Lesson 2: Data Types

Table of Contents

2.1 Objectives

- Examine the implications of using computation to solve a problem
  - Discuss best practices for using computation to solve a problem

    - Suggest types of problems that can be solved through computation
    - Show how computation can solve a problem

- Recognize key computer science concepts
    - Identify data types used in Python scripting
    - Identify data structures used in Python scripting
    - Define variables and strings
    - Recognize how queries operate

- Demonstrate the ability to build basic scripts using Python scripting language
    - Use various data types and structures in Python scripting
    - Collect data using Python scripting
    - Extract data using Python scripting
    - Develop advanced data structures using Python scripting

---

2.2. Overview

The following material is divided into the following parts:

- Lesson
- Guided Exercise
- Practical Exercises

Instructor Guidance: **Refer back to Lesson 1 and relate the four steps of problem solving using Computational Thinking (Decomposition, Pattern Recognition, Abstraction, & Algorithm Design) to lessons, exercises, examples, student questions/comments, etc., as appropriate throughout this lesson.**

2.3. Review

2.3.1. Computational Thinking

The basic steps for approaching a computational problem are:

1. **Decomposition:** Breaking down data, processes, or problems into smaller, manageable parts
2. **Pattern Recognition:** Observing patterns, trends, and regularities in data
3. **Abstraction:** Identifying the general principles that generate these patterns
4. **Algorithm Design:** Developing the step by step instructions for solving this and similar problems

Exercise: Write some pseudocode to solve the following problem: Given a number, print the squared value.

> Remember that pseudocode is just a set of steps, not actual working code.

In [ ]:

```
## YOUR PSEUDOCODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## No code required.

## 0. Create a number in a variable
## 1. Multiply that number by itself
## 2. Store the result in a variable
## 3. Print the variable
```

2.3.2. Variables

Exercise: Create a variable of any type and then call the `print()` function on it.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
text = "Hello world!"
print(text)
```

2.3.3. Jupyter Notebook: Keyboard Shortcuts

Exercise: Add a new code cell below, type some code to print the sentence `'Hello Lesson 2!'`, and run it using keyboard shortcuts.

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
text = "Hello Lesson 2!"
print(text)
```

2.4. Lesson: Data Types

2.4.1. Introduction to Data Types

References

- Python: Data Types (https://docs.python.org/2/library/datatypes.html)
- Python: Type() (https://docs.python.org/2.7/library/functions.html#type)

There are many data types in Python, but to keep things simple, we will begin by covering the basics: Boolean, integer, float, string, and the None Type. See the table below for short descriptions of these basic data types.

Plain Language Description	Python Data Type	`type()` Output
True or False	Boolean	`bool`
Whole Numbers	Integer	`int`
Decimal Numbers	Float	`float`
Sequences of characters including letters, numbers, formatting characters, and/or punctuation	String	`str`
Nothingness, non-existence; not the same as zero or empty	None	`NoneType`

Why do data types matter?

Different data objects (e.g. some text, a number, or a list of things) are given specific data types so that computers know how to use, manipulate, and store them. A Python program will get confused if you tell it to add `'one'`, which is text, and `2`, which is a number. People can make the mental leap that `'one'` also means the numeric value `1`, but a computer program cannot make this leap on its own. It can sometimes be difficult to know the type of a data object just by looking at it. Luckily, there is a built-in function that tells us the type of any argument we pass it: the `type()` function. Remember, functions always have parentheses after their name and functions' inputs always go inside of those parentheses. Try calling `type()` in the cells below.

In [ ]:

```
boolean_variable = True
type(boolean_variable)
```

In [ ]:

```
integer_variable = 3
type(integer_variable)
```

In [ ]:

```
integer_variable + integer_variable
```

In [ ]:

```
string_variable = 'This is a string.'
type(string_variable)
```

In [ ]:

```
string_variable + string_variable
```

Even when given the same exact command, Python can perform different operations depending on the type of data that it is working with. When told to add two numbers together, Python knows to perform addition. However, when told to add two strings together, Python knows that addition is not possible, and instead concatenates them.

In [ ]:

```
integer_variable + string_variable
```

If Python is not sure how two data types relate to each other for a given operation, it will throw a `TypeError`, telling you that it doesn't know which operation to perform. Check your data types using the `type()` function and make sure that they are what you expected.

Exercise: In the code cell below, assign a value to a variable, then use the `type()` function to output the data type for that variable.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
item = 'some text'
type(item)
```

Exercise: In the code cell below, use the variables that we created above and type a period after them (ex: `integer.`), then hit `tab` to view the available methods. Notice the differences in which methods are available to each data type.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
boolean_variable.
integer_variable.
string_variable.
```

2.4.2. Numbers (Integers and Floats) and Arithmetic Operators

References:

- TutorialsPoint: Python - Basic Operators (https://www.tutorialspoint.com/python/python_basic_operators.htm)

The difference between integers and floats is the decimal point, which is sometimes called a floating point. Integers are whole numbers and do not include a decimal point. Even in a number like like `2.0`, the presence of a decimal makes it a float. However, Python treats equivalent numbers as equal even if they are of different types.

You can use integers and floats together in arithmetic expressions.

In [ ]:

```
2.5 + 10
```

> NOTE: There is a difference between Python 2.7 (which you may have on your work computer) and 3.5 (which we are using in this class) when doing division with integers.
>
> - Python 2.7: Division operations involving only integers will result in a rounded down integer.
>   - `2/5 = 0`
> - Python 3.5: Division operations involving only integers will result in a float.
>   - `2/5 = 0.4`
>
> You can get Python 2.7 to output the float result by making one of the numbers in your expression a float (i.e. `2/5.0`)

Just like any data value, we can assign numbers explicitly to variables or create them with expressions.

In [ ]:

```
side_of_square = 5

area_of_square = side_of_square * side_of_square
area_of_square
```

Exercise: Write an expression with both integers and floats, and assign the result to a variable.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
result = (1 + 50.7 - 2.5) * 3
result
```

2.4.2.1. Casting

We can convert other data types to integers and floats using the `int()` and `float()` functions. Using these functions to change a value's data type is called casting. Other data types have their own functions for casting data objects. In the code cell below, notice how we have to cast the string `'4'` to an integer so that we can calculate the perimeter of a triangle.

In [ ]:

```
side = '4'

side = int(side)

perimeter = side + side + side
perimeter
```

Instructor Guidance: Optionally, comment out the integer casting to demonstrate what would go wrong if you tried the addition with `side` as a string.

Exercise: Run the following code cell. What is causing the error, and how can we change the code so it runs without throwing an error?

In [ ]:

```
my_num = '100.5'
my_num = int(my_num)
my_num
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##

# No code required
# Error cause: '100.5' must be cast to a float, not an int, because of the decimal point
# Fix: Change 'int' to 'float'
# Note: Once the value is a float, you can then cast to an int if you want
```

2.4.2.2. Arithmetic Operators

Operator	Operation
**	exponent
*	multiplication
/	division
//	floor division - rounds down the result of division
%	modulus - returns only the remainder of division
+	addition
-	subtraction

View the arithmetic expressions in the cells below. Note that each `print()` statement below contains two elements: a string, and the expression itself. The arithmetic expression is evaluated and its result is printed next to the string.

In [ ]:

```
print('10.2 + 0.8 =', 10.2 + 0.8)
print('4 - 6 =', 4 - 6)
print('8 * 2 =', 8 * 2)
print('10 / 3 =', 10 / 3)
```

In [ ]:

```
print('10 // 3 =', 10 // 3)
print('10 % 3 =', 10 % 3)
print('2 ** 3 =', 2 ** 3)
```

Python will evaluate arithmetic expressions in the order P,E,MD,AS: parentheses, exponents, multiplication and division (floor division, modulus), then addition and subtraction. For groupings of multiplication and division, as well as for groupings of addition and subtraction, the order is based on position and works from left to right. Work through the formula/expression below to understand how it produces the result, then run it to check if you were right.

In [ ]:

```
3 - 36 / (2 * 3) ** 2
```

Exercise: Write a new arithmetic expression in the code cell below. Include a string-to-number casting function.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_num = '100'

result = int(my_num) // 6
result
```

2.4.2.3. Assignment Operators

In Python, a single equals sign (`=`) is used to assign values to variables. The assignment operators below are used to adjust and assign values simultaneously. These operators are really just abbreviations of longer equations. See the examples below, then create your own.

Operator	Operation
+=	adds right operand to the left operand, then assigns the result to left operand
-=	subtracts right operand from the left operand, then assigns the result to left operand
*=	multiply right operand by the left operand, then assigns the result to left operand

In [ ]:

```
n = 100
n += 1
n
```

Above, the `+=` adds `1` to `n` and then resets `n` to that new value (in this case `101`). The above is equivalent to the code below, where we add `1` to `n` explicitly and assign the result back to `n`.

In [ ]:

```
n = 100
n = n + 1
n
```

Exercise: Write your own equation using an assignment operator below.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
n = 4
n *= 3
n
```

Instructor Guidance: Any mathematical operator can be applied as an assignment operator, even modulus or floor division.

2.4.3. Strings

References:

- Python: Strings (https://docs.python.org/3.4/tutorial/introduction.html#strings)
- Python: string - Common string operations (https://docs.python.org/3.4/library/string.html#module-string)
- Wikipedia: ASCII (https://simple.wikipedia.org/wiki/ASCII)

Strings are immutable sequences of zero or more characters. In Python 3, strings can include extended character sets such as Unicode. Example Unicode characters are shown below. Strings are always wrapped in quotation marks. We can use single quotes (`''`), double quotes (`""`), or three sets of quotation marks often called triple quotes (`"""` `"""`).

2.4.3.1. Creating Strings

Use a set of single, double, or triple quotation marks to create a string.

In [ ]:

```
empty_string_2 = ''
empty_string_2
```

In [ ]:

```
first_name = 'George'
first_name
```

In [ ]:

```
last_name = "Washington"
last_name
```

If you try and write multi-line strings using single or double quotes, you will get an error telling you that it did not expect the end of line: it was expecting a closing quote on that same line, as you can see below.

In [ ]:

```
nce_address = 'NATIONAL GEOSPATIAL-INTELLIGENCE AGENCY
7500 GEOINT Drive
Springfield, Virginia 22150
571-557-5400'

print(nce_address)
```

Triple quotes allow you to define a string that spans multiple lines. Note how whitespace (new lines, spaces, and tabs) is included as part of the string and the formatting is preserved.

In [ ]:

```
ncw_address = '''NATIONAL GEOSPATIAL-INTELLIGENCE AGENCY
3200 S 2nd St
St. Louis, Missouri 63118
(571) 557-5400'''

print(ncw_address)
```

We also create strings by casting other data types using the `str()` function.

In [ ]:

```
str(3.1415)
```

Exercise: Create a string in the cell below and store it into a variable. Then, print the variable.

In [ ]:

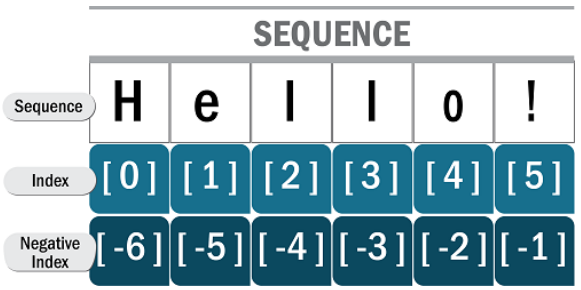
```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
s = "This is a string because it's wrapped in quotes"
print(s)
```

2.4.3.2. Indexing and Slicing a String

Strings in Python are treated as sequences of characters. So Python assigns numeric indexes to each item, starting from zero. Each index also has a corresponding negative index, which the graphic below illustrates. We can access individual characters or substrings of characters (called slices) using index numbers.



In [ ]:

```
sample_string = 'four score and seven years ago'
sample_string[1]
```

Notice what happens if we try to access an index that is not contained in our sequence.

In [ ]:

```
sample_string[100]
```

Slicing

Slicing is similar to indexing except you are accessing multiple items instead of a single element. A slice can be of any size, and to create one we need to specify its start and stop indexes.

Syntax	Description
`string_variable[i]`	Returns the character at index `i` from the string
`string_variable[start:stop]`	Returns the characters from `start` up to but not including `stop`

Below, we access all characters from index `5` up to (but not including) index `10`.

In [ ]:

```
sample_string[5:10]
```

Below, we get all characters from index `0` up to index `4`. If you do not specify a start index, it is assumed to be `0`.

In [ ]:

```
sample_string[:4]
```

If you do not specify a stop index, the slice will include everything from the start index up to (and including) the end of the string. Below, we access all characters from index `15` onward.

In [ ]:

```
sample_string[-3:]
```

Exercise: Create a string in the cell below and save it to a variable. Then, print a single character and a slice of that string.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
s = 'This string includes some numbers: 1, 2, 700'

s_slice = s[-3:]
s_index = s[8]

print(s_slice, s_index)
```

2.4.3.3. Methods and Operations

String methods are functions that can be applied to manipulate strings. These methods all return modified copies of the original string and leave the original string unaltered.

The `.upper()` method produces a new string where all alphabetic characters are upper case. The `.lower()` method produces a new string where all alphabetic characters are lower case.

Syntax	Description
`string_variable.upper()`	Returns an upper case string
`string_variable.lower()`	Returns a lower case string

In [ ]:

```
greeting = 'hello'
greeting.upper()
```

In [ ]:

```
farewell = 'GOODBYE'
farewell.lower()
```

Note that neither of these original values are changed, as we see below:

In [ ]:

```
print(greeting, farewell)
```

If you would like to hold on to the new version instead of just displaying the altered copy, you have to save it to variable. If you save it to the same variable name, we call that "overwriting the variable."

In [ ]:

```
# Saving to new variable:
new_greeting = greeting.upper()

# Overwriting the variable:
farewell = farewell.lower()
```

The `.strip()` method removes leading and trailing characters from a string. If you do not tell it what to strip, `.strip()` will just remove whitespace characters (spaces, newlines, etc.). The `.find()` method looks for a substring and returns the index location of the first character of the first instance of the substring if it finds it; otherwise it returns `-1`. The `.replace()` method returns a copy of the original string where every instance of a specified substring (the first argument) is replaced with a replacement string (the second argument).

Syntax	Description
`string_variable.strip(chars)`	Removes any character found in `chars` from the beginning and end of a string
`string_variable.find(substring)`	Searches a string for `substring` and returns the index location of first character of the first instance of `substring` or `-1` if substring is not found
`string_variable.replace(substring, new)`	Replaces all occurrences of `substring` with `new`

In [ ]:

```
spacey_str = '       1234032E    '
spacey_str.strip()
```

In [ ]:

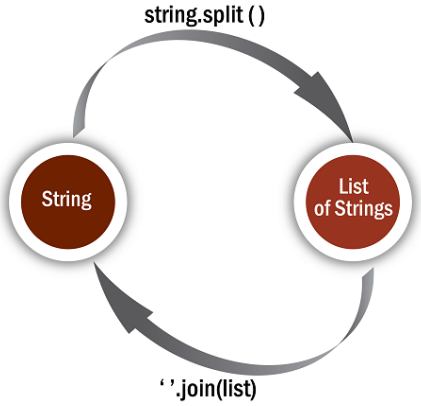
```
states = 'Alabama, Alaska, Arizona, Arkansas...'
states.find('Arizona')
```

In [ ]:

```
date_str = '01/01/2000'
date_str.replace('/', '-')
```

The `.split()` method creates a list of strings by splitting the original string on a delimiter that we specify.

The delimiter can be a single character or multiple characters, and the default behavior if no argument is passed is to split on whitespace characters. The `.join()` method does the inverse of `.split()`. It creates a single string from a list of strings by joining them around a delimiter. The relationship between the `.split()` and `.join()` functions can be described by this graphic:



Syntax	Description
`string_variable.split(substring)`	Splits a string into a list of strings using `substring` as a delimiter
`string_variable.join(list_of_strings)`	Creates one string from a list of strings by joining them with `string_variable`

In [ ]:

```
sentence = 'A bunch of words separated by spaces'
sentence.split()
```

In [ ]:

```
date = ['05', '07', '2018']
'/'.join(date)
```

Exercise: Take the dates below and reformat them both to the following format: `mm/dd/yyyy`.

In [ ]:

```
date_1 = '01-02-2018'
date_2 = '--12.18.1983--'
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
date_1 = date_1.replace('-', '/')
date_2 = date_2.strip('-').replace('.', '/')

date_1, date_2
```

We can concatenate, or chain together, two strings using the plus sign (+). We can repeat strings with the multiplication sign. Just like in math, multiplication is just repeated addition.

In [ ]:

```
f_name = 'G.'
l_name = 'van Rossum'

f_name + ' ' + l_name
```

In [ ]:

```
'-' * 50
```

Exercise: Using slicing and string concatenation (+) reformat the dates below in the following format: `mm/dd/yyyy` .

In [ ]:

```
date_3 = '04052016'
date_4 = '1776,07.04'
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
date_3 = date_3[:2] + '/' + date_3[2:4] + '/' + date_3[4:]
date_4 = date_4[-5:-3] + '/' + date_4[-2:] + '/' + date_4[:4]

date_3, date_4
```

2.4.4. The None Type

References:

- W3 Schools: None (https://www.w3schools.com/python/ref_keyword_none.asp)

The None data type only has one possible value, `None`, which is used to indicate the absence of data. `None` is not the same as zero, an empty list, or an empty string. The `None` type is used to indicate an empty cell in a table or spreadsheet. Other uses of `None` will become evident when we begin to create functions in this course.

In [ ]:

```
type(None)
```

`None` is not equivalent to empty data structures such as the empty string (`''`), or the empty list (`[]`), nor is it equivalent to zero (`0`).

In [ ]:

```
None == 0
```

In [ ]:

```
None == ''
```

2.5. Guided Exercise: Solving a Practical Exercise

Throughout this week, at the end of every lesson and guided exercise there will be multiple practical exercises to practice applying the lesson content.

Guidelines to solving Practical Exercises:

1. Read the question carefully and discuss with a neighbor what the problem is asking.
2. Pseudocode! Without any code, write out the steps that you should take to solve this problem.
3. Convert the pseudocode steps into actual code.
4. Compare with either input/output tables or neighbors to confirm that your code is giving you the correct answer

Remember: Coding is not a one person mission, so talk to your neighbors!

Practical Exercise Example: Write code to covert Celsius temperatures to Fahrenheit. The formula is as follows:

```
fahrenheit = (celsius * (9/5)) + 32
```

Example Input	Expected Output
100	212

Example Input	Expected Output
0	32
-40	-40

> NOTE: The input/output table is meant to show expected outputs for given example inputs. Use it to check your code.

Pseudocode:

- Input: A number of degrees in Celsius
- Output: A number of degrees in Fahrenheit

1. Use the formula given to do the conversion
2. Print or display the result

In [ ]:

```
celsius = 100

fahrenheit = (celsius * (9/5)) + 32
fahrenheit
```

Now that we have the final answer, we can refer back to the input/output table in order to check that we have the correct answer. Then, if those outputs match, try the other outputs to make sure your code is working for a variety of inputs.

In [ ]:

```
celsius = 0

fahrenheit = (celsius * (9/5)) + 32
fahrenheit
```

After trying all the inputs, if we keep getting the expected output then we have successfully solved the problem being asked.

Instructor Guidance: Refer back to Lesson 1 and relate the four steps of problem-solving using Computational Thinking (Decomposition, Pattern Recognition, Abstraction, & Algorithm Design) as appropriate throughout these exercises.

Instructor Guidance: The practical exercises deemed most important for this lesson, due to content and/or a cumulative result, which should be completed first in the interest of maximum training value in relation to time are Practical Exercises 1-5. Ensure you go over the exercise solutions and (as necessary) the processes to arrive at the solutions with the students.

Instructor Guidance: **Follow-up questions are designed to be asked by the facilitators individually as each student completes the task and has it looked at by a facilitator.**

2.6. Practical Exercises

2.6.1. Practical Exercise 1: Casting

References:

- PythonSpot: Datatype casting (https://pythonspot.com/datatype-casting/)

Cast the following two values into floats and then add them together.

In [ ]:

```
costs_2016 = '912765.56'
costs_2017 = '864920.21'
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

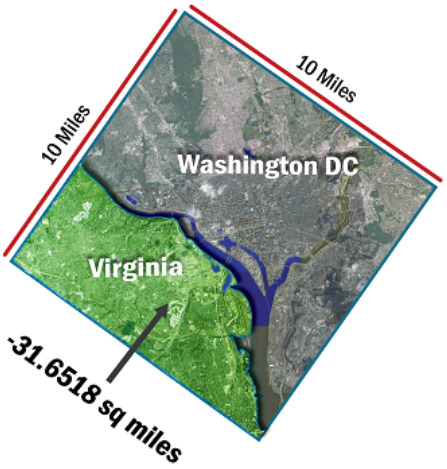
In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
float(costs_2016) + float(costs_2017)
```

2.6.2. Practical Exercise 2: How Big is Washington, D.C.?

In accordance with the Residence Act of 1790, Washington, D.C. was declared to be a square diamond 10 miles wide by 10 miles wide using land ceded from Maryland and Virginia. However, in the 1840s the Virginia state government said, "Give us our land back! We want those citizens to vote in our commonwealth." and Virginia got 31.6518 square miles of land back from the nation's capital.

How big is D.C. today? Assign numeric values to variables before doing calculations. Use `print()` to give your final answer as a full sentence.



HINT: Two ways to find the area of a square are: (`length_of_a_side ** 2`) or (`length_of_a_side * length_of_a_side`)

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
side = 10.0

old_area = side * side

VA_land = 31.6518

new_area = old_area - VA_land

print('The area of Washington, D.C. is', new_area, 'square miles.')
```

2.6.3. Practical Exercise 3: Cleaning Data

In the code cell below, there are four string values representing coordinates in Degrees-Minutes-Seconds (DMS) format. Standardize the format of each the coordinates below. Each example requires different code to fix, so treat each coordinate seperately. Do not attempt to write one script to reformat all of the examples at once. The desired formats are as follows:

- longitude: `DDDMMSS + E or W` (Example: `1104523E`)
- latitude: `DDMMSS + N or S` (Example: `432602N`)

> NOTE: Latitude ranges from 0 to 90 degrees, and longitude ranges from 0 to 180 degrees.

Do not attempt to write one script to reformat all of the examples at once, but rather treat each case separately. For example, for the input `c2 = ' 1200001E '`, one acceptable answer is `c2.strip()`, even though this same code wouldn't fix any of the other examples.

Example Input	Expected Output
'n435421'	'435421N'
' 1200001E '	'1200001E'
'781130-(S) '	'781130S'
'e 093.48.71'	'0934871E'

In [ ]:

```
c1 = 'n435421'
c2 = ' 1200001E '
c3 = '781130-(S) '
c4 = 'e 093.48.71'
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
c1 = c1[1:] + c1[0].upper()

c2 = c2.strip()

c3 = c3[:c3.find('-')] + c3.strip()[-2]

c4 = c4.replace('.', '')[2:] + c4[0].upper()

print(c1, c2, c3, c4)
```

2.6.4. Practical Exercise 4: Radio Signal Travel Time

Calculate to see how long, in hours, a radio signal (which travels at the speed of light) takes to reach Earth from the Voyager 1 spacecraft. First use comments in the code cell below to write out the steps of calculation or unit conversion that are needed to find the answer. Then write the code to execute these steps. Make your answer print out as a complete sentence. HINT: Start with the speed of light in kilometers per hour to write less code.

Helpful values	
Speed of light in kilometers per hour	`1079252848.8`
Voyager 1 distance from Earth in kilometers	`20836509700`

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
voyager_1_dist_in_km = 20836509700

c = 1079252848.8    #km/hr


message_time = voyager_1_dist_in_km / c


print('It takes', message_time, 'hours for a message to reach Earth')
```

2.6.5. Practical Exercise 5: Restaurant Bill

Calculate the overall cost of a restaurant bill given the pre-tax total, tax rate, and tip percentage. Apply a tip to the post-tax cost of the meal to get your final output.

Example Input	Expected Output

`bill = 44.50`
`tax = .0675`
`tip = .15`
	`54.63`

`bill = 100`
`tax = .09`
`tip = .20`
	`130.80`

`bill = 567.34`
`tax = .0525`
`tip = .20`
	`716.55`

HINT: Multiply a value by `(1 + percentage)` to get the sum of the original value plus that percentage of the original value.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
meal = 567.34
tax = 0.0525
tip = 0.20

meal = meal + (meal * tax)
total = meal + (meal * tip)

print('The total cost of the meal is $' + str(round(total,2))+ '.')
```

2.6.6. Practical Exercise 6: Hours, Minutes, and Seconds

Problem 1: Given a number of seconds (int or float), output the equivalent amount of time in minutes and seconds as a string.

Example Input	Expected Output
`seconds = 257`	`'4 minutes and 17 seconds'`
`seconds = 7904`	`'131 minutes and 44 seconds'`
`seconds = 1000`	`'16 minutes and 40 seconds'`
`seconds = 123123`	`'2052 minutes and 3 seconds'`

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
seconds = 123123

m = seconds // 60
s = seconds % 60

print(str(m) + ' minutes and ' + str(s) + ' seconds')
```

Problem 2: Incorporate the number of hours into the output as well so that the output is in the format `'X hours, Y minutes, and Z seconds'`.

Example Input	Expected Output

Example Input	Expected Output
seconds = 257	'0 hours, 4 minutes, and 17 seconds'
seconds = 7904	'2 hours, 11 minutes, and 44 seconds'
seconds = 1000	'0 hours, 16 minutes, and 40 seconds'
seconds = 123123	'34 hours, 12 minutes, and 3 seconds'

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
seconds = 7904

h = seconds // 3600
m = (seconds % 3600) // 60
s = seconds % 60


print(str(h) + ' hours, '+ str(m) + ' minutes and ' + str(s) + ' seconds')
```

2.6.7. Practical Exercise 7: String Slicing Exercises

Problem 1: Using slicing and concatenation, print out James Bond's name as 'Bond, James Bond.'

In [ ]:

```
name = 'James Bond'
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##

name = 'James Bond'

last_name = name[6:]

answer = last_name + ', ' + name
print(answer)
```

Problem 2: Slice the following text to just get the mission, start date, location, and coordinates (without the labels).

HINT: Use the `.find()` string function to search for specific text and find the index location, then use the location as a start or stop index in a slice.

In [ ]:

```
mission_text = '''Your mission, should you choose to accept it:
You must infiltrate the terrorist group SPECTRE and retrieve the ancient artifact

Start Date: 07/18/2025
Location: Hidden Base in the Pacific Ocean
Coordinates: 15.121054, 177.902503
This message will self-destruct in five seconds'''
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
mission_start = mission_text.find(':')
date_start = mission_text.find('Start Date: ')
location_start = mission_text.find('Location: ')
coord_start = mission_text.find('Coordinates: ')
ending = mission_text.find('This message will self-destruct in five seconds')

mission = mission_text[mission_start+2:date_start]
date = mission_text[date_start+12:location_start]
location = mission_text[location_start + len('Location: '):coord_start]
coords = mission_text[coord_start + len('Coordinates: '):ending]

print(mission)
print(date)
print(location)
print(coords)
```

2.6.8. [Challenge] Practical Exercise 8: Slicing with the Step Parameter

There is an optional parameter in slicing which controls what is called the *step*. By default the step is 1, so it captures every character. If you set the step to 2, it would get every other character. A step of 3 would get every third, and so on. Run the code cells below to see some examples of using step in a slice.

In [ ]:

```
text = 't h i s   i s   a   s t r i n g   e x a m p l e'
```

In [ ]:

```
text[::2]
```

In [ ]:

```
text[::3]
```

In [ ]:

```
text[::4]
```

In [ ]:

```
text[::-1]
```

Problem 1: The message in the following code cell was sent by a fellow field agent. Decode it by slicing every 4th character using the step parameter.

In [ ]:

```
message = '075303557674 dwxpgsjldxeetsgavafshsyedfs dtdhthieddelvfhp'
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
message[::4]
```

Problem 2: Use the step parameter to reverse your name, and fix the capitalization.

Example Input	Expexted Output
'Jimmy'	'Ymmij'
'Sacha'	'Ahcas'
'Doris'	'Sirod'

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_name = 'Jimmy'

my_name[::-1].title()
```

2.6.9. [Challenge] Practical Exercise 9: Processing Markup Languages

Below is data in a fake markup language. The tags (e.g. `<a>` , `</a>`) make it possible to programmatically find and extract elements of the data. Using a combination of string methods, we can write a script that will analyze this data. We want to process the data into a format we can more easily use. Here's the sample data we'll be using:

```
s1 = '<a>USGS_datastore_5A</a> <b>Democratic Republic of Congo (DRC)</b> <c>Moun
t Nyiragongo</c> <d>3470 meters</d>'
s2 = '<a>USGS_datastore_1N</a> <b>United States of America (USA)</b> <c>Mount Wh
itney</c> <d>4421 meters</d>'
s3 = '<a>USGS_datastore_8C</a> <b>French Republic (FR)</b> <c>Mont Blanc</c> <d>
4808 meters</d>'
```

Problem 1: Write a script that takes a letter representing a tag (e.g. `'a'`) and constructs the full open tag string (`'<a>'`) and close tag string (`'</a>'`).

Example Input	Expected Output
`tag = 'a'`	`open_tag = '<a>'` `close_tag = '</a>'`
`tag = 'b'`	`open_tag = '<b>'` `close_tag = '</b>'`
`tag = 'd'`	`open_tag = '<d>'` `close_tag = '</d>'`

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
tag = 'a'

open_tag = '<' + tag + '>'
close_tag = '</' + tag + '>'

print(open_tag, close_tag)
```

Problem 2: Write a script that takes a string like the ones in our sample data and a letter representing a tag (e.g. `'a'`), and outputs the data value stored inside the tag in the input string.

HINT: Use the code you wrote above to construct the full tags, then use `.find()` to find the indexes you want to use in your slice.

Example Input	Expected Output
`data = s1` `tag = 'a'`	`'USGS_datastore_5A'`
`data = s2` `tag = 'b'`	`'United States of America (USA)'`
`data = s3` `tag = 'd'`	`'4808 meters'`

In [ ]:

```
s1 = '<a>USGS_datastore_5A</a> <b>Democratic Republic of Congo (DRC)</b> <c>Mount Nyirago
s2 = '<a>USGS_datastore_1N</a> <b>United States of America (USA)</b> <c>Mount Whitney</c>
s3 = '<a>USGS_datastore_8C</a> <b>French Republic (FR)</b> <c>Mont Blanc</c> <d>4808 mete
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
data = s1
tag = 'a'

open_tag = '<' + tag + '>'
close_tag = '</' + tag + '>'

start = data.find(open_tag)
end = data.find(close_tag)

output = data[start+3:end]
output
```

2.7. Appendix

Basic Data Types

Plain Language Description	Python Data Type	`type()` Output

Plain Language Description	Python Data Type	`type()` Output
True or False	Boolean	`bool`
Whole Numbers	Integer	`int`
Decimal Numbers	Float	`float`
Sequences of characters including letters, numbers, formatting characters, and punctuation	String	`str`
Nothingness, non-existence; not the same as zero or empty	None	`NoneType`

Arithmetic Operators

Operator	Operation
`**`	exponent
`*`	multiplication
`/`	division
`//`	floor division - rounds down the result of division
`%`	modulus - returns only the remainder of division
`+`	addition
`-`	subtraction

Assignment Operators

Operator	Operation
`+=`	adds right operand to the left operand, then assigns the result to left operand
`-=`	subtracts right operand from the left operand, then assigns the result to left operand
`*=`	multiply right operand by the left operand, then assigns the result to left operand

String Methods and Operations

Syntax	Description
`string_variable[i]`	Returns the character at index `i` from the string
`string_variable[start:stop]`	Returns the characters from `start` up to but not including `stop`
`string_variable.upper()`	Returns an upper case string
`string_variable.lower()`	Returns a lower case string
`string_variable.strip(chars)`	Removes any character found in `chars` from the beginning and end of a string
`string_variable.find(substring)`	Searches a string for `substring` and returns the index location of first character of the first instance of `substring` or `-1` if substring is not found

Syntax	Description
`string_variable.replace(substring, new)`	Replaces all occurrences of `substring` with `new`
`string_variable.split(substring)`	Splits a string into a list of strings using `substring` as a delimiter
`string_variable.join(list_of_strings)`	Creates one string from a list of strings by joining them with `string_variable`
`s1 + s2`	Concatenate `s1` and `s2`
`s1 * n`	Repeat `s1` `n` times

Casting Functions

Syntax	Description
`int(x)`	Returns an integer interpretation of `x`, if such an interpretation is possible
`float(x)`	Returns a float interpretation of `x`, if such an interpretation is possible
`str(x)`	Returns a string interpretation of `x`, if such an interpretation is possible
`bool(x)`	Returns a boolean interpretation of `x`, if such an interpretation is possible