Python Native Data Types & Methods

Introduction

Welcome to the lesson on Python native data types and methods. In this lesson, we will go over the native data types that you will work with when writing code in Python (or any programming language).

Table of Contents

Data Types

Introduction to Data Types

- Python: Data Types

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

Description	Python Data Type	Displayed `type()`
True or False	Boolean	`bool`
Whole Numbers	Integer	`int`
Decimal Numbers	Float	`float`
Sequences of characters	String	`str`
Nothing or Non-existent	None	`NoneType`

Different data objects (e.g. text, numbers, or a list of things) are given specific data types so that computers know how store them in memory. Behind the scenes, these different data types use memory differently. While you can look at a string representation of the number `1`, which looks like `"1"`, the way the computer stores the two are different and operations between the two cannot happen until certain methods are used. A Python program will give you an error if you try to execute an arithmetic function between the two. Try it out below:

In [ ]:
```
1 + '3'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 1 + '3'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Casting

To get around this problem, there are built-in methods that you can use to change dta from one type to another (with limitations, of course). These functions allow us to cast dta from one type to another by wrapping our target data with the appropriate method. See below for examples:

In [ ]:
```
#  Same as the problem above but casting the string to an integer
1 + int('3')
```

Out[ ]:
```
4
```

In [ ]:
```
1 + float('3')
```

Out[ ]:
```
4.0
```

In [ ]:
```
str(1) + '3'
```

Out[ ]:
```
'13'
```

One limitation to keep in mind with casting is you will need to ensure the data type will fit the format of the casted result. One example is trying to cast a string representation of a float to an integer. Integers do not have decimal points, so if you tried to cast `4.5` to a float, Python will throw a `ValueError`

In [ ]:
```
int('4.56')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[8], line 1
----> 1 int('4.56')

ValueError: invalid literal for int() with base 10: '4.56'
```

Booleans

The most basic data type in Python, booleans serve the most important role when it comes to writing a computer program: It allows a program to know what to do. A boolean is binary. It is yes or no, on or off, 1 or 0, `True` or `False`. While this is an easy concept on its own, it will still take some practice to plan out how you are going to use booleans to construct a useful script that will do what you want.

While you will sometimes explictly define a variable as a boolean value, booleans are typically used in comparison operators where the result of a boolean expression is what you're looking at to allow your program to make decisions. For example, in a theoretical program of a thermostat, you can set the `current_temperature` to the output of the thermometer and your thermostat program can periodically evaluate the temperature against a set threshold to switch the heater or air conditioning on or off. So you could write code that looks like this:

In [ ]:
```
current_temperature = 67

if current_temperature < 68: # <--- Results in True or False and determines which line below runs
    heater_on = True
else:
    heater_on = False
```

Numbers

The two types of numbers we see in programming are integers (whole numbers) and floating point (decimals). The presence of a decimal point means that a number is a floating point number. Python treats integers and floats that represent the same value as equal, so `5.0` is equal to `5`.

You can perform arithmetic between numbers with Python. Python will calculate result between integers and floats.

In [ ]:
```
type(23)
```

Out[ ]:
```
int
```

In [ ]:
```
type(34.5)
```

Out[ ]:
```
float
```

In [ ]:
```
34 + 6.5
```

Out[ ]:
```
40.5
```

Operations between floats and integers will result in the most precise form. if a calculation between an integer and a float occurs, the result will always be a float because it is more precise.

Mathematical Operators

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

Like you learned in math class, Python performs arithmetic operations according to the rule of operations: Parenthesis, Exponents, Multiplication / Division, Addition / Subtraction (PEMDAS). Below are some examples that you can try out.

In [ ]:
```
(1 + 70) / 4.3 + 8 ** 5
```

Out[ ]:
```
32784.51162790698
```

In [ ]:
```
20 // 4
```

Out[ ]:
```
5
```

In [ ]:
```
21 % 4
```

Out[ ]:
```
1
```

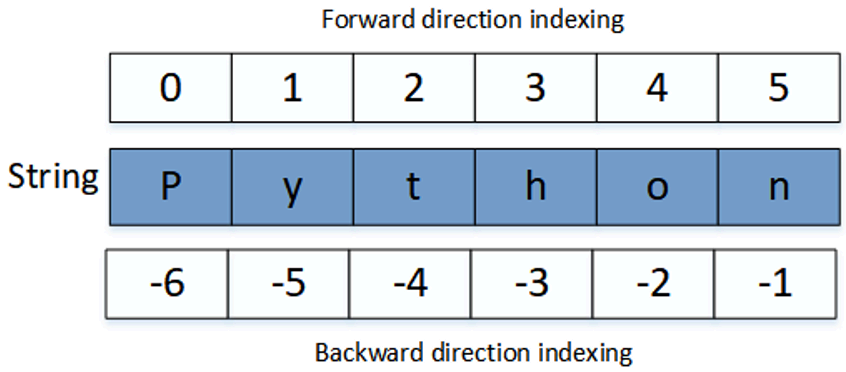
In [ ]:
```
20 / 4
```

Out[ ]:
```
5.0
```

Strings

- Python: String - Common string operations

Strings, formally called *string literals*, are immutable sequences of zero or more characters. They are enclosed with single, double, or triple quotes depending on the use case. Since strings are sequences, or collections of characters, we can access any portion of a string with *indexing*. The syntax for accessing a portion of a string is with the `[]` characters. As a reminder, Python uses zero-based indexing for forward direction indexing, which is nothing more than starting your counting at zero. For indexing from the end of a string, Python starts at `-1`.



In [ ]:
```
my_string = "This is my basic string"
# accessing individual indexes and printing them out
print(my_string[0])
print(my_string[1])
print(my_string[2])
print(my_string[3])
print(my_string[4])
```

```
T
h
i
s
```

String Immutability

The term immutable means that once a string is created, it cannot be changed in-place. It doesn't mean that we cannot "change" a string, but the way we do it is through completely re-defining, or creating completely new strings with changes.

With other collections in Python, such as lists, you can re-define a specific index of the list using the same bracket notation we saw in the above example. Because strings are immutable, we would get an error if we tried to change a letter in place.

In [ ]:
```
my_string = 'This is my string.'
my_string[-1] = "!" # Trying to change the last index
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[18], line 2
      1 my_string = 'This is my string.'
----> 2 my_string[-1] = "!" # Trying to change the last index

TypeError: 'str' object does not support item assignment
```

Nothing stops us from *redefining* a string, however.

In [ ]:
```
print(my_string) # printing out what was assigned above...
my_string = "This is my changed string" # re-defining my_string
my_string
```

```
This is my string.
```

Out[ ]:
```
'This is my changed string'
```

String Operations & Methods

Slicing Strings

As we saw with indexing, string slicing also uses bracket syntax. The difference is the presence of one or more colons which will tell python which indices to start (optional), stop, and step (optional).

The syntax for string slicing is `string[start:stop:step]`. `start` and `step` are optional parameters. If omitted, `start` defaults to `0`. `step` defaults to `1`.

In [ ]:
```
my_string[11:18]
```

Out[ ]:
```
'changed'
```

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

In [ ]:
```
my_string[:4]
```

Out[ ]:
```
'This'
```

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 [ ]:
```
my_string[-6:]
```

Out[ ]:
```
'string'
```

Reversing Strings

You can use string slicing to reverse a string by using -1 as the step.

In [ ]:
```
my_string[::-1]
```

Out[ ]:
```
'gnirts degnahc ym si sihT'
```

Below, assign a new string to a variable. Also, print a single character and a slice of the string.

In [ ]:
```
### Enter code here ###
```

Strings have their own methods in Python. These methods are used to return a modified copy of a string and leave the original string untouched.

`str.upper()`

- **Description**: Converts all characters in a string to uppercase.
- **Use Case**: Useful for ensuring consistent formatting or performing case-insensitive comparisons.

In [ ]:
```
text = "hello, world"
upper_text = text.upper()  # Result: "HELLO, WORLD"
upper_text
```

Out[ ]:
```
'HELLO, WORLD'
```

`str.lower()`

- **Description**: Converts all characters in a string to lowercase.
- **Use Case**: Useful for ensuring consistent formatting or performing case-insensitive comparisons.

In [ ]:
```
text = "Hello, World"
lower_text = text.lower()  # Result: "hello, world"
lower_text
```

Out[ ]:
```
'hello, world'
```

`str.capitalize()`

- **Description**: Capitalizes the first character of a string and makes all others lowercase.
- **Use Case**: Useful for capitalizing names or titles.

In [ ]:
```
text = "python programming"
capitalized_text = text.capitalize()  # Result: "Python programming
capitalized_text
```

Out[ ]:
```
'Python programming'
```

`str.title()`

- **Description**: Capitalizes the first character of each word in a string.
- **Use Case**: Useful for creating title case text.

In [ ]:
```
text = "python programming"
title_text = text.title()  # Result: "Python Programming"
title_text
```

Out[ ]:
```
'Python Programming'
```

`str.strip()`

- **Description**: Removes leading and trailing whitespace from a string.
- **Use Case**: Useful for cleaning user input or when working with data that may have inconsistent spacing.

In [ ]:
```
text = "          This is some text.   "
print(text)
stripped_text = text.strip()  # Result: "This is some text."
stripped_text
```

```
          This is some text.
```

Out[ ]:
```
'This is some text.'
```

`str.startswith()`

- **Description**: Checks if a string starts with a specified prefix.
- **Use Case**: Useful for filtering or categorizing strings.

In [ ]:
```
text = "Hello, World"
starts_with_hello = text.startswith("Hello")  # Result: True
starts_with_hello
```

Out[ ]:
```
True
```

`str.endswith()`

- **Description**: Checks if a string ends with a specified suffix.
- **Use Case**: Useful for filtering or categorizing strings.

In [ ]:
```
text = "Hello, World"
ends_with_world = text.endswith("World")  # Result: True
ends_with_world
```

Out[ ]:
```
True
```

`str.find()`

- **Description**: Searches for a substring within a string and returns the index of the first occurrence (or -1 if not found).
- **Use Case**: Useful for locating and extracting specific information from text.

In [ ]:
```
text = "Python is a powerful language. Python is fun."
index = text.find("Python")  # Result: 0
index
```

Out[ ]:
```
0
```

`str.count()`

- **Description**: Counts the number of non-overlapping occurrences of a substring in a string.
- **Use Case**: Useful for various text analysis tasks.

In [ ]:
```
text = "Python is a popular language, and Python is versatile."
count = text.count("Python")  # Result: 2
```

```
count
```

Out[ ]: 2

`str.replace()`

- **Description**: Replaces all occurrences of a specified substring with another string.
- **Use Case**: Useful for text transformations and substitutions.

In [ ]:
```
date_str = '03/14/2024'
formatted_date = date_str.replace('/', '-') # Result: 03-14-2024
print(date_str)
print(formatted_date)
```

```
03/14/2024
03-14-2024
```

k. `str.split()`

- **Description**: Splits a string into a list of substrings based on a specified delimiter.
- **Use Case**: Useful for tokenizing text or processing data.

In [ ]:
```
date_str = '03/14/2024'
split_date = date_str.split('/') # Result: ['03', '14', '2024']
split_date
```

Out[ ]: ['03', '14', '2024']

`str.join()`

- **Description**: Joins a list of strings into a single string using the current string as a separator.
- **Use Case**: Useful for constructing formatted text from a list of elements.

In [ ]:
```
merged_date = '-'.join(split_date)
merged_date
```

Out[ ]: '03-14-2024'

Other Methods (Conditionals)

Sometimes, you will want to evaluate a string before doing anything else with it. The below methods can be used to evaluate a string for certain characteristics. They will all return a boolean.

`str.isalpha()`

- **Description**: Checks if all characters in a string are alphabetic (letters).
- **Use Case**: Useful for validating input that should contain only letters.

In [ ]:
```
my_string = "This is my string"
my_string.isalpha()
```

Out[ ]: False

`str.isnumeric()`

- **Description**: Checks if all characters in a string are numeric (digits).
- **Use Case**: Useful for validating input that should contain only digits.

In [ ]:
```
my_string = "123"
my_string.isnumeric()
```

Out[ ]: True

`str.isalnum()`

- **Description**: Checks if all characters in a string are alphanumeric (letters or digits).
- **Use Case**: Useful for validating input that should be a combination of letters and digits.

In [ ]:
```
my_string = "ABC123"
my_string.isalnum()
```

Out[ ]: True

`str.islower()`

- **Description**: Checks if all characters in a string are lowercase letters.
- **Use Case**: Useful for verifying that text is in lowercase.

In [ ]:
```
my_string = "ABC123"
my_string.isupper()
```

Out[ ]: True

`str.isupper()`

- **Description**: Checks if all characters in a string are uppercase letters.
- **Use Case**: Useful for verifying that text is in uppercase.

In [ ]:
```
my_string = "ABC123"
my_string.islower()
```

Out[ ]: False

Interpolated Strings

Python allows embedded expressions inside of strings. The syntax invloves putting the letter `f` in front of the opening quote and using curly braces `{}` to surround your expression.

In [ ]:
```
name = 'Beth'
greeting = f"Hello, {name}!"
greeting
```

Out[ ]:
```
'Hello, Beth!'
```

Escape Sequences

Escape sequences allow you to include special characters in strings. They start with a backslash . Some common escape sequences include:

`\n` : Newline
`\t` : Tab
`\` : Backslash
`\"` : Double quote
`\'` : Single quote

Example with `\n` :

In [ ]:
```
multi_line = "This is\na multi-line\nstring."
print(multi_line)
```

```
This is
a multi-line
string.
```

The `None` Type

The `None` data type has only one value: `None`, which really just means the absence of data. `None` does not mean zero, or an empty string. You can equate `None` to an empty cell in a spreadsheet. You will find that some scripts will initialize a variable as `None` before they are used or functions use `None` when they do not need to "return" a value.

In [ ]:
```
type(None)
```

Out[ ]:
```
NoneType
```

You will find that `None` is not equivalent to zero or empty data structures, such as lists (`[]`).

In [ ]:
```
print(None == [])
print(None == '')
print(None == 0)
```

```
False
False
False
```

Conclusion

In this lesson, we explored Python data types to include Booleans, Numbers (Floats and Integers), Strings, and the None type. This comprehensive lecture covered all the major built-in string methods in Python, providing detailed explanations, use cases, and examples that you can practice with.

Exercise

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 = 55.50`
`tax = .0675`
`tip = .15`
	`68.13`

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

`bill = 796.34`
`tax = .0525`
`tip = .20`
	`1005.78`

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

In [ ]:
```
## Enter Code Below
```

Additional Exercises

- Exclamations
- Esrever
- DNA