If Statements

University of Wyoming COSC 1010

Adapted from: *Python Crash Course 3rd Ed* By Eric Matthes

If Statements

- Programming often involves evaluating a set of conditions
- And then decide what action to take as a result
- Python's `if` statements allow you to examine the current state of a program and respond appropriately

If Statements

- Today we will learn how to:
  - Write conditional tests, letting you check any condition
  - Write simple `if` statements
  - Create more complex statements
  - Apply this concept to lists, writing a `for` loop to handle most items in a list

Starting Simple

- We are going tos tart off with a test that lets you respond to situations correctly
- Imagine you have a list of cities
- And you want to print each car with proper casing
- City names are proper nouns, so should begin with a capital
- But what if you want only a specific city in all upper?

In [ ]:
```
cities = ["laramie","casper","jackson hole","cheyenne"]

for city in cities:
    if city == "laramie":
        print(city.upper())
    else:
        print(city.title())
```

```
LARAMIE
Casper
Jackson Hole
Cheyenne
```

Starting Simple

- That code follows a loop through the list
- It looks for the value `'laramie'` , printing in upper case when found
- All other cases are printed in lowercase
- This combines many of the things we will talk about

Conditional Tests

- At the heart of every `if` statement is an expression that can be evaluated
- These expressions evaluate to either `True` or `False`
- Python uses these to determine whether the code in an `if` should be executed

Conditional Tests

- If the expression evaluates to `True` Python executes the code
- If it evaluates to `False` then Python ignores the code within the if statement

Checking for Equality

- Most conditional tests compare the current value of a variable to a value of interest
- the simplest conditions check whether the value of a variable is equal to the value of interest
- Comparisons are done with the double equals `==` operator
- You need to use `==` when doing equality comparisons, as `=` is assignment

In [ ]:
```
#Explained in the next cell
city = "Laramie"
print(city=="Laramie")
```

```
True
```

Checking for Equality

- The first line assigns `'Laramie'` to the variable `city`

    - This is done using `=`
- The following line checks whether the value of city is equivalent to `'Laramie'`
    - Using `==`
- The *equality operator* returns `True` if the values on the left and right side match
- `False` if they don't

Checking for Equality

- A single `=` is really a *statement*, you might read the first line as:
    - "Set the value of city equal to 'Laramie'
- On the other hand a double equals asks a question
    - "Is the value of city equal to 'Laramie'"

In [ ]:
```
city = "Laramie"
print(city=="Cheyenne")
print(city=="laramie")
```

```
False
False
```

Ignoring Case when Checking Equality

- Testing for equality is case sensitive in Python
- Two values with different capitalization
- If case matters this behavior is advantageous
- If case doesn't matter it can be inconvenient

Ignoring Case when Checking Equality

- So what do we do if case doesn't matter and we want to compare elements?

In [ ]:
```
city = 'Laramie'
print(city.upper()=='laramie'.upper())
```

```
True
```

Ignoring Case when Checking Equality

- that test will return true no matter how `Laramie` is formatted
- We are setting the value in the variable adn the value we are testing against to be the same case

- `.upper()` doesn't change the value, just the casing

Ignoring Case when Checking Equality

- Websites enforce certain rules for the data that users enter
- It is done in a similar way to this
- It could be to ensure your email is correct, regardless of the capitalization you entered
- or to ensure all usernames are truly unique

Checking for Inequality

- When you want to determine if two values are not equal the *inequality operator* can be used
  - `!=`
- The syntax to using this is the same as it is for equality
- But, this will return `True` when things are not equal and `False` if they are equal

In [ ]:
```
city = "Laramie"
city_two = "Cheyenne"

print(city == city_two)
if city == city_two:
    print(f"{city} is equal to {city_two}")

print(city != city_two)
if city != city_two:
    print(f"{city} is not equal to {city_two}")
```

```
False
True
Laramie is not equal to Cheyenne
```

Checking for Inequality

- That code compares the values in the variables `city` and `city_two`
- If the two variables match Python returns `False` when using `!=`
- IF the two variables do not match `!=` returns `True`
- Many comparisons will check for equality, but inequality can also be useful

Numerical Comparisons

- Testing numerical values is straight forward

- Numerical comparisons can use both `==` and `!=`
- As well as other operators

In [ ]:
```
age_to_drive = 16

print(age_to_drive == 16)
print(age_to_drive == 15)
```

```
True
False
```

In [ ]:
```
age_to_drive = 16

print(age_to_drive != 16)
print(age_to_drive != 15)
```

```
False
True
```

Numerical Comparisons

- Many mathematical comparisons can also be included in conditional statements
  - less than `<`
  - greater than `>`
  - less than or equal to `<=`
  - greater than or equal to `>=`
- Each mathematical comparison can be used within an if statement

In [ ]:
```
age_to_drive = 16
print(age_to_drive < 16)
print(age_to_drive > 16)
print(age_to_drive <= 16)
print(age_to_drive >= 16)
```

```
False
False
True
True
```

Checking Multiple Conditions

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

Using `and` to Check Multiple Conditions

- Sometimes you may need to check if two statements are both `True` simultaneously
- To do so the `and` keyword is used to combine the two conditionals
- If each passes the overall expression evaluates to `True`
- If either test fails, or both, the expression evaluates to `False`

In [ ]:
```
age_zero = 22
age_one = 20
print(age_zero >= 21)
print(age_one >= 21)
print(age_zero >= 21 and age_one >= 21)
```

```
True
False
False
```

Using `and` to Check Multiple Conditions

- To improve readability you can use parenthesis
  - `((age_zero >= 21) and (age_one >= 21))`
- They aren't required but it makes it more clear what is going on

Using `or` to Check Multiple Conditions

- The `or` keyword allows you to check multiple conditions like `and`
- But, this condition passes when one or both of the conditions are true
- An `or` expression only fails when both statements are false

In [ ]:
```
age_zero = 22
age_one = 20
print(age_zero >= 21)
print(age_one >= 21)
print(age_zero >= 21 or age_one >= 21)
```

```
True
False
True
```

Checking Whether a Value is in a List

- Sometimes it is important to check whether a list contains a value
- You may want to check to see if a username already exists in a list
- To determine if a value is in a list `in` can be used

- Thinking back to our list of cities, what if we want to add new cities only if they aren't already there

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

if "Laramie" in cities:
    print("Laramie already in the list")
```

```
Laramie already in the list
```

Checking Whether a Value is in a List

- The keyword `in` tells Python to check for the existence of `Laramie` in the cities
- This technique is very powerful
- It allows you to avoid adding duplicates to your list
- this can also be used on strings to see if a *substring* exists within

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

if " " in cities[0]:
    print(f"There is a space in {cities[0]}")

if " " in cities[2]:
    print(f"There is a space in {cities[2]}")
```

```
There is a space in Jackson Hole
```

Checking Whether a Value is Not in a List

- Other times you may want to check to see if an element is *not* in a list
- Suppose you have a list of dis-allowed words
- You can ensure a word is not in the list before allowing it to be posted

In [ ]:
```
banned_phrases = ["Go Rams","Boise is a state"]

if "Go Pokes" not in banned_phrases:
    print("Go Pokes")
```

```
Go Pokes
```

Checking Whether a Value is Not in a List

- The if statement should be pretty clear `if "Go Pokes" not in banned_phrases`
- Python returns true if that statement is not in the list

Boolean Expressions

- As you learn more about programming you will here the term *boolean expression*
- This is another name for a conditional test
- A *boolean value* is either `True` or `False`, just like the value of a conditional expression after evaluation
- Boolean values are often used to keep track of certain conditions
- Boolean values provide an efficient way to track the state of a program

`if` Statements

- When you understand conditional tests, you can start writing `if` statements
- Several kinds of `if` statements exist
- Your choice of which to use depends on the number of conditions needed to test
- We've already seen if statements, but now it is time for a deeper dive

Simple `if` Statements

- The simplest kinds of `if` statements only have one test and one action
- You can put any conditional test in the first line
- And just about any action in the indented block following the test
- If the conditional test evaluates to `True` the code block is executed
- If the conditional executes to `False` the code is ignored

In [ ]:
```
if True:
    print('This will print')

if False:
    print('This not will print')

if not True:
    print('This not will print')

if not False:
    print('This will print')
```

```
This will print
This will print
```

In [ ]:
```
age = 19

if age >= 18:
    print("you can vote!")
```

```
you can vote!
```

Simple `if` Statements

- Indentation plays the role role in `if` statements as it does in `for` loops
- All indented lines after an `if` statement will be executed if the test passes
- The entire block of indented items will be ignored if the test fails
- You can have as many lines of code within the block as you need

`if-else` Statements

- Often you may want to take one action when a condition passes, or another if it fails
- Python's `if-else` syntax makes this possible
- An `if-else` block is similar to a simple `if` statement
- The else allows you to define an action if the conditional test fails

In [ ]:
```
age = 17

if age >= 18:
    print("You are old enough to vote!")
else:
    print("You can't yet vote")
```

```
You can't yet vote
```

`if-else` Statements

- If the condition passes the code in the first block runs
- If the test evaluates to `False` the `else` block is executed
- In the previous example the age was less than 18, so the `else` triggered
- This code works because there are only two possible situations to evaluate
- Python will always execute one of the two cases

The `if-elif-else` Chain

- Often times you will need to test more than two situations
- To evaluate these Python's `if-elif-else` syntax can be used
- Python will execute only one block in the chain
- It runs each conditional test in order, until one passes

- When a test passes Python runs the code in its corresponding block, skipping the rest

The `if-elif-else` Chain

- Many real world situations involve multiple options
- Consider a movie ticket that has a varying price based on age
  - Attendees under the age of four could be $2.00
  - 5-64 could be $12.00
  - And finally seniors 65 and over could be $7.00

In [ ]:
```
age = 12
if age < 4:
    price = 2
elif age < 65:
    price = 12
else:
    price = 7

print(f"Your cost of admission is ${price}.00")
```

```
Your cost of admission is $12.00
```

The `if-elif-else` Chain

- The indented lines set the value of `price` based on age
- After the price is set a separate `print()` is used to display the price
- In addition to being more efficient, this revised code is easier to modify
- To change the text of the message, you only have to alter one line

Using Multiple `elif` Blocks

- You can use as many `elif` blocks in your code as needed
- Suppose the previous example added a rate for children less than 18
- Most of the code would remain unchanged
- You would simply add a new `elif` block to it to check

In [ ]:
```
age = 12
if age < 4:
    price = 2
elif age < 18:
    price = 10
elif age < 65:
    price = 12
```

```
else:
    price = 7

print(f"Your cost of admission is ${price}.00")
```

```
Your cost of admission is $10.00
```

Omitting the `else` block

- Python does not require an `else` block at the end of an `if-elif` chain
- Sometimes an `else` is useful
- Other times it may be clearer to use another `elif`
- Or in yet another case you may simply not want a default option

In [ ]:
```
age = 12
if age < 4:
    price = 2
elif age < 18:
    price = 10
elif age < 65:
    price = 12
elif age >= 65:
    price = 7

print(f"Your cost of admission is ${price}.00")
```

```
Your cost of admission is $10.00
```

Omitting the `else` block

- The code shown is functionally the same as the previous code
- But now, every code block has to pass a condition before being run
- The `else` block is a catch all statement, matching any condition that didn't match one of the checks
- If you have a specific final condition use an `elif`
- Otherwise, use an `else` if you want a catch all

Testing Multiple Conditions

- The `if-elif-else` chain is powerful, but it is only appropriate when you need one test to pass
- As soon as one passing test is found, the rest are skipped
- This behavior is beneficial because it's efficient and allows you to test for a specific condition

- Sometimes it's important to check all conditions of interest
- In this case a series of `if` statements can be used
- This technique makes sense when more than one conditional could be `True`

In [ ]:
```
cities = ["laramie","casper","jackson hole","cheyenne"]

if 'laramie' in cities:
    print("Laramie is in Wyoming")
if 'cheyenne' in cities:
    print("Cheyenne is in Wyoming")
```

```
Laramie is in Wyoming
Cheyenne is in Wyoming
```

Testing Multiple Conditions

- We start with a list of cities
- The first if statement checks to see if "laramie" is in the list
- The second checks for "cheyenne"
- As these aren't `if-elif` statements both ifs can be true and execute their block of code
- If you want only block to execute then you should use `if-elif-else`
- If multiple blocks can execute then `if-if-if` can be used

Using `if` Statements with Lists

- When coming lists and `if` statements you can do some powerful things
- You can watch for special values that need to be treated differently than other values
- You can efficiently manage changing conditions
- You can also begin to prove that your code works as you expect!

Checking for Special Items

- Now you (hopefully) have a basic understanding of conditional tests and `if` statements
- That knowledge can be applied to look for special items in a list
- We'll work with our cities list once again to see this in action
- How could we make a check that prints all cities with an `a` n them?

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

for city in cities:
```

```
    if 'a' in city.lower():
        print(f"{city} has an 'a' in it!")
```

```
Laramie has an 'a' in it!
Casper has an 'a' in it!
Jackson Hole has an 'a' in it!
```

Checking for Special Items

- Alternatively you can check for individual items in the list
- This can be done with the equality operator ==
- Just remember when checking strings you need to ensure the casing is the same

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

for city in cities:
    if city.lower() == 'laramie':
        print(f"{city} is the home of the University of Wyoming")
```

```
Laramie is the home of the University of Wyoming
```

Checking That a List is Not Empty

- Thus far our code has made a simple assumption
  - That the lists have at least one item
- Remember some lists will start empty and will be built via user input
- So, you can't always assume a list will have elements

In [ ]:
```
user_home_cities = []

if user_home_cities:
    for city in cities:
        print(city)
else:
    print("No home cities")
```

```
No home cities
```

Checking That a List is Not Empty

- There we start with an empty list
- Prior to doing the loop we check to ensure there are elements in the list
- And if there aren't we print out a message to that end
- Can you think of another way to check if a list is empty?

In [ ]:
```
user_home_cities = []
```

```
if len(user_home_cities) == 0:
    print("No cities stored")
else:
    print(f"There are {len(user_home_cities)} stored")
```

```
No cities stored
```

Using Multiple Lists

- Sometimes you will need to have multiple lists
- And make comparisons between the two
- Suppose you work at A Pizzeria and a customer is trying to add toppings to their Pizza
- You would have a pre-defined list of allowable toppings
- And a list of their requested toppings

In [ ]:
```
avail_toppings = ['pepperoni', 'extra cheese', 'green peppers', 'bacon']

requested_toppings = ['extra cheese', 'green onions', 'pepperoni']

for req_top in requested_toppings:
    if req_top in avail_toppings:
        print(f"Adding {req_top}")
    else:
        print(f"{req_top}  not available")
```

```
Adding extra cheese
green onions  not available
Adding pepperoni
```

Using Multiple Lists

- We defined two lists
  - One for toppings that are available (could also be a tuple)
  - One for the requested toppings
- Then in the loop we checked to verify if each requested topping is available