Data Types Recap

In Python, text in between quotes -- either single or double quotes -- is a string data type. An integer is a whole number, without a fraction, while a float is a real number that can contain a fractional part. For example, 1, 7, 342 are all integers, while 5.3, 3.14159 and 6.0 are all floats. When attempting to mix incompatible data types, you may encounter a **TypeError**. You can always check the data type of something using the *type()* function.

Implicit vs Explicit Conversion

As we saw earlier in the video, some data types can be mixed and matched due to implicit conversion. Implicit conversion is where the interpreter helps us out and automatically converts one data type into another, without having to explicitly tell it to do so.

By contrast, explicit conversion is where we manually convert from one data type to another by calling the relevant function for the data type we want to convert to. We used this in our video example when we wanted to print a number alongside some text. Before we could do that, we needed to call the *str()* function to convert the number into a string. Once the number was explicitly converted to a string, we could join it with the rest of our textual string and print the result.

Study Guide: Expressions and Variables

This study guide provides a quick-reference summary of what you learned in this lesson and serves as a guide for the upcoming practice quiz.

In the Expressions and Variables segment, you learned about expressions, variables, and the data types: string, integer, and float. You learned how to convert a value from one data type to another and you learned how to resolve a few common errors in Python.

Terms

- **expression** - a combination of numbers, symbols, or other values that produce a result when evaluated
- **data types** - classes of data (e.g., string, int, float, Boolean, etc.), which include the properties and behaviors of instances of the data type (variables)
- **variable** - an instance of a data type class, represented by a unique name within the code, that stores changeable values of the specific data type
- **implicit conversion** - when the Python interpreter automatically converts one data type to another
- **explicit conversion** - when code is written to manually convert one data type to another using a data type conversion function:
    - **str()** - converts a value (often numeric) to a **string** data type
    - **int()** - converts a value (usually a float) to an **integer** data type
    - **float()** - converts a value (usually an integer) to a **float** data type

Coding skills

Skill Group 1

- Use the assignment operator = to assign values to variables
- Use basic arithmetic operators with variables to create expressions
- Use explicit conversion to change a data type from float to string

```
# The following lines assign the variable to the left of the =
# assignment operator with the values and arithmetic expressions
# on the right side of the = assignment operator.
hotel_room = 100
tax = hotel_room * 0.08
```

```
total = hotel_room + tax
room_guests = 4
share_per_person = total/room_guests

# This line outputs the result of the final calculation stored
# in the variable "share_per_person"
print("Each person needs to pay: " + str(share_per_person)) # change a data type
```

Skill Group 2

- Output multiple string variables on a single line to form a sentence
- Use the plus (+) connector or a comma to connect strings in a print() function
- Create spaces between variables in a print() function

```
# The following 5 lines assign strings to a list of variables.
salutation = "Dr."
first_name = "Prisha"
middle_name = "Jai"
last_name = "Agarwal"
suffix = "Ph.D."

print(salutation + " " + first_name + " " + middle_name + " " + last_name + ", " + suffix)

# The comma as a string ", " adds the conventional use of a comma plus a
# space to separate the last name from the suffix.

# Alternatively, you could use commas in place of the + connector:
print(salutation, first_name, middle_name, last_name,",", suffix)
# However, you will find that this produces a space before a comma within a string.
```

Skill Group 3

- Resolve TypeError caused by a data type mismatch issue
- Use an explicit conversion function

```
print("5 * 3 = " + (5*3))

# Resolution:
# print("5 * 3 = " + str(5*3))
#
# To avoid a type error between the string and the integer within the
# print() function, you can make an explicit data type conversion by
# using the str() function to convert the integer to a string.
```

Skill Group 4

- Resolve a ZeroDivisionError caused by an attempt to divide by 0

```
numerator = 7
denominator = 0   # Possible resolution: Change the denominator value
result = numerator / denominator
```

```
print(result)

# One possible assumption for a number divided by zero error might
# include the issue of a null value as a denominator (could happen when
# using a loop to iterate over values in a database). In such cases, the
# desired outcome may be to leave the numerator value intact. The
# numerator value can be preserved by reassigning the denominator with
# the integer value of 1. The result would then equal the numerator.
```

Python practice information

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket

Defining Functions Recap

We looked at a few examples of built-in functions in Python, but being able to define your own functions is incredibly powerful. We start a function definition with the def keyword, followed by the name we want to give our function. After the name, we have the parameters, also called arguments, for the function enclosed in parentheses. A function can have no parameters, or it can have multiple parameters. Parameters allow us to call a function and pass it data, with the data being available inside the function as variables with the same name as the parameters. Lastly, we put a colon at the end of the line.

After the colon, the function body starts. It's important to note that in Python the function body is delimited by indentation. This means that all code indented to the right following a function definition is part of the function body. The first line that's no longer indented is the boundary of the function body. It's up to you how many spaces you use when indenting -- just make sure to be consistent. So if you choose to indent with four spaces, you need to use four spaces everywhere in your code.

Returning Values Using Functions

Sometimes we don't want a function to simply run and finish. We may want a function to manipulate data we passed it and then return the result to us. This is where the concept of return values comes in handy. We use the return keyword in a function, which tells the function to pass data back. When we call the function, we can store the returned value in a variable. Return values allow our functions to be more flexible and powerful, so they can be reused and called multiple times.

Functions can even return multiple values. Just don't forget to store all returned values in variables! You could also have a function return nothing, in which case the function simply exits.

Study Guide: Functions

This study guide provides a quick-reference summary of what you learned in this lesson and serves as a guide for the upcoming practice quiz.

In the Functions segment, you learned how to define and call functions, utilize a function's parameters, and return data from a function. You also learned how to differentiate and convert between different data types utilizing variables. Plus, you learned a few best practices for writing reusable and readable code.

Terms

- **return value** - the value or variable returned as the end result of a function
- **parameter (argument)** - a value passed into a function for use within the function
- **refactoring code** - a process to restructure code without changing functionality

Knowledge

- The purpose of the **def()** keyword is to define a new function.
- Best practices for writing code that is readable and reusable:
    - **Create a reusable function** - Replace duplicate code with one reusable function to make the code easier to read and repurpose.
    - **Refactor code** - Update code so that it is self-documenting and the intent of the code is clear.
    - **Add comments** - Adding comments is part of creating self-documenting code. Using comments allows you to leave notes to yourself and/or other programmers to make the purpose of the code clear.

Coding skills

Skill Group 1

- Use a function that accepts multiple parameters
- Return a result value

```
# This function calculates the number of days in a variable number of
# years, months, and days. These variables are provided by the user and
# are passed to the function through the function's parameters.
def find_total_days(years, months, days):
# Assign a variable to hold the calculations for the number of days in
# a year (years*365) plus the number of days in a month (months*30) plus
# the number of days provided through the "days" parameter variable.
    my_days = (years*365) + (months*30) + days
# Use the "return" keyword to send the result of the "my_days"
# calculation to the function call.
    return my_days

# Function call with user provided parameter values.
print(find_total_days(2,5,23))
```

Skill Group 2

- Use a function to return the result of a measurement conversion
- Use arithmetic operators to perform a calculation
- Combine text with a function call within a print() statement
- Convert the return value from a float data type to a string for the print() function
- Call the function and perform a calculation on the return value within a print() statement

```
# This function converts fluid ounces to milliliters and returns the
# result of the conversion.
def convert_volume(fluid_ounce):
# Calculate value of the "ml" variable using the parameter variable
# "fluid_ounce". There are approximately 29.5 milliliters in 1 fluid
```

```
# ounce.
    ml = fluid_ounce * 29.5
# Return the result of the calculation.
    return ml

# Call the conversion from within the print() function using 2 fluid
# ounces. Convert the return value from a float to a string.
print("The volume in millimeters is " + str(convert_volume(2)))

# Call the function again and double the 2 fluid ounces from within
# the print function.
print("The volume in millimeters is " + str(convert_volume(2)*2))
# Alternative calculation:
# print("The volume in millimeters is " + str(convert_volume(4))
```

Python practice information

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket

Comparison Operators with Equations

The following examples demonstrate how to use comparison operators with the data types **int** (integers, whole numbers) and **float** (number with a decimal point or fractional value). Comparison operators return Boolean results. As you learned previously, Boolean is a data type that can hold only one of two values: **True** or **False**.

The comparison operators include:

- **==** (equality)
- **!=** (not equal to)
- **>** (greater than)
- **<** (less than)
- **>=** (greater than or equal to)
- **<=** (less than or equal to)

PART 1: Equality == and Not Equal To != Operators

In Python, you can use comparison operators to compare values. When a comparison is made, Python returns a Boolean result: **True** or **False**. Note that Boolean data types are not string data types (Boolean **True** is not equal to the string "True").

- To check if two values are the same, use the **equality operator**: **==**
- To check if two values are not the same, use the **not equal to operator**: **!=**

The print() function can be used to display the results of the comparisons.

Examples:

```
print(32 == 30+2)   # The == operator checks if the 2 values are
True                # equal to each other. If they are equal,
                    # Python returns a True result.


print(5+10 == 6+7)  # If the two values are not equal, as in the
False               # expression 5+10 == 6+7 (or 15 == 13), Python
                    # returns a False result.


print(10-4 != 10+4) # The != operator checks if the 2 values are
True                # NOT equal to each other. If true, Python
                    # returns a True result.


print(9/3 != 3*1)   # In this last example, 9/3 != 3*1 (or 3 != 3)
False               # is false. So, Python returns a False value.
```

The equality == operator versus the equals = operator

It is important to note that the equality == comparison operator performs a different task than the equals = assignment operator. The equals = operator assigns the value on the right side of the equals = to the object (e.g., a variable) on the left side of the equals = operator.

Examples:

```
# The = equals assignment operator is used to assign a value to a
# variable.

my_variable = 3*5           # Assigns a value to my_variable
print(my_variable)          # Printing the variable returns the
15                          # value assigned to the variable.



# The == equality comparison operator checks if the values of the two
# expressions on either side of the == operator are equivalent to one
# another.

print(my_variable == 3*5)   # Printing the variable returns a Boolean
True                        # True or False result.
```

PART 2: Greater Than > and Less Than < Operators

The comparison operators greater than > and less than < also return a **True** or **False** Boolean result after comparing two values.

- To check if one value is larger than another value, use the greater than operator: >
- To check if one value is smaller than another value, use the less than operator: <

Examples:

```
print(11 > 3*3)         # The > operator checks if the left value is
True                    # greater than the right value. If true, it
                        # returns a True result.


print(4/2 > 8-4)        # If the > operator finds that the left value
False                   # is NOT greater than the right value, the
                        # comparison will return a False result.


print(4/2 < 8-4)        # The < operator checks  if the left value is
True                    # less than the right side. If true, the
                        # comparison returns a True result.


print(11 < 3*3)         # If the < operator finds that the left side is False

                        # NOT less than the right value, Python returns
False                   # a False result.
```

PART 3: Greater Than or Equal to >= and Less Than or Equal to <= Operators

Like the other comparison operators, the greater than or equal to **>=** and less than or equal to **<=** operators return a **True** or **False** Boolean result when a comparison is made.

- To check if one value is larger than or equal to another value, use the greater than or equal to operator: **>=**
- To check if one value is smaller than or equal to another value, use the less than or equal to operator: **<=**

Examples:

```
print(15 <= 18/2)   # If the <= comparison determines that the left
False               # value is NOT less than or equal to the right
                    # value, the comparison returns a False result.
                    # right value. Again, if one of the two
                    # conditions is true, Python returns a True
                    # result.

print(12*2 <= 30)   # The <= operator checks if the left value is
True                # less than or equal to the right value. In
                    # this case, the left value is less than the
print(18/2 >= 15)   # If the >= comparison determines that the left False
False               # value is NOT greater than or equal to the
                    # right, it returns a False result.
                    # If one of these conditions is true,
                    # Python returns a True result. In this case
                    # the two values are equal. So, the comparison
                    # returns a True result.
```

```
print(12*2 >= 24)   # The >= operator checks if the left value is
True                # greater than or equal to the right value.
```

PART 4: Practice

If you would like more practice using comparison operators, feel free to create your own comparisons using the code block below. Note that there is no feedback associated with this code block.

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket

Key takeaways

Python comparison operators return Boolean results: **True** or **False**.

Symbol	Name	Expression	Description
==	Equality operator	a == b	a is equal to b
!=	Not equal to operator	a != b	a is **not** equal to b
>	Greater than operator	a > b	a is larger than b
>=	Greater than or equal to operator	a >= b	a is larger than or equal to b
<	Less than operator	a < b	a is smaller than b
<=	Less than or equal to operator	a <= b	a is smaller than or equal to b

Resources for more information

For more information about the concepts covered in these practice exercises, please visit:

- Order of Operations - A refresher on the mathematical Order of Operations.
- Python Comparison Operators with Syntax and Example - Provides examples of more complex comparisons.
- Raise numbers to a power: here's how to exponentiate in Python - Explains multiple methods for calculating exponents in Python.

Comparison Operators with Strings

In this reading, you will learn more about what comparison operators can and cannot do. If you use the **==** (equality) and **!=** (not equal to) operators with strings, you can check if two strings contain the same text or not. You can also alphabetize strings using **>** (greater than), **<** (less than), **>=** (greater than or equal to), **<=** (less than or equal to) comparison operators. As with numeric data types, comparison operators used with strings will return Boolean (**True**, **False**) results.

PART 1: Equality == and Not Equal to != Operators with Strings

In Python, you can use comparison operators to compare strings. The equality **==** and the not equal to **!=** operators are helpful when you need to search for a specific string in a body of text, a log file, a spreadsheet, a database, and more. You can also check user input strings to compare them to another string. Note that Boolean data types are not string data types (Boolean **True** is not equal to the string "True").

Examples:

```
# The == operator can check if two strings are equal to each other.
# If they are equal, the Python interpreter returns a True result.
print("a string" == "a string")
True


# In this example, the equality == comparison is between "4 + 5" and
# 4 + 5. Since the left data type is a string and the right data type
# is an integer, the two values cannot be equal. So, the comparison
# returns a False result.
print("4 + 5" == 4 + 5)
False


# The != operator can check if the two strings are NOT equal to each
# other. If they are indeed not equal, then Python returns a True result.
print("rabbit" != "frog")
True


# In this example, the variable event_city has been assigned the string
# value "Shanghai". This variable is compared to a static string,
# "Shanghai", using the != operator. As, the strings "Shanghai" and
# "Shanghai" are the same, the comparison of "Shanghai" != "Shanghai"
# is false. Accordingly, Python will return a False result.
event_city = "Shanghai"
print(event_city != "Shanghai")
False

# This last example illustrates the result of trying to compare two
# items of different data types using the equality == operator. The
# two items are not equal, so the comparison returns False.
print("three" == 3)
False
```

PART 2: The Greater Than > and Less Than < Operators

The comparison operators greater than **>** and less than **<** can be used to alphabetize words in Python. The letters of the alphabet have numeric codes in Unicode (also known as ASCII values). The uppercase letters A to Z are represented by the Unicode values 65 to 90. The lowercase letters a to z are represented by the Unicode values 97 to 122.

Uppercase		Uppercase		Lowercase		Lowercase	
Unicode #	Character	Unicode #	Character	Unicode #	Character	Unicode #	Character
65	A	78	N	97	a	110	n
66	B	79	O	98	b	111	o
67	C	80	P	99	c	112	p
68	D	81	Q	100	d	113	q
69	E	82	R	101	e	114	r
70	F	83	S	102	f	115	s
71	G	84	T	103	g	116	t
72	H	85	U	104	h	117	u
73	I	86	V	105	i	118	v
74	J	87	W	106	j	119	w
75	K	88	X	107	k	120	x
76	L	89	Y	108	l	121	y
77	M	90	Z	109	m	122	z

- To check if the first letter(s) of a string have a larger Unicode value (meaning the letter is closer to 122 or lowercase z) than the first letter of another string, use the greater than operator: **>**
- To check if the first letter(s) of a string have a smaller Unicode value (meaning the letter is closer to 65 or uppercase A) than the first letter of another string, use the less than operator: **<**

Like numeric comparisons with the greater than **>** and less than **<** operators, comparisons between strings also return Boolean **True** or **False** results.

Examples:

```
# cycle through each letter of each string, from left to right until it
# finds two letters that have different Unicode values. In this example,
# both strings share the initial substring "sun", but then have
# different letters with different Unicode values in the fourth place
# in each string. So, the fourth letters 'b' and 't' of the two
# strings are used for the comparison. Since 'b' does not have a higher
# Unicode value than 't', the comparison returns a False result.
print("sunbathe" > "suntan")
False


# If two identical strings are compared using the less than < comparison
# operator, this will produce a False result because they are equal.
print("Lima" < "Lima")
False


# This last example illustrates the result of trying to compare two


# If the strings have the same first few letters, the comparison will
# return a True result.
print("Brown" < "brown")
True
# chart above, you can see that all lowercase letters have higher
# Unicode values than uppercase letters. We can see that B has a
```

```
# Unicode value of 66 and b has a Unicode value of 98. This
# comparison is the same as 66 < 98, which is true. So, Python will

# The less than < operator checks if the left string has a lower
# Unicode value than the right string. If you reference the Unicode
print("Wednesday" > "Friday")
True

# returns a True result. Since W has a Unicode value of 87, and you can
# easily calculate that F has a Unicode value of 70, this comparison is
# the same as 87 > 70. As this is true, Python will return a True
# result.
# The greater than > operator checks if the left string has a higher
# Unicode value than the right string. If true, the Python interpreter
```

PART 3: The Greater Than or Equal To >= and Less Than or Equal To <= Operators

The greater than or equal to **>=** and less than or equal to **<=** operators can be used with strings as well. Like the other comparison operators, they will return a **True** or **False** Boolean result when a comparison is made between two strings.

- To check if a string has a larger or equal Unicode value than the first letter(s) of another string, use the greater than or equal to operator: **>=**
- To check if a string has a smaller or equal Unicode value than the first letter(s) of another string, use the less than or equal to operator: **<=**

At this point, you should be familiar with how comparison operators work in Python. Can you determine what the results will be from the comparisons listed below? When you are ready to check your answers, click Run.

1. "my computer" >= "my chair"
2. "Spring" <= "Winter"
3. "pineapple" >= "pineapple"

```
# Use the Unicode chart in Part 2 to determine if the Unicode values of
# the first letters of each string are higher, lower, or equal to one
# another.


var1 = "my computer" >= "my chair"
var2 = "Spring" <= "Winter"
var3 = "pineapple" >= "pineapple"

print("Is \"my computer\" greater than or equal to \"my chair\"? Result: ", var1)
print("Is \"Spring\" less than or equal to \"Winter\"? Result: ", var2)
print("Is \"pineapple\" less than or equal to \"pineapple\"? Result: ", var3)
```

PART 4: Practice

If you would like more practice using the comparison (**==**, **!=**, **>**, **<**, **>=**, **<=**) operators with strings, feel free to create your own comparisons using the code block below. Note that there is no feedback associated with this code block.

For additional Python practice, the following links are for several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket

Key takeaways

Python comparison operators return Boolean results (**True** or **False**) with strings:

Expression	Description
"a" **==** "a"	If string "a" is identical to string "a", returns True. Else, returns False
"a" **!=** "b"	If string "a" is **not** identical to string "b"
"a" **>** "b"	If string "a" has a larger Unicode value than string "b"
"a" **>=** "b"	If the Unicode value for string "a" is greater than or equal to the Unicode value of string "b"
"a" **<** "b"	If string "a" has a smaller Unicode value than string "b"
"a" **<=** "b"	If the Unicode value for string "a" is smaller than or equal to the Unicode value of string "b"

Resources for more information

For more information about the concepts covered in these practice exercises, please visit:

- Python String Comparison: A Step-by-Step Guide (with Examples) - A quick reference guide to using comparison operators with strings. Includes part of a Unicode table that displays all of the Unicode values for both uppercase and lowercase letters.
- Comparing Strings using Python - Provides more advanced examples of using comparison operators with strings.

Logical Operators

Logical operators are used to construct more complex expressions. You can make complex comparisons by joining comparison statements together using the logical operators: **and**, **or**, **not**. Complex comparisons return a Boolean (**True** or **False**) result.

- **and**
  - Both sides of the statement being evaluated must be True for the whole statement to be True.
  - Example: (5 > 1 **and** 5 < 10) = **True**
- **or**
  - If either side of the comparison is True, then the whole statement is True.
  - Example: (color = "blue" **or** color = "green") = **True**
- **not**
  - Inverts the Boolean result of the statement immediately following it. So, if a statement evaluates to True, and we put the not operator in front of it, it would become False.
  - Example: (**not** "A" == "A") = **False**

PART 1: The and Logical Operator

In Python, you can use the logical operator **and** to connect more than one comparison. This type of complex comparison is used to check if two comparison statements are both True or not. You might use the **and** operator when you need to execute a block of code, but only if two different conditions are true. For example, you might want to write a script that automates sending you an emergency alert if a server stops responding *and* there is an unusual increase in employees opening trouble tickets.

Example 1:

The following model demonstrates the use of the **and** logical operator to join comparisons between two mathematical expressions. The description below the example explains the order in which Python will process the line of code.

```
# Example 1

print((6*3 >= 18) and (9+9 <= 36/2))
```

In the example above, the following activities were completed by Python in the following order:

1. Python solves the numerical expressions using the order of operations. **(6*3 >= 18) and (9+9 <= 36/2) becomes (18 >= 18) and (18 <= 18)**
2. Python compares the results of the numerical expressions using the comparison operators (in this case >= and <=). **(18 >= 18) and (18 <= 18) becomes True and True**
3. Python checks if both sides of the logical operator "and" are true. **True and True become True**
4. Python returns a Boolean value: True or False. **The complex comparison returns a True result.**

Example 2:

In this next example, "Nairobi" < "Milan" and "Nairobi" > "Hanoi", the **and** logical operator is connecting two string comparison statements. You learned previously that using the greater than and less than operators on strings will test the alphabetical order (technically Unicode values) of the strings. So, this complex comparison is checking if "Nairobi" is alphabetized before "Milan" (False) AND after "Hanoi" (True).

This comparison returns a False result because both sides of the logical operator are not True. A comparison statement like this might be used to iterate through a list of names to check if they are alphabetized in the correct order.

```
# Example 2

print("Nairobi" < "Milan" and "Nairobi" > "Hanoi")
```

PART 2: The or Logical Operator

The **or** logical operator tests two conditions to determine if at least one side of the **or** logical operator is True. The result of the test can be used to trigger a block of code if at least one condition is present.

Syntax:

```
Expression1 or Expression2
```

Returns Booleans:

Expression1	Expression2	Returns Result
True	True	True
True	False	True
False	True	True
False	False	False

Examples:

```
# True or True returns True
print((15/3 < 2+4) or (0 >= 6-7))
True

# False or True returns True
```

```
print(country == "New York City" or city == "New York City")
True

# True or False returns True
print(16 <= 4**2 or 9**(0.5) != 3)
True

# False or False returns False
print("B_name" > "C_name" or "B_name" < "A_name”)
False
```

PART 3: The not Logical Operator

The **not** logical operator inverts the value of the comparison expression. This is a helpful tool when you want to execute a block of code as long as a certain condition is **not** present.

- If the conditional expression is True, the **not** logical operator can be added to make the expression **not** True (False).
- If the conditional expression is False, the **not** logical operator can be added to make the expression **not** False (True).

Syntax:

```
not expression
```

Example 1:

```
# Test Example 1:

x = 2*3 > 6
print("The value of x is:")
print(x)

print("")  # Prints a blank line

print("The inverse value of x is:")
print(not x)
```

Example 2:

Can you determine the result of the following comparison?

```
# What happens when you negate a False statement?
# Click Run when you are ready to check your answer.


today = "Monday"
print(not today == "Tuesday")


# The "today" variable states today is Monday. This makes the comparison
# "today == Tuesday" False. The logical operator "not" inverts the False
# result to become True. In other words, this expression asks if it is
```

```
# false that today is not Tuesday. More succinctly, "not False" means
# True."
```

PART 4: Practice

If you would like more practice using the logical (**and**, **or**, **not**) operators, feel free to create your own comparisons using the code block below. Note that there is no feedback associated with this code block.

For additional Python practice, the following links are for several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket

Key takeaways

When Python logical operators are used with comparison operators, the interpreter will return Boolean results (**True** or **False**):

Expression	Description
a **==** a **and** a **!=** b	True if both sides are True, otherwise False.
a **>** b **or** a **<=** c	True if either side is True. False if both sides are False.
not a **==** b	True if the statement is False, False if the statement is True.

Resources for more information

For more information about the concepts covered in these practice exercises, please visit:

- Understanding Boolean Logic in Python 3 - A handy guide for reviewing both logical and conditional operators.

---

if Statements Recap

We can use the concept of **branching** to have our code alter its execution sequence depending on the values of variables. We can use an *if* statement to evaluate a comparison. We start with the *if* keyword, followed by our comparison. We end the line with a colon. The body of the *if* statement is then indented to the right. If the comparison is ***True***, the code inside the *if* body is executed. If the comparison evaluates to ***False,*** then the code block is skipped and will not be run.

---

else Statements and the Modulo Operator

We just covered the *if* statement, which executes code if an evaluation is true and skips the code if it's false. But what if we wanted the code to do something different if the evaluation is false? We can do this using the *else* statement. The *else* statement follows an *if* block, and is composed of the keyword *else* followed by a colon. The body of the *else* statement is indented to the right, and will be executed if the above *if* statement doesn't execute.

We also touched on the modulo operator, which is represented by the percent sign: **%**. This operator performs integer division, but only returns the remainder of this division operation. If we're dividing 5 by 2, the quotient is 2, and the remainder is 1. Two 2s can go into 5, leaving 1 left over. So 5%2 would return 1. Dividing 10 by 5 would

give us a quotient of 2 with no remainder, since 5 can go into 10 twice with nothing left over. In this case, 10%5 would return 0, as there is no remainder.

Complex Branching with elif Statements

Building off of the *if* and *else* blocks, which allow us to branch our code depending on the evaluation of one statement, the *elif* statement allows us even more comparisons to perform more complex branching. Very similar to the *if* statements, an *elif* statement starts with the *elif* keyword, followed by a comparison to be evaluated. This is followed by a colon, and then the code block on the next line, indented to the right. An *elif* statement must follow an *if* statement, and will only be evaluated if the *if* statement was evaluated as false. You can include multiple *elif* statements to build complex branching in your code to do all kinds of powerful things!

Study Guide: Conditionals

This study guide provides a quick-reference summary of what you learned in this lesson and serves as a guide for the upcoming practice quiz.

In the Conditionals segment, you learned about the built-in Python operators used for comparing values and the logical operators for making complex comparisons. You also learned how to use operators in if-else-elif blocks.

Knowledge

Comparison operators with numerical values

Comparison expressions return a Boolean result (True or False).

- x **==** y If x is equal to y, return True. Else, return False.
- x **!=** y If x is not equal to y, return True. Else, return False.
- x **<** y If x is less than y, return True. Else, return False.
- x **<=** y If x is less than or equal to y, return True. Else, return False.
- x **>** y If x is greater than y, return True. Else, return False.
- x **>=** y If x is greater or equal to y, return True. Else, return False.

Comparison operators with strings

Comparison expressions with strings also return a Boolean result (True or False).

- "x" **==** "y" If the words are the same, return True. Else, return False.
- "x" **!=** "y" If the words are **not** the same, return True. Else, return False.

When used with strings, the following comparison expressions will alphabetize the strings.

- "x" **<** "y" If string "x" has a smaller Unicode value than string "y", return True. Else, return False.
- "x" **<=** "y" If the Unicode value for string "x" is smaller than or equal to the Unicode value of string "y", return True. Else, return False.
- "x" **>** "y" If string "x" has a larger Unicode value than string "y", return True. Else, return False.
- "x" **>=** "y" If the Unicode value for string "x" is greater than or equal to the Unicode value of string "y", return True. Else, return False.

Unicode values for the alphabet

Uppercase		Uppercase		Lowercase		Lowercase	
Unicode #	Character	Unicode #	Character	Unicode #	Character	Unicode #	Character
65	A	78	N	97	a	110	n
66	B	79	O	98	b	111	o
67	C	80	P	99	c	112	p
68	D	81	Q	100	d	113	q
69	E	82	R	101	e	114	r
70	F	83	S	102	f	115	s
71	G	84	T	103	g	116	t
72	H	85	U	104	h	117	u
73	I	86	V	105	i	118	v
74	J	87	W	106	j	119	w
75	K	88	X	107	k	120	x
76	L	89	Y	108	l	121	y
77	M	90	Z	109	m	122	z

The Unicode numbering for the alphabet starts at 65 for capital letter A and runs to 90 for capital letter Z. Then, the lowercase alphabet values start at 97 for lowercase a and run to 122 for lowercase z. Using these Unicode numbers, capital A's code is less than the codes of all other letters, which Python interprets as the beginning of the alphabet. Lowercase z's code is greater than the codes of all other letters, which Python interprets as the ultimate end of the English alphabet.

Logical operators

Logical operators are used to combine comparison expressions and also return Boolean results (True or False).

- comparison1 **and** comparison2
  - Returns a True result if both comparison1 **and** comparison2 are true.
  - If they are not both true, return False.
- comparison1 **or** comparison2
  - Returns a True result if either comparison1 and/or comparison2 are True.
  - If neither comparison is true, return False.
- **not** comparison1
  - Returns the inverse Boolean value of the comparison.
    - Returns a True result if comparison1 is false.
    - If comparison1 is true, then returns False.

Syntax of an if-elif-else block

```
if condition1:
    action1
elif condition2:
    action2
else:
    action3
```

- If condition1 is True:
  - Then perform action1 and exit if-elif-else block
- If condition2 is True:

    - Then perform action2 and exit if-elif-else block
- If neither condition1 nor condition2 are True:
    - Then perform action3 and exit if-elif-else block

Coding skills

Skill Group 1

- Use a comparison operator with numbers
- Use a comparison operator to alphabetize strings

```
# The value of 10*4 (40) is greater than 14+23 (37), therefore this
# comparison expression will return the Boolean value of True.


print(10*4 > 14+23) # Should print True

# The letter "t" has a Unicode value of 116 and the letter "s" has a
# Unicode value of 115. Since 116 is not less than 115, the
# comparison of "tall" < "short" (or 116 < 115) is False.

print("tall" < "short")  # Should print False
```

Skill Group 2

- Use a function with the def() keyword
- Pass a parameter to the function
- Use an if-elif-else statement
- Assign strings to variables
- Use conditional operators
- Return a value# This function accepts one variable as a parameter

```
def translate_error_code(error_code):

# The if-elif-else block assesses the value of the variable
# passed to the function as a parameter. The if statement uses
# the equality operator == to test the value of the variable.
# This test returns a Boolean (True/False) result.
    if error_code == "401 Unauthorized":
# If the comparison above returns True, then the indented
# line(s) inside the if-statement will run. In this case, the
# action is to assign a string to the translation variable.
# The remainder of the if-elif-else block will not run.
# The Python interpreter will skip to the next line outside of
# the if-elif-else block. In this case, the next line is the
# return value statement.
        translation = "Server received an unauthenticated request"

# If the initial if-statement returns a False result, then the
# first elif-statement will run a different test on the value
# of the variable.
    elif error_code == "404 Not Found":
```

```
# If the first elif-statement returns a True result, then the
# indented line(s) inside the first elif-statement will run.
# After this line, the remainder of the if-elif-else block will
# not run. The Python interpreter will skip to the next line
# outside of the if-elif-else block.
        translation = "Requested web page not found on server"

# If both the initial if-statement and the first elif-statement
# return a False result, then the second elif-statement will
# run.
    elif error_code == "408 Request Timeout":
# If the second elif-statement returns a True result, then the
# indented line(s) inside the second elif-statement will run.
# After this line, the remainder of the if-elif-else block will
# not run. The Python interpreter will skip to the next line
# outside of the if-elif-else block.
        translation = "Server request to close unused connection"

# If the conditional tests above do not produce a True result
```

Skill Group 3

- Use an if-elif-else statement with:
    - comparison operators
    - logical operators

```
# Sets value of the "number" variable
number = 25

# The "number" variable will first be compared to 5. Since it is
# False that "number" is not less than or equal to 5, the expression indented
# under this line will be ignored.
if number <= 5:
   print("The number is 5 or smaller.")

# Next, the "number" variable will be compared to 33. Since it is
# False that "number" is equal to 33, the expression indented under
# this line will be ignored.
elif number == 33:
   print("The number is 33.")

# Then, the "number" variable will be compared to 32 and 6. Since it
# is True that 25 is less than 32 and greater than 6, the Python
# interpreter will print "The number is less than 32 and/or greater
# than 6." Then, it will exit the if-elif-else statement and the remainder
# of the if-elif-else statement will be ignored.
elif number < 32 and number >= 6:
   print("The number is less than 32 and greater than 6.")

else:
   print("The number is " + str(number))
```

```
# Expected output is:
# The number is less than 32 and greater than 6.
```

Skill Group 4

- Use an if statement to calculate a return value
- Use conditional operators
- Recall the arithmetic operators // and %

```
# This function rounds a variable number up to the nearest 10x value
def round_up(number):
  x = 10
# The floor division operator will calculate the integer value of
# "number" divided by x: 35 // 10 will return the integer 3.
  whole_number = number // x
# The modulo operator will calculate the remainder value of "number"
# divided by x: 35 % 10 will return the remainder value 5.
  remainder = number % x
# If the remainder is greater than 0:
  if remainder >= 5:
# Return x multiplied by the (whole_number+1) to round up
    return x*(whole_number+1)
# Else, return x multiplied by the whole_number to round down
  return x*whole_number

# Calls the function with the parameter value of 35.
print(round_up(35)) # Should print 40
```

Python practice information

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket

Study Guide: Module 2 Graded Quiz

It is time to prepare for the Module 2 graded quiz. Please review the following items from this Module before starting the Module 2 Graded Quiz. If you would like to refresh your memory on these materials, please revisit the Study Guides located before each Practice Quiz in Module 2 : Study Guide: Expressions and Variables, Study Guide: Functions, and Study Guide: Conditionals.

Knowledge

- How to assign values to variables and use them in code

- How to construct a function and use function parameters
- How comparison and logical operators can be used,
- How comparison and logical operators behave with different data types
- What type of results simple and complex comparisons produce
- How to alphabetize strings using comparison operators
- What must appear after the **if** and **elif** keywords
- What the **elif** keyword does
- When an **if, elif,** or **else**-statement will execute
- How to use the floor division // and modulo % operators and why
- How to use logical operators with comparison operators to develop complex conditional statements within an **if-elif-else** block
- Best practices for coding and their benefits
- What "self-documenting code" means

There may be a few questions on the quiz that will ask you about *either* the *output* of a small block of code *or the value of part of the code.* Make sure to read the instructions carefully on those questions.

Coding skills

Skill Group 1

- Use a function with the def() keyword
- Pass a parameter to the function
- Use an if-elif-else block to set specific conditions for a variety of actions
- Assign strings to variables
- Use comparison operators
- Return a value
- Call the function in a print statement and pass parameter to the function

```
# A function is created with the def() keyword. The parameter
# variable "time_as_string" is passed to the function through a
# call to the function.
def task_reminder(time_as_string):

    # The following if-elif-else block assigns various strings to
    # the variable "task" depending on specific conditions. The
    # test conditions are set using the == equality comparison
    # operator. In this case, the time passed through the
    # "time_as_string" parameter variable is tested as the
    # specific condition. So, if the time  is "11:30 a.m.", then
    # "task" is assigned the value: "Run TPS report".
    if time_as_string == "8:00 a.m.":
        task = "Check overnight backup images"
    elif time_as_string == "11:30 a.m.":
        task = "Run TPS report"
    elif time_as_string == "5:30 p.m.":
        task = "Reboot servers"
    # The else statement is a catchall for all other values of
    # the "time_as_string" parameter variable not listed in the
    # if-elif block of code.
    else:
        task = "Provide IT Support to employees"

    # This line returns the value of "task" to the function call.
    return task
```

```
# This line calls the function and passes a parameter
# ("10:00 a.m.") to the function.
print(task_reminder("10:00 a.m."))
# Should print "Provide IT Support to employees"
```

Skill Group 2

- Predict the output of expressions written with Python's syntax.
- Requires an understanding of:
  - Arithmetic and logical operators
  - How functions return and print values
  - How if-elif-else statements work
  - Comparison operators

```
# Example 1
# Evaluate the output of this print statement

def product(a, b):
        return(a*b)

print(product(product(2,4), product(3,5)))

##################################

# Example 2
# Evaluate the output of this print statement

def difference(a, b):
        return(a-b)

def sum(a, b):
        return(a+b)

print(difference(sum(2,2), sum(3,3)))


##################################


# Example 3
# Evaluate the Boolean output of this comparison


print((5 >= 2*4) and (5 <= 4*3))


##################################


# Example 4
# Evaluate the value of the comparison in the if statement
```

```
x = 3
```

Skill Group 3

- Create an if-elif-else statement with:
  - a complex conditional statement using both comparison and logical operators
  - values assigned to variables
  - arithmetic operators, including the modulo % operator

```
def get_remainder(x, y):

  if x == 0 or y == 0 or x ==y:
    remainder = 0
  else:
    remainder = (x % y) / y
  return remainder


print(get_remainder(10, 3))
```

Reminder: Correct syntax is critical

Using precise syntax is critical when writing code in any programming language, including Python. Even a small typo can cause a syntax error and the automated Python-coded quiz grader will mark your code as incorrect. This reflects real life coding errors in the sense that a single error in spelling, case, punctuation, etc. can cause your code to fail. Coding problems caused by imprecise syntax will always be an issue whether you are learning a programming language or you are using programming skills on the job. So, it is critical to start the habit of being precise in your code now.

No credit will be given if there are any coding errors on the automated graded quizzes - including minor errors. Fortunately, you have 3 optional retake opportunities on the graded quizzes in this course. Additionally, you have unlimited retakes on practice quizzes and can review the videos and readings as many times as you need to master the concepts in this course.

Now, before starting the graded quiz, please review this list of common syntax errors coders make when writing code.

Common syntax errors:

- Misspellings
- Incorrect indentations
- Missing or incorrect key characters:
  - Parenthetical types - (curved), [square], { curly }
  - Quote types - "straight-double" or 'straight-single', “curly-double” or ‘curly-single’
  - Block introduction characters, like colons - :
- Data type mismatches
- Missing, incorrectly used, or misplaced Python reserved words
- Using the wrong case (uppercase/lowercase) - Python is a case-sensitive language

Resources

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

- Welcome to Python

- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket