String Indexing and Slicing

String indexing allows you to access individual characters in a string. You can do this by using square brackets and the location, or index, of the character you want to access. It's important to remember that Python starts indexes at 0. So to access the first character in a string, you would use the index [0]. If you try to access an index that's larger than the length of your string, you'll get an **IndexError**. This is because you're trying to access something that doesn't exist! You can also access indexes from the end of the string going towards the start of the string by using negative values. The index [-1] would access the last character of the string, and the index [-2] would access the second-to-last character.

You can also access a portion of a string, called a slice or a substring. This allows you to access multiple characters of a string. You can do this by creating a range, using a colon as a separator between the start and end of the range, like [2:5].

This range is similar to the range() function we saw previously. It includes the first number, but goes to one less than the last number. For example:

```
>>> fruit = "Mangosteen" >>> fruit[1:4] 'ang'
```

The slice *includes* the character at index 1, and *excludes* the character at index 4. You can also easily reference a substring at the start or end of the string by only specifying one end of the range. For example, only giving the end of the range:

```
>>> fruit[:5] 'Mango'
```

This gave us the characters from the start of the string through index 4, *excluding* index 5. On the other hand this example gives is the characters *including* index 5, through the end of the string:

```
>>> fruit[5:] 'steen'
```

You might have noticed that if you put both of those results together, you get the original string back!

```
>>> fruit[:5] + fruit[5:] 'Mangosteen'
```

Cool!

Basic String Methods

In Python, strings are immutable. This means that they can't be modified. So if we wanted to fix a typo in a string, we can't simply modify the wrong character. We would have to create a new string with the typo corrected. We can also assign a new value to the variable holding our string.

If we aren't sure what the index of our typo is, we can use the string method *index* to locate it and return the index. Let's imagine we have the string **"lions tigers and bears"** in the variable **animals**. We can locate the index that contains the letter **g** using *animals.index("g")*, which will return the index; in this case 8. We can also use substrings to locate the index where the substring begins. *animals.index("bears")* would return 17, since that's the start of the substring. If there's more than one match for a substring, the index method will return the first match. If we try to locate a substring that doesn't exist in the string, we'll receive a **ValueError** explaining that the substring was not found.

```
animals = "lions tigers and bears"
animals.index("g")
```

```
animals = "lions tigers and bears"
animals.index("bears")
```

We can avoid a ValueError by first checking if the substring exists in the string. This can be done using the ***in*** keyword. We saw this keyword earlier when we covered *for* loops. In this case, it's a conditional that will be either True or False. If the substring is found in the string, it will be True. If the substring is not found in the string, it will be False. Using our previous variable **animals**, we can do **"horses" in animals** to check if the substring "horses" is found in our variable. In this case, it would evaluate to False, since horses aren't included in our example string. If we did **"tigers" in animals**, we'd get True, since this substring is contained in our string.

```
animals = "lions tigers and bears"
"horses" in animals
```

```
animals = "lions tigers and bears"
"tigers" in animals
```

Advanced String Methods

We've covered a bunch of String class methods already, so let's keep building on those and run down some more advanced methods.

The string method **lower** will return the string with all characters changed to lowercase. The inverse of this is the **upper** method, which will return the string all in uppercase. Just like with previous methods, we call these on a string using dot notation, like **"this is a string".upper()**. This would return the string **"THIS IS A STRING"**. This can be super handy when checking user input, since someone might type in all lowercase, all uppercase, or even a mixture of cases.

You can use the **strip** method to remove surrounding whitespace from a string. Whitespace includes spaces, tabs, and newline characters. You can also use the methods **lstrip** and **rstrip** to remove whitespace only from the left or the right side of the string, respectively.

The method **count** can be used to return the number of times a substring appears in a string. This can be handy for finding out how many characters appear in a string, or counting the number of times a certain word appears in a sentence or paragraph.

If you wanted to check if a string ends with a given substring, you can use the method **endswith**. This will return True if the substring is found at the end of the string, and False if not.

The **isnumeric** method can check if a string is composed of only numbers. If the string contains only numbers, this method will return True. We can use this to check if a string contains numbers before passing the string to the **int()** function to convert it to an integer, avoiding an error. Useful!

We took a look at string concatenation using the plus sign, earlier. We can also use the **join** method to concatenate strings. This method is called on a string that will be used to join a list of strings. The method takes a list of strings to be joined as a parameter, and returns a new string composed of each of the strings from our list joined using the initial string. For example, **" ".join(["This","is","a","sentence"])** would return the string **"This is a sentence"**.

The inverse of the join method is the **split** method. This allows us to split a string into a list of strings. By default, it splits by any whitespace characters. You can also split by any other characters by passing a parameter.

String Formatting

You can use the **format** method on strings to concatenate and format strings in all kinds of powerful ways. To do this, create a string containing curly brackets, **{}**, as a placeholder, to be replaced. Then call the format method on the string using *.format()* and pass variables as parameters. The variables passed to the method will then be used to replace the curly bracket placeholders. This method automatically handles any conversion between data types for us.

If the curly brackets are empty, they'll be populated with the variables passed in the order in which they're passed. However, you can put certain expressions inside the curly brackets to do even more powerful string formatting operations. You can put the name of a variable into the curly brackets, then use the names in the parameters. This allows for more easily readable code, and for more flexibility with the order of variables.

You can also put a formatting expression inside the curly brackets, which lets you alter the way the string is formatted. For example, the formatting expression **{:.2f}** means that you'd format this as a float number, with two digits after the decimal dot. The colon acts as a separator from the field name, if you had specified one. You can also specify text alignment using the greater than operator: **>**. For example, the expression **{:>3.2f}** would align the text three spaces to the right, as well as specify a float number with two decimal places. String formatting can be very handy for outputting easy-to-read textual output.

String Reference Guide

In Python, there are a lot of things you can do with strings. In this guide, you'll find the most common string operations and string methods.

String operations

- **len(string) -** Returns the length of the string
- **for character in string** - Iterates over each character in the string
- **if substring in string** - Checks whether the substring is part of the string
- **string[i]** - Accesses the character at index **i** of the string, starting at zero
- **string[i:j]** - Accesses the substring starting at index **i**, ending at index **j** minus 1. If **i** is omitted, its value defaults to **0**. If **j** is omitted, the value will default to **len(string)**.

String methods

- **string.lower()** - Returns a copy of the string with all lowercase characters
- **string.upper()** - Returns a copy of the string with all uppercase characters
- **string.lstrip()** - Returns a copy of the string with the left-side whitespace removed
- **string.rstrip()** - Returns a copy of the string with the right-side whitespace removed
- **string.strip()** - Returns a copy of the string with both the left and right-side whitespace removed
- **string.count(substring)** - Returns the number of times substring is present in the string
- **string.isnumeric()** - Returns True if there are only numeric characters in the string. If not, returns False.
- **string.isalpha()** - Returns True if there are only alphabetic characters in the string. If not, returns False.
- **string.split()** - Returns a list of substrings that were separated by whitespace (whitespace can be a space, tab, or new line)
- **string.split(delimiter)** - Returns a list of substrings that were separated by whitespace or a delimiter
- **string.replace(old, new)** - Returns a new string where all occurrences of old have been replaced by new.
- **delimiter.join(list of strings)** - Returns a new string with all the strings joined by the delimiter

Check out the official documentation for all available String methods.

Formatting Strings Guide

Python offers different ways to format strings. In the video, we explained the format() method. In this reading, we'll highlight three different ways of formatting strings. For this course you only need to know the format() method. But on the internet, you might find any of the three, so it's a good idea to know that the others exist.

Using the format() method

The format method returns a copy of the string where the {} placeholders have been replaced with the values of the variables. These variables are converted to strings if they weren't strings already. Empty placeholders are replaced by the variables passed to format in the same order.

```
# "base string with {} placeholders".format(variables)
```

```
example = "format() method"

formatted_string = "this is an example of using the {} on a string".format(example)

print(formatted_string)

"""Outputs:
this is an example of using the format() method on a string
"""
```

If the placeholders indicate a number, they're replaced by the variable corresponding to that order (starting at zero).

```
# "{0} {1}".format(first, second)

first = "apple"
second = "banana"
third = "carrot"

formatted_string = "{0} {2} {1}".format(first, second, third)

print(formatted_string)

"""Outputs:
apple carrot banana
"""
```

If the placeholders indicate a field name, they're replaced by the variable corresponding to that field name. This means that parameters to format need to be passed indicating the field name.

```
# "{var1} {var2}".format(var1=value1, var2=value2)

"{:exp1} {:exp2}".format(value1, value2)
```

If the placeholders include a colon, what comes after the colon is a formatting expression. See below for the expression reference.

Official documentation for the format string syntax

```
# {:d} integer value
'{:d}'.format(10.5) → '10'
```

Formatting expressions

Expr	Meaning	Example
{:d}	integer value	'{:d}'.format(10.5) → '10'
{:.2f}	floating point with that many decimals	'{:.2f}'.format(0.5) → '0.50'
{:.2s}	string with that many characters	'{:.2s}'.format('Python') → 'Py'
{:<6s}	string aligned to the left that many spaces	'{:<6s}'.format('Py') → 'Py '
{:>6s}	string aligned to the right that many spaces	'{:>6s}'.format('Py') → ' Py'
{:^6s}	string centered in that many spaces	'{:^6s}'.format('Py') → ' Py '

Check out the official documentation for all available expressions.

Old string formatting (Optional)

The format() method was introduced in Python 2.6. Before that, the % (modulo) operator could be used to get a similar result. While this method is **no longer recommended** for new code, you might come across it in someone else's code. This is what it looks like:

"base string with %s placeholder" % variable

The % (modulo) operator returns a copy of the string where the placeholders indicated by % followed by a formatting expression are replaced by the variables after the operator.

"base string with %d and %d placeholders" % (value1, value2)

To replace more than one value, the values need to be written between parentheses. The formatting expression needs to match the value type.

"%(var1) %(var2)" % {var1:value1, var2:value2}

Variables can be replaced by name using a dictionary syntax (we'll learn about dictionaries in an upcoming video).

"Item: %s - Amount: %d - Price: %.2f" % (item, amount, price)

The formatting expressions are mostly the same as those of the format() method.

Check out the official documentation for old string formatting.

Formatted string literals (Optional)

This feature was added in Python 3.6 and isn't used a lot yet. Again, it's included here in case you run into it in the future, but it's not needed for this or any upcoming courses.

A formatted string literal or f-string is a string that starts with 'f' or 'F' before the quotes. These strings might contain {} placeholders using expressions like the ones used for format method strings.

The important difference with the format method is that it takes the value of the variables from the current context, instead of taking the values from parameters.

Examples:

```
>>> name = "Micah"

>>> print(f'Hello {name}')

Hello Micah

>>> item = "Purple Cup"

>>> amount = 5

>>> price = amount * 3.25

>>> print(f'Item: {item} - Amount: {amount} - Price: {price:.2f}')

Item: Purple Cup - Amount: 5 - Price: 16.25
```

Check out the official documentation for f-strings.

Study Guide: Strings

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. The string readings in this section are great syntax guides to help you on the Strings Practice Quiz.

In the Strings segment, you learned about the parts of a string, string indexing and slicing, creating new strings, string methods and operations, and formatting strings.

Knowledge

String Operations and Methods

- **.format()** - String method that can be used to concatenate and format strings.
    - **{:.2f}** - Within the .format() method, limits a floating point variable to 2 decimal places. The number of decimal places can be customized.
- **len(string)** - String operation that returns the length of the string.
- **string[x]** - String operation that accesses the character at index [x] of the string, where indexing starts at zero.
- **string[x:y]** - String operation that accesses a substring starting at index [x] and ending at index [y-1]. If x is omitted, its value defaults to 0. If y is omitted, the value will default to len(string).
- **string.replace(old, new)** - String method that returns a new string where all occurrences of an old substring have been replaced by a new substring.
- **string.lower()** - String method that returns a copy of the string with all lowercase characters.

Coding skills

Skill Group 1

- Use a **for** loop to iterate through each letter of a string.
- Add a character to the front of a string.
- Add a character to the end of a string.
- Use the **.lower()** string method to convert the case (uppercase/lowercase) of the letters within a string variable. *This method is often used to eliminate cases as a factor when comparing two strings. For example, all lowercase "cat" is not equal to "Cat" because "Cat" contains an uppercase letter. To be able to compare the two strings to see if they are the same word, you can use the .lower() string method to remove capitalization as a factor in the string comparison.*

```
# This function accepts a given string and checks each character of
# the string to see if it is a letter or not. If the character is a
# letter, that letter is added to the end of the string variable
# "forwards" and to the beginning of the string variable "backwards".
def mirrored_string(my_string):

    # Two variables are initialized as string data types using empty
    # quotes. The variable "forwards" will hold the "my_string"
    # minus any character that is not a letter. The "backwards"
    # variable will hold the same letters as "forwards", but in
    # in reverse order.
    forwards = ""
    backwards = ""

    # The for loop iterates through each character of the "my_string"
    for character in my_string:

        # The if-statement checks if the character is not a space.
        if character.isalpha():
```

```
            # If True, the body of the loop adds the character to the
            # to the end of "forwards" and to the front of
            # "backwards".
            forwards += character
            backwards = character + backwards

        # If False (meaning the character is not a letter), no action
        # is needed. This coding approach results prevents any
        # non-alphabetical characters from being written to the
        # "forwards" and "backwards" variables. The for loop will
        # restart until all characters in "my_string" have been
        # processed.

    # The final if-statement compares the "forwards" and "backwards"
    # strings to see if the letters are the same both forwards and
    # backwards. Since Python is case sensitive, the two strings will
    # need to be converted to use the same case for this comparison.
    if forwards.lower() == backwards.lower():
        return True
    return False
```

Skill Group 2

- Use the **format()** method, with **{}** placeholders for variable data, to create a new string.
- Use a formatting expression, like **{:.2f}**, to format a float variable and configure the number of decimal places to display for the float.

```
# This function converts measurement equivalents. Output is formatted
# as, "x ounces equals y pounds", with y limited to 2 decimal places.
def convert_weight(ounces):

    # Conversion formula: 1 pound = 16 ounces
    pounds = ounces/16

    # The result is composed using the .format() method. There are two
    # placeholders in the string: the first is for the "ounces"
    # variable and the second is for the "pounds" variable. The second
    # placeholder formats the float result of the conversion
    # calculation to be limited to 2 decimal places.
    result = "{} ounces equals {:.2f} pounds".format(ounces,pounds)
    return result


print(convert_weight(12)) # Should be: 12 ounces equals 0.75 pounds
print(convert_weight(50.5)) # Should be: 50.5 ounces equals 3.16 pounds
print(convert_weight(16)) # Should be: 16 ounces equals 1.00 pounds
```

Skill Group 3

- Within the **format()** parameters, select characters at specific index [ ] positions from a variable string.
- Use the **format()** method, with **{}** placeholders for variable data, to create a new string.

```
# This function generates a username using the first 3 letters of a
# user’s last name plus their birth year.
def username(last_name, birth_year):

    # The .format() method will use the first 3 letters at index
    # positions [0,1,2] of the "last_name" variable for the first
    # {} placeholder. The second {} placeholder concatenates the user’s
    #  "birth_year" to that string to form a new string username.
    return("{}{}".format(last_name[0:3],birth_year))


print(username("Ivanov", "1985"))
# Should display "Iva1985"
print(username("Rodríguez", "2000"))
# Should display "Rod2000"
print(username("Deng", "1991"))
# Should display "Den1991"
```

Skill Group 4

- Use the **.replace()** method to replace part of a string.
- Use the **len()** function to get the number of index positions in a string.
- Slice a string at a specific index position.

```
# This function checks a given schedule entry for an old date and, if
# found, the function replaces it with a new date.
def replace_date(schedule, old_date, new_date):

    # Check if the given "old_date" appears at the end of the given
    # string variable "schedule".
    if schedule.endswith(old_date):

        # If True, the body of the if-block will run. The variable "n" is
        # used to hold the slicing index position. The len() function
        # is used to measure the length of the string "new_date".
        p = len(old_date)

        # The "new_schedule" string holds the updated string with the
        # old date replaced by the new date. The schedule[:-p] part of
        # the code trims the "old_date" substring from "schedule"
        # starting at the final index position (or right-side) counting
        # towards the left the same number of index positions as
        # calculated from len(old_date). Then, the code schedule[-p:]
        # starts the indexing position at the slot where the first
        # character of the "old_date" used to be positioned. The
        # .replace(old_date, new_date) code inserts the "new_date" into
        # the position where the "old_date" used to exist.
        new_schedule = schedule[:-p] + schedule[-p:].replace(old_date, new_date)

        # Returns the schedule with the new date.
        return new_schedule

    # If the schedule does not end with the old date, then return the
    # original sentence without any modifications.
```

```
    return schedule


print(replace_date("Last year’s annual report will be released in March 2023", "2023", "20
24"))
# Should display "Last year’s annual report will be released in March 2024"
print(replace_date("In April, the CEO will hold a conference", "April", "May"))
# Should display "In April, the CEO will hold a conference"
print(replace_date("The convention is scheduled for October", "October", "June"))
# Should display "The convention is scheduled for June"
```

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

Lists Defined

Lists in Python are defined using square brackets, with the elements stored in the list separated by commas: **list = ["This", "is", "a", "list"]**. You can use the **len()** function to return the number of elements in a list: **len(list)** would return **4**. You can also use the **in** keyword to check if a list contains a certain element. If the element is present, it will return a True boolean. If the element is not found in the list, it will return False. For example, **"This" in list** would return True in our example. Similar to strings, lists can also use indexing to access specific elements in a list based on their position. You can access the first element in a list by doing **list[0]**, which would allow you to access the string **"This"**.

In Python, lists and strings are quite similar. They’re both examples of sequences of data. Sequences have similar properties, like (1) being able to iterate over them using **for loops**; (2) support indexing; (3) using the **len** function to find the length of the sequence; (4) using the plus operator **+** in order to concatenate; and (5) using the **in** keyword to check if the sequence contains a value. Understanding these concepts allows you to apply them to other sequence types as well.

Modifying Lists

While lists and strings are both sequences, a big difference between them is that lists are mutable. This means that the contents of the list can be changed, unlike strings, which are immutable. You can add, remove, or modify elements in a list.

You can add elements to the end of a list using the **append** method. You call this method on a list using dot notation, and pass in the element to be added as a parameter. For example, **list.append("New data")** would add the string "New data" to the end of the list called list.

If you want to add an element to a list in a specific position, you can use the method **insert**. The method takes two parameters: the first specifies the index in the list, and the second is the element to be added to the list. So **list.insert(0, "New data")** would add the string "New data" to the front of the list. This wouldn't overwrite the existing element at the start of the list. It would just shift all the other elements by one. If you specify an index that’s larger than the length of the list, the element will simply be added to the end of the list.

You can remove elements from the list using the **remove** method. This method takes an element as a parameter, and removes the first occurrence of the element. If the element isn't found in the list, you'll get a **ValueError** error explaining that the element was not found in the list.

You can also remove elements from a list using the **pop** method. This method differs from the remove method in that it takes an index as a parameter, and returns the element that was removed. This can be useful if you don't know what the value is, but you know where it's located. This can also be useful when you need to access the data and also want to remove it from the list.

Finally, you can change an element in a list by using indexing to overwrite the value stored at the specified index. For example, you can enter **list[0] = "Old data"** to overwrite the first element in a list with the new string "Old data".

Tuples

As we mentioned earlier, strings and lists are both examples of sequences. Strings are sequences of characters, and are immutable. Lists are sequences of elements of any data type, and are mutable. The third sequence type is the tuple. Tuples are like lists, since they can contain elements of any data type. But unlike lists, tuples are immutable. They're specified using parentheses instead of square brackets.

You might be wondering why tuples are a thing, given how similar they are to lists. Tuples can be useful when we need to ensure that an element is in a certain position and will not change. Since lists are mutable, the order of the elements can be changed on us. Since the order of the elements in a tuple can't be changed, the position of the element in a tuple can have meaning. A good example of this is when a function returns multiple values. In this case, what gets returned is a tuple, with the return values as elements in the tuple. The order of the returned values is important, and a tuple ensures that the order isn't going to change. Storing the elements of a tuple in separate variables is called unpacking. This allows you to take multiple returned values from a function and store each value in its own variable.

Iterating Over Lists Using Enumerate

When we covered *for* loops, we showed the example of iterating over a list. This lets you iterate over each element in the list, exposing the element to the for loop as a variable. But what if you want to access the elements in a list, along with the index of the element in question? You can do this using the **enumerate()** function. The enumerate() function takes a list as a parameter and returns a tuple for each element in the list. The first value of the tuple is the index and the second value is the element itself.

List Comprehension Examples

You can create a list from a sequence using a **for** loop, but there's a more streamlined way to do this by using a list comprehension. List comprehensions allow you to create a new list from a sequence or a range in a single line.

Simple List Comprehension

For example, **[x*2 for x in range(1,11)]** is a simple list comprehension. This single line of code iterates over a range from 1 to 10, multiplies each element in the range by 2, and creates a new list from all multiples of 2 from 2 to 20.

```
### Simple List Comprehension
print("List comprehension result:")

# The following list comprehension compacts several lines
# of code into one line:
print([x*2 for x in range(1,11)])
```

```
### Long form for loop
print("Long form code result:")

# The list comprehension above accomplishes the same result as
# the long form version of the code:
my_list = []
for x in range(1,11):
   my_list.append(x*2)
print(my_list)


# Click Run to compare the two results.
```

List Comprehension with Conditional Statement

You can also use conditionals with list comprehensions to build even more complex and powerful statements. You can do this by appending an if statement to the end of the list comprehension. For example, **[x for x in range(1,101) if x % 10 == 0]** generates a new list containing all the integers divisible by 10 from 1 to 100. The if statement evaluates each value in the range from 1 to 100 to check if it's evenly divisible by 10. If it is, the number is added to a new list.

```
### List Comprehension with Conditional Statement
print("List comprehension result:")

# The following list comprehension compacts multiple lines
# of code into one line:
print([ x for x in range(1,101) if x % 10 == 0 ])

### Long form for loop with nested if-statement
print("Long form code result:")

# The list comprehension above accomplishes the same result as
# the long form version of the code:
my_list = []
for x in range(1,101):
  if x % 10 == 0:
    my_list.append(x)
print(my_list)


# Click Run to observe the two results.
```

List comprehensions can be really powerful, but they can also be complex, resulting in code that's hard to read. Be careful when using them, since it might make it more difficult for someone else looking at your code to easily understand what the code is doing. It is a best practice to add descriptive comments about any list comprehensions used in your code. This helps to communicate the purpose of list comprehensions to other coders. Comments will also help you remember the goal of the code when performing future code additions and maintenance.

Practice exercise

This exercise will walk you through how to write a list comprehension to create a list of squared numbers (n*n). It needs to return a list of squares of consecutive numbers between "start" and "end" *inclusively*. For example, squares(2, 3) should return a list containing [4, 9].

1. The function receives the variables "start" and "end" through the function parameters.
2. In the **return** line, start by entering the list brackets [ ]
3. Between the brackets [ ], enter the arithmetic expression to square a variable "n".
4. To the right of the square expression, write a **for** loop that iterates over "n" in a range from the "start" to "end" variables.
5. Ensure the "end" range value is included in the **range()** by adding 1 to it.
6. Run your code to see if it works! If needed, the solution to this code is included in the "Study Guide: List Operations and Methods" reading under "Skill Group 2" (list comprehensions).

```
def squares(start, end):
    return ___


print(squares(2, 3))  # Should print [4, 9]
print(squares(1, 5))  # Should print [1, 4, 9, 16, 25]
print(squares(0, 10)) # Should print [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Study Guide: Lists Operations and Methods

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 Lists and Tuples segment, you learned about the differences between lists and tuples, how to modify the contents of a list, how to iterate over lists and tuples, how to use the enumerate() function, and how to create list comprehensions.

Knowledge

Common sequence operations

Lists and tuples are both sequences and they share a number of sequence operations. The following common sequence operations are used by both lists and tuples:

- **len(sequence)** - Returns the length of the sequence.
- **for element in sequence** - Iterates over each element in the sequence.
- **if element in sequence** - Checks whether the element is part of the sequence.
- **sequence[x]** - Accesses the element at index [x] of the sequence, starting at zero
- **sequence[x:y]** - Accesses a slice starting at index [x], ending at index [y-1]. If [x] is omitted, the index will start at 0 by default. If [y] is omitted, the len(sequence) will set the ending index position by default.
- **for index, element in enumerate(sequence)** - Iterates over both the indices and the elements in the sequence at the same time.

List-specific operations and methods

One major difference between lists and tuples is that lists are mutable (changeable) and tuples are immutable (not changeable). There are a few operations and methods that are specific to changing data within lists:

- **list[index] = x** - Replaces the element at index [n] with x.
- **list.append(x)** - Appends x to the end of the list.
- **list.insert(index, x)** - Inserts x at index position [index].

- **list.pop(index)** - Returns the element at [index] and removes it from the list. If [index] position is not in the list, the last element in the list is returned and removed.
- **list.remove(x)** - Removes the first occurrence of x in the list.
- **list.sort()** - Sorts the items in the list.
- **list.reverse()** - Reverses the order of items of the list.
- **list.clear()** - Deletes all items in the list.
- **list.copy()** - Creates a copy of the list.
- **list.extend(other_list)** - Appends all the elements of other_list at the end of list

List comprehensions

A list comprehension is an efficient method for creating a new list from a sequence or a range in a single line of code. It is a best practice to add descriptive comments about any list comprehensions used in your code, as their purpose can be difficult to interpret by other coders.

- **[expression for variable in sequence]** - Creates a new list based on the given sequence. Each element in the new list is the result of the given expression.
- Example: **my_list = [x*2 for x in range(1,11)]**
- **[expression for variable in sequence if condition]** - Creates a new list based on a specified sequence. Each element is the result of the given expression; elements are added only if the specified condition is true.
    - Example: **my_list = [x for x in range(1,101) if x % 10 == 0]**

Coding skills

Skill Group 1

- Use a **for** loop to modify elements of a list.
- Use the **list.append(old,new)** method.
- Use the **string.endswith()** and **string.replace()** methods to modify the elements within a list.

```
# This block of code changes the year on a list of dates.
# The "years" list is given with existing elements.
years = ["January 2023", "May 2025", "April 2023", "August 2024", "September 2025", "Decem
ber 2023"]



# The variable "updated_years" is initialized as a list data type
# using empty square brackets []. This list will hold the new list
# with the updated years.
updated_years = []

# The for loop checks each "year" element in the list "years".
for year in years:

    # The if-statement checks if the "year" element ends with the
    # substring "2023".
    if year.endswith("2023"):

        # If True, then a temporary variable "new" will hold the
        # modified "year" element where the "2023" substring is
        # replaced with the substring "2024".
        new = year.replace("2023","2024")

        # Then, the list "updated_years" is appended with the changed
```

```
        # element held in the temporary variable "new".
        updated_years.append(new)

    # If False, the original "year" element will be appended to the
    # the "updated_years" list unchanged.
    else:
        updated_years.append(year)


print(updated_years)
# Should print ["January 2024", "May 2025", "April 2024", "August 2024", "September 2025",
 "December 2024"]
```

Skill Group 2

- Use a list comprehension to return values

```
# This list comprehension creates a list of squared numbers (n*n). It
# accepts two integer variables through the function's parameters.
def squares(start, end):

# The list comprehension calculates the square of a variable integer
# "n", where "n" ranges from the "start" to "end" variables inclusively.
# To be inclusive in a range(), add +1 to the end of range variable.
    return [n*n for n in range(start,end+1)]


print(squares(2, 3))  # Should print [4, 9]
print(squares(1, 5))  # Should print [1, 4, 9, 16, 25]
print(squares(0, 10)) # Should print [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Skill Group 3

- Use the **string[index]** method within a list comprehension.
- Use a list comprehension to modify elements of a list.
- Use the **string.replace()** method within a list comprehension.

```
# This block of code also changes the year on a list of dates using a
# different approach than demonstrated in Skill Group 1. By using a
# list comprehension, you can see how it is possible to refactor the
# code to a shorter, more efficient code block.

# The "years" list is given with existing elements.
years = ["January 2023", "May 2025", "April 2023", "August 2024", "September 2025", "Decem
ber 2023"]

# The list comprehension below creates a new list "updated_years" to
# hold the command to replace the "2023" substring of the "year"
# element with the substring "2024". This action will be executed if
# the last 4 indices of the "year" string is equal to the substring
# "2023". If false (else), the "year" element will be included in the
# new list "updated_years" unchanged.
updated_years = [year.replace("2023","2024") if year[-
4:] == "2023" else year for year in years]
```

```
print(updated_years)
# Should print ["January 2024", "May 2025", "April 2024", "August 2024", "September 2025",
 "December 2024"]
```

Skill Group 4

- Use the **string.split()** method to separate a string into a list of individual words.
- Iterate over the new list using a **for** loop.
- Modify each element in the list by slicing the element's string at a given [1:] index position and appending the substring to the end of the element.
- Convert a list back into a string.

```
# This function splits a given string into a list of elements. Then, it
# modifies each element by moving the first character to the end of the
# element and adds a dash between the element and the moved character.
# For example, the element "2two" will be changed to "two-2". Finally,
# the function converts the list back to a string, and returns the
# new string.
def change_string(given_string):

# Initialize "new_string" as a string data type by using empty quotes.
    new_string = ""
    # Split the "given_string" into a "new_list", with each "element"
    # holding an individual word from the string.
    new_list = given_string.split()

    # The for loop iterates over each "element" in the "new_list".
    for element in new_list:

        # Convert the list into a "new_string" by using the assignment
        # operator += to concatenate the following items:
        # + Each list "element" (starting at index position [1:]),
        # + a dash "-",
        # + append the first character of the "element" (using the index
        # [0]) to the end of the "element", and finally,
        # + a space " " to separate each "element" in the "new_string".
        new_string += element[1:] + "-"  + element[0] + " "

    # Return the list that has been converted back into a string.
    return new_string


print(change_string("1one 2two 3three 4four 5five"))
# Should print "one-1 two-2 three-3 four-4 five-5"
```

Skill Group 5

- Use the **string.join()** method to concatenate a string that provides a list name and its elements

```
# This function accepts a list name and a list of elements, and returns
# a string with the format: "The "list_name" list includes: element1,
# element2, element3".
```

```
def list_elements(list_name, elements):

    # This task can be completed in a single line of code. The
    # concatenation of strings, "list_name", and the list "elements" can
    # occur on the return line. In this case, the string "The " is added
    # to the "list_name", plus the string " list includes: ", then the
    # "elements" are joined using a comma to separate each element of the
    # list.
    return "The " + list_name + " list includes: " + ", ".join(elements)


print(list_elements("Printers", ["Color Printer", "Black and White Printer", "3-
D Printer"]))
# Should print "The Printers list includes: Color Printer, Black and White Printer, 3-
D Printer"
```

Resources

For additional information about list and tuple operations and methods, please visit:

- Common Sequence Operations - Official python.org documentation for list, tuple, and range sequence operations.
- Lists - Official python.org documentation for list operations and methods.
- Tuples - Official python.org documentation for tuple operations and methods

---

Dictionaries Defined

Dictionaries are another data structure in Python. They're similar to a list in that they can be used to organize data into collections. However, data in a dictionary isn't accessed based on its position. Data in a dictionary is organized into pairs of keys and values. You use the key to access the corresponding value. Where a list index is always a number, a dictionary key can be a different data type, like a string, integer, float, or even tuples.

When creating a dictionary, you use curly brackets: **{}**. When storing values in a dictionary, the key is specified first, followed by the corresponding value, separated by a colon. For example, **animals = { "bears":10, "lions":1, "tigers":2 }** creates a dictionary with three key value pairs, stored in the variable animals. The key "bears" points to the integer value 10, while the key "lions" points to the integer value 1, and "tigers" points to the integer 2. You can access the values by referencing the key, like this: **animals["bears"]**. This would return the integer 10, since that's the corresponding value for this key.

You can also check if a key is contained in a dictionary using the **in** keyword. Just like other uses of this keyword, it will return True if the key is found in the dictionary; otherwise it will return False.

Dictionaries are mutable, meaning they can be modified by adding, removing, and replacing elements in a dictionary, similar to lists. You can add a new key value pair to a dictionary by assigning a value to the key, like this: **animals["zebras"] = 2**. This creates the new key in the animal dictionary called zebras, and stores the value 2. You can modify the value of an existing key by doing the same thing. So **animals["bears"] = 11** would change the value stored in the bears key from 10 to 11. Lastly, you can remove elements from a dictionary by using the **del** keyword. By doing **del animals["lions"]** you would remove the key value pair from the animals dictionary.

Iterating Over Dictionaries

You can iterate over dictionaries using a *for* loop, just like with strings, lists, and tuples. This will iterate over the sequence of keys in the dictionary. If you want to access the corresponding values associated with the keys, you could use the keys as indexes. Or you can use the **items** method on the dictionary, like **dictionary.items()**. This

method returns a tuple for each element in the dictionary, where the first element in the tuple is the key and the second is the value.

If you only wanted to access the keys in a dictionary, you could use the **keys()** method on the dictionary: **dictionary.keys()**. If you only wanted the values, you could use the **values()** method: **dictionary.values()**.

Study Guide: Dictionary Methods

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 Dictionary segment, you learned about the properties of the Python dictionary data type, how dictionaries differ from lists, how to iterate over the contents of a dictionary, and how to use dictionaries with lists and strings.

Knowledge

Python dictionaries are used to organize elements into collections. Dictionaries include one or more keys, with one or more values associated with each key.

Syntax

```
my_dictionary = {keyA:value1,value2, keyB:value3,value4}
```

Operations

- **len(dictionary)** - Returns the number of items in a dictionary.
- **for key, in dictionary** - Iterates over each key in a dictionary.
- **for key, value in dictionary.items()** - Iterates over each key,value pair in a dictionary.
- **if key in dictionary** - Checks whether a key is in a dictionary.
- **dictionary[key]** - Accesses a value using the associated key from a dictionary.
- **dictionary[key] = value** - Sets a value associated with a key.
- **del dictionary[key]** - Removes a value using the associated key from a dictionary.

Methods

- **dictionary.get(key, default)** - Returns the value corresponding to a key, or the default value if the specified key is not present.
- **dictionary.keys()** - Returns a sequence containing the keys in a dictionary.
- **dictionary.values()** - Returns a sequence containing the values in a dictionary.
- **dictionary[key].append(value)** - Appends a new value for an existing key.
- **dictionary.update(other_dictionary)** - Updates a dictionary with the items from another dictionary. Existing entries are updated; new entries are added.
- **dictionary.clear()** - Deletes all items from a dictionary.
- **dictionary.copy()** - Makes a copy of a dictionary.

Dictionaries versus Lists

Dictionaries are similar to lists, but there are a few differences:

Both dictionaries and lists:

- are used to organize elements into collections;
- are used to initialize a new dictionary or list, use empty brackets;
- can iterate through the items or elements in the collection; and
- can use a variety of methods and operations to create and change the collections, like removing and inserting items or elements.

Dictionaries only:

- are unordered sets;
- have keys that can be a variety of data types, including strings, integers, floats, tuples;.
- can access dictionary values by keys;
- use square brackets inside curly brackets { [ ] };
- use colons between the key and the value(s);
- use commas to separate each key group and each value within a key group;
- make it quicker and easier for a Python interpreter to find specific elements, as compared to a list.

Dictionary Example:

```
pet_dictionary = {"dogs": ["Yorkie", "Collie", "Bulldog"], "cats": ["Persian", "Scottish F
old", "Siberian"], "rabbits": ["Angora", "Holland Lop", "Harlequin"]}


print(pet_dictionary.get("dogs", 0))
# Should print ['Yorkie', 'Collie', 'Bulldog']
```

Lists only:

- are ordered sets;
- access list elements by index positions;
- require that these indices be integers;
- use square brackets [ ];
- use commas to separate each list element.

List Example:

```
pet_list  = ["Yorkie", "Collie", "Bulldog", "Persian", "Scottish Fold", "Siberian", "Angor
a", "Holland Lop", "Harlequin"]


print(pet_list[0:3])
# Should print ['Yorkie', 'Collie', 'Bulldog']
```

Coding skills

Skill Group 1

- Iterate over the key and value pairs of a dictionary using a **for** loop with the **dictionary.items()** method to calculate the sum of the values in a dictionary.

```
# This function returns the total time, with minutes represented as
```

```
# decimals (example: 1 hour 30 minutes = 1.5), for all end user time
# spent accessing a server in a given day.


def sum_server_use_time(Server):

    # Initialize the variable as a float data type, which will be used
    # to hold the sum of the total hours and minutes of server usage by
    # end users in a day.
    total_use_time = 0.0

    # Iterate through the "Server" dictionary's key and value items
    # using a for loop.
    for key,value in Server.items():

        # For each end user key, add the associated time value to the
        # total sum of all end user use time.
        total_use_time += Server[key]

    # Round the return value and limit to 2 decimal places.
    return round(total_use_time, 2)

FileServer = {"EndUser1": 2.25, "EndUser2": 4.5, "EndUser3": 1, "EndUser4": 3.75, "EndUser
5": 0.6, "EndUser6": 8}

print(sum_server_use_time(FileServer)) # Should print 20.1
```

Skill Group 2

- Concatenate a value, a string, and the key for each item in the dictionary and append to the end of a new list[ ] using the **list.append(x)** method.
- Iterate over keys with multiple values from a dictionary using nested **for** loops with the **dictionary.items()** method.

```
# This function receives a dictionary, which contains common employee
# last names as keys, and a list of employee first names as values.
# The function generates a new list that contains each employees' full
# name (First_name Last_Name). For example, the key "Garcia" with the
# values ["Maria", "Hugo", "Lucia"] should be converted to a list
# that contains ["Maria Garcia", "Hugo Garcia", "Lucia Garcia"].


def list_full_names(employee_dictionary):
    # Initialize the "full_names" variable as a list data type using
    # empty [] square brackets.
    full_names = []

    # The outer for loop iterates through each "last_name" key and
    # associated "first_name" values, in the "employee_dictionary" items.
    for last_name, first_names in employee_dictionary.items():

        # The inner for loop iterates over each "first_name" value in
        # the list of "first_names" for one "last_name" key at a time.
        for first_name in first_names:
```

```
            # Append the new "full_names" list with the "first_name" value
            # concatenated with a space " ", and the key "last_name".
            full_names.append(first_name+" "+last_name)

    # Return the new "full_names" list once the outer for loop has
    # completed all iterations.
    return(full_names)


print(list_full_names({"Ali": ["Muhammad", "Amir", "Malik"], "Devi": ["Ram", "Amaira"], "C
hen": ["Feng", "Li"]}))
# Should print ['Muhammad Ali', 'Amir Ali', 'Malik Ali', 'Ram Devi', 'Amaira Devi', 'Feng
Chen', 'Li Chen']
```

Skill Group 3

- Use the **dictionary[key] = value** operation to associate a value with a key in a dictionary.
- Iterate over keys with multiple values from a dictionary, using nested **for** loops and an **if**-statement, and the **dictionary.items()** method.
- Use the **dictionary[key].append(value)** method to add the key, a string, and the key for each item in the dictionary.

```
# This function receives a dictionary, which contains resource
# categories (keys) with a list of available resources (values) for a
# company's IT Department. The resources belong to multiple categories.
# The function should reverse the keys and values to show which
# categories (values) each resource (key) belongs to.


def invert_resource_dict(resource_dictionary):
  # Initialize a "new_dictionary" variable as a dict data type using
  # empty {} curly brackets.
    new_dictionary = {}
    # The outer for loop iterates through each "resource_group" and
    # associated "resources" in the "resource_dictionary" items.
    for resource_group, resources in resource_dictionary.items():
        # The inner for loop iterates over each "resource" value in
        # the list of "resources" for one "resource_group" key at a time.
        for resource in resources:
            # The if-statement checks if the current "resource" value has
            # been appended as a key to the "new_dictionary" yet.
            if resource in new_dictionary:
                # If True, then append the "resource_group" as a value to the
                # "resource", which is now the key.
                new_dictionary[resource].append(resource_group)
            # If False (else), then add the "resource" as a new key with the
            # "resource_group" as a value for that key.
            else:
                new_dictionary[resource] = [resource_group]
    # Return the new dictionary once the outer for loop has completed
    # all iterations.
    return(new_dictionary)
```

```
print(invert_resource_dict({"Hard Drives": ["IDE HDDs", "SCSI HDDs"],
        "PC Parts":  ["IDE HDDs", "SCSI HDDs", "High-
end video cards", "Basic video cards"], "Video Cards": ["High-
end video cards", "Basic video cards"]}))
# Should print {'IDE HDDs': ['Hard Drives', 'PC Parts'], 'SCSI HDDs': ['Hard Drives', 'PC
Parts'], 'High-
end video cards': ['PC Parts', 'Video Cards'], 'Basic video cards': ['PC Parts', 'Video Ca
rds']}
```

Resources

For additional information about dictionaries, please visit:

- Mapping Types — dict - Official python.org documentation for dictionary methods
- Python Dictionaries - Tutorial with interactive code blocks for practicing using dictionary methods and operations

---

Study Guide: Module 4 Graded Quiz

It is time to prepare for the Module 4 Graded Quiz. Please review the following items from this module before beginning the quiz. If you would like to refresh your memory on these materials, please also revisit the Study Guides located before each practice quiz in Module 4: Study Guide: Strings, Study Guide: Lists Operations and Methods, and Study Guide: Dictionary Methods.

Knowledge

- How to output a list of the keys in a Python dictionary.
- How to determine the output of a string index range used on a string.
- Determine what a list should contain after the .insert() method is used on the list.
- How to replace a specific word in a sentence with the same word in all uppercase letters.
- How to use a dictionary to count the frequency of letters in a string.

Operations, Methods, and Functions

- **String Methods, Operations, and Functions**
  - .upper()
  - .lower()
  - .split()
  - .format()
  - .isnumeric()
  - .isalpha()
  - .replace()
  - string index [ ]
  - len()
- **List Operations and Methods**
  - .reverse()
  - .extend()
  - .insert()
  - .append()
  - .remove()
  - .sort()
  - list comprehensions [ ]

    - list comprehensions [ ] with if condition
- **Dictionary Operations and Methods**
    - .items()
    - .update()
    - .keys()
    - .values()
    - .copy()
    - dictionary[key]
    - dictionary[key] = value

Coding Skills

Skill 1: Using string methods

- Separate numerical values from text values in a string using **.split()**.
- Iterate over the elements in a string.
- Test if the element contains letters with **.isalpha()**.
- Assign the elements of the split string to new variables.
- Trim any extra white space using **.strip()**.
- Format a string using **.format()** and **{ }** variable placeholders.

```
def sales_prices(item_and_price):
    # Initialize variables "item" and "price" as strings
    item = ""
    price = ""
    # Create a variable "item_or_price" to hold the result of the split.
    item_or_price = item_and_price.split()

    # For each element "x" in the split variable "item_or_price"
    for x in item_or_price:

        # Check if the element is a number
        if x.isalpha():

            # If true, assign the element to the "item" string variable and add a space
            # for any item names containing multiple words, like "Winter fleece jacket".
            item += x + " "

        # Else, if x is a number (if x.isalpha() is false):
        else:
            # Assign the element to the "price" string variable.
            price = x

    # Strip the extra space to the right of the last "item" word
    item = item.strip()

    # Return the item name and price formatted in a sentence
    return "{} are on sale for ${}".format(item,price)


# Call to the function
print(sales_prices("Winter fleece jackets 49.99"))
# Should print "Winter fleece jackets are on sale for $49.99"
```

- Use the len() function to measure a string.

```
# This function accepts a string variable "data_field".
def count_words(data_field):

    # Splits the string into individual words.
    split_data = data_field.split()

    # Then returns the number of words in the string using the len()
    # function.
    return len(split_data)

    # Note that it is possible to combine the len() function and the
    # .split() method into the same line of code by inserting the
    # data_field.split() command into the the len() function parameters.

# Call to the function
count_words("Catalog item 3523: Organic raw pumpkin seeds in shell")
# Output should be 9
```

Skill 2: Using list methods

- Reverse the order of a list using the **.reverse()** method.
- Combine two lists using the **.extend()** method.

```
# This function accepts two variables, each containing a list of years.
# A current "recent_first" list contains [2022, 2018, 2011, 2006].
# An older "recent_last" list contains [1989, 1992, 1997, 2001].
# The lists need to be combined with the years in chronological order.
def record_profit_years(recent_first, recent_last):

    # Reverse the order of the "recent_first" list so that it is in
    # chronological order.
    recent_first.reverse()

    # Extend the "recent_last" list by appending the newly reversed
    # "recent_first" list.
    recent_last.extend(recent_first)

    # Return the "recent_last", which now contains the two lists
    # combined in chronological order.
    return recent_last

# Assign the two lists to the two variables to be passed to the
# record_profit_years() function.
recent_first = [2022, 2018, 2011, 2006]
recent_last = [1989, 1992, 1997, 2001]



# Call the record_profit_years() function and pass the two lists as
# parameters.
print(record_profit_years(recent_first, recent_last))
# Should print [1989, 1992, 1997, 2001, 2006, 2011, 2018, 2022]
```

Skill 3: Using a list comprehension

- Use a list comprehension [ ] as a shortcut for creating a new list from a range.
- Include a calculation with a **for** loop **in** a **range** with 2 parameters (lower, upper+1).

```
# The function accepts two parameters: a start year and an end year.
def list_years(start, end):

    # It returns a list comprehension that creates a list of years in a for
    # loop using a range from the start year to the end year (inclusive of
    # the upper range year, using end+1).
    return [year for year in range(start, end+1)]


# Call the years() function with two parameters.
print(list_years(1972, 1975))
# Should print [1972, 1973, 1974, 1975]
```

- Use a list comprehension [ ] with a **for** loop and an **if** condition.

```
# The function accepts two variable integers through the parameters and
# returns all odd numbers between x and y-1.
def odd_numbers(x, y):


# This list comprehension uses a for loop to iterate through values
# of n in a range from x to y, with the value of y excluded (meaning
# keep the default range() function behavior to exclude the
# end-of-range value from the range). Since an incremental value is not
# specified, the range function uses the default increment of +1.
# The if condition checks n to test if the number is odd using the
# modulo operator. This condition is written to check if n is divided
# by 2, that the remainder is not 0.
    return [n for n in range(x, y) if n % 2 != 0]


# Call the years() function with two parameters.
print(odd_numbers(5, 15))
# Should print [5, 7, 9, 11, 13]
```

Skill 4: Using dictionary methods

- Iterate through the keys and values of a dictionary.
- Return the keys and values in a formatted string using the .format() function.

```
# The network() function accepts a dictionary "servers" as a parameter.
def network(servers):

    # A string variable is initialized to hold the "result".
    result = ""

    # For each "hostname" (key) and "IP address" (value) in the "server" dictionary items.
..
    for hostname, IP_address in servers.items():
```

```
        # A string identifying the hostname and IP address for each server is added
        # to the "result" variable. The string .format() function and is used to plug
        # the hostname and IP_address variables into the designated {} placeholders
        # within the string.
        result += "The IP address of the {} server is {}".format(hostname, IP_address) + "
\n"

    # Return the "result" variable string.
    return result

# Call the "network" function with the dictionary.
print(network({"Domain Name Server":"8.8.8.8", "Gateway Server":"192.168.1.1", "Print Serv
er":"192.168.1.33", "Mail Server":"192.168.1.190"}))

# Should print:
# The IP address of the Domain Name Server server is 8.8.8.8
# The IP address of the Gateway Server server is 192.168.1.1
# The IP address of the Print Server server is 192.168.1.33
# The IP address of the Mail Server server is 192.168.1.190
```

- Create a copy of a dictionary.
- Iterate through the values of the new dictionary.
- Change each value in the new dictionary, while keeping the same keys.

```
# The scores() function accepts a dictionary "game_scores" as a parameter.
def reset_scores(game_scores):

    # The .copy() dictionary method is used to create a new copy of the "game_scores".
    new_game_scores = game_scores.copy()

    # The for loop iterates over new_game_scores items, with the player as the key
    # and the score as the value.
    for player, score in new_game_scores.items():

        # The dictionary operation to assign a new value to a key is used
        # to reset the grade values to 0.
        new_game_scores[player] = 0

    return new_game_scores

# The dictionary is defined.
game1_scores = {"Arshi": 3, "Catalina": 7, "Diego": 6}

# Call the "reset_scores" function with the "game1_scores" dictionary.
print(reset_scores(game1_scores))
# Should print {'Arshi': 0, 'Catalina': 0, 'Diego': 0}
```

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