Introduction to YAML in Python for Beginners

geekflare.com/python-yaml-intro/

November 8, 2020

YAML Ain't Markup Language (YAML) is a data serialization language for most programming languages. Let's understand in detail.

YAML a strict superset of **JSON**, so anything written in **JSON** can be parsed to **YAML**. It's mostly used for configuration files in projects and it's super easy to understand and read the code.

The file extension for YAML files is `.yaml` or `.yml`

In this tutorial, we are going to learn about different data types that are present in YAML and work with YAML in Python. By the end of this tutorial, you will be able to understand the YAML and its syntax.

> I assume you are familiar with Python if not check out Python learning resources.

The YAML follows indentation syntax similar to Python. But, it doesn't allow **tab** (remember it while writing YAML files) for indentation.

Setup

Install a Python module called the `pyyaml` to work with **YAML** files.

```
pip install pyyaml
```

Copy and paste the following code in a file and save it as `yaml_script.py`

```
import yaml

yaml_file = open("learn_yaml.yaml", 'r')
yaml_content = yaml.load(yaml_file)

print("Key: Value")
for key, value in yaml_content.items():
    print(f"{key}: {value}")
```

- We'll use the above script to convert the YAML code to Python and check different data types.
- Create a file called `learn_yaml.yaml` and practice different examples in it, that we are going to discuss in this tutorial.

Without further ado let's jump into the data types section of YAML.

Data Type in YAML

Everything in the **YAML** is a **key-value** pair.

We have a different name for key-value pairs in different programming languages like dictionaries, hashes, objects, etc.., These are building blocks of YAML.

The keys can be strings(quoted or normal), floats, or integers (Support may change in a future update). And values can be of any data type that YAML supports.

Let's see different data types presents in YAML.

Numbers

YAML supports integers, floating numbers, and exponential floating numbers.

```
integer: 123
float: 123.123
exponential_float: 1.34e+3
```

When you evaluate the above code with Python script you will get the result as follows.

```
Key: Value
integer: 123
float: 123.123
exponential_float: 1340.0
```

We can represent the values in different number systems like **decimal, octal,** and **hexadecimal**.

- The leading **zero(0)** represents it's an octal number.
- The prefix **0x** represents its hexadecimal value.

See the example below.

```
integer: 123
octal: 0123
hexa: 0x123
```

We can see the converted decimal values of octal and hexadecimal value by running our Python script. You should see the exact output as follows.

```
Key: Value
integer: 123
octal: 83
hexa: 291
```

Another exciting thing about YAML is that we can represent **NAN(Not A Number)** and **Infinity**.

```
not_a_number: .NAN
infinity: .inf
negative_infinity: -.inf
```

Run the Python script, you will see converted values of **NAN,** and **inf** to Python.

```
Key: Value
not_a_number: nan
infinity: inf
negative_infinity: -inf
```

That's it for the numeric types in YAML.

Strings

Strings in YAML can be represented **with or without quotes**. Both are similar. Unlike JSON there is no strict rule to put every string in quotes. But, if we need to use the **escape sequences**, then we must use **double-quotes**.

Let's see some examples of strings.

```
string: Hi, there I am a string
string with escape character: Hi, I am a newline \n character. And I am not
working :(
string with working escape character: "Hi, I am a newline \n character. And I am
working :)"
```

The newline character in the section key-value pair works as expected. As we already said, we need to use double quotes to work with escape sequences.

We have used double quotes in the next key-value pair and it works as expected. Interpret the above YAML code with our Python script. You will get the result as follows.

```
Key: Value
string: Hi, there I am a string
string with escape character: Hi, I am a newline \n character. And I am not
working :(
string with working escape character: Hi, I am a newline
character. And I am working :)
```

There two special characters in YAML that we can use to write multiple sentences as value to a key. Let's say we have to split a long sentence into multiple lines. In this type of scenario, we can use a **fold (greater than >)** or **block (pipe |)** character to write multiple lines.

What's the difference between **fold** and **block** characters? Coming to it.

Fold character won't interpret the newlines, whereas **block** character does.

Let's see the examples.

```
multiple lines string with fold character: >
  This is a
  multiple line
  string with fold
  character. Remember to use
  indentation. Newlines won't be
  interpreted.
multiple lines string with block character: |
  This is a
  multiple line
  string with fold
  character. Remember to use
  indentation. Newlines will be
  interpreted.
```

Run the Python script, then you will see the difference between the **fold** and **block** characters. And don't forget to use indentation.

```
Key: Value
multiple lines string with fold character: This is a multiple line string with
fold character. Remember to use indentation. Newlines won't be interpreted.

multiple lines string with block character: This is a
multiple line
string with fold
character. Remember to use
indentation. Newlines will be
interpreted.
```

Booleans

In YAML, we can represent the boolean value **True** and **False** in three different ways. Look at them.

- The values **True, On,** and **Yes** are considered as **True** in YAML.
- The values **False, Off,** and **No** are considered as **False** in YAML.

```
random_key_1: True
random_key_2: On
random_key_3: Yes
random_key_4: False
random_key_5: Off
random_key_6: No
```

If you interpret the above YAML code, then you will get the first 3 key values as **True** and the next 3 key values as **False**.

```
Key: Value
random_key_1: True
random_key_2: True
random_key_3: True
random_key_4: False
random_key_5: False
random_key_6: False
```

Null

YAML supports the null value similar to JSON. We can use the keyword **null** or the symbol **tilde(~)** to define the null value in YAML. YAML got pretty alternatives right? Yeah, they are kind of cool.

```
null_case_1: null
null_case_2: ~
```

Run the Python script. You will get both values as **None** as Python uses **None** instead of the **null** keyword.

```
Key: Value
null_case_1: None
null_case_2: None
```

Arrays

We can specify the arrays similar to Python in YAML. Or we can write all the array elements in separate lines preceded **hyphen (-)**. Let's see an example for each representation.

```
programming_languages: [Python, JavaScript, C, HTML, CSS]
libraries: [React, TailwindCSS]
```

In the above YAML code, we have used **square brackets** similar to Python lists. Let's see another way to represent arrays in YAML (looks like markdown lists).

```
programming_languages:
  - Python
  - JavaScript
  - C
  - HTML
  - CSS
libraries:
  - React
  - TailwindCSS
```

If you use our Python script to interpret the above examples. You will get the output as follows.

```
programming_languages: [Python, JavaScript, C, HTML, CSS]
libraries: [React, TailwindCSS]
```

We can have dictionaries in the list not only strings, numbers, etc..,

```
programming_languages:
- Python:
author: Guido van Rossum
- JavaScript:
author: Brendan Eich
- C:
author: Dennis Ritchie
libraries:
- React:
popularity: High
- TailwindCSS:
popularity: High
```

You will get an array of dictionaries if you interpret the above YAML code with our Python script.

```
Key: Value
programming_languages: [{'Python': {'author': 'Guido van Rossum'}}, {'JavaScript':
{'author': 'Brendan Eich'}}, {'C': {'author': 'Dennis Ritchie'}}]
libraries: [{'React': {'popularity': 'High'}}, {'TailwindCSS': {'popularity':
'High'}}]
```

Dictionaries

We have already seen the syntax of dictionaries in the above examples. To quickly recap dictionaries are key-value pairs. We can have any valid data type as a value to the key. YAML even supports the nested dictionaries.

Let's see an example.

```
dictionary:
  i am key: i am value

nested dictionary:
  nested key:
    i am nested key: i am nested value
```

Interpret the above code you will see the same result as follows.

```
Key: Value
dictionary: {'i am key': 'i am value'}
nested dictionary: {'nested key': {'i am nested key': 'i am nested value'}}
```

Do you know you can convert a list into a dictionary in Python?

Set

YAML supports another data type called set. Set contains unique values similar to Pythons' set data type. Set items are preceded by the **question mark (?)** like list items preceded by **hyphens (-)**.

We need to mention that the data type is set using **!!set** after the set name.

Look at the following example.

```
i am a set: !!set
  ? 1
  ? 2
  ? 2
  ? 3
```

As set contains only unique values, you won't get **2** two times when you interpret the above YAML code with our Python script.

Let's see the result.

```
Key: Value
set: {1, 2, 3}
```

We can also represent the set similar to Python syntax as follows.

```
i am a set: !!set
  {1, 2, 2, 3}
```

You will get the exact output as the above example.

That's it for the data types in YAML. Let's see some extra features in YAML.

Comments

YAML supports comments. It's great. We can write comments in YAML starting with a **hash (#)** symbol.

```
# I am a comment
yaml is great: # I am a comment too
```

If you interpret the above YAML code, then you will get an empty key with a null value.

```
Key: Value
yaml is great: None
```

YAML doesn't support multi-line comments. We have to write multiple lines starting with a hash for multi-line comments similar to Python.

Anchors

Anchors allow us to copy the content of a key wherever we want in the entire document. This is very handy if you like to duplicate some content in the document.

To use anchors, we have to define a name for it as a variable name in programming languages. And then we can use it across the document wherever we want.

We can define anchor name using **&** and use it with *****. Let's see an example.

```
# duplicate_data is the name of the anchor
data: &duplicate_data This content is to duplicate

# dopying the data
duplicate_data: *duplicate_data
```

In the above example, we have used a **duplicate_data** anchor to copy the value of the **data** key. Both the keys contain the same values if you interpret the above YAML.

```
Key: Value
data: This content is to duplicate
duplicate_data: This content is to duplicate
```

Conclusion

I hope you got a good understanding of YAML. Now, you can use YAML in your next project config file. You can refer to the YAML official website to get more advanced stuff.

Happy coding

TAGS:

Python