheit. [Formula : c/5 = f-32/9 [where c = temperature in celsius and f = temperature in fahrenheit]

3. Write a Python program that accepts a word from the user and reverse it, and if word is palindrome then print 'Mirror Case'.

4. Write a Python program to count the number of even and odd numbers from a series of numbers.
   Sample numbers : numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9)

5. Write a Python program that prints all the numbers from 0 to 6 except 3 and 6.

6. Write a Python program to get the Fibonacci series between 0 to 50.
   Note : The Fibonacci Sequence is the series of numbers : 0, 1, 1, 2, 3, 5, 8, 13, 21, …

7. Write a Python program which iterates the integers from 1 to 50. For multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

8. Write a Python program that accepts a string and calculate the number of digits and letters.

9. Write a Python program to check the validity of password input by users. Go to the editor
   Validation :
   - At least 1 letter between [a-z] and 1 letter between [A-Z].
   - At least 1 number between [0-9].
   - At least 1 character from [$#@].
   - Minimum length 6 characters.
   - Maximum length 16 characters.

10. Write a Python program to find numbers between 100 and 400 (both included) where each digit of a number is an even number. The numbers obtained should be printed in a comma-separated sequence.

11. Write a Python program to check whether an alphabet is a vowel or consonant.

12. Write a Python program to convert month name to a number of days of the year 2100.
    Sample Input: July
    Sample Output: 31

13. Write a Python program to check a triangle is equilateral, isosceles or scalene. Go to the editor
    Note :
    - An equilateral triangle is a triangle in which all three sides are equal.
    - A scalene triangle is a triangle that has three unequal sides.

- An isosceles triangle is a triangle with (at least) two equal sides.
  Example:
  Input lengths of the triangle sides:
  x: 6
  y: 8
  z: 12
  Scalene triangle

14. Write a Python program that reads two integers representing a month and day and prints the season for that month and day in Chennai, India.
15. Create a function that counts the number of elements within a list that are greater than 30.