Introduction

How do you write programs that can make decisions? As a Python programmer a lot of your time coding will be spent writing programs that receive, and examine information and then determine what action to take based on that information. Python's **if** statement allows you to write programs that respond in a predetermined way based on current information of condition within your programs.

In this and the next several videos we are going to be writing conditional tests, which will allow you to check any condition within your programs.

In [7]:
```
order_013678 = ['paper', 'pen', 'notepad', 'marker', 'highlighter']

for order in order_013678:
    if order == 'marker':
        print(order.title(), "* Item currently out of stock *")
    else:
        print(order.title())
```

```
Paper
Pen
Notepad
Marker * Item currently out of stock *
Highlighter
```

In this brief example, our program has responded to the condition that a customer order has one item that is out of stock. Imagine you have an online stationary business and a customer has just placed an order with you for paper, pens, notepads, markers, and highlighters. They reach the checkout of your website and they are asked to review and confirm their order. They see the message that one item is out of stock and so they select a different item instead.

- At line one we have created a new variable which is a random customer order number. We then assign a list of items to that variable.
- At line three we define a **for** loop
- At line four our loop checks if the current value of **order** is 'marker'. If it is then the out of stock message is printed alongside it. If the value is anything other than 'marker' it is printed out.

  Let's continue this lesson by looking at **conditional tests**.

Conditional tests

Every **if** statement needs an expression to be evaluated to **True** or **False**. This is called a **conditional test**. In Python, when an **if** statement evaluates to **True** or **False** it uses these values to decide whether the code of an **if** statement should be executed. If a conditional test is True, Python executes the code following the **if** statement. If the test evaluates to **False**, Python will ignore the code following the **if** statement.

Checking for equality

Checking for equality is the act of comparing the current value of a variable to a specific value of interest. For example the login email address of a website customer:

In [8]:
```
email_address = 'tony@example.com'
email_address == 'tony@example.com'
```

Out[8]:
```
True
```

In the example above we have first assigned the email address 'tony@example.com' to the variable **email_address**. This is done using the equals symbol (=). At line two we check if the value of **email_address** is 'tony@example.com'. We do this using a double equals symbol (==). This symbol is called the **equality_operator** and it returns **True** if the value on its left matches the value on its right. If they don't match then it returns **False**.

In [10]:
```
email_address = 'tony@example.com'
email_address == 'frank@example.com'
```

Out[10]:
```
False
```

Ignoring case when checking for equality

When testing for equality in Python remember that Pyhton is case sensitive. For example:

In [11]:
```
name = 'tony'
name == 'Tony'
```

Out[11]:
```
False
```

In [12]:
```
name = 'Tony'
name == 'Tony'
```

Out[12]:
```
True
```

We've all used our email address to login to a website or application. Sometimes, the first initial is automatically capitalized so that 't' of tony@example.com becomes Tony@example.com. Yet the website or application still allows us to login. How is this? As we have seen in the two examples above 't' is not the same as 'T'. When creating the login features of a website or application case should not matter. Let's take a look:

In [13]:
```
email_address = 'Tony@example.com'
email_address.lower() == 'tony@example.com'
```

Out[13]:
```
True
```

In this example we assign the capitalized email address 'Tony@example.com' to the variable **email_address**. At line two we use Python's **lower()** function to lower any capitalized letter in **email_address**. We then compare the lowercase email address to 'tony@example.com. The two email addresses match so Python returns **True**.

The **lower** function doesn't change the value that was originally in **email_address**, which means that this comparison does not affect the original variable.

There are multiple examples where this type of comparison takes place, such as login forms of websites and applications.

Checking for inequality

Now that we know how to check if two values are equal to one another, how do we check when they are not equal? To do this we combine an exclamation mark (!) with an equals symbol (=). In Python as in many programming languages the exclamation mark represents **not**. Let's look at an example:

In [15]:
```
my_birthday = 'november'

if my_birthday != 'november':
    print('Your birthday is not this month!')
```

In this example the code at line 3 compares the value of **my_birthday** to the value of 'march'. If these two values do not match, Python returns **True** and executes the code after the **if** statement. If the two values were to match, Python would return **False** and would not run the code following the **if** statement.

In our example the value of **my_birthday** is not (!=) 'march' and so the print function is executed.

Numerical comparisons

Let's now learn how to test numbers:

In [16]:
```
age = 21
age == 21
```

Out[16]:
```
True
```

We can test to see if two numbers are not equal to each other.

In [17]:
```
unlock_code = 2121

if unlock_code != 40:
    print("Incorrect code. Two attempts remaining.")
```

```
Incorrect code. Two attempts remaining.
```

Image you have a phone that requires a four digit PIN code to unlock it. In the example above, a user has entered the unlock code 2121 into the phone. At line 3 the conditional test passes because the value of **unlock_code** is not equal (!=) to 2121, instead the current value of unlock_code is 40. Because the test passes the print statement is executed informing the user that the PIN code entered is incorrect.

We can also use mathematical operators in our conditional statements. These operators include, less than (<), less than or equal to <=), greater than (>), and greater than or equal to (>=). Let's take a look:

In [18]:
```
age = 18
age < 21
```

Out[18]:
```
True
```

In [19]:
```
age = 18
age <= 21
```

Out[19]:
```
True
```

In [20]:
```
age = 18
age > 21
```

Out[20]:
```
False
```

In [21]:
```
age = 18
age >= 21
```

Out[21]:
```
False
```

Checking multiple conditions

There will be times when you need to check several conditions at the same time. For example, you may be writing a program that requires two conditions to be **True**, username and password, in order to perform an action. There may be times when you have two conditions and only one needs to be **True** to perform an action.

Using and to check multiple conditions

To check if two conditions are both **True** at the same time we use the **and** keyword. This keyword allow us to combine two conditional tests. If each test passes, then the overall expression will return true. If either test fails or if both tests fail then the expression will return false. Let's take a look:

In [22]:
```
age_1 = 16
age_2 = 18

age_1 >= 18 and age_2 >= 18
```

Out[22]:
```
False
```

In the code example above we defined two age variables, **age_1** and **age_2**. At line 4 we check if both ages are 18 or older. As **age_1** is not greater than or equal to 18 the overall expression returns false. Let's take a look at another example:

In [23]:
```
age_1 = 19
age_2 = 20

age_1 >= 18 and age_2 >= 18
```

Out[23]:
```
True
```

In this example we have another two age variables, **age_1** and **age_2**. This time both variables are over 18. The expression on line 4 checks if **age_1** and **age_2** are greater than or equal to 18. As both variables are the expression returns **True**.

Using or to check multiple conditions

We can also check multiple conditions using the **or** keyword. This keyword will allow an expression to return **True** when either one or both tests pass. An **or** expression will fail only when both individual tests fail. Let's take a look at an example:

In [24]:
```
age_1 = 20
age_2 = 17

age_1 >= 18 or age_2 >= 18
```

Out[24]:
```
True
```

Again we have two age variables. Our test at line 4 checks to see if **age_1** or **age_2** is greater than or equal to 18. As only one test passes our expression returns **True**. Let's look at two failing tests:

In [25]:
```
age_1 = 17
age_2 = 16

age_1 >= 18 or age_2 >= 18
```

Out[25]:
```
False
```

Is a value in a list

We've looked at lists quite a bit during previous video. Let's revisit them again and this time learn how to check if a value is in a list. Why would this be necessary? You could have a website that allows customers to sign up with a username. All current usernames are stored on a list and to prevent duplication you want to check if a new username already exists.

To perform this check we use the keyword **in**. Let's take a look:

In [26]:
```
customers = ['tstaunton', 'jsmith', 'fmurphy']
'tstaunton' in customers
```

Out[26]:
```
True
```

In [27]:
```
customers = ['tstaunton', 'jsmith', 'fmurphy']
'frank001' in customers
```

Out[27]:
```
False
```

In the two examples above we use the **in** keyword to tell Python to scan the lists, **customers**, and check if 'tstaunton' or 'frank001' is in them. As you can see in example one our code returns **True** and in example two our code returns **False**.

Checking if a value is not in a list

As well as checking if a value is in a list, it is also helpful to know how to check if a value is not in a list. Let's take a look:

In [29]:
```
not_for_sale = ['red car', 'blue car', 'green car']
for_sale = 'red car'

if for_sale not in not_for_sale:
    print(f"{for_sale}, this car is in stock.")
```

In this example, if the value of **for_sale** is not in the list **not_for_sale**, Python returns **True** and executes the print statement.

Booleans

So far we've been using conditional tests to make our programs return **True** or **False** depending on the state of the program. Another term for conditional tests is **Boolean expressions**. A **boolean value** can be **True** or **False**. Boolean values are an efficient way to track the state of a program or a particular condition that is of importance to your program.

Exercises

- Write two conditional test that creates a number guessing game for your user. Have each test return **True** and **False**.
- Did you use a string or an integer to represent your number? Change your code to use the opposite
- Edit your code to test using the **lower()** method
- Write another guessing game this time using greater than, less than, greater than or equal to, and less than or equal to

In [30]:
```
my_number = 'seven'
print("Is your number seven?")
print(my_number == 'seven')
```

```
Is your number seven?
True
```

If statements

We've already seen and used **if** statements but there are several different kinds of **if** statement. Which one you choose to use depends on the number of conditions that you need to test. Let's take a look:

If statements

In [31]:
```
age = 18
if age >= 18:
    print("You are old enough to vote!")
```

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

We've seen this **if** statement before. At line 2 Python checks to see whether the value of **age** is greater than or equal to 18. If the condition is True, execute the **print** statement, which it does! If the test failed this program would do nothing.

Just as it did with **for** loops, indentation plays an important part of **if** statements.

if conditional_test:
  execute code

All indented lines after an **if** statement will be executed if the test passes. If the test fails all indented lines will be ignored. You can have as many lines of code as you like in the block following an **if** statement.

In [32]:
```
age = 16

if age >= 16:
    print("You are old enough to drive.")
    print("Do you have a license?")
```

```
You are old enough to drive.
Do you have a license?
```

In this code example the conditional test passes and so the code block that follows it is executed. If the test failed nothing would be output and the program would stop.

if-else statements

In [33]:
```
age = 15

if age >= 16:
    print("You are old enough to drive.")
else:
    print("Sorry you are not old enogh to drive.")
```

```
Sorry you are not old enogh to drive.
```

In this code example, we have defined a variable, **age**, at line 1. We then check if this variable is greater than or equal to 16, if it is then our program executes a **print** statement. What's different about this code is that we have added an **else** block at line 5. If the test **age >= 16** were to fail the **else** block would execute. In this example the test does fail and so the **else** block does execute.

Combing an **if** statement with an **else** statement allows our programs to perform one action when a test passes and a different action when it doesn't. In Python this syntax is

called an **if-else** block. It's very similar to an **if** statement but the addition of the **else** statement allows us to define an action or a set of actions that are executed when the condition fails.

The **if-else** block works very well when you want to execute one of two possible actions.

- If **True** then do X
- If **False** then do Y

if-elif-else chain

As we discussed, the **if-else** block works very well in situations that require your programs to execute one of two possible actions. But you are going to need to test more than two possible actions and when you do you can use Python's **if-elif-else** syntax. In an **if-elif-else** chain Python runs each conditional test in order until one passes. Let's take a look:

In [35]:
```
years_of_service = 10

if years_of_service <= 1:
    print("Your bonus is $50.")
elif years_of_service <= 5:
    print("Your bonus is $100")
else:
    print("Please contact your manager")
```

```
Please contact your manager
```

The **if** statement at line 3 tests if an employee has been with the company for 1 year or less. If the test passes, the 50 dollar bonus message is printed out and Python will skip the rest of the program. At line 5 we have the **elif** keyword, this is another **if** test which will run if the first test fails. When Python reaches line 5 it knows that the employee has worked with the company for more than 1 year. If the second test succeeds, which means that the employee is with the company less than 5 years but more than 1, the 100 dollar bonus message is printed out and the **else** block is ignored. If the tests at lines 3 and 5 fail then Python will execute the code at line 7, the **else** block.

In this example, the test at line 3 fails, so Python moves to line 5 and this test succeeds and runs the **print** statement at line 6.

Go ahead and edit the number at line 1 so that the print statement on line 8 is executed.

There is a way to make this example more efficient:

In [36]:
```
years_of_service = 1

if years_of_service <= 1:
    bonus = 50
elif years_of_service <= 5:
    bonus = 100
else:
    bonus = "Yet to be confirmed. Please contact your manager."

print(f"You bonus is ${bonus}.")
```

```
You bonus is $50.
```

In this example we set the value of an employees bonus at lines 4, 6 and 8. With the bonus now set within the **if-elif-else** chain, we have an unindented **print()** statement at line 10. This print statement will receive the correct bonus value and print it to the screen. This is code is simpler and easier to maintain.

Multiple elif blocks

The bonus example above is very limited. But Python allows us to us as many **elif** blocks as required. Let's update the bonus example:

In [39]:
```
years_of_service = 4

if years_of_service <= 1:
    bonus = 50
elif years_of_service <= 3:
    bonus = 100
elif years_of_service <= 5:
    bonus = 150
else:
    bonus = 200

print(f"For {years_of_service} years of service, your bonus is ${bonus}.")
```

```
For 20 years of service, your bonus is $200.
```

This is a nice little piece of code. Let's walk-through it:

- At line 1 we define a variable called 'years_of_service' and give this variable the value of 4
- Our first test is at line 3, and tests to see of the 'years_of_service' variable is less than or equal to 1, if so the bonus of 50 is applied
- The second test at line 5, checks if the 'years_of_service' variable is less then or equal to 3, and if so apply a bonus of 100
- The third test at line 7, checks if the 'years_of_service' variable is less then or equal to 5, and if so apply a bonus of 150
- The **else** block at line 9 is a catch-all. If the three tests above fail then Python knows that your years of service is greater than 5 and so applies a bonus of 200.
- Finally at line 12 we have a **print** statement which will output output the value of the variable 'years_of_service' with the correct bonus amount.

Python does not require an **else** block at the end of an **if_elif** chain. Sometimes using an **else** block is useful and sometimes it is not. It may be easier and make for more readable code to use an additional **elif** statement that will catch the necessary condition. Let's take a look at what I mean:

In [ ]:
```
years_of_service = 20

if years_of_service <= 1:
    bonus = 50
elif years_of_service <= 3:
    bonus = 100
elif years_of_service <= 5:
    bonus = 150
```

```
elif years_of_service > 5:
    bonus = 200

print(f"For {years_of_service} years of service, your bonus is ${bonus}.")
```

Now at line 9 any employee with 'years_of_service' greater than 5 will be awarded a bonus of 200.

When possible it is better to use an **elif** block in your code rather than the **else** block. Why? The **elif** block will match a specific condition in your code, in this case greater then 5. Whereas the **else** block is a catch-all and can lead to unexpected results.

Testing multiple conditions

The examples we have just seen are very helpful, and powerful and will be appropriate for many situations when you need just one test to pass.

But what happens when you need your program to check **all** conditions? When you require this, you should use a series of **if** statements with no **elif** or **else** blocks. This technique is used when more then one condition could be **True** and you need to act on every condition that is **True**. Let's take a look:

In [40]:
```
car_extras = ['heated seats', 'parking sensor', 'leather seats']

if 'heated seats' in car_extras:
    print("Adding heated seats to car order.")
if 'parking sensor' in car_extras:
    print("Adding parking sensor to car order.")
if 'leather seats' in car_extras:
    print("Adding leather seats to car order.")

print("\nCar order is ready for review by customer.")
```

```
Adding heated seats to car order.
Adding parking sensor to car order.
Adding leather seats to car order.

Car order is ready for review by customer.
```

In this code example we have a list at line 1, which contains additional extras requested by a customer who is purchasing a car. The **if** statement at line 3 checks to see whether the customer added heated seats to their car order. If they did a confirmation message is printed out. The test for parking sensor at line 5 is an **if** statement, obviously not an **elif** or **else** statement. This test will run whether the previous test passed or not. The test at line 7 checks whether leather seats were requested by the customer, this check is performed no matter the outcome of the previous two tests. So, we have three independent tests which are executed every time the program is run.

This code would not work correctly or produce the expected results if we used **elif** or **else** statements in our code.

If you want only one block of code to run, use an **if-elif-else** chain. If more than one block of code needs to be run, use a series of independent **if** statements.

Exercises

- Write an **if** statement to check if it is going to rain. If it is then print the message 'Don't forget your umbrella'.
- What happens when this program fails? can you modify it so that even if the test for rain fails a message is printed?
- Write a program that provides a variable with the assigned value of the current day of the week. For example, when you are doing this exercise if it is Saturday then assign the value of 'saturday' to the variable 'day_of_week'
  - If the day of the week is Monday, Tuesday, Wednesday, Thursday, Friday, output a message to say it's a weekday
  - If the day of the week is Saturday or Sunday, output a message to say it's the weekend
  - Use an **if_elif_else** chain to complete this exercise
  - Can you improve on this program?

In [43]:
```
day_of_week = 'Thursday'

if day_of_week == 'saturday':
    print('It\'s the weekend.')
elif day_of_week == 'sunday':
    print('It\'s the weekend.')
else:
    print('It\'s a weekday.')
```

```
It's a weekday.
```

Using if statements with lists

When we start to combine **if** statements with **lists** we give our programs more options and power. We can monitor for specific values, we can manage changing conditions and much more.

Checking for special items

For the next example let's imagine a calendar application that manages our days of the week and our schedule. Whenever an item is added to your calendar by a colleague a message appears. Let's take a look:

In [44]:
```
days_of_week = ['monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'sat

for day in days_of_week:
    print(f"A new meeting has been added for {day.title()}.")

print("Here is your schedule")
```

```
A new meeting has been added for Monday.
A new meeting has been added for Tuesday.
A new meeting has been added for Wednesday.
A new meeting has been added for Thursday.
A new meeting has been added for Friday.
A new meeting has been added for Saturday.
A new meeting has been added for Sunday.
Here is your schedule
```

Simple enough, but we don't want to be working on the weekends. How Might we change this example to make sure don't work on either Saturday or Sunday? We could simply remove Saturday and Sunday from the list but then this would not make for a very good calendar application if we could only schedule items for weekdays. Let's take a look:

In [45]:
```
days_of_week = ['monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'sat

for day in days_of_week:
    if day == 'saturday' or day == 'sunday':
        print("Sorry Weekends are not avaiable for appointments.")
    else:
        print(f"A new meeting has been added for {day.title()}.")
```

```
A new meeting has been added for Monday.
A new meeting has been added for Tuesday.
A new meeting has been added for Wednesday.
A new meeting has been added for Thursday.
A new meeting has been added for Friday.
Sorry Weekends are not avaiable for appointments.
Sorry Weekends are not avaiable for appointments.
```

This is a nice, powerful piece of code so let's step through it:

- As before, at line 1 we have a list containing the days of the week
- At line 3 we have a **for** loop which iterates over the list **days_of**week **and assigns each item of the list to the variable** day__
- At line 4 we check to see if the variable **day** is equal to Saturday of Sunday and if it is print out a message.
- At line 6 the **else** block continues adding meeting to our schedule

There is a lot more that can be added and improved in this program. Why don't you experiment with it a little?

Checking that a list is not empty

So far every list that we have worked with has contained items in it. But what happens when a list is empty and why would we have an empty list? Imagine on your website you have the option allow users to sign-up to your newsletter. In the beginning the list of subscribers would be empty but would become populated at visitors to your website subscribed. Let's take a look:

In [46]:
```
email_subscribers = []

if email_subscribers:
    for subscriber in email_subscribers:
        print(f"{subscriber}, thank you for subscribing to my newsletter.")
else:
    print("No subscribers yet.")
```

```
No subscribers yet.
```

In this example, we start out with an empty list at line 1. At line 3 we have an **if** statement. In Python, when the name of a list is used in an **if** statement, it returns **True** if

the list contains at least one item and returns **False** if the list is empty.

If **email_subscribers** passes the conditional test the **for** loop at line 5 is executed. If the conditional test fails a message is printed telling us that we have no subscribers.

The list is empty which means the conditional test at line 3 fails, the **for** loop at line 4 is not executed and the **print** statement at line 7 is run.

If the list were not empty, subscribers to our email list would be printed to the screen. If you want go ahead and add some email addresses to the list 'email_subscribers'.

Using multiple lists

In our previous calendar example the code was a bit clunky. Let's see how we can improve that example by using multiple lists.

In [48]:
```
available_days = ['monday', 'tuesday', 'thursday']
requested_days = ['saturday', 'sunday', 'wednesday', 'monday']

for day in requested_days:
    if day in available_days:
        print(f"{day.title()} is open for appointments.")
    else:
        print(f"Sorry, {day.title()} is not available for appointments.")
print("\nThank you")
```

```
Sorry, Saturday is not available for appointments.
Sorry, Sunday is not available for appointments.
Sorry, Wednesday is not available for appointments.
Monday is open for appointments.

Thank you
```

In this example we define a list of days that available to make appointments at line 1. It's worth pointing out that if list of days never changed then you could define this as a tuple rather than a list.

Next, we define a list of days that someone has requested to make an appointment on.

At line 4 we loop through the list of requested days. Inside the loop, we first check to see if each requested day is in the list of available days, this is done at line 5. If it is, we make this day available for an appointment to the user. If the requested day is not in the list of available days, the **else** block at line 7 is executed. This block prints a message informing the user that the requested day is unavailable for appointments. Phew!

Exercises

- Make a list of the days of the week. Write a piece of code that will print a greeting that includes the day of the week. Loop through the list and print another greeting that says 'tomorrow is...'
- If the day of the week is a Saturday or Sunday, print a message that tells the users it's the weekend
- Otherwise print a greeting that says it's a weekday

- To the program you have just created, add an **if** test to make sure the list of days of the week is not empty
- If the list is empty print a message
- Remove all the elements from the list and make sure that the correct message is printed out
- Make a list of days of the week that you have been requested for appointments, meetings or lectures
- Make another list of days during the week that you are unavailable for appointments, be sure that days of the week from this list are also in your list of requested days
- Loop through the unavailable list to see if any element is included in the requested list. If it has, print a message informing the user that you are unavailable for appointments on that day. If a day of the week is available print a message to the user letting them know.
- Make sure your lists are case sensitive. 'Monday' and 'monday' should be the same day of the week. A customer should not be able to book an appointment on 'Monday' and 'monday'.