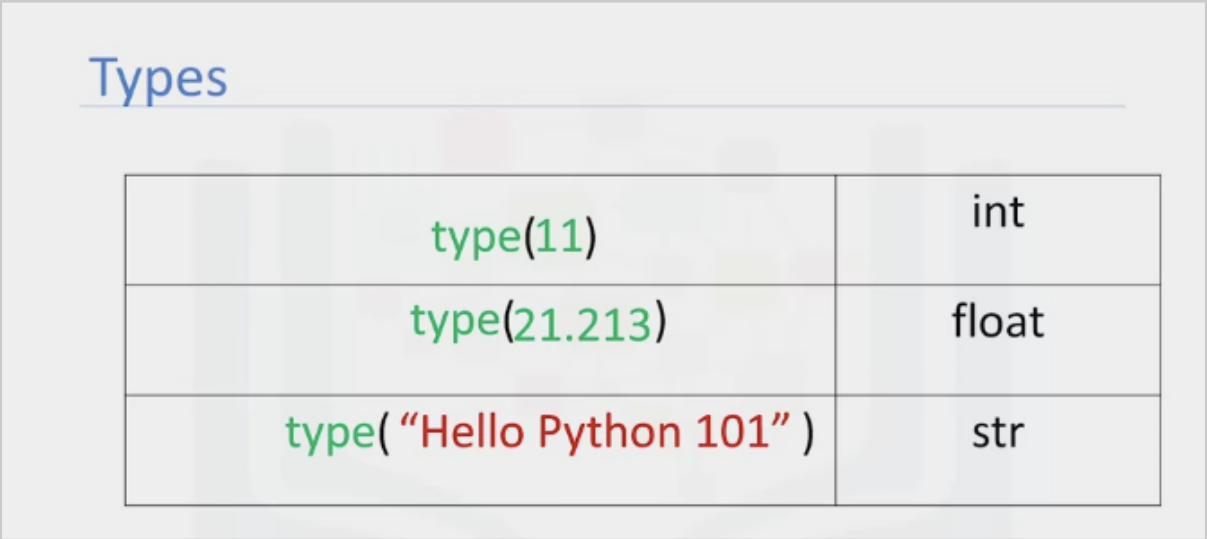
Week 1

#Data Science/4 - Python for Data Science, AI & Development#

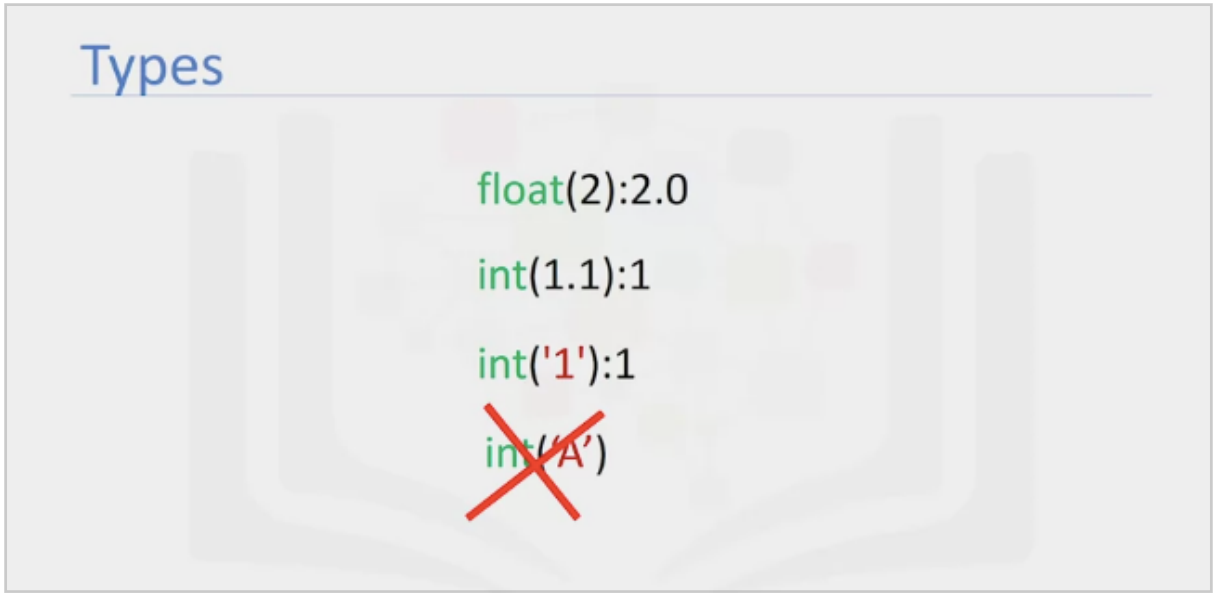
Types

Types

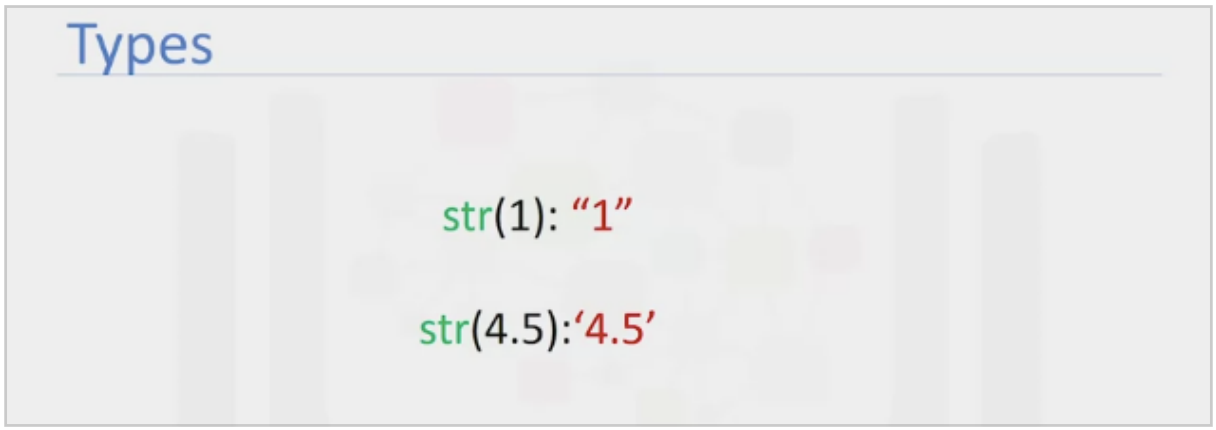
A type is how Python represents different types of data. In this video, we will discuss some widely used types in Python. You can have different types in Python.
They can be integers like 11, real numbers like 21.213, they can even be words. Integers, real numbers, and words can be expressed as different data types. The following chart summarizes three data types for the last examples.



The first column indicates the expression. The second column indicates the data type. We can see the actual data type in Python by using the type command. We can have int, which stands for an integer and float that stands for float, essentially a real number. The type string is a sequence of characters. Here are some integers. **Integers** can be negative or positive. It should be noted that there is a finite range of integers but it is quite large. **Floats** are real numbers. They include the integers but also numbers in between the integers. Consider the numbers between 0 and 1. We can select numbers in between them. These numbers are floats. Similarly, consider the numbers between 0.5 and 0.6. We can select numbers in between them. These are floats as well. We can continue the process zooming in for different numbers. Of course there is a limit but it is quite small. You can change the type of the expression in Python, this is called **typecasting**. You can **convert** an int to a float. For example, you can convert or cast the integer 2 to a float 2.0. Nothing really changes, **if you cast a float to an integer, you must be careful**.



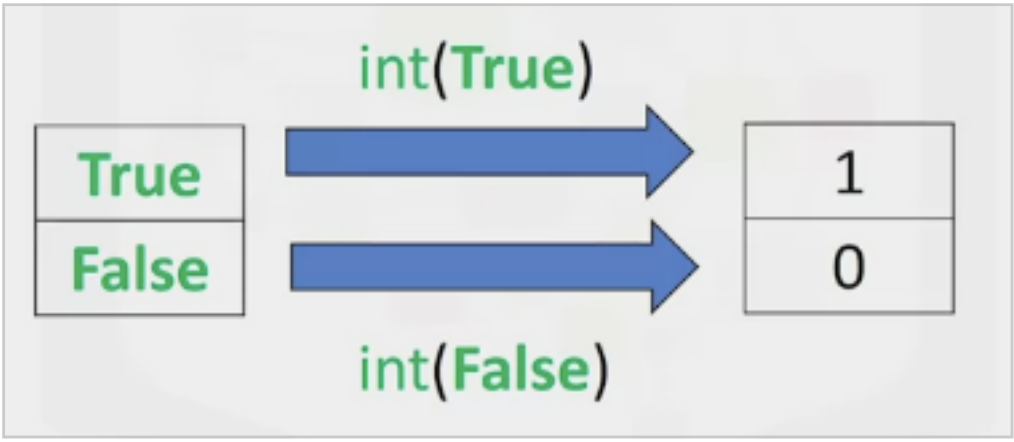
For example, if you cast the float 1.1 to 1, you will lose some information. If a string contains an integer value, you can convert it to int. If we convert a string that contains a non-integer value, we get an error. Check out more examples in the lab.



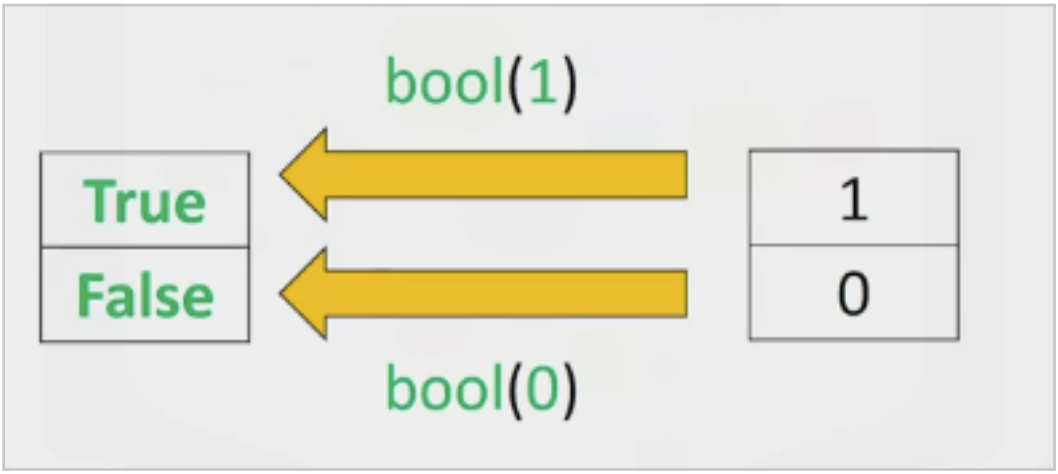
You can convert an int to a string or a float to a string. Boolean is another important type in Python.

A Boolean can take on two values. The first value is True, just remember we use an uppercase T. Boolean values can also be False with an uppercase F. Using the type

command on a Boolean value, we obtain the term bool. This is short for Boolean, if we cast a Boolean True to an integer or float, we will get a 1. If we cast a Boolean False to an integer or float, we get a 0.

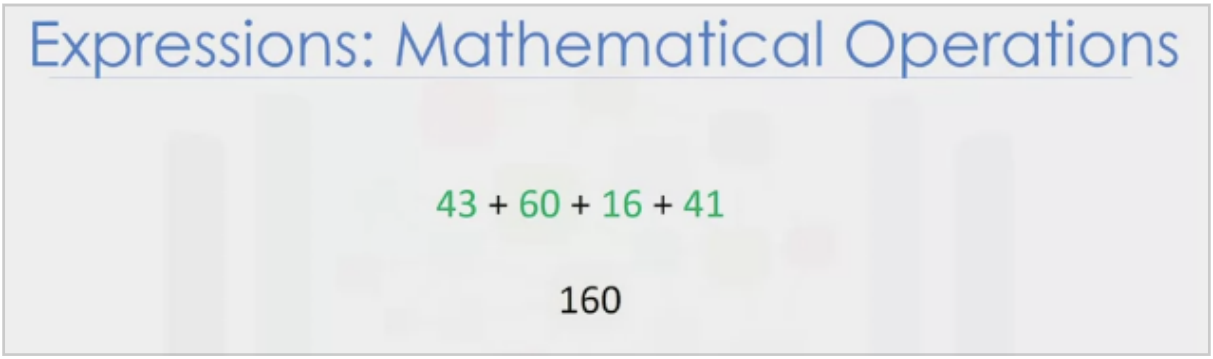


If you cast a 1 to a Boolean, you get a True. Similarly, if you cast a 0 to a Boolean, you get a False.



Expressions and Variables

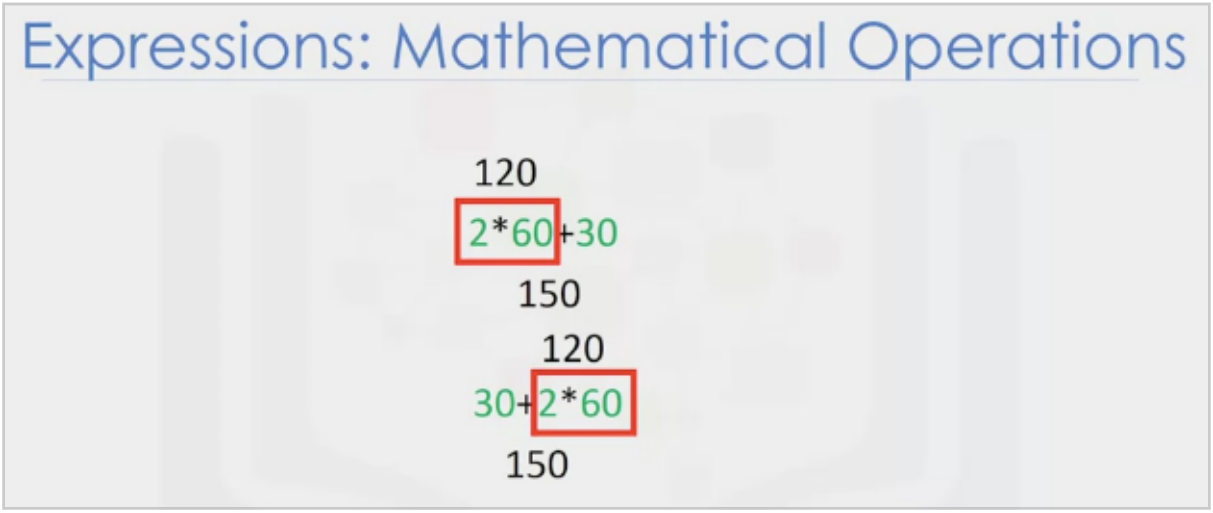
In this video, we will cover expressions and variables. **Expressions** describe a type of operation the computers perform. Expressions are operations the python performs. For example, basic **arithmetic operations** like adding multiple numbers.



The result in this case is 160. We call the **numbers operands**, and the **math symbols** in this case, addition, are called **operators**. We can perform operations such as subtraction

using the subtraction sign. In this case, the result is a negative number. We can perform multiplication operations using the asterisk. The result is 25. In this case, the operands are given by negative and asterisk. We can also perform division with the forward slash- 25 / 5 is 5.0; 25 / 6 is approximately 4.167.

In Python 3, the version we will be using in this course, both will result in a float. We can use the **double slash for integer division**, where **the result is rounded**. Be aware, in some cases the results are not the same as regular division. Python follows mathematical conventions when performing mathematical expressions.



The following operations are in a different order. In both cases, Python performs multiplication, then addition, to obtain the final result. There are a lot more operations you can do with Python, check the labs for more examples. We will also be covering more complex operations throughout he course. The expressions in the parentheses are performed first. We then multiply the result by 60. The result is 1,920.

Now, let's look at **variables**.

We can use variables to store values. In this case, we assign a value of 1 to the variable my*variable using the assignment operator, i.e, the equal sign. We can then use the value somewhere else in the code by typing the exact name of the variable. We will use a colon to denote the value of the variable. We can assign a new value to my*variable using the assignment operator. We assign a value of 10. The variable now has a value of 10. The old value of the variable is not important. We can store the results of expressions. For example, we add several values and assign the result to x. X now stores the result. We can also perform operations on x and save the result to a new variable-y. Y now has a value of 2.666. We can also perform operations on x and assign the value x. The variable x now has a value: 2.666. As before, the old value of x is not important. We can use the type command in variables as well. It's good practice to use meaningful variable names; so, you don't have to keep track of what the variable is doing.

Let say, we would like to convert the number of minutes in the highlighted examples to number of hours in the following music data-set. We call the variable, that contains the total number of minutes, total*min. It's common to use the underscore to represent the start*

*of a new word. You could also use a capital letter. We call the variable that contains the total number of hours, total*hour. We can obtain the total number of hours by dividing total_min by 60. The result is approximately 2.367 hours. If we modify the value of the first variable, the value of the variable will change. The final result values change accordingly, but we do not have to modify the rest of the code.

Lab

Objectives

After completing this lab you will be able to:

- Write basic code in Python
- Work with various types of data in Python
- Convert the data from one type to another
- Use expressions and variables to perform operations

Table of content

Say "Hello" to the world in Python

When learning a new programming language, it is customary to start with an “hello world” example. As simple as it is, this one line of code will ensure that we know how to print a string in output and how to execute code within cells in a notebook.

```
# Try your first Python output
print('Hello, Python!')
```

```
Hello, Python!
```

After executing the cell above, you should see that Python prints Hello, Python!. Congratulations on running your first Python code!

What version of Python are we using?

There are two popular versions of the Python programming language in use today: Python 2 and Python 3. The Python community has decided to move on from Python 2 to Python 3, and many popular libraries have announced that they will no longer support Python 2. Since Python 3 is the future, in this course we will be using it exclusively. How do we know that our notebook is executed by a Python 3 runtime? We can look in the top-right hand corner of this notebook and see "Python 3".
We can also ask Python directly and obtain a detailed answer. Try executing the following code:

```
# Check the Python Version
import sys
print(sys.version)
```

```
3.6.6+ (default, Aug 30 2021, 19:21:08)
[GCC Apple LLVM 12.0.5 (clang-1205.0.22.11)]
```

Writing comments in Python

In addition to writing code, note that it's always a good idea to add comments to your code. It will help others understand what you were trying to accomplish (the reason why you wrote a given snippet of code). Not only does this help **other people** understand your

code, it can also serve as a reminder **to you** when you come back to it weeks or months later.
To write comments in Python, use the number symbol # before writing your comment. When you run your code, Python will ignore everything past the # on a given line.

```
# Practice on writing comments
print('Hello, Python!') # This line prints a string
# print('Hi')


Hello, Python!
```

After executing the cell above, you should notice that This line prints a string did not appear in the output, because it was a comment (and thus ignored by Python).
The second line was also not executed because `print('Hi')` was preceded by the number sign `(#)` as well! Since this isn't an explanatory comment from the programmer, but an actual line of code, we might say that the programmer *commented out* that second line of code.

Errors in Python

Everyone makes mistakes. For many types of mistakes, Python will tell you that you have made a mistake by giving you an error message. It is important to read error messages carefully to really understand where you made a mistake and how you may go about correcting it.
For example, if you spell `print` as `frint`, Python will display an error message. Give it a try:

```
# Print string as error message
frint("Hello, Python!")


---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-33-313a1769a8a5> in <module>
      1 # Print string as error message
      2
```

```
----> 3 frint("Hello, Python!")
NameError: name 'frint' is not defined
```

The error message tells you:

1. where the error occurred (more useful in large notebook cells or scripts), and
2. what kind of error it was (NameError)

Here, Python attempted to run the function `frint`, but could not determine what `frint` is since it's not a built-in function and it has not been previously defined by us either.
You'll notice that if we make a different type of mistake, by forgetting to close the string, we'll obtain a different error (i.e., a `SyntaxError`). Try it below:

```
# Try to see built-in error message
print("Hello, Python!)
```

```
  File "<ipython-input-58-f0b5a635e1a2>", line 3
    print("Hello, Python!)
                          ^
SyntaxError: EOL while scanning string literal
```

Does Python know about your error before it runs your code?
Python is what is called an interpreted language. Compiled languages examine your entire program at compile time, and are able to warn you about a whole class of errors prior to execution. In contrast, Python interprets your script line by line as it executes it. Python will stop executing the entire program when it encounters an error (unless the error is expected and handled by the programmer, a more advanced subject that we'll cover later on in this course).

Try to run the code in the cell below and see what happens:

```
# Print string and error to see the running order

print("This will be printed")
frint("This will cause an error")
print("This will NOT be printed")
```

```
This will be printed
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-59-af59af1b345d> in <module>
      2
      3 print("This will be printed")
----> 4 frint("This will cause an error")
      5 print("This will NOT be printed")
NameError: name 'frint' is not defined
```

Exercise: Your First Program

Generations of programmers have started their coding careers by simply printing "Hello, world!". You will be following in their footsteps.

In the code cell below, use the `print()` function to print out the phrase: `Hello, world!`

```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
print('Hello, world!')
```

```
Hello, world!
```

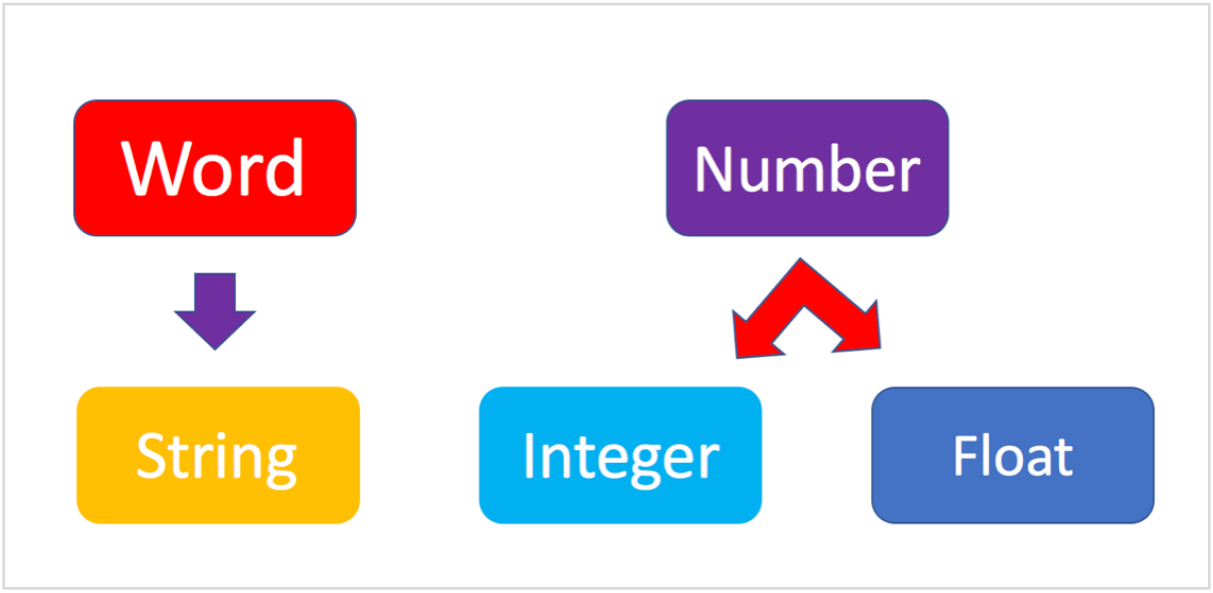
Now, let's enhance your code with a comment. In the code cell below, print out the phrase: `Hello, world!` and comment it with the phrase `Print the traditional hello world` all in one line of code.

```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
print('Hello, world!') #Print the traditional hello world
```

```
Hello, world!
```

Types of objects in Python

Python is an object-oriented language. There are many different types of objects in Python. Let's start with the most common object types: *strings*, *integers* and *floats*. Anytime you write words (text) in Python, you're using *character strings* (strings for short). The most common numbers, on the other hand, are *integers* (e.g. -1, 0, 100) and *floats*, which represent real numbers (e.g. 3.14, -42.0).



The following code cells contain some examples.

```
# Integer
11
```

```
# Float
2.14
```

```
# String
"Hello, Python 101!"
```

You can get Python to tell you the type of an expression by using the built-in `type()` function. You'll notice that Python refers to integers as `int`, floats as `float`, and character strings as `str`.

```
```

```
# Type of 12
type(12)
```

```
int
```

```
# Type of 2.14
type(2.14)
```

```
float
```

```
# Type of "Hello, Python 101!"
type("Hello, Python 101!")
```

```
str
```

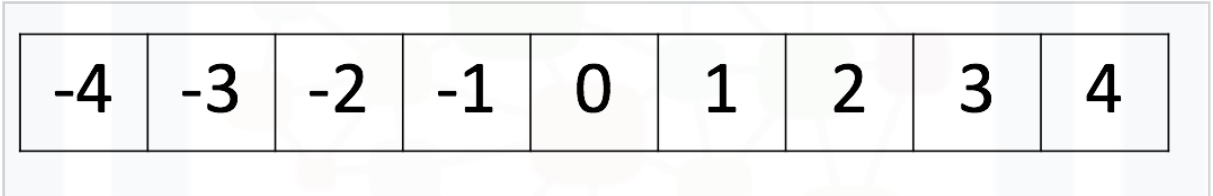
In the code cell below, use the `type()` function to check the object type of `12.0`.

```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
type(12.0)
```

```
float
```

Integers

Here are some examples of integers. Integers can be negative or positive numbers:



We can verify this is the case by using, you guessed it, the `type()` function:

```
# Print the type of -1
type(-1)
```

```
int
```

```
# Print the type of 4
type(4)
```

```
int
```

```
# Print the type of 0
type(0)
```

```
int
```

Float

Floats represent real numbers; they are a superset of integer numbers but also include "numbers with decimals". There are some limitations when it comes to machines representing real numbers, but floating point numbers are a good representation in most cases. You can learn more about the specifics of floats for your runtime environment, by checking the value of `sys.float_info`. This will also tell you what's the largest and smallest number that can be represented with them.
Once again, can test some examples with the `type()` function:

```
# Print the type of 1.0

type(1.0) # Notice that 1 is an int, and 1.0 is a float
```

```
float
```

```
# Print the type of 0.5
type(0.5)
```

```
float
```

```
# Print the type of 0.56
type(0.56)
```

```
float
```

Converting from one object type to a different object type

You can change the type of the object in Python; this is called typecasting. For example, you can convert an *integer* into a *float* (e.g. 2 to 2.0).

Let's try it:

```
# Verify that this is an integer
type(2)
```

```
int
```

Converting integers to floats

Let's cast integer 2 to float:

```
# Convert 2 to a float
float(2)
```

```
2.0
```

```
# Convert integer 2 to a float and check its type
type(float(2))
```

```
float
```

When we convert an integer into a float, we don't really change the value (i.e., the significand) of the number. However, if we cast a float into an integer, we could potentially lose some information. For example, if we cast the float 1.1 to integer we will get 1 and lose the decimal information (i.e., 0.1):

```
# Casting 1.1 to integer will result in loss of information
int(1.1)
```

```
1
```

Converting from strings to integers or floats

Sometimes, we can have a string that contains a number within it. If this is the case, we can cast that string that represents a number into an integer using `int()`:

```
# Convert a string into an integer
int('1')
```

```
1
```

But if you try to do so with a string that is not a perfect match for a number, you'll get an

error. Try the following:

```
# Convert a string into an integer with error
int('1 or 2 people')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-76-b78145d165c7> in <module>
      1 # Convert a string into an integer with error
      2
----> 3 int('1 or 2 people')
ValueError: invalid literal for int() with base 10: '1 or 2 people'
```

You can also convert strings containing floating point numbers into *float* objects:

```
# Convert the string "1.2" into a float
float('1.2')
```

```
1.2
```

Converting numbers to strings

If we can convert strings to numbers, it is only natural to assume that we can convert numbers to strings, right?

```
# Convert an integer to a string
str(1)
```

```
'1'
```

And there is no reason why we shouldn't be able to make floats into strings as well:

```
# Convert a float to a string
str(1.2)
```

```
'1.2'
```

Boolean data type

Boolean is another important type in Python. An object of type *Boolean* can take on one of two values: `True` or `False`:

```
# Value true
True
```

```
True
```

Notice that the value `True` has an uppercase "T". The same is true for `False` (i.e. you must use the uppercase "F").

```
# Value false
False
```

```
False
```

When you ask Python to display the type of a boolean object it will show `bool` which stands for *boolean*:

```
# Type of True
type(True)
```

```
bool
```

```
# Type of False
type(False)
```

```
bool
```

We can cast boolean objects to other data types. If we cast a boolean with a value of `True` to an integer or float we will get a one. If we cast a boolean with a value of `False` to an integer or float we will get a zero. Similarly, if we cast a 1 to a Boolean, you get a `True`. And if we cast a 0 to a Boolean we will get a `False`. Let's give it a try:

```
# Convert True to int
int(True)
```

```
1
```

```
# Convert 1 to boolean
bool(1)
```

```
True
```

```
# Convert 0 to boolean
bool(0)
```

```
False
```

```
# Convert True to float
float(True)
```

```
1.0
```

Exercise: Types

What is the data type of the result of: `6 / 2`?

```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
6/2
```

```
3.0
```

What is the type of the result of: `6 // 2`? (Note the double slash `//`.)

```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
6//2
```

```
3
```

```
type(6//2) # int, as the double slashes stand for integer division
```

Expression and Variables

Expression

Expressions in Python can include operations among compatible types (e.g., integers and floats). For example, basic arithmetic operations like adding multiple numbers:

```
# Addition operation expression
43 + 60 + 16 + 41
```

```
160
```

Expressions in Python can include operations among compatible types (e.g., integers and floats). For example, basic arithmetic operations like adding multiple numbers:

```
# Subtraction operation expression
50 - 60
```

```
-10
```

We can do multiplication using an asterisk:

```
# Multiplication operation expression
5 * 5
```

```
25
```

We can also perform division with the forward slash:

```
# Division operation expression
25 / 5
```

```
5.0
```

```
# Division operation expression

25 / 6
```

```
4.166666666666667
```

As seen in the quiz above, we can use the double slash for integer division, where the result is rounded down to the nearest integer:

```
# Integer division operation expression
25 // 5
```

```
5
```

```
# Integer division operation expression
25 // 6
```

```
4
```

Exercise: Expression

Let's write an expression that calculates how many hours there are in 160 minutes:

```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
160/60
```

```
2.6666666666666665
```

```
160/60
# Or
160//60
```

Python follows well accepted mathematical conventions when evaluating mathematical expressions. In the following example, Python adds 30 to the result of the multiplication (i.e., 120).

```
# Mathematical expression
30 + 2 * 60
```

```
150
```

And just like mathematics, expressions enclosed in parentheses have priority. So the following multiplies 32 by 60.

```
# Mathematical expression
(30 + 2) * 60
```

```
1920
```

Variables

Just like with most programming languages, we can store values in *variables*, so we can use them later on. For example:

```
# Store value into variable
x = 43 + 60 + 16 + 41
```

To see the value of `x` in a Notebook, we can simply place it on the last line of a cell:

```
# Print out the value in variable
x
```

```
160
```

We can also perform operations on `x` and save the result to a new variable:

```
# Use another variable to store the result of the operation between variable
and value
y = x / 60
y
```

```
2.6666666666666665
```

If we save a value to an existing variable, the new value will overwrite the previous value:

```
# Overwrite variable with new value
x = x / 60
x
```

```
2.6666666666666665
```

It's a good practice to use meaningful variable names, so you and others can read the code and understand it more easily:

```
# Name the variables meaningfully
```

```
total_min = 43 + 42 + 57 # Total length of albums in minutes
total_min
```

```
142
```

In the cells above we added the length of three albums in minutes and stored it in `total_min`. We then divided it by 60 to calculate total length `total_hours` in hours. You can also do it all at once in a single expression, as long as you use parenthesis to add the albums length before you divide, as shown below.

```
# Name the variables meaningfully
total_hours = total_min / 60 # Total length of albums in hours
total_hours
```

```
2.3666666666666667
```

If you'd rather have total hours as an integer, you can of course replace the floating point division with integer division (i.e., `//`).

```
# Complicate expression
total_hours = (43 + 42 + 57) / 60  # Total hours in a single expression
total_hours
```

```
2.3666666666666667
```

Exercise: Expression and Variables in Python

What is the value of `x` where `x = 3 + 2 * 2`

```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
x = 3+2*2
```

```
x
```

```
7
```

What is the value of `y` where `y = (3 + 2) * 2`?

```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
y=(3+2)*2
y
```

```
10
```

What is the value of `z` where `z = x + y`?

```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
z=x+y
z
```

```
17
```

Operations With Strings

In Python, a **string** is a s**equence of characters**:

- A string is contained within two quotes. You could also use single quotes.
- A string can be spaces or digits.
- A string can also be special characters.
- We can bind or assign a string to another variable.

It is helpful to think of a string as an ordered sequence. Each element in the sequence can be accessed using an index represented by the array of numbers. The first index can be

accessed as follows: We can access index six. Moreover, we can access the 13th index. We can also use negative indexing with strings. The last element is given by the index negative one. The first element can be obtained by index negative 15 and so on. We can bind a string to another variable. It is helpful to think of string as a list or tuple. We can treat the string as a sequence and perform sequence operations. We can also input a stride value as follows: The two indicates we'd select every second variable. We can also incorporate slicing. In this case, we return every second value up to index four. We can use the len command to obtain the length of the string. As there are 15 elements, the result is 15. We can concatenate or combine strings. We use the addition symbols. The result is a new string that is a combination of both. We can replicate values of a string. We simply multiply the string by the number of times we would like to replicate it- in this case, three. The result is a new string. The new string consists of three copies of the original string. This means you cannot change the value of the string, but you can create a new string. For example, you can create a new string by setting it to the original variable and concatenate it with a new string. The result is a new string that changes from Michael Jackson to Michael Jackson is the best. Strings are immutable. Back slashes represent the beginning of escape sequences. Escape sequences represent strings that may be difficult to input. For example, backslashes "n" represent a new line. The output is given by a new line after the backslashes "n" is encountered. Similarly, backslash "t" represents a tab. The output is given by a tab where the backslash, "t" is. If you want to place a backslash in your string, use a double backslash. The result is a backslash after the escape sequence. We can also place an "r" in front of the string. Now, let's take a look at string methods. Strings are sequences and as such, have apply methods that work on lists and tuples. Strings also have a second set of methods that just work on strings. When we apply a method to the string A, we get a new string B that is different from A. Let's do some examples. Let's try with the method "Upper". This method converts lowercase characters to uppercase characters. In this example, we set the variable A to the following value. We apply the method "Upper", and set it equal to B. The value for B is similar to A, but all the characters are uppercase. The method replaces a segment of the string- i.e. a substring with a new string. We input the part of the string we would like to change. The second argument is what we would like to exchange the segment with. The result is a new string with a segment changed. The method find, find substrings. The argument is the substring you would like to find. The output is the first index of the sequence. We can find the substring Jack. If the substring is not in the string, the output is negative one. Check the labs for more examples.

Lab

Objectives

After completing this lab you will be able to:

- Work with Strings
- Perform operations on String
- Manipulate Strings using indexing and escape sequences

Table of Contents

What are Strings?

The following example shows a string contained within 2 quotation marks:

```
# Use quotation marks for defining string
"Michael Jackson"
```

```
'Michael Jackson'
```

We can also use single quotation marks:

```
# Use single quotation marks for defining string
'Michael Jackson'
```

```
'Michael Jackson'
```

A string can be a combination of spaces and digits:

```
#Digitals and spaces in string
'1 2 3 4 5 6 '
```

```
'1 2 3 4 5 6 '
```

A string can also be a combination of special characters :

```
# Special characters in string
'@#2_#]&*^%$'
```

```
'@#2_#]&*^%$'
```

We can print our string using the print statement:

```
# Print the string
print("hello!")
print('Hello woldd!')
```

```
Hello woldd!
```

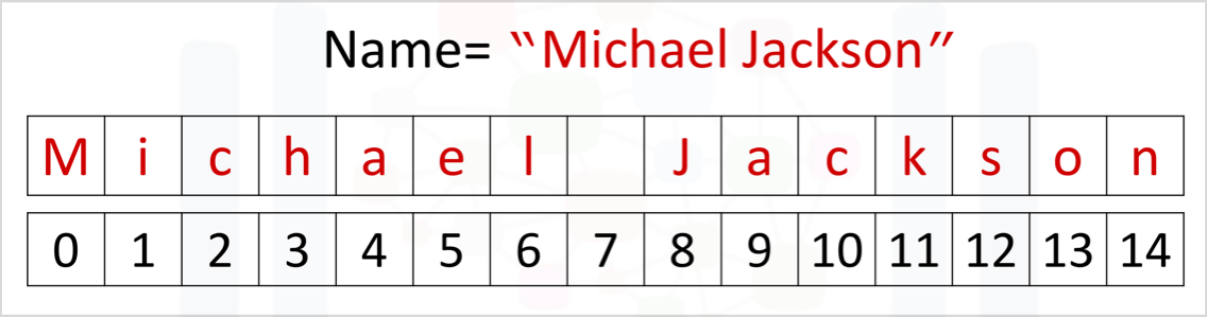
We can bind or assign a string to another variable:

```
# Assign string to variable
name = "Michael Jackson"
name
```

```
'Michael Jackson'
```

Indexing

It is helpful to think of a string as an ordered sequence. Each element in the sequence can be accessed using an index represented by the array of numbers:



The first index can be accessed as follows:

Because indexing starts at 0, it means the first index is on the index 0.

```
# Print the first element in the string
print(name[0])
```

M

We can access index 6:

```
# Print the element on index 6 in the string
print(name[6])
```

l

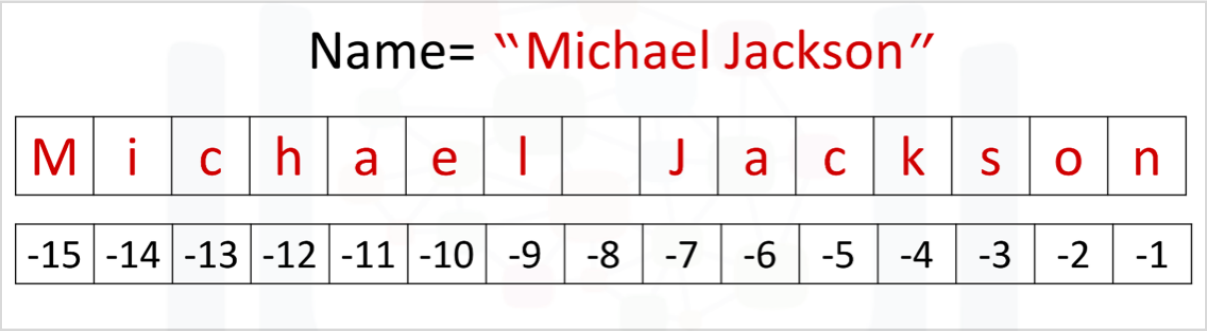
Moreover, we can access the 13th index:

```
# Print the element on the 13th index in the string
print(name[13])
```

o

Negative Indexing

We can also use negative indexing with strings:



Negative index can help us to count the element from the end of the string.

The last element is given by the index -1:

```
# Print the last element in the string
print(name[-1])
```

n

The first element can be obtained by index -15:

```
# Print the first element in the string
print(name[-15])
print(name[0])
```

M
M

We can find the number of characters in a string by using `len` , short for length:

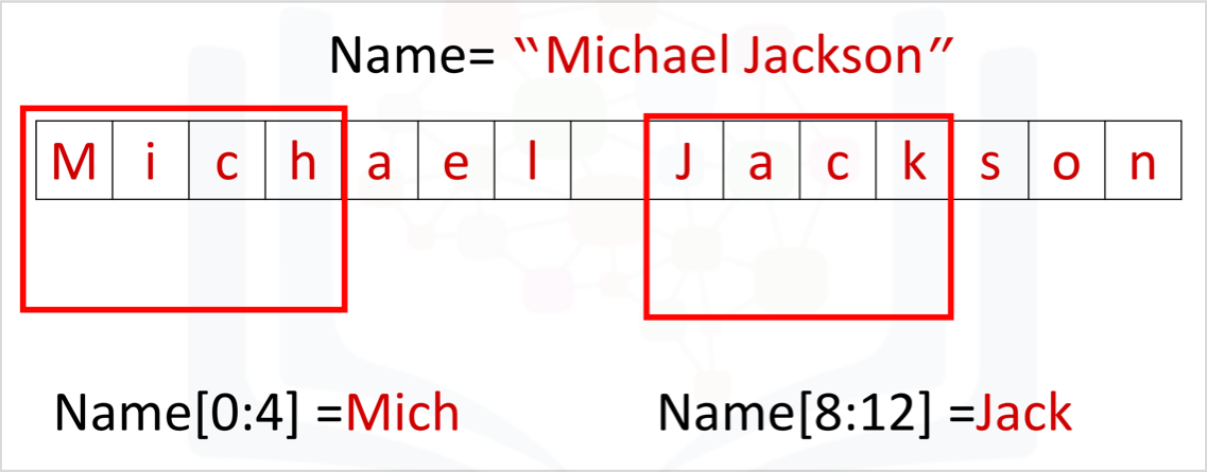
```
```

```
# Find the length of string
len("Michael Jackson")
```

15

Slicing

We can obtain multiple characters from a string using slicing, we can obtain the 0 to 4th and 8th to the 12th element:



[Tip]: When taking the slice, the first number means the index (start at 0), and the second number means the length from the index to the last element you want (start at 1)

```
# Take the slice on variable name with only index 0 to index 3
name[0:4]
```

'Mich'

```
# Take the slice on variable name with only index 8 to index 11
name[8:12]
```

'Jack'

Stride

We can also input a stride value as follows, with the '2' indicating that we are selecting every second variable:


```
# Get every second element. The elments on index 1, 3, 5 ...
name[::2]
```

'McalJcsn'

We can also incorporate slicing with the stride. In this case, we select the first five elements and then use the stride:

```
# Get every second element in the range from index 0 to index 4
name[0:5:2]
```

'Mca'

Concatenate Strings

We can concatenate or combine strings by using the addition symbols, and the result is a new string that is a combination of both:

```
# Concatenate two strings
statement = name + "is the best"
statement
```

```
```

'Michael Jacksonis the best'

To replicate values of a string we simply multiply the string by the number of times we would like to replicate it. In this case, the number is three. The result is a new string, and this new string consists of three copies of the original string:

```
# Print the string for 3 times
3 * " Michael Jackson"
```

' Michael Jackson Michael Jackson Michael Jackson'

You can create a new string by setting it to the original variable. Concatenated with a new string, the result is a new string that changes from Michael Jackson to “Michael Jackson is the best".

```
# Concatenate strings

name = "Michael Jackson"
name = name + " is the best"
name
```

'Michael Jackson is the best'

Escape Sequences

Back slashes represent the beginning of escape sequences. Escape sequences represent strings that may be difficult to input. For example, back slash "n" represents a new line. The output is given by a new line after the back slash "n" is encountered:

```
# New line escape sequence
print(" Michael Jackson \n is the best" )
```

Michael Jackson
is the best

Similarly, back slash "t" represents a tab:

```
# Tab escape sequence
print(" Michael Jackson \t is the best" )
```

Michael Jackson is the best

If you want to place a back slash in your string, use a double back slash:

```
# Include back slash in string
print(" Michael Jackson \\ is the best" )
```

Michael Jackson \ is the best

We can also place an `"r"` before the string to display the backslash:

```
# r will tell python that string will be display as raw string
print(r" Michael Jackson \ is the best" )
```

Michael Jackson \ is the best

String Operations

There are many string operation methods in Python that can be used to manipulate the data. We are going to use some basic string operations on the data.
Let's try with the method `upper`; this method converts lower case characters to upper case characters:

```
# Convert all the characters in string to upper case
a = "Thriller is the sixth studio album"
```

```
print("before upper:", a)
b = a.upper()
print("After upper:", b)
```

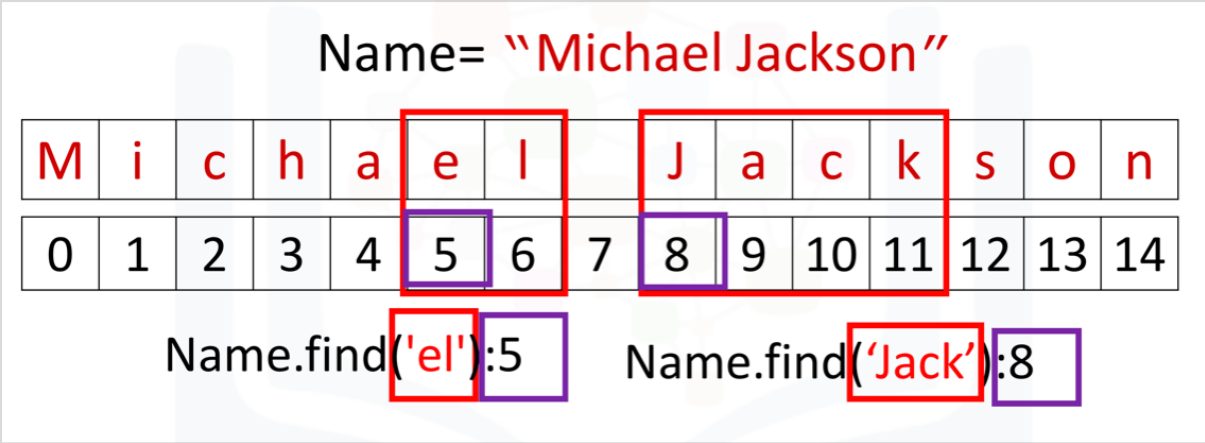
before upper: Thriller is the sixth studio album
After upper: THRILLER IS THE SIXTH STUDIO ALBUM

The method `replace` replaces a segment of the string, i.e. a substring with a new string. We input the part of the string we would like to change. The second argument is what we would like to exchange the segment with, and the result is a new string with the segment changed:

```
# Replace the old substring with the new target substring is the segment has been found in the string
a = "Michael Jackson is the best"
b = a.replace('Michael', 'Janet')
b
```

'Janet Jackson is the best'

The method `find` finds a sub-string. The argument is the substring you would like to find, and the output is the first index of the sequence. We can find the sub-string `jack` or `el`.




```
# Find the substring in the string. Only the index of the first elment of substring in string will be the output
name = "Michael Jackson"
```

```
name.find('el')
```

5

```
# Find the substring in the string.
name.find('Jack')
```

8

If the sub-string is not in the string then the output is a negative one. For example, the string 'Jasdfasdasdf' is not a substring:

```
# If cannot find the substring in the string
name.find('Jasdfasdasdf')
```

-1

Quiz on Strings

What is the value of the variable `a` after the following code is executed?

```
# Write your code below and press Shift+Enter to execute
a = "1"
a
```

'1'

What is the value of the variable `b` after the following code is executed?

```
# Write your code below and press Shift+Enter to execute
b = "2"
b
```

'2'

What is the value of the variable `c` after the following code is executed?

```
# Write your code below and press Shift+Enter to execute
c = a + b
c
```

'12'

Consider the variable `d` use slicing to print out the first three elements:

```
# Write your code below and press Shift+Enter to execute
d = "ABCDEFG"
print(d[0:3])
```

ABC

Use a stride value of 2 to print out every second character of the string `e`:

```
# Write your code below and press Shift+Enter to execute
e = 'clocrkr1e1c1t'
print(e[::2])
```

correct

Print out a backslash:

```
# Write your code below and press Shift+Enter to execute
print('\\')
print(r"\ ")
```

```
\
\
```

Convert the variable `f` to uppercase:

```
# Write your code below and press Shift+Enter to execute

f = "You are wrong"
f=f.upper()
f
```

```
'YOU ARE WRONG'
```

Consider the variable `g`, and find the first index of the sub-string `snow`:

```
# Write your code below and press Shift+Enter to execute

g = "Mary had a little lamb Little lamb, little lamb Mary had a little lamb \
Its fleece was white as snow And everywhere that Mary went Mary went, Mary went
\
Everywhere that Mary went The lamb was sure to go"
print(g.find('snow'))
g
```

```
95
```

'Mary had a little lamb Little lamb, little lamb Mary had a little lamb Its fleece was white as snow And everywhere that Mary went Mary went, Mary went Everywhere that Mary went The lamb was sure to go'

In the variable `g`, replace the sub-string `Mary` with `Bob`:

```
# Write your code below and press Shift+Enter to execute
print(g)
g = g.replace('Mary', 'Bob')
g
```

```
‘Mary had a little lamb Little lamb, little lamb Mary had a little lamb Its fleece was white as snow And everywhere that Mary went Mary went, Mary went Everywhere that Mary went The lamb was sure to go’
```

```
'Bob had a little lamb Little lamb, little lamb Bob had a little lamb Its fleece was white as snow And everywhere that Bob went Bob went, Bob went Everywhere that Bob went The lamb was sure to go'
```