# MASTERING YAML

Sri Adilakshmi Marrivada

# TABLE OF CONTENTS

# 1 YAML OVERVIEW

**YAML** is a data serialization language *(textual representation of typed/cyclical data graphs)* which is one of the most popular programming languages. It is designed as human readable and interact easily with other scripting languages such as Perl, Python, etc. YAML is often used for configuration files that are parsed and read by a programming language or framework. YAML is widely used not only for configuration settings but also in log files, object persistence, internet messaging, cross language data sharing, complex data structures and many other places.

YAML was first proposed by **Clark Evans in 2001** and designed it together with **Brian Ingerson**, and **Oren Ben-Kiki** and was originally abbreviated as **Yet Another Markup Language** but it was later backronymed (recursive acronym) to **YAML Ain't Markup Language** to emphasize that YAML is data-oriented but not really a markup language *(marking up various elements of a text document)*.

YAML builds upon the structures and concepts described by XML, HTML, Perl, Python, C and other programming languages. YAML is also a superset of JSON, making JSON files valid in YAML.

YAML gained its popularity due to its explicit features:

- YAML is very much readable by humans. It enables the representation of intricate data structures in a comprehensible way. This is demonstrated by the fact that even the home page of the official YAML website is presented as a YAML document.
- YAML is simple to read and straightforward to understand. It supports complex data structures with minimal syntax overhead.
- YAML is easy to implement and use.
- YAML saves code in plain text only which can be easily extended and version controlled.
- YAML has consistent data model making data serialization and deserialization easy.
- YAML is fast to load and easy to process in memory.

## 2 XML VS JSON VS YAML

YAML is a data serialization language similar to XML or JSON but YAML is much more human readable and concise. **XML** (eXtensible Markup Language) is a markup language that came up in early 1990s whereas **JSON** (Java Script Object Notation) and **YAML** (YAML Ain't Markup Language) are data formats. In fact, YAML is a superset of JSON meaning it includes all features that JSON has and many more, allowing valid JSON files to be parsed as YAML.

The key differences between XML and JSON and YAML are:

- XML uses opening tag such as `<abc>` and closing tag such as `</abc>` to define the elements and stores data in a tree structure, whereas JSON uses braces *{ }* and brackets *[ ]* to represent data like a map with key-value pairs while YAML uses while space indentation and hyphen **–** to represent data in list or sequence format and in the form of a map with key-value pairs.
- XML syntax is more verbose whereas JSON syntax is explicit with strict syntax requirements but YAML has very minimal syntax.
- XML requires huge storage and network bandwidths since opening and closing tags need to be carried out for each data element. JSON is lighter compared to XML and YAML is even lighter compared to both.
- XML and YAML allow to define comments but JSON does not allow comments.
- XML and JSON are harder to read but YAML is easy to read and understand.
- XML is best for complex projects where strict schema and validation is required. JSON is best for web applications and data exchange over HTTP. YAML is best for human-edited configuration files along with JSON features.

**XML Format:**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<person>
  <name>John</name>
  <age>30</age>
  <gender>Male</gender>
  <hobbies>
    <hobby>Travelling</hobby>
    <hobby>Reading</hobby>
    <hobby>Music Playing</hobby>
  </hobbies>
  <address>
    <street>101 Avenue</street>
    <city>New York</city>
    <country>USA</country>
  </address>
</person>
```

**JSON Format:**

```
{
  person {
    "name": "John",
    "age": 30,
    "gender": "Male",
    "hobbies": [
      "Travelling",
      "Reading",
      "Music Playing"
    ],
    "address": {
      "street": "101 Avenue",
      "city": "New York",
      "country": "USA"
    }
  }
}
```

**YAML Format:**

```
person:
  name: John
  age: 30
  gender: Male
  hobbies:
  - Travelling
  - Reading
  - Photography
  address:
    street: 101 Avenue
    city: New York
    country: USA
```

Since YAML is designed as human readable and understandable, YAML format is widely used for writing configuration files in DevOps and cloud infrastructure such as **Docker Compose** (`docker-compose.yaml`), **Kubernetes** manifests (`deployment.yaml`, `service.yaml`), **GitHub actions** (`.github/workflows/*.yaml`), Ansible playbooks, CI/CD pipeline configs (**GitLab**, **CircleCI**), etc. YAML format is also used in **OpenAPI Specification** for RESTful API generation.

# 3 YAML SYNTAX RULES

YAML's syntax primarily uses indentation to define data structures like key-value pairs and lists, and follows other syntax rules.

- YAML uses Python-style indentation to determine the structure and indicate nesting. YAML does not allow tab characters for indentation to maintain portability across systems, so one or more whitespaces *(literal space characters)* must be used instead of tabs. The number of whitespaces to use for indentation is not fixed but it should be consistent across the YAML file to avoid cross platform issues *(ideally, 2 white spaces are used for indentation)*.
- Each key-value pair which is the primary structure for organizing data in YAML, must be properly separated by a colon *( `:` )* followed by a space to avoid misconfigurations. Any trailing spaces at the end of key-value pair should be avoided as they can cause issues while parsing the YAML file.
- YAML is case sensitive which applies to all of its elements, particularly the keys in key-value pairs. For example, YAML identifies `data:` and `Data:` as two separate keys, hence consistent casing must be used throughout the YAML file to avoid errors.
- Each list item must be defined on a new line, prefixing with a hyphen and a space *(−  )*, and at the same level of indentation.
- Strings need not be quoted in general but quotes (single quote or double quote) are required for strings containing special characters such as `[` or `]` or `,` or `@`, etc. or reserved words to avoid misinterpretation.
- YAML allows multi-line strings in folded style (using `>` character) to preserve all new lines or literal style (using `|` character) to replace new lines with spaces.
- Comments in YAML can be defined with a pound or hash symbol *(#)*. It is always a best practice to use comments for describing the code, making it easy to understand.
- YAML allows to store multiple documents (each document has its own YAML structure) in a single file with each document separated by three hyphens *(−−−)* on a new line.
- YAML files can optionally begin with 3 hyphens *(−−−)* to indicate the start of the document and optionally end with 3 dots *( `...` )* to indicate the end of the document.
- Each YAML file must use either `.yml` or `.yaml` extension.

# 4 YAML BASIC SYNTAX

YAML data type structures are similar to that of Perl and Python. In YAML, there are three fundamental structures *(also termed as nodes in YAML)* which are **Scalars** (data types), **Mappings** (key-value pairs) and **Sequences** (lists). YAML follows some key formatting principles to define its data structure.

## 4.1 Comments

YAML is a superset of JSON but unlike JSON, YAML supports comments which will be helpful to explain the document code. YAML supports only single line comment that can be represented using hash or pound character (**#**) but multi-line or block comments are not supported. However, multi-line comments can be represented using **#** at the beginning of each line. YAML allows to specify comments as a single line or after a value (inline).

```
# This is a single line comment
person:
  name: John # this is an inline comment
  age: 30
```

## 4.2 Indentation

YAML heavily relies on **whitespace and indentation** to indicate nesting which makes easy to read and understand by humans. YAML follows the Python-like indentation style for defining hierarchical relationships and nesting. YAML allows only white spaces for indentation and tab characters are not allowed. The number of spaces used for indentation does not matter as long as they are consistent at each nesting level to avoid any parsing issues.

```
person: # Nesting level 1 (zero spaces used for indentation)
  name: John # Nesting level 2 (2 spaces used for indentation)
  address:
    street: 101 Avenue # Nesting level 3 (4 spaces used for indentation)
    city: New York
    country: USA
```

## 4.3 Key-Value Pairs

YAML represents data as key-value pairs. The key-value pair is the basic building block of YAML. The key-value pair is represented by a `key` separated by a `colon` followed by a `space` and `value` such as `<key>: <value>`. YAML also provides a flexibility of marking a key with **?**

(a question mark followed by a space) indicator while value must be marked with `:` (a colon followed by a space) indicator.

```
name: John
age: 30
```

## 4.4 Scalars

Scalars (or Scalar Nodes) in YAML are nothing but values which are represented in various types such as string, integer, float, boolean, etc.

YAML scalars supports many data types including:

- **Numeric** datatypes which are `integer`, `float`, `hexadecimal`, `binary`, etc. to represent a number value in various formats.
- **String** datatype which is `string` to represent a text value.
- **Boolean** datatype which is `boolean` to represent true or false value.
- **Date** datatype which is `timestamp` to represent date and/or time value.

**Numeric Scalars:**

YAML supports many numeric datatypes including `integer` to represent a whole number, `float` to represent a number with decimal point, `exponential` to represent a number in scientific notation, `octal` to represent a base-8 number which is prefixed with `0o` *(this type is commonly used for file permissions),* `hexadecimal` to represent a base-16 number which is prefixed with `0x` and `binary` to represent a number in binary format. Note that some numeric interpretations vary between YAML 1.1 vs 1.2 version. For example, octal scalar must be prefixed with `0o` in YAML 1.2 while it was prefixed with `0` alone in YAML 1.1 version.

YAML also supports special numeric data types such as `infinity` to represent an infinite number which is denoted using `.inf` value, `negative-infinity` to represent a non-positive infinite number which is denoted using `-.inf` value and `not-a-number` to represent a non-number which is denoted using `.nan` value.

```
age: 30 # Decimal integer scalar
ageHex: 0x1E # Hexadecimal integer scalar
ageOctal: 0o36 # Octal integer scalar
ageBinary: 11110 # Binary integer scalar

heightCM: 170.5 # Decimal floating-point scalar
smallscale: 1.0e-10 # Exponential scalar representing 0.0000000001
```

```
largescale: 1.0e+5 # Positive exponential scalar representing 1,00,000

inf: .inf # Infinity scalar
inf_: -.inf # Negative infinity scalar
nan: .nan # Not-a-number scalar
```

**String Scalars:**

YAML automatically detects string scalars so that string values need not be explicitly quoted unlike in JSON, Python and other programming languages.

```
name: John #string scalar
userID: user12 #string scalar
```

However, it is necessary to enclose a string scalar in quotations *(using either single quotation mark ' or double quotation mark ")* when the value contains any special characters such as **#**, **&**, **?**, **|**, **:**, *****, **@**, **%**, **-**, **{**, **}**, **[**, **]**, **>**, **<**, **=**, **!** , **'**, **"**, **,** , **\**, etc. or when the value must be explicitly represented as string *(for example, an integer to be treated as a string).*

```
id: "12345" # Quoted to represent number as string

name: 'John D''Ove' # Single quoted to escape internal single quote with
another single quote

address: "123 Main Street, Apt #4B" # Double quoted for special characters
```

**Boolean Scalars:**

YAML identifies boolean values with keywords `true` or `yes` or `on` for positive statement and keywords `false` or `no` or `off` for negative statement. Note that different versions of YAML may have different rules for interpreting boolean scalars. For example, in YAML 1.1, `yes` and `no` are valid boolean scalars while they are not in YAML 1.2 and so it is recommended to use `true` and `false` for boolean scalars to avoid potential parsing and compatibility issues since `true` and `false` boolean values are standard and widely supported by the most YAML processors.

```
name: John Doe

# Boolean with True/False
verified: True
active: False

# Boolean with Yes/No
acceptEmails: Yes
```

```
receiveSMS: No

# Boolean with On/Off
notifications: on
awayMode: off
```

**Date-Time Scalars:**

A date-time scalar in YAML represents a specific point in time. YAML supports many timestamp formats including:

- **Canonical (ISO 8601 subset) Format**: It is used to specify date and time with milliseconds and UTC (Coordinated Universal Time) in `YYYY-MM-DDThh:mm:ss.sZ` format where `Z` indicates UTC *(for example `2020-10-20T01:23:34.1Z`)*.

- **ISO 8601 Format**: It is used to specify date and time with milliseconds and time zone offset in `YYYY-MM-DDThh:mm:ss.s±hh:mm` format. *(e.g. `2020-10-20T01:23:34.10-05:00`)*.

- **Space-separated Format:** It is used to specify date and time values with or without time zone offset separated by spaces in `YYYY-MM-DD hh:mm:ss.s ±h` format or `YYYY-MM-DD hh:mm:ss.s ±hh:mm` format. By default, YAML uses `UTC` time zone when offset is not explicitly specified *(e.g. `2020-10-20 01:23:34.10 -5`)*.

- **Date only Format**: It is used to specify only the date in `YYYY-MM-DD` format. In this case, YAML assumes the time part as `00:00:00Z` which is midnight UTC *(e.g. `2020-10-20`)*.

Though YAML supports many date-time formats, **ISO 8601** format is mostly commonly used to maintain consistency across different systems.

```
name: John Doe

birth_date: 1990-05-15 # Date only format with time assumed as 00:00:00Z

register_time: 2023-12-14T21:59:43.10Z # Canonical format with UTC timestamp

last_login: 2024-01-02T08:35:50.10-05:00 # ISO 8601 format with time zone
offset (-05:00 indicates Eastern Daylight Time)

last_updated: 2024-01-02 08:35:51 -05:00 # Space-separated format with time
zone offset
```

## 4.5 Mappings

Mappings (or Mapping Nodes) in YAML represent an unordered collection of unique key/value pairs. YAML mappings work like dictionaries in Python or hash maps in Java or associative arrays in other programming languages. Due to the unordered nature of mapping, the order of key-value pairs within a mapping may not be the same after parsing or serialization (though many parsers try to maintain the order during parsing). Each entry in a mapping consists of a unique key and its corresponding value where the key can be a string or number and the value can be of any valid YAML data type such as scalar *(string, number, boolean, null)* or sequence or a mapping. Mappings in YAML files can be nested by increasing the indentation level and new mappings can be created at the same level by resolving the previous one.

YAML mappings are represented in **block style** using indentation as below:

```
# Mapping 1
name: John Doe #key: value (string)

# Mapping 2
age: 30 #key: value (number)

# Mapping 3
contact:
  email: jane.doe@example.com #key: value (string)
  phone: #key: value (nested mapping)
    type: home
    details:
      country_code: +1
      area_code: 123
      number: +1-123-456-7890
```

Since YAML is a superset of JSON, the mapping can also be represented in JSON format as **flow style** by enclosing data in curly brackets `{}` and items separated by a comma.

```
{name: John Doe, age: 30, contact: {email: jane.doe@example.com, phone:
{type: home, details: {country_code: +1, area_code: 123, number: +1-123-456-
7890}}}}
```

## 4.6 Sequences

Sequences (or Sequence Nodes) in YAML represent a set of values listed in a specific order. Sequence works like a list in Python or an array in Bash and other programming languages. Each item in sequence is represented with a `hyphen` and a `space` followed by a `value`.

Sequences can be nested using the proper indentation and can also be embedded into a mapping. In YAML, nested sequences need not start with a new line and when a sequence is embedded into a mapping, indentation is not mandatory.

YAML sequences are represented in **block style** using indentation and hyphen as below:

```
# Sequence 1
skills:
  - Python
  - Java
  - - Core Java # Nested sequence started on same line
    - Advanced Java

# Sequence 2
hobbies: # Sequence in a mapping with indentation
  - Travelling
  - Photography
  - Reading: # Nested sequence of a mapping without indentation
    - Fantasy
    - Science Fiction
    - Historical Fiction
```

YAML sequences can also be represented in JSON format as **flow style** by enclosing sequence data in square brackets **[ ]** and items separated by comma.

```
{skills: [Python, Java, [Core Java, Advanced Java]], hobbies: [Travelling,
Photography, {fav_genres: [Fantasy, Science Fiction, Historical Fiction]}]}
```

## 4.7 Nulls

Nulls represent unknown or undefined values and can be used in various ways depending on the context. A null value in YAML can be represented using either `null` keyword or tilde (~) special character or empty value.

- A null can be used when the value of data element is unknown or undefined.
- A null can be used to represent an empty value, such as an empty list or an empty string.
- A null can be used as a placeholder value when the actual value is not yet known.
- A null value can be used to represent data that is missing or incomplete.
- A null value can be used with optional parameter such that if the parameter is not specified, the `null` value will be used.
- A null value can be used to represent when the data is invalid.

```
# Null using null keyword
name: null

# Null using ~ character
age: ~

# Null as an empty value
gender:
```

# 5 YAML ADVANCED SYNTAX

YAML offers several advanced features beyond basic key-value pairs and lists, enhancing its utility for complex configurations and data structures.

## 5.1 Multi-line Strings

YAML has the feasibility of representing large blocks of text in **folded style** or **literal style**.

In folded style, a string can be specified in multiple lines (as a block) using the *folded block scalar* (**>**) in YAML file and it is interpreted as a single line without newline characters. Such strings are termed as **Folding strings**. Folding strings can be useful for defining long text fields, such as descriptions, that should appear as a single line after interpretation.

```
example: >
 this looks like
 a multiline string,
  but it is actually not.
```

The above YAML code is interpreted as below:

```
example: "this looks like a multiline string, but it is actually not."
```

In literal or block style, a string can be specified in multiple lines (as a block) using the *literal block scalar* (**|**) in YAML file and it is interpreted as multiple lines with newline characters. Such strings are termed as **Block strings**. Block strings is particularly helpful when defining shell commands.

```
example: |
 This is
  a real
  multiline string.
```

The above YAML code is interpreted with new line characters (`\n`) as below:

```
example: "This is\n a real\n multiline string."
```

When dealing with multi-line strings, **Chomp modifiers** can be used to define how YAML can interpret newlines at the end of the multi-line string. YAML provides two chomp modifiers, *keep* (indicated by **+**) and *strip* (indicated by **−**) to preserve or remove trailing newlines of multi-line string. These modifiers must be used along with folded (**>**) or block (**|**) indicators.

**Chomp in Folded style:**

```
example: >+
 This is a
  single line with an empty line at the end.

```

The above YAML code is interpreted as below:

```
example: "This is a single line with an empty line at the end.\n"
```

**Chomp in Blocked style:**

```
example: |-
 This is a
  single line with an empty line at the end.

```

The above YAML code is interpreted as below:

```
example: "This is a\n single line with an empty line at the end."
```

YAML follows three chomp modes when handling multiline blocks containing new lines:

- **Clip mode**: This is the default behavior when no chomp indicator is specified. It preserves the first trailing newline and removes any additional trailing empty lines.

```
example: |
 This is a multiline string.
 It has three trailing empty lines.



```

The above YAML code is interpreted as below:

```
example: "This is a multiline string.\nIt has three trailing empty
lines.\n"
```

- **Strip mode**: This is indicated by **–** character in the string block header to remove all trailing spaces and empty lines including the first trailing newline.

```
example: |-
 This is a multiline string.
 It has three trailing empty lines.



```

The above YAML code is parsed as below:

```
"This is a multiline string.\nIt has three trailing empty lines."
```

- **Keep**: This is indicated by **+** character in the string block header to preserve all trailing spaces and empty lines as they appear in the YAML block.

```
example: |+
 This is a multiline string.
 It has three trailing empty lines.



```

The above YAML code is parsed as below:

```
"This is a multiline string.\nIt has three trailing empty
lines.\n\n\n"
```

## 5.2 Documents

In YAML, a **document** refers to a single, complete set of data structures contained within a file. A single YAML file can contain more than one document where each document is considered as a standalone YAML file. This multi-document feature is helpful to organize related but independent configurations within a single file instead of multiple files, making it easier to manage and parse. Each document in a single YAML file can contain its own distinct data structures with scalars, sequences and mappings which means that duplicate keys that are not allowed within a single document can be repeated across different documents in the same file.

YAML files can begin with **Document Start Marker** represented using 3 hyphens (`---`) to indicate the start of the document and optionally end with **Document End Marker** represented using 3 dots (`...`) to indicate the end of the document. When a YAML file has a single document, using `---` is optional but when it contains multiple documents, each document must be separated by a line containing `---` which acts as a document separator.

```
--- # This is optional for the first document but good to use to indicate
start of file
# Document 1: Application configuration
name: MyApp
version: 1.0
environment: production

--- # This is mandatory for the second document
# Document 2: User definition
name: John Doe
email: john.doe@example.com

--- # This is mandatory for the third document
# Document 3: Deployment settings
target: Kubernetes
namespace: default
replicas: 3
resources:
  cpu: 200m
  memory: 256MB

... # This is optional but good to use to indicate end of file
```

In the above example, `name` key is unique in `Document 1` and `Document 2` but repeated across documents.

Though Document End Marker is optional, it is useful in some cases. For example, when YAML is used in a stream or over network, it might be important to tell the receiver that the document has ended.

A YAML document can be started with either mapping block or sequence block. When a YAML document is started with a mapping, then YAML expects only a series of mappings whereas when it is started with a sequence, then YAML expects only a series of sequences.

**YAML started with a mapping block**

```
---
name: John
age: 30
is_employed: true
gender: Male
address:
  street: 101 Avenue
  city: New York
  country: USA
hobbies:
  - Travelling
  - Reading
  - Music Playing
```

When this YAML code is viewed as JSON, it reflects as below:

```
{"name": "John", "age": 30, "is_employed": true, "gender": "Male",
"address": {"street": "101 Avenue", "city": "New York", "country": "USA"},
{"hobbies": ["Travelling", "Reading", "Music Playing"]}}
```

**YAML started with a sequence block**

```
---
- Travelling
- Reading
- Music Playing
- Hours_Played: 100
- Instruments:
  - guitar
  - violin
```

When this YAML code is viewed as JSON, it reflects as below:

```
["Travelling", "Reading", "Music Playing", {"Hours_Played": 100},
{"Instruments": ["guitar", "violin"]}]
```

## 5.3 Directives

**YAML directives** are instructions to a YAML processor that appear at the beginning of a YAML document, before the actual data content. Directives are denoted by a percent sign (%) followed by the directive name and any parameters.

YAML directives are declared before a document begins and must be followed by **---** *(called as **directives end marker** or **document start marker**)* which separates directives from the document's content. YAML directives are applicable to its following document but not the entire YAML file.

YAML offers two standard directives:

- %YAML directive is used to explicitly declare the YAML version that the document confides to. This is helpful to ensure compatibility and consistent parsing across different YAML processors. This directive is declared at the beginning of a YAML document which is followed by the document start marker (---).

```
%YAML 1.2
---
# This document adheres to YAML 1.2 version
name: John Doe
age: 30
is_active: true
```

  In the above example, the %YAML 1.2 directive indicates that the document should be parsed according to the YAML 1.2 specification. In YAML 1.2, only true and false are explicitly recognized as booleans, unlike YAML 1.1 which also recognized yes, no, on, and off. Also, YAML applies a %YAML directive only to the single document immediately following it. If a YAML file contains multiple documents, each document that requires a specific version directive would need its own %YAML declaration.

- %TAG directive allows to define the tag handles or shorthand prefixes for URIs (Uniform Resource Identifiers) associated with custom or application-specific data types and these short hands are used in node tags. Basically, this directive helps to create an alias or a shortcut for a longer tag identifier within the YAML document. This makes custom tags more concise and readable within the document.

  %TAG directive must be declared along with a tag handle (shorthand) and a tag prefix (which is generally a URI) at the beginning of the YAML document.

```
%TAG tag_handle tag_prefix
---
```

  There can be three types of tag handles that can be used with tags and %TAG directives.

  - **Primary tag handle** is simply an exclamation mark (**!**) which by default associates to **!** tag prefix. This handle is commonly used for custom tags or for explicitly declaring a node (or value) type with a local name.

- **Secondary tag handle** is written as double exclamation marks `!!` which by default associates to `tag:yaml.org,2002` tag prefix. This handle is commonly used for YAML's built-in data types.
- **Named tag handle** defines a custom handle name in between exclamation marks like `!abc!` which is used with `%TAG` directive to create a short alias for longer URI. This handle is commonly used for global tags declaration.

```
# Declare TAG directive
%TAG !py! tag:yaml.org,2002:python/object:
---
- !py!__main__.MyClass # Using the custom tag handle
```

In the above example, `!py!` is the named tag handle or alias defined for the URI `tag:yaml.org,2002:python/object`. When this tag handle is used with in the document, it will be replaced with the URI. So, the sequence item `!py!__main__.MyClass` is evaluated to `tag:yaml.org,2002:python/object:__main__.MyClass` during parsing.

## 5.4 Tags

**YAML tags** are identifiers that can be used within a YAML document to explicitly denote the data type of a value (or node) or provide additional metadata. YAML derives data types *(implicit typing inference)* when no explicit tag is provided which simplifies YAML documents making human-readable.

In YAML, tags are broadly classified into two types:

1. **Non-Specific Tags:**

   Non-specific tags are the default tags assigned to nodes (values) when the data type of a node (a value) is not explicitly declared in YAML document. They bring in the **implicit typing** feature which automatically determines the data type of values based on the data provided *(for example, a string that starts with a digit is auto-interpreted as a number, and a number that is enclosed in quotes is auto-interpreted as a string)*. YAML parser infers data types automatically. For instance, strings are auto tagged to `tag:yaml.org,2002:str`, maps are auto tagged to `tag:yaml.org,2002:map`, sequences are auto tagged to `tag:yaml.org,2002:seq`, etc.

The non-specific tag is typically denoted with **!** (single exclamation mark) so that when a node has the **!** tag, it informs the YAML parser to interpret the node as a basic type (string, map, or sequence) resolving to `tag:yaml.org,2002:str` or `tag:yaml.org,2002:map` or `tag:yaml.org,2002:seq` based on its structure. For example, `! 133` value is forcefully interpreted as basic string type though it appears as a number.

```
# Implicitly inferred as a string
name: John Doe

# Implicitly inferred as an integer
age: 30

# Implicitly inferred as a float
height: 172.5

# Implicitly inferred as a boolean
active: true

# Implicitly inferred as null
phone: ~

# Implicitly inferred as a sequence
hobbies:
  - Travelling
  - Photography

# Implicitly inferred as a mapping
address:
  street: 101 Avenue
  city: New York
  country: USA
  zip: "12345" # Explicitly quoted to infer as a string

# Explicitly inferred as a string, overriding potential number inference
phone: ! 9192939495
```

2. **Specific Tags:**
   Specific tags in YAML provide a flexibility to explicitly declare the data type of a value or provide additional metadata beyond the implicit type reference.

   In YAML, specific tags can be defined using standard tags or custom tags:

   - **Standard Tags:**
     Standard tags are used to explicitly declare the data type of a value to avoid any misinterpretations and ensure data is parsed correctly by different applications. Tags for

datatype can be declared using double exclamation marks **!!** (secondary tag handle) followed by the name of data type such as `!!int`, `!!str`, `!!seq`, etc. By default, the **!!** expands to `tag:yaml.org,2002:` so the `!!int` tag maps to the built-in YAML integer type as `tag:yaml.org,2002:int`, similarly `!!str` tag maps to `tag:yaml.org,2002:str`, `!!seq` maps to `tag:yaml.org,2002:seq`, etc.

Note that if a single exclamation mark is used for tagging such as `!int`, it is identified as a local tag which requires a tag resolution mechanism to resolve `!int` to a specific URI that defines the `int` type.

The most commonly used standard tags are:

- Scalar type tags including `!!str` (represents text data), `!!int` (represents whole number), `!!float` (represents number with fractional parts or a floating-point number), `!!binary` (represents binary data encoded in base64), `!!null` (represents a null or empty value), `!!bool` (represents truth values such as true or false), `!!timestamp` (represents a point in time, as date and/or time), etc.
- Collection type tags including `!!map` (represents a mapping with unordered collection of key-value pairs), `!!seq` (represents a sequence with ordered collection of items), `!!set` (represents an unordered collection of unique values) and `!!omap` (represents a mapping with ordered sequence of key-value pairs).

```
# Implicitly inferred as a string
name: John Doe

# Explicitly declared to treat as an integer
height: !!int 172.5 # Treat 172.55 an integer, truncating the decimal

# Explicitly declared to treat as a boolean
active: !!bool true

# Explicitly declared to treat as null
phone: !!null ~

# Explicitly declared to treat as a sequence
hobbies: !!seq
  - Travelling
  - Photography

# Explicitly declared to treat as a mapping
address: !!map
  street: 101 Avenue
  city: New York
  country: USA
```

- **Custom Tags:**

  Custom tags are used to represent complex data types that are not covered by the built-in scalars (strings, integers, floats, booleans) and collections (maps and sequences). Custom tags are useful to create custom schemas for special configurations or data serialization and represent complex and user-defined data structures and domain-specific concepts or objects.

  Parsing YAML with custom data types involves defining the custom data structure or class (e.g., `!Person` custom type representing a `Person` object with `name`, `age`, and `city` attributes) and implementing serialization/deserialization mechanism for the custom type in the chosen programming language. This allows the YAML parser to recognize and correctly convert the specific structures within the YAML document into corresponding classes or data structures defined in the programming code.

  YAML allows to create two types of custom tags – **Local tags** and **Global tags** defining their scope.

  - **Local Tags:**

    Local tags are declared and used within a single YAML document and are not globally unique. These are helpful to define custom data types that are only relevant within the context of that specific YAML document. Local tags are denoted using primary tag handle **!** followed immediately by the custom tag name without any space such as `!tag`.

    ```
    ---
    people:
      - !Person # local tag for a custom Person data type
        name: John Doe
        age: 30
        city: New York
      - !Person
        name: Jane Smith
        age: 25
        city: Los Angeles
    ```

In the above example, `!Person` is a custom local tag used to identify the subsequent data as representing the `Person` object with `name`, `age` and `city` attributes within this specific document.

By default, the primary tag handle **!** is associated to tag prefix **!**. However, this default behavior can be overridden by declaring tag prefix beginning with **!** using `%TAG` directive.

```
%TAG !emp! !Employee
---
people:
  - !emp!Person # local tag with named handle for a custom Person
data type
    name: John Doe
    age: 30
    city: New York
```

In the above example, `!emp!` is the named tag handle or alias defined for the local tag prefix `!Employee`. When this tag handle is used with in the document, it will be replaced with the tag prefix. So, the sequence item `!emp!Person` is evaluated to `!EmployeePerson` during parsing.

- **Global Tags:**
  Global tags are custom URIs used to reference a specific data type or schema in YAML using Universal Resource Identifier (URI). This helps to represent custom data structures that are specific to an application or domain. Global tags must be unique across all applications in the environment.

  A global tag must be declared before the start of the YAML document using the **`%TAG`** directive followed by the tag handle and tag prefix. For global tag declaration, YAML allows to use either primary tag handle **!** *(though it is commonly used for local tags)* or secondary tag handle **!!** *(though it is commonly used for standard tags)* or named tag handle *(commonly used for global tags)*, but the tag prefix must begin with `tag:` to specify URI *(which generally includes domain name and year to ensure global uniqueness)* such as `%TAG !example! tag:example.com,2025:`

```
# Global tag declaration with primary tag handle !
%TAG ! tag:example.com,2025:
--- !person # Top level object is a custom type !person
name: John Doe
age: 30
skills: # Sequence of skills
  - !skill # Each skill is a custom type !skill
    name: Programming
    level: Expert
  - !skill
    name: Cloud Computing
    level: Advanced
```

In the above example, global tag is created with tag handle `!` which is associated with `tag:example.com,2025:` URI. Then, `!person` is defined at the start of the document indicating the subsequent data within the document is a custom type `!person` where `!` represents `tag:example.com,2025:` and it has scalars representing person's `name` and `age` along with `skills` mapping that has a sequence of skills in which each skill is a custom type `!skill` where `!` represents `tag:example.com,2025:`

YAML allows to declare multiple global tags using multiple `%TAG` directives to define tag prefixes, which makes easy to refer to specific tag URIs within a document. While multiple `%TAG` directives can be used in a single YAML document, each directive must define a unique prefix or a unique URI for a given prefix.

For example:

```
%TAG !person! tag:example.com,2024:person/ # General person tag
declaration
%TAG !employee! tag:example.com,2024:person/employee/ # Specific
person employee tag declartion
%TAG !contractor! tag:example.com,2024:person/contractor/ #
Specific person contractor tag declaration
---
employees:
  - !employee!FullTime: # Tag for a full-time employee
      id: EMP001
      firstName: Alice
      lastName: Smith
      department: Engineering
      startDate: 2023-01-15
      salary: 75000.00
  - !employee!PartTime: # Tag for a part-time employee
      id: EMP002
      firstName: Bob
      lastName: Johnson
      department: Marketing
      startDate: 2024-03-01
      hoursPerWeek: 20

contractors:
  - !contractor!Individual: # Tag for an individual contractor
      id: CON001
      firstName: Charlie
      lastName: Brown
      company: Independent Solutions
      contractEndDate: 2025-12-31
      hourlyRate: 50.00
  - !contractor!Agency: # Tag for a contractor from an agency
      id: CON002
      firstName: Diana
      lastName: Miller
      company: Global Talent Agency
      contractEndDate: 2026-06-30
      agencyFee: 15.00
```

In the above YAML code, `%TAG !person! tag:example.com,2024:person/` defines `!person!` short alias for the longer URI `tag:example.com,2024:person/`. Here, `!person!` acts as a general tag handle for any person-related data. While it is not directly used in any data, it helps to organize and define the other specific person tags within the YAML document, providing a clear hierarchy and structure for custom data types. Then a `!employee!` tag handle is defined for employee-specific data using a more specific URI prefix and a `!contractor!` tag handle is defined for contractor-specific data with its own URI prefix. These are used to define different types of employees (using `!employee!FullTime`, !`employee!PartTime` tags) and contractors (using `!contractor!Individual`, `!contractor!Agency` tags).

Standard tags, local tags and global tags can be used depending on the specific needs of YAML data and its intended use:

- Standards tags are recommended when explicit declaration of common YAML data types (strings, integers, mappings, etc.) is needed to avoid any misinterpretations and ensure data is parsed correctly by different applications.
- Local tags can be used when custom data types are only relevant to a specific YAML file and don't require global uniqueness or external referencing.

- Global tags are preferred when data schemas need to be shared across multiple YAML files or different applications ensuring consistent data representation and type validation.

**Language-specific Tags:**
YAML also allows to use language-specific tags, which extend the basic YAML type system to represent objects from a specific programming language. These tags indicate that the subsequent YAML data represents an instance of a specific object or class within that programming language. When a YAML parser encounters a language-specific tag, it uses its knowledge of that tag and the associated language's object model to instantiate the correct object type and populate the object attributes from the YAML data.

Language-specific tags are denoted with a double exclamation mark (**`!!`**) followed by the language and then the specific object or type such as `!!python/object`, `!!java/object`, `!!ruby/object`, `!!perl/regexp`, etc. For example, a Python YAML parser uses `!!python/object:module.ClassName` tag to represent an instance of a Python class, `!!python/tuple` tag to represent a Python tuple, `!!python/set` tag to represent a Python set, `!!python/complex` tag to represent a Python complex number.

```
# Represents a Python complex number
number: !!python/complex 1+2j

# Represents a Python tuple
? !!python/tuple [ 5, 7 ]: Fifty Seven

# Represents an instance of a custom Python class
person: !!python/object:__main__.person
  name: John Doe
  age: 30
```

Note that constructing objects using language-specific tags can introduce security vulnerabilities if YAML documents are loaded from untrusted sources. To avoid this, always use safe loading functions *(such as `yaml.safe_load()` in **PyYAML**)* when dealing with untrusted YAML data, as these functions limit the types of objects that can be constructed.

## 5.5 Schema

A Schema is the connection between Tags in YAML and Classes (or data types) in the programming language. Schemas in YAML can be thought of as the way that YAML parser resolves or understands data in a YAML file. YAML uses schemas to determine how tags are interpreted and resolved to native data types.

YAML 1.2 specifically uses three default schemas that build up on each other:

- **FailSafe Schema**: It is the most minimalist schema which only understands strings (`!!str`), maps (`!!map`) and sequences (`!!seq`) and is guaranteed to work for any YAML file due to its simplicity, but it does not support any complex tags.

- **JSON Schema**: It is a foundational schema designed to reliably parse equivalent YAML and JSON files to the same end result. This is the most commonly used schema and is the starting point for most schemas. It understands all types supported within JSON format including integers (`!!int`), floating numbers (`!!float`), Booleans (`!!bool)` and nulls (`!!null)` in addition to the ones supported by the FailSafe schema.

- **Core Schema:** It is the default YAML schema which is an extension of the JSON schema, offering more flexible matching rules for nulls and booleans by supporting the same type but in multiple forms, making YAML files more human-readable. For example, `null` or `Null` or `NULL` will all be resolved to the same `null` type and `true` or `True` or `TRUE` will all be resolved to the same `boolean` type.

  `!!null` matching rule is defined as `null` or `Null` or `NULL` or ~ or empty scalar
  `!!bool` matching rule is defined as `true` or `True` or `TRUE` for true statement and `false` or `False` or `FALSE` for false statement
  `!!int` Integer Base 10 matching rule is defined as `[-+]? [0-9]+`
  `!!int` Octal matching rule is defined as `0o [0-7]+`
  `!!int` Hex matching rule is defined as `0x [0-9a-fA-F]+`
  `!!float` Float matching rule is defined as `[-+]? ( \. [0-9]+ | [0-9]+ ( \. [0-9]* )? ) ( [eE] [-+]? [0-9]+ )?`
  `!!float` infinity matching rule is defined as `[-+]?` ( `\.inf` or `\.Inf` or `\.INF` )
  `!!float` Not a Number matching rule is defined as `\.nan` or `\.NaN` or `\.NAN`

Note that YAML 1.1 version does not have schema concept but it has **Types** concept where the mandatory types/tags are `!!str`, `!!map` and `!!seq` and some optional types are `!!int`, `!!float`, `!!null`, `!!bool`, `!!binary`, `!!timestamp`, etc.

YAML 1.2 also allows to create custom schemas based on the JSON Schema since YAML Schema is a small extension of **JSON Schema Draft** 4. To create a custom schema, a fundamental understanding of JSON Schema concepts like data types *(`type`)* such as `string`, `number`, `integer`, `boolean`, `object`, `array` and `null`, expected properties within JSON object *(`properties`)*, required fields *(`required`)*, limitation on values *(`minLength`/`maxLength` for strings, `minimum`/`maximum` for numbers, `minItems`/`maxItems` for arrays, expected `format`)*, allowed patterns *(`pattern`)*, restrict values *(`enum`)*, validate instance in one or any or all schemas *(`oneOf`, `anyOf`, `allOf`)*, etc. is essential. Refer to official JSON schema website to know more.

Custom YAML schema file can be created either in JSON format with `.json` file extension or in YAML format with `.yaml` file extension. This file will describe the expected structure of YAML documents. Custom schemas are helpful when configuration files include custom objects or when language specific object serialization needs to be created.

For example, the following YAML schema defines a person:

```
# Example custom_schema.yaml
$schema: http://json-schema.org/draft-07/schema#
title: My Custom YAML Schema
description: Schema for a person

type: object
properties:
  name:
      type: string
      description: Name of the person
      minLength: 3
    age:
      type: integer,
      description: Age of the person,
      minimum: 0,
      maximum: 120
    email:
      type: string,
      description: Email address of the person,
      format: email
required:
  - name
  - age
```

This schema specifies that a person is an object with three properties: `name`, `age` and `email`. The `name` property must be a `string` with minimum of `3` characters, the `age` property must be an `integer` ranging from `0` to `120` and `email` property must be a `string` to be specified in `email` format. Out of these 3 properties, the `name` and `age` properties are marked as required which must be specified in the YAML document where this schema is being referenced.

Once the custom schema is defined, it can be referenced at the top of the YAML file by including a comment with `$schema:` keyword.

```
# $schema: ./custom_schema.yaml
person:
  - name: John Doe
    age: 30
    email: jane.doe@example.com
  - name: Jane Smith
    age:25
```

A YAML processor often also implements a language specific schema for serializing objects. Below is how serializing of generic objects looks like in several languages:

```
---
# Ruby Psych
dice: !ruby/Object:Dice [3, 6]

---
# perl YAML::XS, YAML.pm, YAML::Syck (Dump and Load)
dice: !!perl/array:Dice [3, 6]

---
# perl YAML.pm, YAML::Syck (Load)
dice: !perl/array:Dice [3, 6]

---
# Pyyaml
dice: !!python/object/new:__main__.Dice
  - !!python/tuple [3, 6]
```

## 5.6 Mapping Key

In a YAML mapping, keys are typically scalar values like strings, numbers, or booleans. However, YAML also supports complex keys where the key itself is a more complex data structure (such as a mapping or sequence) or contains special characters. Complex keys are explicitly denoted by a question mark **?** followed by the key's content. When a key is prefixed with ? , it signifies that a complex mapping key is defined. This explicit mapping syntax tells the YAML parser that the entire block following the question mark until the colon ' should be treated as the key, even if it contains elements that would normally be interpreted as values or part of the structure.

The mapping key is primarily used in complex or ambiguous scenarios where there is a chance that key might be misinterpreted as a value or a sequence item. For example, if a key itself contains special characters or needs to be multiline, using ? helps to explicitly define it as a key.

```
---
name: John Doe

# Explicit key-value pair using '?' for a multi-line key
? |
  This is a
  multi-line key
: This is the value for the multi-line key.

# Explicit key-value pair with a list as key
? - item1
  - item2
: This is the value associated with the list key
```

## 5.7 Anchors & Aliases

YAML **Anchors** and **Aliases** are powerful features that facilitate the reuse of data within a YAML document to avoid redundancy and improve maintainability by allowing to define a block of configuration once and then reference it multiple times throughout the YAML document. They are especially useful for creating templates or for sharing common data between multiple parts of an application. YAML anchors and aliases cannot contain the **[**, **]**, **{**, **}**, and **,** characters.

An **Anchor** is a named reference point defined on a YAML node (scalar, mapping, or sequence) to mark a specific value or a block of data for reuse. It is denoted by using ampersand symbol (**&**) followed by a unique name.

```
# Define a base person location using an anchor
person_location: &default_location
  location: New York
  country: USA
```

In the above example, `&default_location` defines an anchor named `default_location` for the `person_location` mapping.

An **Alias** is a reference to a previously defined anchor. It is denoted by using an asterisk symbol (*****) followed by the name of the anchor. When an alias is used, the value or block of data associated with the referenced anchor is effectively copied to the alias's location in the document.

```
person:
  - name: John Doe # First person, referring default location
    age: 30
    location: *default_location
  - name: Jane Smith # Second person, referring default location
    age: 25
    location: *default_location
```

In the above example, `*default_location` is an alias used to reference content in `default_location` anchor which was defined earlier.

YAML provides a feature to override the content referenced in the anchor while using the alias. The merge key denoted by `<<:` allows merging an aliased mapping into the current mapping. If same keys are used in aliased mapping and current mapping, the current mapping's value takes precedence.

```
# Define a base person configuration using an anchor
person_defaults: &default_person_info
  name: Unknown
  occupation: Software Engineer
  skills:
    - Programming
    - Problem-solving
    - Communication
  location: New York
  country: USA
person:
  - <<: *default_person_info # Merge the common person attributes
    name: John Doe # Override the default name
    age: 30
  - <<: *default_person_info # Merge the common person attributes
```

```
    name: Jane Smith # Override the default name
    age: 25
    skills: # Override the entire skills list
    - Project Management
    - Team Leadership
    - Budgeting
```

In the above example, both `John Doe` and `Jane Smith` inherit content from the `&default_person_info` anchor, with `John Doe` overriding `name` and `Jane Smith` overriding `name` and `skills`. This eliminates the need to repeat `occupation`, `location` and `country` for each person reducing redundancy and simplifying maintenance.

## 5.8 Escape Sequences

YAML files treat **#**, **&**, **?**, **|**, **:**, *****, **@**, **%**, **-**, **{**, **}**, **[**, **]**, **<**, **>**, **=**, **!**, **'**, **"**, **,** , **\**, etc. as special characters. When these special characters are actually part of the data or value, they can be escaped in many different ways though enclosing a string in double quotes is the most common way to escape special characters.

YAML offers **Escape Sequences** to indicate if the character should be interpreted as a special character or as part of the scalar. There are four types of Escape Sequences that can be used in YAML:

- **Single Quoted Escapes** allows to enclose a string in single quotes (`'`) which treats all characters (except single quote itself) literally, meaning backslashes are not interpreted as special character. To include a literal single quote within a single-quoted string, it must be escaped by using two consecutive single quotes (`''`).

```
text: 'This string contains a literal backslash: \\ and a single
quote: '\''Hello'\'''
```

- **Double Quoted Escapes** allows to enclose a string in double quotes (`"`) and use backslash for escape sequences to represent special characters like new lines (`\n`), tab (`\t`), or to include backslash (`\\`) or even double quotes (`\"`) within the string.

```
text: "This string has a tab \t and a newline: \n along with backslash
and a double quote: \"Hello\\\""
```

- **Entity Escapes** are HTML characters that have special meaning in HTML and can be used in YAML to represent special characters within data so that characters are treated as literal content rather than indicators of structure. A string using entity escapes must be enclosed in double quotes (").

  For example, entity escapes like
  `&#x20;` is used to represent space character.
  `&#58;` is used to represent colon character.
  `&lt;` is used to represent less than character.
  `&gt;` is used to represent greater than character.
  `&amp;` is used to represent ampersand character.

```
person:
  name: John Doe
  title: "Senior Developer &amp; Team Lead" # Using &amp; for &
character
  contact:
    email: "jane.doe&#x40;example.com"  # Using &#x40; for @ character
  skills:
    - Programming: "Java, Python"
    - Web Development: "HTML5, CSS3, &lt;Frameworks&gt;" # Using &lt;
for < character and &gt; for > character
  experience:
    - company: "Tech Solutions, Inc."
      role: "Lead Developer (2020 &ndash; Present)" # Using &ndash;
for en dash
    - company: "Creative Minds Corp."
      role: "Junior Developer (2018 &mdash; 2020)" # Using &mdash; for
em dash
```

- **Unicode Escapes** can also be used to represent special characters in format `\uXXXX` where XXXX is the hexadecimal Unicode point. This is particularly useful when dealing with characters that might not be easily typable or displayed in editor, or when need to embed Unicode characters within a string that also contains special YAML characters. A string using Unicode escapes must be enclosed in double quotes (").

  For example, Unicode escapes like
  `\u0020` is used to represent space character.
  `\u0027` is used to represent single quote character.
  `\u0022` is used to represent double character.

```
# Representing unicode characters
"Name in Japanese": "\u5C71\u7530 \u592A\u90CE" # Yamada Tarō (山田 太郎
) in Japanese language

# Combining unicode escapes with other escape sequences
copyright: "Copyright symbol: \u00A9\nNewline and tab: \t"
```

## 5.9 Advanced Data Types

Beyond basic scalars (strings, numbers, booleans) and collections (sequences and mappings), YAML supports advanced data types such as ordered mappings, sets and pairs that provide greater flexibility while representing data.

- **Ordered Mappings:**

  The standard YAML mappings are unordered in nature which means when a YAML parser processes a mapping, it does not guarantee that the order of key-value pairs in the input file is preserved after parsing. To maintain the order of key-value pairs, YAML provides the `!!omap` tag that defines an ordered sequence of unique key-value pairs.

```
person: !!omap
  - name: John Doe
  - age: 30
  # Representing ordered mapping in flow style
  - address: !!omap [street: 101 Avenue, city: New York, country: USA]
```

Some YAML parsers may not support ordered mappings in which case the order can be stored separately in a YAML sequence which is used to iterate through the mapping.

```
ordered_person_data:
  order:
    - name
    - age
    - location
  person:
    name: John Doe
    age: 30
    location: USA
```

In the above example, the `order` sequence explicitly defines the desired order of keys from the `person` mapping. The application logic then uses this order sequence to iterate through the `person` mapping in the specified order.

- **Sets:**

  In YAML, a set is a mapping that represents as an unordered collection of unique keys where each key has a `null` value. A YAML set can also be explicitly declared using `!!set` tag.

  A set in YAML can be represented in three ways:

  - Using the **?** indicator explicitly indicates a set by placing a **?** before each item, signifying that item is a key in a mapping with a null value.
  - Using a flow-style notation similar to a JSON object with keys to represent a mapping where all values are implicitly null.
  - Using the `!!set` tag explicitly indicates a set.

```
person:
  name: Alice Smith
  skills: # Represents a set with ? indicator
    ? Python
    ? Java
  hobbies: !!set # Represents an explicit set with !!set tag
    ? Reading
    ? Hiking
  contact: { email, phone }# Represents a set in flow style
  address: # Represents a set since all keys has null values
    city: null
    country: null
```

  In the above example, the **?** before each item signifies that it is a key in a mapping, and implicitly considered to have a null value (since no value is provided).

- **Pairs:**

  YAML pairs, denoted using `!!pairs` tag, defines a mapping with ordered sequence of key-value pairs where keys can be duplicated. Using `!!pairs` tag preserves the order in which key-value pairs appear even after parsing. This is useful in scenarios where both the order of key-value pairs and the possibility of duplicate keys are significant. For example, it can be used to represent data structures that are ordered lists of named values, which is commonly seen when modeling data like XML documents. However, many programming languages do not have a direct native equivalent for this data type, in which case a library support is needed for handling `!!pairs` when parsing YAML.

```
name: John Doe
age: 30
address: !!pairs
  - home:
      street: 10 Downing Street
      city: London
      country: UK
      start_date: 2010-01-01
      end_date: 2015-12-31
  - work:
      street: 1600 Pennsylvania Avenue NW
      city: Washington DC
      country: USA
      start_date: 2016-01-01
      end_date: 2018-06-30
  - work:
      street: 17 Avenue
      city: New York
      country: USA
      start_date: 2018-07-01
      end_date: null # null implies current or ongoing
```

In the above example, the `work` address appears twice indicating a move from one work location to another, with both previous and current address listed in the order worked.

## 6 YAML INDICATORS

It is important to understand that YAML uses specific characters (called as indicators) for specific functionality to describe the content of YAML document.

Some of these indicators are listed below:

**_** indicator denotes a block sequence entry

**?** indicator denotes a mapping key

**:** indicator denotes a mapping value

**,** indicator denotes flow collection entry

**[** indicator denotes the start of a flow sequence

**]** indicator denotes the end of a flow sequence

**{** indicator denotes the start of a flow mapping

**}** indicator denotes the end of a flow mapping

**#** indicator denotes comments

**&** indicator denotes anchor node

***** indicator denotes alias node

**!** indicator denotes tag handle

**|** indicator denotes a literal block scalar

**>** indicator denotes a folded block scalar

**'** indicator is used to surround a single quoted flow scalar

**"** indicator is used to surround a double quoted flow scalar

**%** indicator denotes the directive used

YAML provides an official Reference Card which lists out all characters used for specific YAML representation.

```
# This is a YAML example showcasing all it's indicators
# Lines starting with # are ignored by parsers.

%YAML 1.2 # It specifies the version that this document adheres to
%TAG !app! tag:mydomain.com,2025: # It creates `!app!` shorthand name for
the specified URI

--- # It indicates the start of a YAML document
server_config:
  # Mappings
  host: "api.example.com" # Quoted to avoid misinterpretation due to .
character
  port: 8080
  # Sequences
  endpoints:
    - /data
    - /status
    - /healthz
    # Flow styles use brackets and braces for in-line collections
    - [error, timeout, redirect]

  # Scalars
  is_secure: true # Boolean scalar
  user_agent: 'MyApp/1.0 (#version-info)' # Single-quoted scalar with
special characters
  status_message: "Server is running.\nAll systems green." # Single-quoted
scalar to allow escape sequences

  # Block scalars
  literal_text: | # Literal block to preserve line breaks and indentation
    This is a block of text.
      The indentation here is preserved.
    Newlines are kept exactly as written.

  folded_text: > # Folded block to fold newlines into spaces
    This is a very long text that can
    be folded into a single line.

      Blank lines, however,
      will remain as line breaks.

  # Anchor and Alias
  default_user: &DEFAULT_USER # Anchored to reuse later
    name: guest
    roles: [read-only]

  current_user: *DEFAULT_USER # Aliased to replace with anchor content

  # Tags
  admin_user: !app!user # Using named tag handle defined by %TAG directive
above
    name: admin
    roles: !!seq [admin, read-only] # Using built-in type to represent a
list
  binary_data: !!binary "QmFzZTY0IGV4YW1wbGU=" # Using built-in type to
represent binary value

# Multiple documents
--- # It indicates a second, independent document
? - production # `?` indicator is used for non-scalar keys
  - database_host
:
  - primary_db: "prod-db-1"
  - secondary_db: "prod-db-2"
... # It indicates the end of the file or a document
```

# 7 YAML VALIDATORS

A YAML file can be created or modified using any text editor such as VS Code (with YAML plugins), PyCharm (with built-in YAML support), Sublime Text (with YAML syntax highlighting) but it is recommended to validate YAML file for any syntactical errors like using tabs in place of spaces, weird issues like repeated characters, trailing spaces, etc. using YAML linting tools.

There are many CLI and online tools available to ensure correct YAML structure is followed:

- **CLI Tools:**

  CLI tools such as yamllint, yamle, yq, yaml-validator, yaml-lint, etc. are available in market to validate and format YAML files. The most popular tool is **yamllint** which is a Python based linter that not only checks for syntax validity, but also for weird nesses like key repetition and cosmetic problems such as lines length, trailing spaces, indentation, etc. Follow this documentation on installing `yamllint` via command line.

- **Online Tools:**
  The popular online tools are YAML Lint (checks for YAML syntax errors and cosmetic issues), JSON Formatter's YAML Validator (validates and converts JSON to YAML format) and Online YAML Parser (converts YAML to JSON or Python format) that can be used to parse YAML structure online.

Additionally, several programming languages can be utilized for parsing and validation. The commonly used YAML parsers include **PyYAML** (`pyyaml` library) or Ruamel YAML (`ruamel.yaml` library) for Python, **SnakeYAML** (`snakeyaml` library) for Java, **Rapid YAML** (`ryaml` library) for C++, **JS YAML** (`js-yaml` library) for Java Script, **Go YAML** (`go-yaml` library) for Go, **YAML DotNet** (`YamlDotNet` library) for .Net, **YAML Rust** (`yaml-rust` library) for Rust language and many more.

For example:

- Parse in **Python** language with `pyyaml` library installed:

```
python -c "from yaml import load, Loader; load(open('test.yml'),
Loader=Loader)"
```

- Parse in **Ruby** language with `rbyaml` package installed:

```
ruby -rbyaml -e "p yaml.load(STDIN.read)" < test.yml
```

- Parse in **Perl** language with `YAML.pm` module installed:

```
perl -MYAML -e 'use YAML;YAML::LoadFile("./test.yml")'
```

# 8 YAML IN PYTHON

As Python being the most widely used programming language, it is good to understand how to read, write, validate, and edit YAML file effectively in Python. The most popular YAML library that comes with Python is `pyyaml` which is primarily built on YAML 1.1 specification. As an alternative to PyYAML, `ruamel.yaml` library is also widely used with Python as it has greater support for YAML 1.2 specification including better JSON compatibility while it has backward compatibility of YAML 1.1. Note that `ruamel.yaml` is an enhanced version of the standard PyYAML library and its primary distinguishing feature is round-trip capability, which preserves a YAML file's original formatting, comments, and style even after it gets loaded, modified, and saved again by a program.

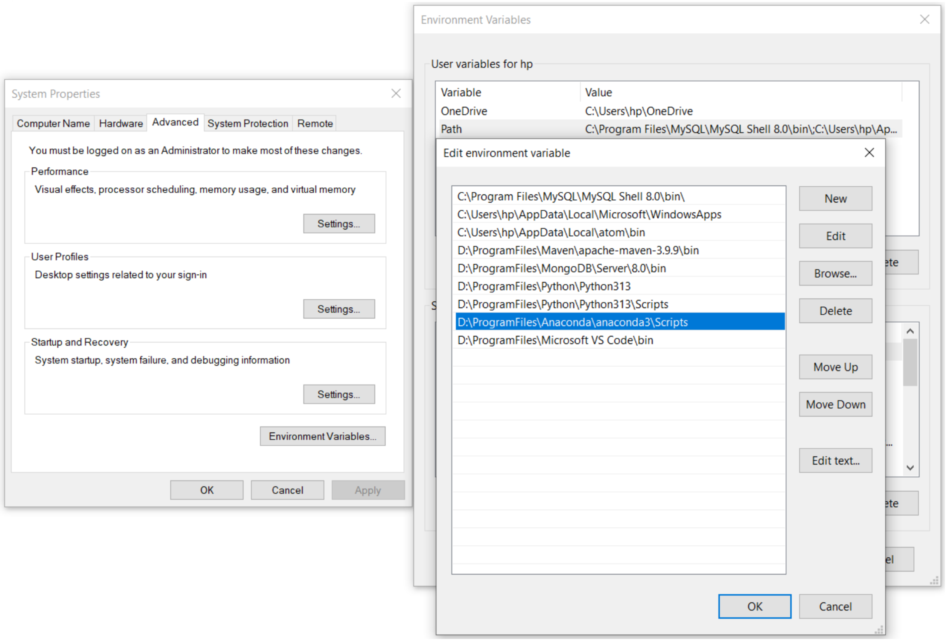
PyYAML is most widely used due to its simplicity and effectiveness supporting all Python 2 and Python 3 releases. With `PyYAML` library, Python developers can easily read from and write to YAML files including simple configuration settings or complex data structures, making it simple to integrate YAML into projects.

## 8.1 Verify Python in Windows

PyYAML requires a working installation of Python on any operating system such as Windows, Linux or MacOS. For the latest versions of PyYAML such as **PyYAML 6.0** and later, **Python 3.5** or higher version is required.

If Python software is not already installed in your operating system, you can install it from the Python Downloads website but it is recommended to install Anaconda Distribution which is an open source platform that comes with prepackaged with a vast collection of commonly used libraries built for Python and R programming languages that are heavily used in Data Science, Data Engineer and Data Analytics fields. Refer to Official Anaconda Installation Guide on how to install it.

Once Python or Anaconda is installed, configure the `PATH` environment variable defining the standalone Python installation path or Anaconda installation path.



You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your sign-in

Startup and Recovery
System startup, system failure, and debugging information

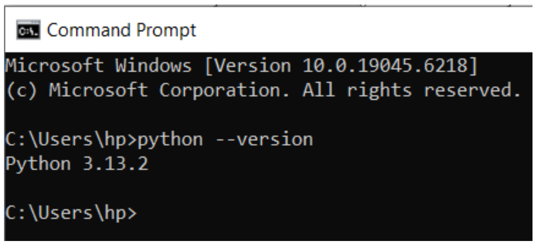
User variables for hp

| Variable | Value |
|---|---|
| OneDrive | C:\Users\hp\OneDrive |
| Path | C:\Program Files\MySQL\MySQL Shell 8.0\bin\;C:\Users\hp\Ap... |

```
C:\Program Files\MySQL\MySQL Shell 8.0\bin\
C:\Users\hp\AppData\Local\Microsoft\WindowsApps
C:\Users\hp\AppData\Local\atom\bin
D:\ProgramFiles\Maven\apache-maven-3.9.9\bin
D:\ProgramFiles\MongoDB\Server\8.0\bin
D:\ProgramFiles\Python\Python313
D:\ProgramFiles\Python\Python313\Scripts
D:\ProgramFiles\Anaconda\anaconda3\Scripts
D:\ProgramFiles\Microsoft VS Code\bin
```

To verify the standalone version of Python installed in Windows, open a new **Command Prompt** or **Windows PowerShell** application and run the following command:

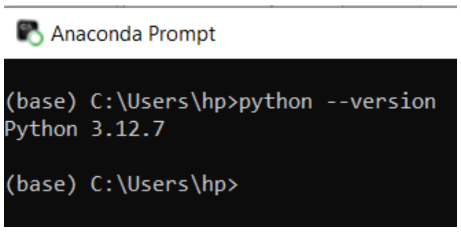
```
python --version
```



```
Microsoft Windows [Version 10.0.19045.6218]
(c) Microsoft Corporation. All rights reserved.

C:\Users\hp>python --version
Python 3.13.2

C:\Users\hp>
```

To verify the Python version installed with Anaconda distribution in Windows, open a new **Anaconda Prompt** application and run the following command:

```
python --version
```



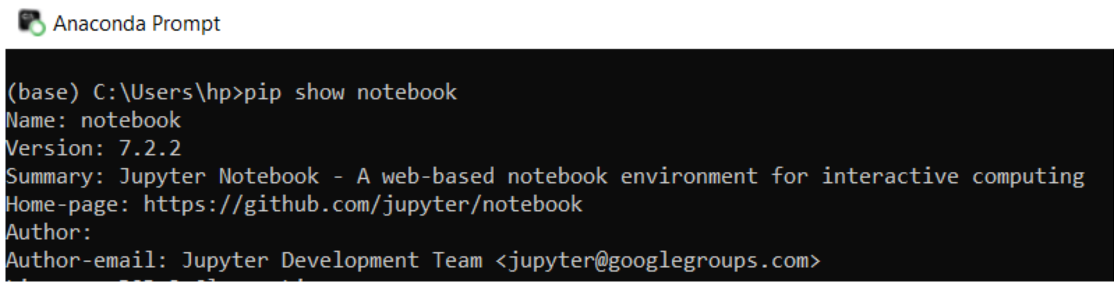
```
(base) C:\Users\hp>python --version
Python 3.12.7

(base) C:\Users\hp>
```

If Anaconda distribution is installed, it allows to verify many preinstalled Python IDEs (Integrated Development Environments) such as **Jupyter Notebook**, **JupyterLab**, **Spyder** etc. *(a Python IDE is recommended to execute Python code and see results line by line)*.

For example, check the **Jupyter Notebook** installation using either of the below ways:

- Open **Anaconda Prompt** and run the following command:

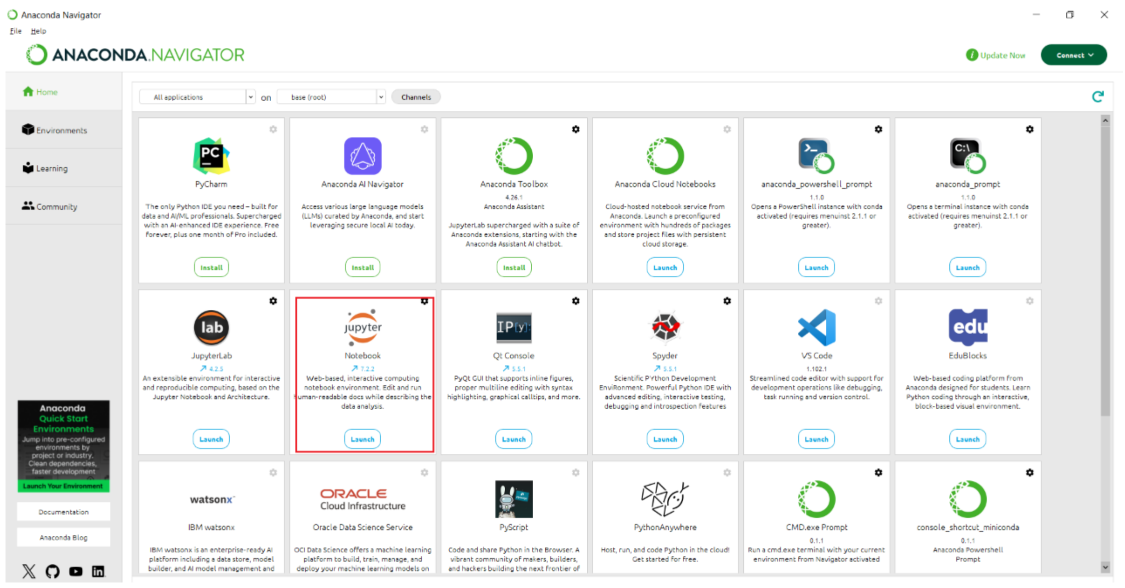
```
pip show notebook
```



Anaconda Prompt

```
(base) C:\Users\hp>pip show notebook
Name: notebook
Version: 7.2.2
Summary: Jupyter Notebook - A web-based notebook environment for interactive computing
Home-page: https://github.com/jupyter/notebook
Author:
Author-email: Jupyter Development Team <jupyter@googlegroups.com>
```

- Open the **Anaconda Navigator** application to see a list of various Python IDEs including **Jupyter Notebook** installed which can be launched directly from the Anaconda Navigator.



PyCharm
The only Python IDE you need – built for data and AI/ML professionals. Supercharged with an AI-enhanced IDE experience. Free forever, plus one month of Pro included.
Install

Anaconda AI Navigator
Access various large language models (LLMs) curated by Anaconda, and start leveraging secure local AI today.
Install

Anaconda Toolbox
4.26.1
Anaconda Assistant
JupyterLab supercharged with a suite of Anaconda extensions, starting with the Anaconda Assistant AI chatbot.
Install

Anaconda Cloud Notebooks
Cloud-hosted notebook service from Anaconda. Launch a preconfigured environment with hundreds of packages and store project files with persistent cloud storage.
Launch

anaconda_powershell_prompt
1.1.0
Opens a PowerShell instance with conda activated (requires menuinst 2.1.1 or greater).
Launch

anaconda_prompt
1.1.0
Opens a terminal instance with conda activated (requires menuinst 2.1.1 or greater).
Launch

JupyterLab
4.2.5
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.
Launch

Notebook
7.2.2
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.
Launch

Qt Console
5.5.1
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.
Launch

Spyder
5.5.1
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Launch

VS Code
1.102.1
Streamlined code editor with support for development operations like debugging, task running and version control.
Launch

EduBlocks
Web-based coding platform from Anaconda designed for students. Learn Python coding through an interactive, block-based visual environment.
Launch

IBM watsonx
IBM watsonx is an enterprise-ready AI platform including a data store, model builder, and AI model management and

Oracle Data Science Service
OCI Data Science offers a machine learning platform to build, train, manage, and deploy your machine learning models on

PyScript
Code and share Python in the Browser. A vibrant community of makers, builders, and hackers building the next frontier of

PythonAnywhere
Host, run, and code Python in the cloud! Get started for free.

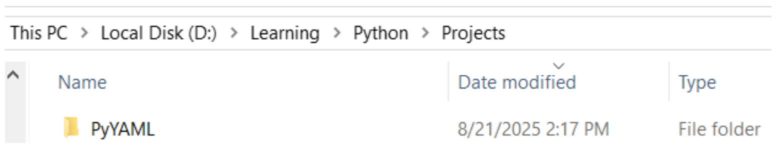
CMD.exe Prompt
0.1.1
Run a cmd.exe terminal with your current environment from Navigator activated

console_shortcut_miniconda
0.1.1
Anaconda Powershell Prompt

## 8.2 Setup Virtual Environment

Before installing Python libraries, it is a best practice to create a project specific folder and setup a virtual environment to isolate dependencies from the existing Python environment, preventing conflicts with other Python projects.

Here, I am creating a new folder named `PyYAML` under `D:\Learning\Python\Projects` folder in my Windows system.

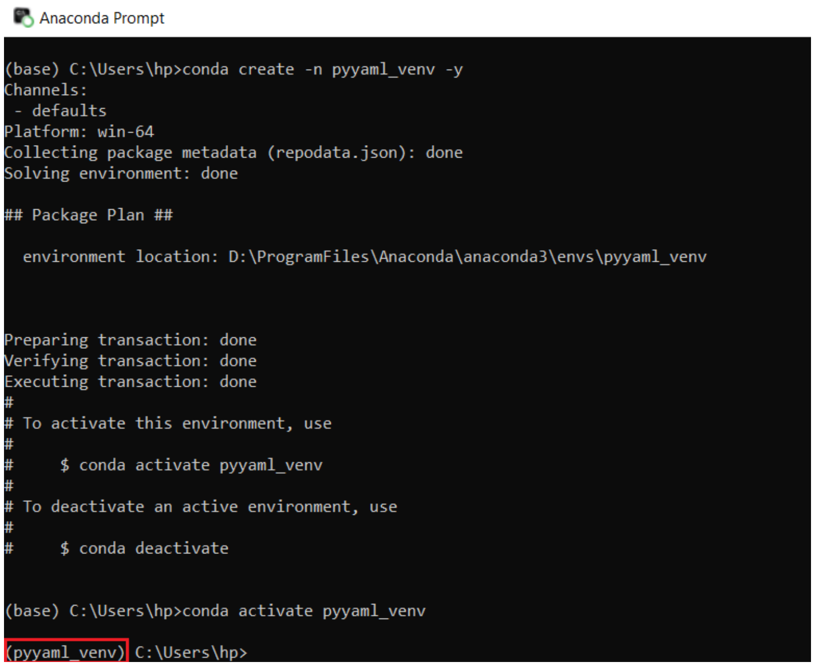


| Name | Date modified | Type |
|---|---|---|
| PyYAML | 8/21/2025 2:17 PM | File folder |

To create a virtual environment using Anaconda, open **Anaconda Prompt** and run the following commands to create and activate the virtual environment named `pyyaml_venv`.

```
conda create -n pyyaml_venv -y
conda activate pyyaml_venv
```

Once the virtual environment is activated, it displays `(pyyaml_venv)` before the prompt.



```
(base) C:\Users\hp>conda create -n pyyaml_venv -y
Channels:
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: D:\ProgramFiles\Anaconda\anaconda3\envs\pyyaml_venv



Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate pyyaml_venv
#
# To deactivate an active environment, use
#
#     $ conda deactivate


(base) C:\Users\hp>conda activate pyyaml_venv

(pyyaml_venv) C:\Users\hp>
```

To create a virtual environment using standalone Python, create a new folder and open a **Command Prompt** and run the following commands to create and activate the virtual environment named `pyayml_venv`.

```
cd <new_folder_path>
python -m venv pyyaml_venv
pyyaml_venv\Scripts\activate
```

Once the virtual environment is activated, it displays `(pyaml_venv)` before the prompt.

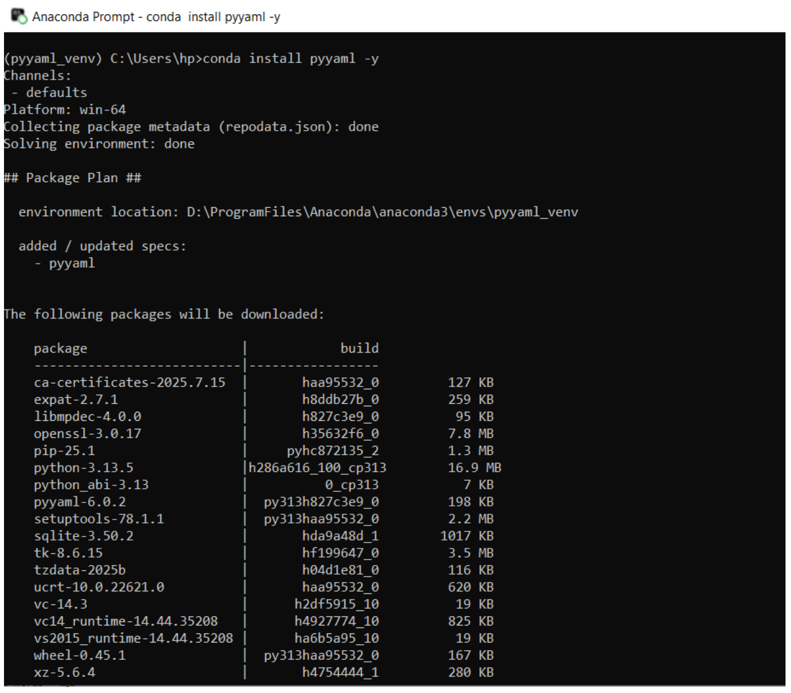
## 8.3 Install PyYAML Library

In the virtual environment created using Anaconda, install the `pyyaml` library using either of the following commands:

```
conda install pyyaml
```

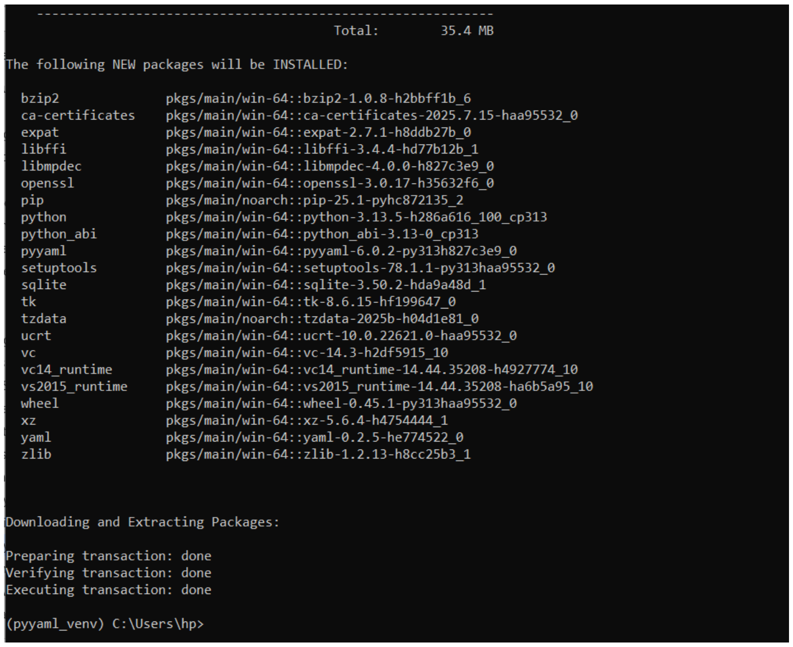
or

```
pip install pyyaml
```



Anaconda Prompt - conda install pyyaml -y

```
(pyyaml_venv) C:\Users\hp>conda install pyyaml -y
Channels:
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: D:\ProgramFiles\Anaconda\anaconda3\envs\pyyaml_venv

  added / updated specs:
    - pyyaml


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    ca-certificates-2025.7.15  |       haa95532_0         127 KB
    expat-2.7.1                |       h8ddb27b_0         259 KB
    libmpdec-4.0.0             |       h827c3e9_0          95 KB
    openssl-3.0.17             |       h35632f6_0         7.8 MB
    pip-25.1                   |     pyhc872135_2         1.3 MB
    python-3.13.5              |h286a616_100_cp313        16.9 MB
    python_abi-3.13            |          0_cp313           7 KB
    pyyaml-6.0.2               |  py313h827c3e9_0         198 KB
    setuptools-78.1.1          |  py313haa95532_0         2.2 MB
    sqlite-3.50.2              |       hda9a48d_1        1017 KB
    tk-8.6.15                  |       hf199647_0         3.5 MB
    tzdata-2025b               |       h04d1e81_0         116 KB
    ucrt-10.0.22621.0          |       haa95532_0         620 KB
    vc-14.3                    |      h2df5915_10          19 KB
    vc14_runtime-14.44.35208   |      h4927774_10         825 KB
    vs2015_runtime-14.44.35208 |      ha6b5a95_10          19 KB
    wheel-0.45.1               |  py313haa95532_0         167 KB
    xz-5.6.4                   |       h4754444_1         280 KB
```



```
    ------------------------------------------------------------
                                           Total:        35.4 MB

The following NEW packages will be INSTALLED:

  bzip2              pkgs/main/win-64::bzip2-1.0.8-h2bbff1b_6
  ca-certificates    pkgs/main/win-64::ca-certificates-2025.7.15-haa95532_0
  expat              pkgs/main/win-64::expat-2.7.1-h8ddb27b_0
  libffi             pkgs/main/win-64::libffi-3.4.4-hd77b12b_1
  libmpdec           pkgs/main/win-64::libmpdec-4.0.0-h827c3e9_0
  openssl            pkgs/main/win-64::openssl-3.0.17-h35632f6_0
  pip                pkgs/main/noarch::pip-25.1-pyhc872135_2
  python             pkgs/main/win-64::python-3.13.5-h286a616_100_cp313
  python_abi         pkgs/main/win-64::python_abi-3.13-0_cp313
  pyyaml             pkgs/main/win-64::pyyaml-6.0.2-py313h827c3e9_0
  setuptools         pkgs/main/win-64::setuptools-78.1.1-py313haa95532_0
  sqlite             pkgs/main/win-64::sqlite-3.50.2-hda9a48d_1
  tk                 pkgs/main/win-64::tk-8.6.15-hf199647_0
  tzdata             pkgs/main/noarch::tzdata-2025b-h04d1e81_0
  ucrt               pkgs/main/win-64::ucrt-10.0.22621.0-haa95532_0
  vc                 pkgs/main/win-64::vc-14.3-h2df5915_10
  vc14_runtime       pkgs/main/win-64::vc14_runtime-14.44.35208-h4927774_10
  vs2015_runtime     pkgs/main/win-64::vs2015_runtime-14.44.35208-ha6b5a95_10
  wheel              pkgs/main/win-64::wheel-0.45.1-py313haa95532_0
  xz                 pkgs/main/win-64::xz-5.6.4-h4754444_1
  yaml               pkgs/main/win-64::yaml-0.2.5-he774522_0
  zlib               pkgs/main/win-64::zlib-1.2.13-h8cc25b3_1



Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done

(pyyaml_venv) C:\Users\hp>
```

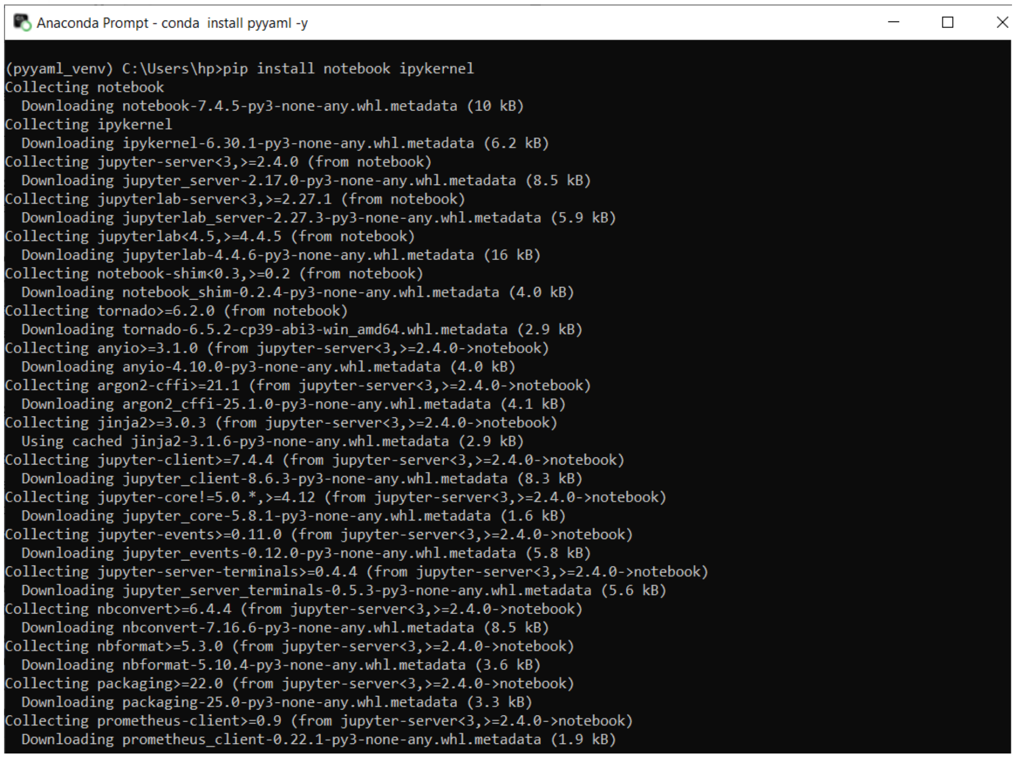
## 8.4 Install Interactive Python IDE

Though this is not mandatory, it is good to use any interactive Python Integrated Development Environment (IDE) such as **Jupyter Notebook**, **Spyder**, etc. to easily write and execute Python code line by line, receiving immediate output on the console.

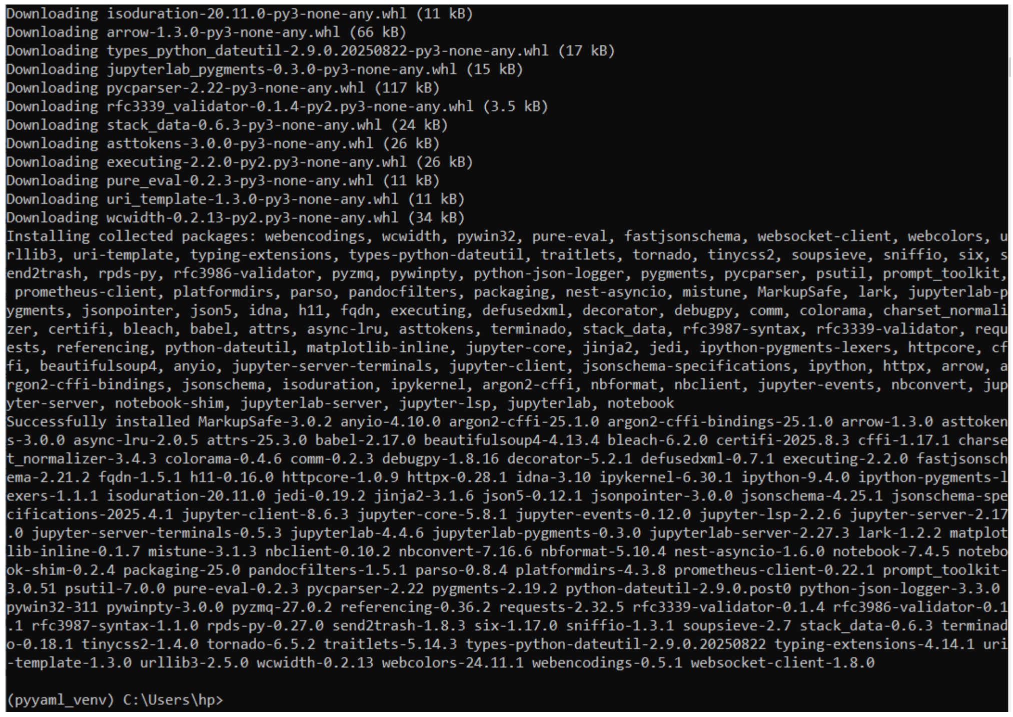
Anaconda distribution by default comes with **Jupyter Notebook** and **JupyterLab** IDEs which are widely used for interactive Python code execution but in a Python virtual environment created using Anaconda (or using standalone Python), Jupyter Notebook needs to be installed manually in the virtual environment.

In the Anaconda virtual environment, run the following command to install **Jupyter Notebook** and **IPython Kernel** (which is needed for Jupyter):

```
conda install jupyter ipykernel
```
or
```
pip install notebook ipykernel
```


Anaconda Prompt - conda  install pyyaml -y

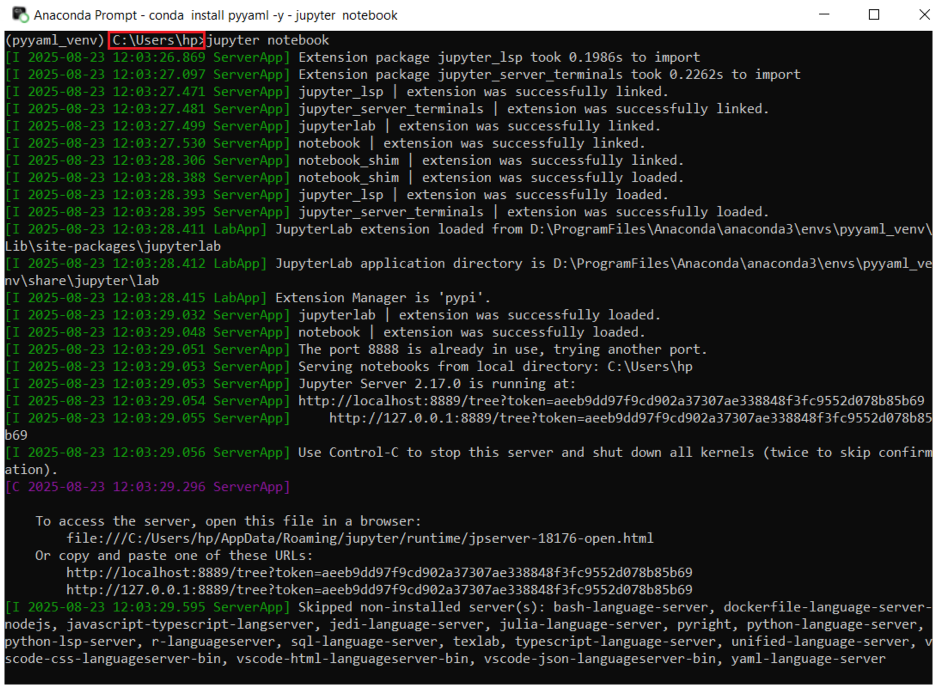
```
(pyyaml_venv) C:\Users\hp>pip install notebook ipykernel
Collecting notebook
  Downloading notebook-7.4.5-py3-none-any.whl.metadata (10 kB)
Collecting ipykernel
  Downloading ipykernel-6.30.1-py3-none-any.whl.metadata (6.2 kB)
Collecting jupyter-server<3,>=2.4.0 (from notebook)
  Downloading jupyter_server-2.17.0-py3-none-any.whl.metadata (8.5 kB)
Collecting jupyterlab-server<3,>=2.27.1 (from notebook)
  Downloading jupyterlab_server-2.27.3-py3-none-any.whl.metadata (5.9 kB)
Collecting jupyterlab<4.5,>=4.4.5 (from notebook)
  Downloading jupyterlab-4.4.6-py3-none-any.whl.metadata (16 kB)
Collecting notebook-shim<0.3,>=0.2 (from notebook)
  Downloading notebook_shim-0.2.4-py3-none-any.whl.metadata (4.0 kB)
Collecting tornado>=6.2.0 (from notebook)
  Downloading tornado-6.5.2-cp39-abi3-win_amd64.whl.metadata (2.9 kB)
Collecting anyio>=3.1.0 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading anyio-4.10.0-py3-none-any.whl.metadata (4.0 kB)
Collecting argon2-cffi>=21.1 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading argon2_cffi-25.1.0-py3-none-any.whl.metadata (4.1 kB)
Collecting jinja2>=3.0.3 (from jupyter-server<3,>=2.4.0->notebook)
  Using cached jinja2-3.1.6-py3-none-any.whl.metadata (2.9 kB)
Collecting jupyter-client>=7.4.4 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading jupyter_client-8.6.3-py3-none-any.whl.metadata (8.3 kB)
Collecting jupyter-core!=5.0.*,>=4.12 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading jupyter_core-5.8.1-py3-none-any.whl.metadata (1.6 kB)
Collecting jupyter-events>=0.11.0 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading jupyter_events-0.12.0-py3-none-any.whl.metadata (5.8 kB)
Collecting jupyter-server-terminals>=0.4.4 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading jupyter_server_terminals-0.5.3-py3-none-any.whl.metadata (5.6 kB)
Collecting nbconvert>=6.4.4 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading nbconvert-7.16.6-py3-none-any.whl.metadata (8.5 kB)
Collecting nbformat>=5.3.0 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading nbformat-5.10.4-py3-none-any.whl.metadata (3.6 kB)
Collecting packaging>=22.0 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading packaging-25.0-py3-none-any.whl.metadata (3.3 kB)
Collecting prometheus-client>=0.9 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading prometheus_client-0.22.1-py3-none-any.whl.metadata (1.9 kB)
```



```
Downloading isoduration-20.11.0-py3-none-any.whl (11 kB)
Downloading arrow-1.3.0-py3-none-any.whl (66 kB)
Downloading types_python_dateutil-2.9.0.20250822-py3-none-any.whl (17 kB)
Downloading jupyterlab_pygments-0.3.0-py3-none-any.whl (15 kB)
Downloading pycparser-2.22-py3-none-any.whl (117 kB)
Downloading rfc3339_validator-0.1.4-py2.py3-none-any.whl (3.5 kB)
Downloading stack_data-0.6.3-py3-none-any.whl (24 kB)
Downloading asttokens-3.0.0-py3-none-any.whl (26 kB)
Downloading executing-2.2.0-py2.py3-none-any.whl (26 kB)
Downloading pure_eval-0.2.3-py3-none-any.whl (11 kB)
Downloading uri_template-1.3.0-py3-none-any.whl (11 kB)
Downloading wcwidth-0.2.13-py2.py3-none-any.whl (34 kB)
Installing collected packages: webencodings, wcwidth, pywin32, pure-eval, fastjsonschema, websocket-client, webcolors, u
rllib3, uri-template, typing-extensions, types-python-dateutil, traitlets, tornado, tinycss2, soupsieve, sniffio, six, s
end2trash, rpds-py, rfc3986-validator, pyzmq, pywinpty, python-json-logger, pygments, pycparser, psutil, prompt_toolkit,
 prometheus-client, platformdirs, parso, pandocfilters, packaging, nest-asyncio, mistune, MarkupSafe, lark, jupyterlab-p
ygments, jsonpointer, json5, idna, h11, fqdn, executing, defusedxml, decorator, debugpy, comm, colorama, charset_normali
zer, certifi, bleach, babel, attrs, async-lru, asttokens, terminado, stack_data, rfc3987-syntax, rfc3339-validator, requ
ests, referencing, python-dateutil, matplotlib-inline, jupyter-core, jinja2, jedi, ipython-pygments-lexers, httpcore, cf
fi, beautifulsoup4, anyio, jupyter-server-terminals, jupyter-client, jsonschema-specifications, ipython, httpx, arrow, a
rgon2-cffi-bindings, jsonschema, isoduration, ipykernel, argon2-cffi, nbformat, nbclient, jupyter-events, nbconvert, jup
yter-server, notebook-shim, jupyterlab-server, jupyter-lsp, jupyterlab, notebook
Successfully installed MarkupSafe-3.0.2 anyio-4.10.0 argon2-cffi-25.1.0 argon2-cffi-bindings-25.1.0 arrow-1.3.0 asttoken
s-3.0.0 async-lru-2.0.5 attrs-25.3.0 babel-2.17.0 beautifulsoup4-4.13.4 bleach-6.2.0 certifi-2025.8.3 cffi-1.17.1 charse
t_normalizer-3.4.3 colorama-0.4.6 comm-0.2.3 debugpy-1.8.16 decorator-5.2.1 defusedxml-0.7.1 executing-2.2.0 fastjsonsch
ema-2.21.2 fqdn-1.5.1 h11-0.16.0 httpcore-1.0.9 httpx-0.28.1 idna-3.10 ipykernel-6.30.1 ipython-9.4.0 ipython-pygments-l
exers-1.1.1 isoduration-20.11.0 jedi-0.19.2 jinja2-3.1.6 json5-0.12.1 jsonpointer-3.0.0 jsonschema-4.25.1 jsonschema-spe
cifications-2025.4.1 jupyter-client-8.6.3 jupyter-core-5.8.1 jupyter-events-0.12.0 jupyter-lsp-2.2.6 jupyter-server-2.17
.0 jupyter-server-terminals-0.5.3 jupyterlab-4.4.6 jupyterlab-pygments-0.3.0 jupyterlab-server-2.27.3 lark-1.2.2 matplot
lib-inline-0.1.7 mistune-3.1.3 nbclient-0.10.2 nbconvert-7.16.6 nbformat-5.10.4 nest-asyncio-1.6.0 notebook-7.4.5 notebo
ok-shim-0.2.4 packaging-25.0 pandocfilters-1.5.1 parso-0.8.4 platformdirs-4.3.8 prometheus-client-0.22.1 prompt_toolkit-
3.0.51 psutil-7.0.0 pure-eval-0.2.3 pycparser-2.22 pygments-2.19.2 python-dateutil-2.9.0.post0 python-json-logger-3.3.0
pywin32-311 pywinpty-3.0.0 pyzmq-27.0.2 referencing-0.36.2 requests-2.32.5 rfc3339-validator-0.1.4 rfc3986-validator-0.1
.1 rfc3987-syntax-1.1.0 rpds-py-0.27.0 send2trash-1.8.3 six-1.17.0 sniffio-1.3.1 soupsieve-2.7 stack_data-0.6.3 terminad
o-0.18.1 tinycss2-1.4.0 tornado-6.5.2 traitlets-5.14.3 types-python-dateutil-2.9.0.20250822 typing-extensions-4.14.1 uri
-template-1.3.0 urllib3-2.5.0 wcwidth-0.2.13 webcolors-24.11.1 webencodings-0.5.1 websocket-client-1.8.0

(pyyaml_venv) C:\Users\hp>
```

Now, run the following command to start **Jupyter Notebook** which by default launches from the current directory (`C:\users\<username>`):

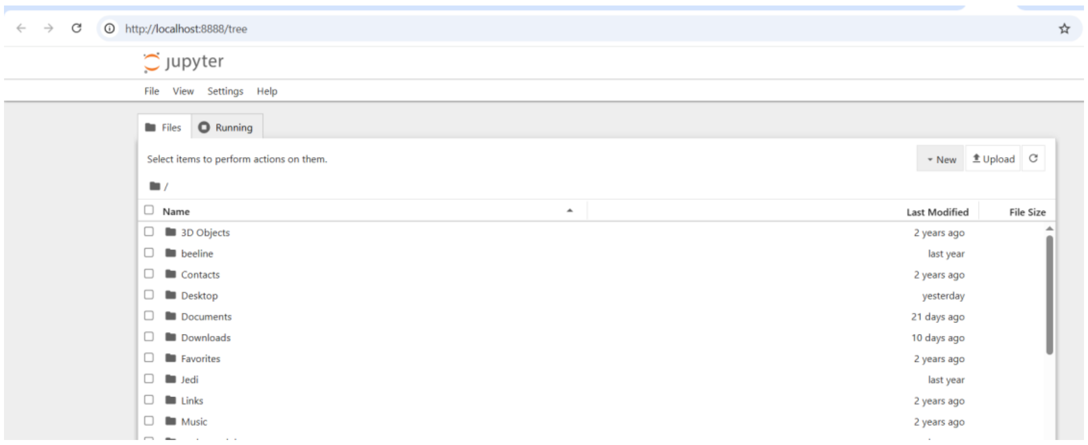
```
jupyter notebook
```



Anaconda Prompt - conda install pyyaml -y - jupyter notebook

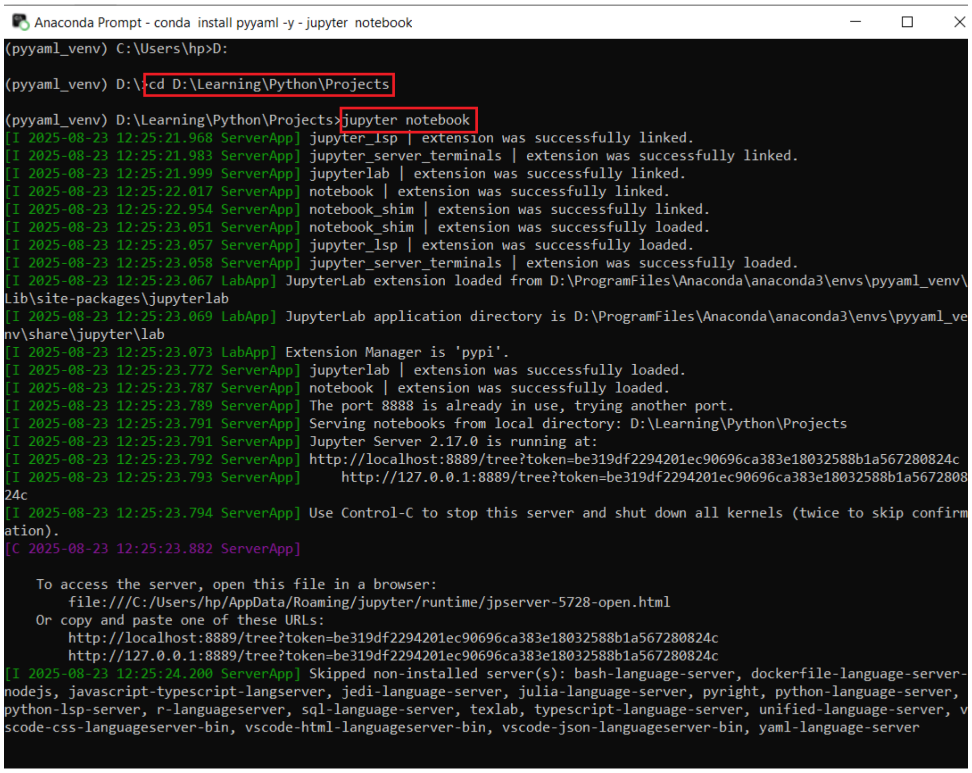
```
(pyyaml_venv) C:\Users\hp>jupyter notebook
[I 2025-08-23 12:03:26.869 ServerApp] Extension package jupyter_lsp took 0.1986s to import
[I 2025-08-23 12:03:27.097 ServerApp] Extension package jupyter_server_terminals took 0.2262s to import
[I 2025-08-23 12:03:27.471 ServerApp] jupyter_lsp | extension was successfully linked.
[I 2025-08-23 12:03:27.481 ServerApp] jupyter_server_terminals | extension was successfully linked.
[I 2025-08-23 12:03:27.499 ServerApp] jupyterlab | extension was successfully linked.
[I 2025-08-23 12:03:27.530 ServerApp] notebook | extension was successfully linked.
[I 2025-08-23 12:03:28.306 ServerApp] notebook_shim | extension was successfully linked.
[I 2025-08-23 12:03:28.388 ServerApp] notebook_shim | extension was successfully loaded.
[I 2025-08-23 12:03:28.393 ServerApp] jupyter_lsp | extension was successfully loaded.
[I 2025-08-23 12:03:28.395 ServerApp] jupyter_server_terminals | extension was successfully loaded.
[I 2025-08-23 12:03:28.411 LabApp] JupyterLab extension loaded from D:\ProgramFiles\Anaconda\anaconda3\envs\pyyaml_venv\
Lib\site-packages\jupyterlab
[I 2025-08-23 12:03:28.412 LabApp] JupyterLab application directory is D:\ProgramFiles\Anaconda\anaconda3\envs\pyyaml_ve
nv\share\jupyter\lab
[I 2025-08-23 12:03:28.415 LabApp] Extension Manager is 'pypi'.
[I 2025-08-23 12:03:29.032 ServerApp] jupyterlab | extension was successfully loaded.
[I 2025-08-23 12:03:29.048 ServerApp] notebook | extension was successfully loaded.
[I 2025-08-23 12:03:29.051 ServerApp] The port 8888 is already in use, trying another port.
[I 2025-08-23 12:03:29.053 ServerApp] Serving notebooks from local directory: C:\Users\hp
[I 2025-08-23 12:03:29.053 ServerApp] Jupyter Server 2.17.0 is running at:
[I 2025-08-23 12:03:29.054 ServerApp] http://localhost:8889/tree?token=aeeb9dd97f9cd902a37307ae338848f3fc9552d078b85b69
[I 2025-08-23 12:03:29.055 ServerApp]     http://127.0.0.1:8889/tree?token=aeeb9dd97f9cd902a37307ae338848f3fc9552d078b85
b69
[I 2025-08-23 12:03:29.056 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirm
ation).
[C 2025-08-23 12:03:29.296 ServerApp]

    To access the server, open this file in a browser:
        file:///C:/Users/hp/AppData/Roaming/jupyter/runtime/jpserver-18176-open.html
    Or copy and paste one of these URLs:
        http://localhost:8889/tree?token=aeeb9dd97f9cd902a37307ae338848f3fc9552d078b85b69
        http://127.0.0.1:8889/tree?token=aeeb9dd97f9cd902a37307ae338848f3fc9552d078b85b69
[I 2025-08-23 12:03:29.595 ServerApp] Skipped non-installed server(s): bash-language-server, dockerfile-language-server-
nodejs, javascript-typescript-langserver, jedi-language-server, julia-language-server, pyright, python-language-server,
python-lsp-server, r-languageserver, sql-language-server, texlab, typescript-language-server, unified-language-server, v
scode-css-languageserver-bin, vscode-html-languageserver-bin, vscode-json-languageserver-bin, yaml-language-server
```

In few seconds, it opens the **Jupyter** application in a web browser, displaying all files and folders available in the current directory.


http://localhost:8888/tree
jupyter
File View Settings Help
Files Running
Select items to perform actions on them.
New Upload
/

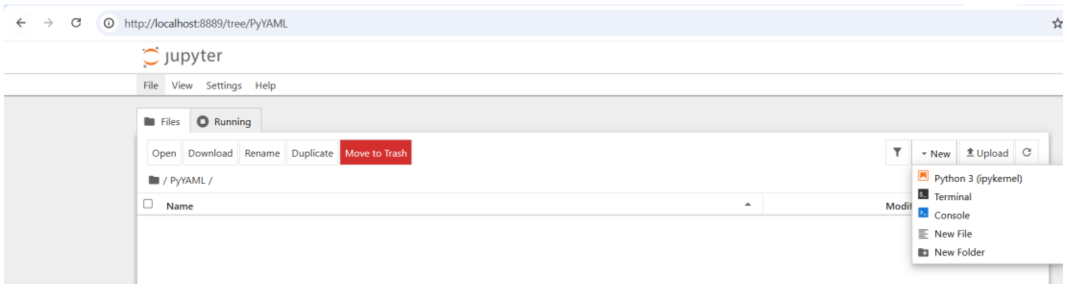
| Name | Last Modified | File Size |
|---|---|---|
| 3D Objects | 2 years ago | |
| beeline | last year | |
| Contacts | 2 years ago | |
| Desktop | yesterday | |
| Documents | 21 days ago | |
| Downloads | 10 days ago | |
| Favorites | 2 years ago | |
| Jedi | last year | |
| Links | 2 years ago | |
| Music | 2 years ago | |

**<u>Note:</u>** To launch Jupyter Notebook from a different directory, navigate to the desired directory (for example, `D:\Learning\Python\Projects`) first and start Jupyter Notebook in the Anaconda virtual environment.

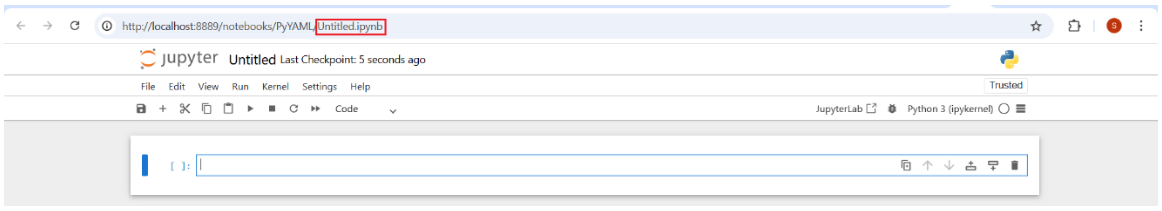


```
(pyyaml_venv) C:\Users\hp>D:

(pyyaml_venv) D:\>cd D:\Learning\Python\Projects

(pyyaml_venv) D:\Learning\Python\Projects>jupyter notebook
[I 2025-08-23 12:25:21.968 ServerApp] jupyter_lsp | extension was successfully linked.
[I 2025-08-23 12:25:21.983 ServerApp] jupyter_server_terminals | extension was successfully linked.
[I 2025-08-23 12:25:21.999 ServerApp] jupyterlab | extension was successfully linked.
[I 2025-08-23 12:25:22.017 ServerApp] notebook | extension was successfully linked.
[I 2025-08-23 12:25:22.954 ServerApp] notebook_shim | extension was successfully linked.
[I 2025-08-23 12:25:23.051 ServerApp] notebook_shim | extension was successfully loaded.
[I 2025-08-23 12:25:23.057 ServerApp] jupyter_lsp | extension was successfully loaded.
[I 2025-08-23 12:25:23.058 ServerApp] jupyter_server_terminals | extension was successfully loaded.
[I 2025-08-23 12:25:23.067 LabApp] JupyterLab extension loaded from D:\ProgramFiles\Anaconda\anaconda3\envs\pyyaml_venv\
Lib\site-packages\jupyterlab
[I 2025-08-23 12:25:23.069 LabApp] JupyterLab application directory is D:\ProgramFiles\Anaconda\anaconda3\envs\pyyaml_ve
nv\share\jupyter\lab
[I 2025-08-23 12:25:23.073 LabApp] Extension Manager is 'pypi'.
[I 2025-08-23 12:25:23.772 ServerApp] jupyterlab | extension was successfully loaded.
[I 2025-08-23 12:25:23.787 ServerApp] notebook | extension was successfully loaded.
[I 2025-08-23 12:25:23.789 ServerApp] The port 8888 is already in use, trying another port.
[I 2025-08-23 12:25:23.791 ServerApp] Serving notebooks from local directory: D:\Learning\Python\Projects
[I 2025-08-23 12:25:23.791 ServerApp] Jupyter Server 2.17.0 is running at:
[I 2025-08-23 12:25:23.792 ServerApp] http://localhost:8889/tree?token=be319df2294201ec90696ca383e18032588b1a567280824c
[I 2025-08-23 12:25:23.793 ServerApp]     http://127.0.0.1:8889/tree?token=be319df2294201ec90696ca383e18032588b1a5672808
24c
[I 2025-08-23 12:25:23.794 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirm
ation).
[C 2025-08-23 12:25:23.882 ServerApp]

    To access the server, open this file in a browser:
        file:///C:/Users/hp/AppData/Roaming/jupyter/runtime/jpserver-5728-open.html
    Or copy and paste one of these URLs:
        http://localhost:8889/tree?token=be319df2294201ec90696ca383e18032588b1a567280824c
        http://127.0.0.1:8889/tree?token=be319df2294201ec90696ca383e18032588b1a567280824c
[I 2025-08-23 12:25:24.200 ServerApp] Skipped non-installed server(s): bash-language-server, dockerfile-language-server-
nodejs, javascript-typescript-langserver, jedi-language-server, julia-language-server, pyright, python-language-server,
python-lsp-server, r-languageserver, sql-language-server, texlab, typescript-language-server, unified-language-server, v
scode-css-languageserver-bin, vscode-html-languageserver-bin, vscode-json-languageserver-bin, yaml-language-server
```

In the **Jupyter** web application, navigate to the desired directory and click on **New** dropdown and select **Python 3 (pykernel)**.



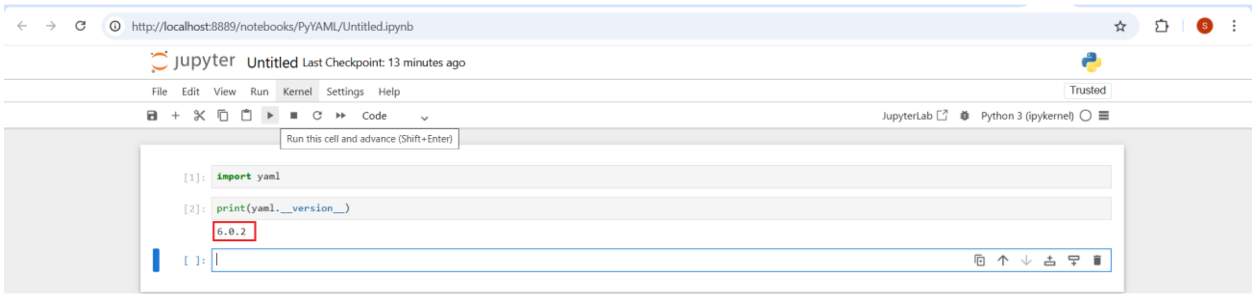
It then creates a file named `Untitled.ipyb` in the directory and opens up that file to execute Python code.


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 5 seconds ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)
[ ]:

## 8.5 Verify PyYAML Version

In any Python interpreter such as **Jupyter Notebook** IDE, run the following Python code to import PyYAML library (`yaml` module) and check its version *(use **Shift + Enter** keys or **Run** button to execute code in Jupyter)*.

```
import yaml
print(yaml.__version__)
```



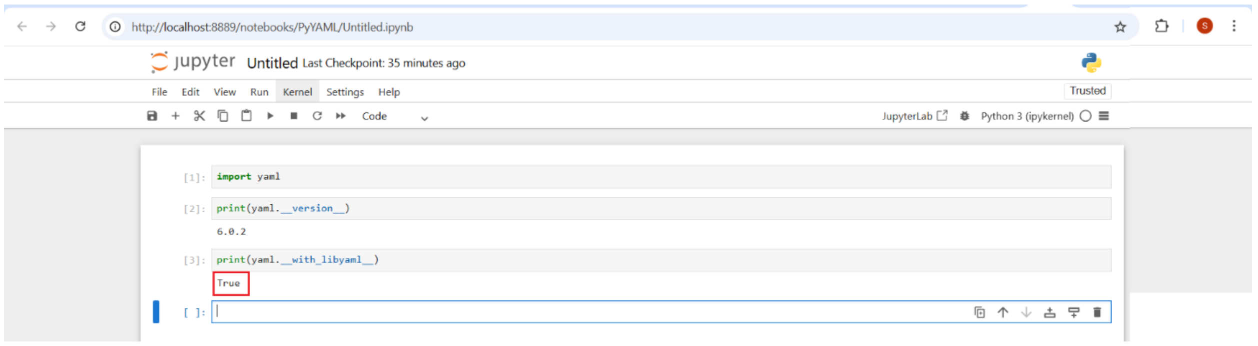
```
[1]: import yaml
[2]: print(yaml.__version__)
6.0.2
```

http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 13 minutes ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)
Run this cell and advance (Shift+Enter)

If Jupyter Notebook is not available, the above code can be executed by launching the standalone Python interpreter using `python` command in **Command Prompt**.

PyYAML often comes with a complied C binding for the [LibYAML](LibYAML) library to make PyYAML run much faster. Run the following code to verify if `LibYAML` library is bundled with the `PyYAML` library *(this code returns `True` if LibYAML has been bundled with the installed PyYAML)*.

```
print(yaml.__with_libyaml__)
```



```
[1]: import yaml
[2]: print(yaml.__version__)
6.0.2
[3]: print(yaml.__with_libyaml__)
True
```

http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 35 minutes ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

## 8.6 Read YAML Document

PyYAML provides various loader classes for parsing YAML content into Python objects, each loader class offering different levels of security and feature support.

There are four primary loader classes in PyYAML:

- `BaseLoader` is the most basic loader which loads the fundamental YAML structures, treating all scalars (like integers, booleans) as strings only and constructs only basic Python objects (`str`, `list`, `dict`), offering the highest level of safety with limited functionality.
- `SafeLoader` loads a safe subset of the YAML document, preventing the execution of arbitrary code embedded within a YAML document. It is useful while loading YAML from untrusted sources since it supports only standard YAML data types (strings, integers, booleans, maps, sequences) and does not construct Python class instances, avoiding potential security risks. For convenience, PyYAML provides `yaml.safe_load()` function that internally utilizes `Loader` class with the `SafeLoader` class to load trusted YAML data.
- `FullLoader` loads the complete YAML document supporting all standard, library, custom tags and complex data types but avoids arbitrary code execution, making it safer than `UnsafeLoader` but less restrictive than `SafeLoader`. This is recommended for trusted YAML data that requires advanced features. For convenience, PyYAML provides `yaml.full_load()` function that internally utilizes `FullLoader` class.
- `UnsafeLoader` (or `Loader` for backwards compatibility) is the less secure loader which loads the complete YAML document and can potentially execute arbitrary code that defines custom Python objects or types. This is not recommended due to security concerns when dealing with external or untrusted YAML files. For convenience, PyYAML provides `yaml.unsafe_load()` function that internally utilizes `UnsafeLoader` class.

These loader classes can be specified to `yaml.load(Loader=FullLoader)` function which by default utilizes `FullLoader` class.

The recommended way of reading a YAML document is by calling `safe_load(stream)` function to safely loads YAML data from a stream (file or string) into Python objects.

## 8.7 Load Document From Stream

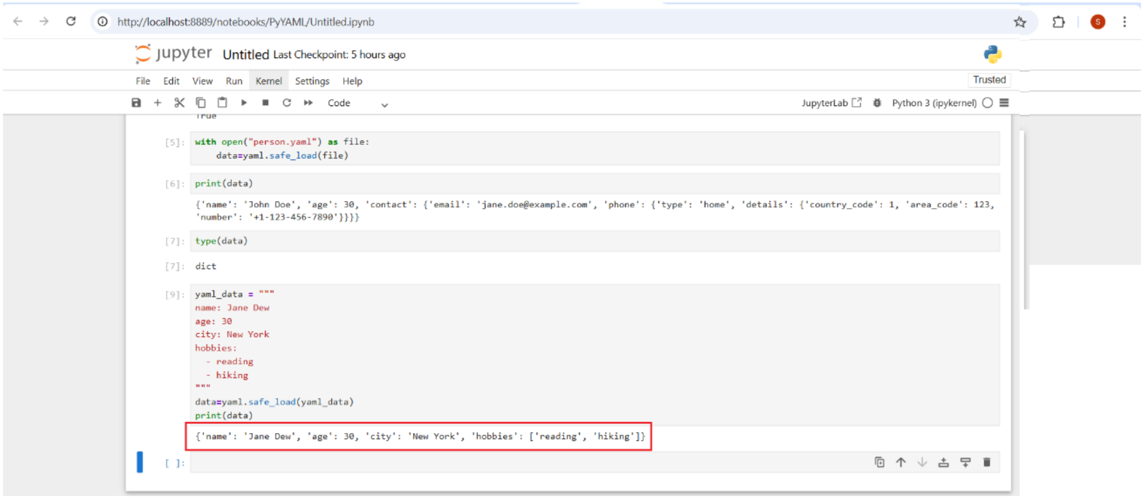
The PyYAML load functions such as `load()`, `safe_load()`, `full_load()` and `unsafe_load()` accept a single argument, which is a universal stream of strings or bytes or file objects.

To read YAML content from a string, run the following code in Jupiter Notebook or Python interpreter:

```
import yaml
yaml_data = """
name: Jane Dew
age: 30
city: New York
hobbies:
  - reading
  - hiking
"""
data=yaml.safe_load(yaml_data)
print(data)
```

When the above code is executed, PyYAML parses the string and deserializes data to a Python dictionary object and displays below output:

```
{'name': 'Jane Dew', 'age': 30, 'city': 'New York', 'hobbies': ['reading',
'hiking']}
```


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 5 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

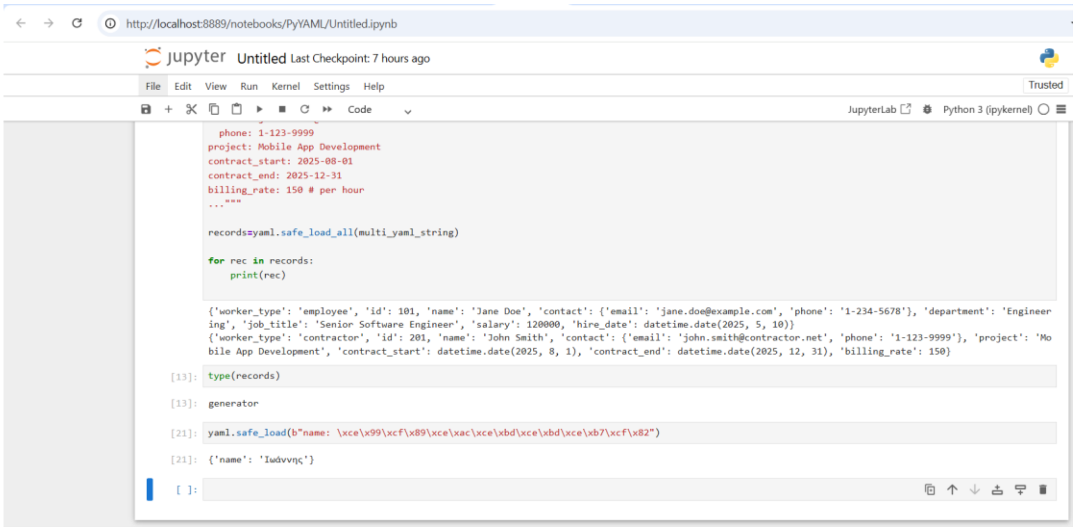
```
True
[5]: with open("person.yaml") as file:
         data=yaml.safe_load(file)
[6]: print(data)
     {'name': 'John Doe', 'age': 30, 'contact': {'email': 'jane.doe@example.com', 'phone': {'type': 'home', 'details': {'country_code': 1, 'area_code': 123,
     'number': '+1-123-456-7890'}}}}
[7]: type(data)
[7]: dict
[9]: yaml_data = """
     name: Jane Dew
     age: 30
     city: New York
     hobbies:
       - reading
       - hiking
     """
     data=yaml.safe_load(yaml_data)
     print(data)
     {'name': 'Jane Dew', 'age': 30, 'city': 'New York', 'hobbies': ['reading', 'hiking']}
[ ]:
```

The below example safely loads YAML data from a byte string:

```
import yaml
yaml.safe_load(b"name: \xce\x99\xcf\x89\xce\xac\xce\xbd\xce\xbd\xce\xb7\xcf\x82")
```

When the above code is executed, PyYAML parses the byte string and detects the UTF-8 encoding and decodes the byte string into a Unicode string, displaying the below output:

```
{'name': 'Ιωάννης'}
```



http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 7 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
  phone: 1-123-9999
project: Mobile App Development
contract_start: 2025-08-01
contract_end: 2025-12-31
billing_rate: 150 # per hour
..."""

records=yaml.safe_load_all(multi_yaml_string)

for rec in records:
    print(rec)
```

```
{'worker_type': 'employee', 'id': 101, 'name': 'Jane Doe', 'contact': {'email': 'jane.doe@example.com', 'phone': '1-234-5678'}, 'department': 'Engineer
ing', 'job_title': 'Senior Software Engineer', 'salary': 120000, 'hire_date': datetime.date(2025, 5, 10)}
{'worker_type': 'contractor', 'id': 201, 'name': 'John Smith', 'contact': {'email': 'john.smith@contractor.net', 'phone': '1-123-9999'}, 'project': 'Mo
bile App Development', 'contract_start': datetime.date(2025, 8, 1), 'contract_end': datetime.date(2025, 12, 31), 'billing_rate': 150}
```

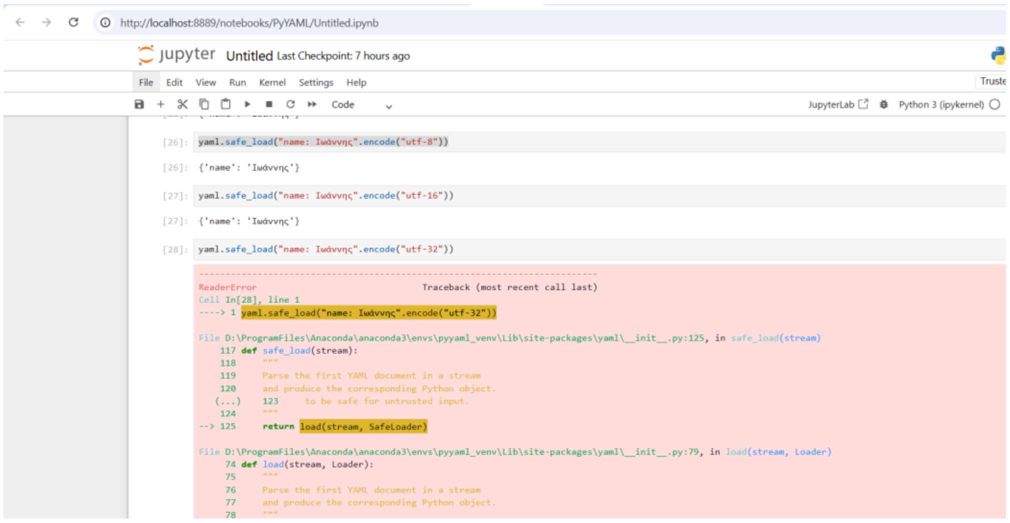
```
[13]: type(records)
[13]: generator
[21]: yaml.safe_load(b"name: \xce\x99\xcf\x89\xce\xac\xce\xbd\xce\xbd\xce\xb7\xcf\x82")
[21]: {'name': 'Ιωάννης'}
[ ]:
```

Note that YAML 1.2 specification supports Unicode encoded with UTF-8, UTF-16, or UTF-32 for compatibility with JSON, but because the PyYAML library builds on YAML 1.1 specification, it supports only UTF-8 and UTF-16 encoding.

For example:

```
import yaml
yaml.safe_load("name: Ιωάννης".encode("utf-8"))
yaml.safe_load("name: Ιωάννης".encode("utf-16"))
yaml.safe_load("name: Ιωάννης".encode("utf-32"))
```

When the above code is executed, it safely loads the string encoded with UTF-8 and UTF-16 but throws error when it tries to load with UTF-32 encoding.

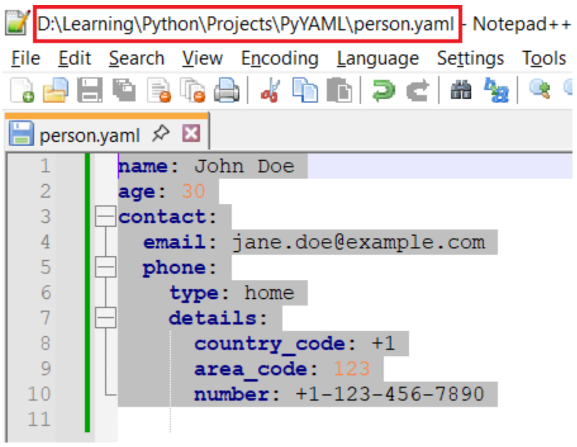


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb

```
[26]: yaml.safe_load("name: Ιωάννης".encode("utf-8"))

[26]: {'name': 'Ιωάννης'}

[27]: yaml.safe_load("name: Ιωάννης".encode("utf-16"))

[27]: {'name': 'Ιωάννης'}

[28]: yaml.safe_load("name: Ιωάννης".encode("utf-32"))

---------------------------------------------------------------------------
ReaderError                               Traceback (most recent call last)
Cell In[28], line 1
----> 1 yaml.safe_load("name: Ιωάννης".encode("utf-32"))

File D:\ProgramFiles\Anaconda\anaconda3\envs\pyyaml_venv\Lib\site-packages\yaml\__init__.py:125, in safe_load(stream)
    117 def safe_load(stream):
    118     """
    119     Parse the first YAML document in a stream
    120     and produce the corresponding Python object.
   (...)
    123     to be safe for untrusted input.
    124     """
--> 125     return load(stream, SafeLoader)

File D:\ProgramFiles\Anaconda\anaconda3\envs\pyyaml_venv\Lib\site-packages\yaml\__init__.py:79, in load(stream, Loader)
     74 def load(stream, Loader):
     75     """
     76     Parse the first YAML document in a stream
     77     and produce the corresponding Python object.
     78     """
```

To read a YAML file through PyYAML, create a file named `person.yaml` in the current directory with the below contents in the file:

```
name: John Doe
age: 30
contact:
  email: jane.doe@example.com
  phone:
    type: home
    details:
      country_code: +1
      area_code: 123
      number: +1-123-456-7890
```



D:\Learning\Python\Projects\PyYAML\person.yaml - Notepad++

person.yaml

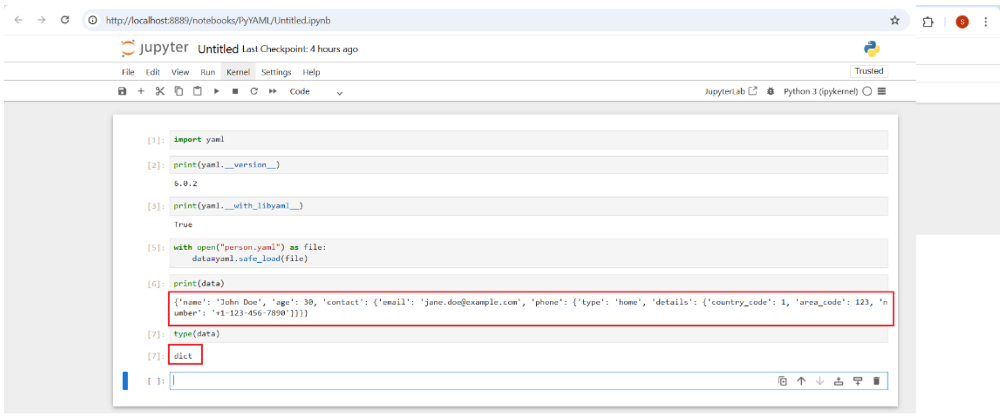
```
name: John Doe
age: 30
contact:
  email: jane.doe@example.com
  phone:
    type: home
    details:
      country_code: +1
      area_code: 123
      number: +1-123-456-7890
```

Run the following code to load `person.yaml` file available in the current directory:

```
import yaml
with open ('person.yaml','r') as file:
    data=yaml.safe_load(file)

print(data)
type(data)
```

When the above code is executed, PyYAML parses the file and deserializes data to a Python dictionary object and displays below output:

```
{'name': 'John Doe', 'age': 30, 'contact': {'email': 'jane.doe@example.com',
'phone': {'type': 'home', 'details': {'country_code': 1, 'area_code': 123,
'number': '+1-123-456-7890'}}}}
```


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 4 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
[1]: import yaml
[2]: print(yaml.__version__)
     6.0.2
[3]: print(yaml.__with_libyaml__)
     True
[5]: with open("person.yaml") as file:
         data=yaml.safe_load(file)
[6]: print(data)
     {'name': 'John Doe', 'age': 30, 'contact': {'email': 'jane.doe@example.com', 'phone': {'type': 'home', 'details': {'country_code': 1, 'area_code': 123, 'n
     umber': '+1-123-456-7890'}}}}
[7]: type(data)
[7]: dict
[ ]:
```

## 8.8 Load Multiple Documents

PyYAML provides `load_all()`, `safe_load_all()`, `full_load_all()`, and `unsafe_load_all()` functions to load multiple YAML documents in a single YAML file or string and converts into a Python generator object. These functions must be used depending on the need but `safe_load_all()` is the most recommended way which avoids any arbitrary code execution while loading multiple YAML documents from untrusted sources.

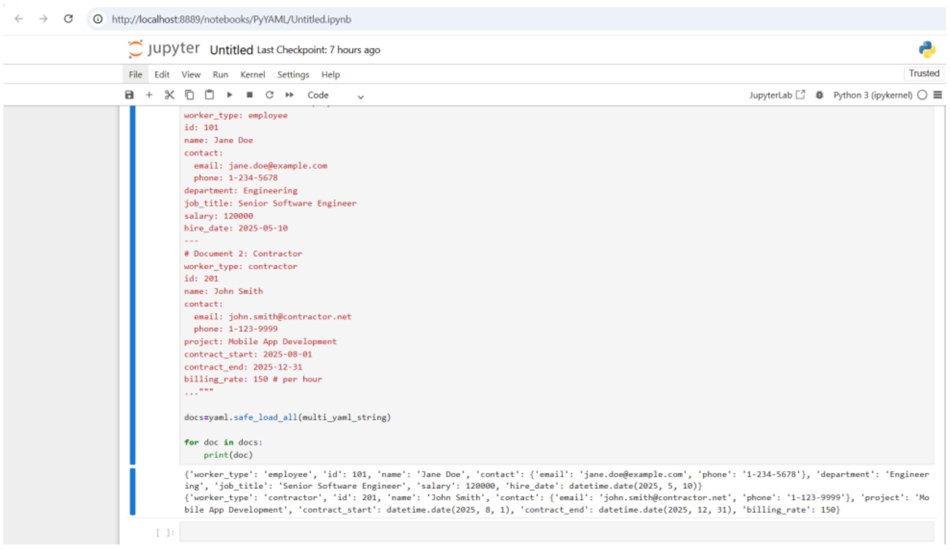
Run the following code to load multiple YAML documents stored in a YAML formatted string:

```
import yaml
multi_yaml_string = """
```

```
---
# Document 1: Full-Time Employee
worker_type: employee
id: 101
name: Jane Doe
contact:
  email: jane.doe@example.com
  phone: 1-234-5678
department: Engineering
job_title: Senior Software Engineer
salary: 120000
hire_date: 2025-05-10
---
# Document 2: Contractor
worker_type: contractor
id: 201
name: John Smith
contact:
  email: john.smith@contractor.net
  phone: 1-123-9999
project: Mobile App Development
contract_start: 2025-08-01
contract_end: 2025-12-31
billing_rate: 150 # per hour
..."""

docs=yaml.safe_load_all(multi_yaml_string)

for doc in docs:
    print(doc)
```

When the above code is executed, PyYAML parses the string and deserializes data to a Python generator object and displays below output:

```
{'worker_type': 'employee', 'id': 101, 'name': 'Jane Doe', 'contact': {'email':
'jane.doe@example.com', 'phone': '1-234-5678'}, 'department': 'Engineering',
'job_title': 'Senior Software Engineer', 'salary': 120000, 'hire_date':
datetime.date(2025, 5, 10)}
{'worker_type': 'contractor', 'id': 201, 'name': 'John Smith', 'contact': {'email':
'john.smith@contractor.net', 'phone': '1-123-9999'}, 'project': 'Mobile App
Development', 'contract_start': datetime.date(2025, 8, 1), 'contract_end':
datetime.date(2025, 12, 31), 'billing_rate': 150}
```


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 7 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
worker_type: employee
id: 101
name: Jane Doe
contact:
  email: jane.doe@example.com
  phone: 1-234-5678
department: Engineering
job_title: Senior Software Engineer
salary: 120000
hire_date: 2025-05-10
---
# Document 2: Contractor
worker_type: contractor
id: 201
name: John Smith
contact:
  email: john.smith@contractor.net
  phone: 1-123-9999
project: Mobile App Development
contract_start: 2025-08-01
contract_end: 2025-12-31
billing_rate: 150 # per hour
..."""

docs=yaml.safe_load_all(multi_yaml_string)

for doc in docs:
    print(doc)
```

```
{'worker_type': 'employee', 'id': 101, 'name': 'Jane Doe', 'contact': {'email': 'jane.doe@example.com', 'phone': '1-234-5678'}, 'department': 'Engineer
ing', 'job_title': 'Senior Software Engineer', 'salary': 120000, 'hire_date': datetime.date(2025, 5, 10)}
{'worker_type': 'contractor', 'id': 201, 'name': 'John Smith', 'contact': {'email': 'john.smith@contractor.net', 'phone': '1-123-9999'}, 'project': 'Mo
bile App Development', 'contract_start': datetime.date(2025, 8, 1), 'contract_end': datetime.date(2025, 12, 31), 'billing_rate': 150}
```

```
[ ]:
```

## 8.9 Write YAML Document

PyYAML provides various dumper classes for parsing Python objects into YAML, each dumper class offering different levels of security and feature support.

There are two primary dumper classes in PyYAML:

- `Dumper` is the base dumper class which serializes a Python object into a YAML stream, including all standard, custom tags and Python-specific tags for custom classes and arbitrary objects. It is helpful when the complete object information needs to be preserved so it can be loaded back into identical Python objects but it is not recommended to use since it can expose to security vulnerabilities causing potential risks. For convivence, PyYAML provides `yaml.dump()` function that internally utilizes `Dumper` class by default.
- `SafeDumper` is a safer alternative to the default `Dumper` since it restricts the types of Python objects that can be serialized, preventing the dumping of arbitrary Python objects that could potentially lead to security vulnerabilities during deserialization. This is generally recommended unless there is a need to dump custom Python class instances. For convenience, PyYAML provides `yaml.safe_dump()` function that internally utilizes `SafeDumper` class.

These dumper classes can be specified to `yaml.dump(Dumper=Dumper)` function which by default utilizes `Dumper` class.

## 8.10 Dump Document Into Stream

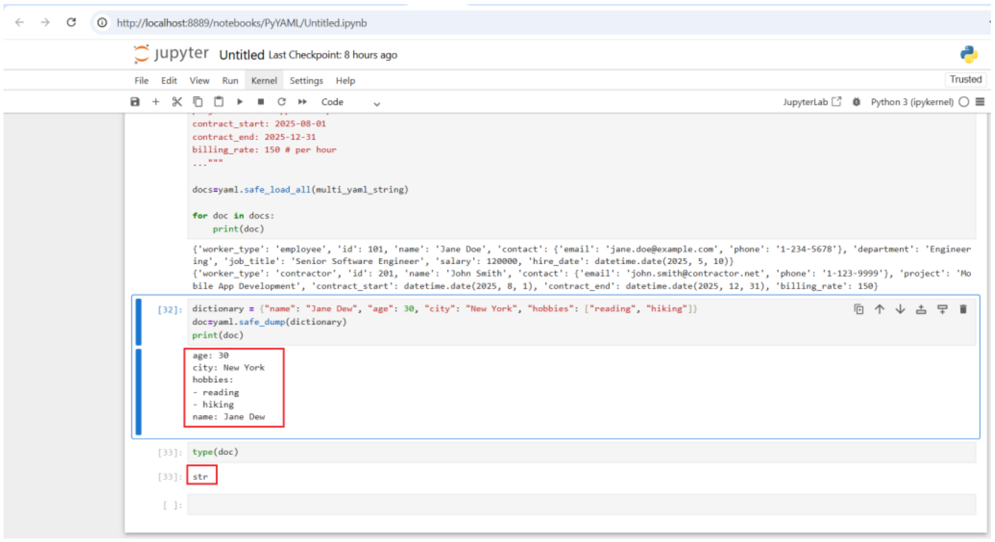
The PyYAML dump functions such as `dump()` and `safe_dump()` accept a primary argument with a dictionary type data.

To dump into a YAML formatted document, run the following code:

```
import yaml
dictionary = {"name": "Jane Dew", "age": 30, "city": "New York", "hobbies":
["reading", "hiking"]}
doc=yaml.safe_dump(dictionary)
print(doc)
type(doc)
```

When the above code is executed, PyYAML parses the dictionary object and serializes data to a YAML formatted string and displays below output:

```
age: 30
city: New York
hobbies:
- reading
- hiking
name: Jane Dew
```


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 8 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
contract_start: 2025-08-01
contract_end: 2025-12-31
billing_rate: 150 # per hour
..."""

docs=yaml.safe_load_all(multi_yaml_string)

for doc in docs:
    print(doc)
```

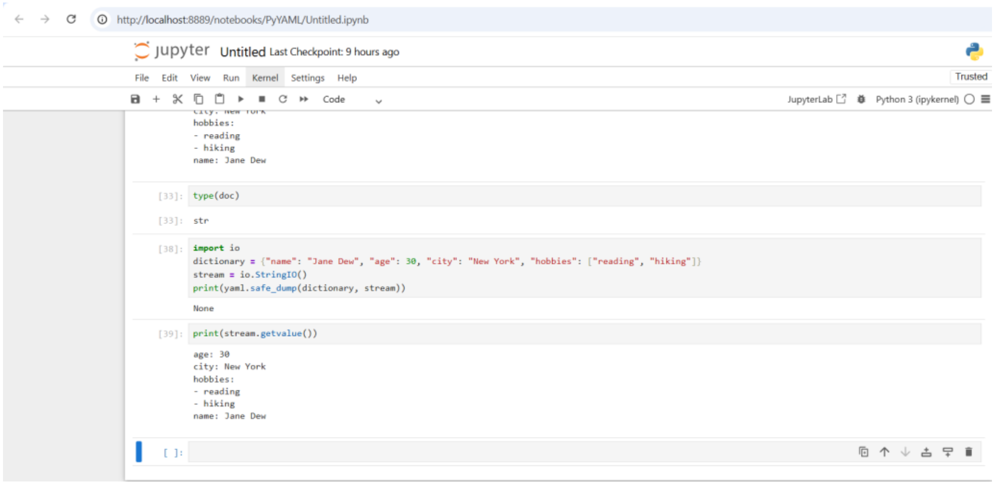
```
{'worker_type': 'employee', 'id': 101, 'name': 'Jane Doe', 'contact': {'email': 'jane.doe@example.com', 'phone': '1-234-5678'}, 'department': 'Engineer
ing', 'job_title': 'Senior Software Engineer', 'salary': 120000, 'hire_date': datetime.date(2025, 5, 10)}
{'worker_type': 'contractor', 'id': 201, 'name': 'John Smith', 'contact': {'email': 'john.smith@contractor.net', 'phone': '1-123-9999'}, 'project': 'Mo
bile App Development', 'contract_start': datetime.date(2025, 8, 1), 'contract_end': datetime.date(2025, 12, 31), 'billing_rate': 150}
```

```
[32]: dictionary = {"name": "Jane Dew", "age": 30, "city": "New York", "hobbies": ["reading", "hiking"]}
      doc=yaml.safe_dump(dictionary)
      print(doc)

      age: 30
      city: New York
      hobbies:
      - reading
      - hiking
      name: Jane Dew

[33]: type(doc)

[33]: str

[ ]:
```

The PyYAML dump functions `dump()` and `safe_dump()` accept a secondary argument to specify the target stream which could be an IO or a file.

When an IO stream is specified to the dump functions, they return `None` and data from the stream needs to be extracted as needed.

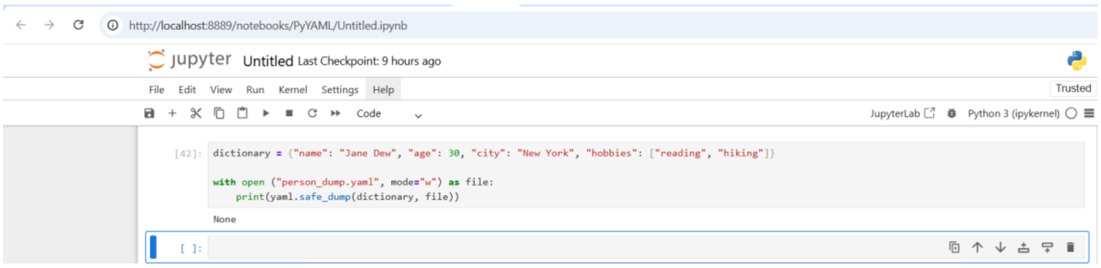
```
import yaml
import io
dictionary = {"name": "Jane Dew", "age": 30, "city": "New York", "hobbies":
["reading", "hiking"]}
stream = io.StringIO()
print(yaml.safe_dump(dictionary, stream))

print(stream.getvalue())
```



```
hobbies:
- reading
- hiking
name: Jane Dew

[33]: type(doc)

[33]: str

[38]: import io
      dictionary = {"name": "Jane Dew", "age": 30, "city": "New York", "hobbies": ["reading", "hiking"]}
      stream = io.StringIO()
      print(yaml.safe_dump(dictionary, stream))

      None

[39]: print(stream.getvalue())

      age: 30
      city: New York
      hobbies:
      - reading
      - hiking
      name: Jane Dew

[ ]:
```

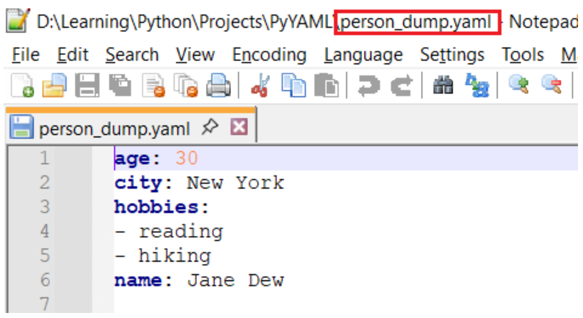
To dump the YAML content into a file, open the file in write mode and dump data as below:

```
import yaml
dictionary = {"name": "John Doe", "age": 30, "city": "New York", "hobbies":
["reading", "hiking"]}

with open ("person_dump.yaml", mode="w") as file:
    print(yaml.safe_dump(dictionary, file))
```



```
[42]: dictionary = {"name": "Jane Dew", "age": 30, "city": "New York", "hobbies": ["reading", "hiking"]}

      with open ("person_dump.yaml", mode="w") as file:
          print(yaml.safe_dump(dictionary, file))

      None

[ ]:
```

Once the above code is executed, verify if `person_dump.yaml` got created with YAML structure.


D:\Learning\Python\Projects\PyYAML\person_dump.yaml - Notepad
File Edit Search View Encoding Language Settings Tools M
person_dump.yaml

```
age: 30
city: New York
hobbies:
- reading
- hiking
name: Jane Dew
```

## 8.11 Dump Multiple Documents

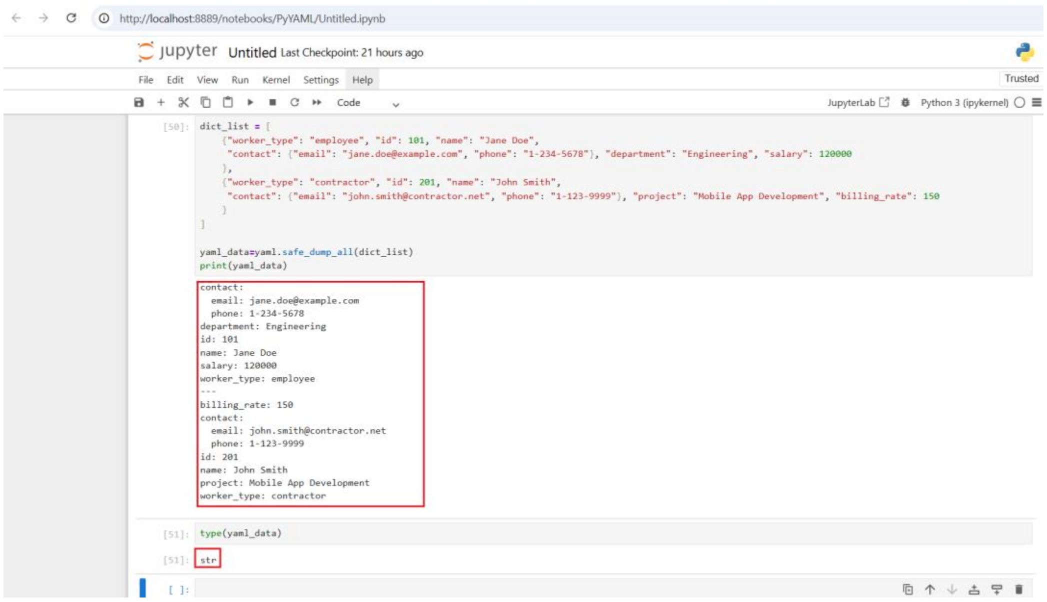
PyYAML provides `dump_all()` and `safe_dump_all()` functions to dump YAML content as multiple documents.

Run the following code to dump into multiple YAML documents:

```
import yaml
dict_list = [
    {"worker_type": "employee", "id": 101, "name": "Jane Doe",
     "contact": {"email": "jane.doe@example.com", "phone": "1-234-5678"},
"department": "Engineering", "salary": 120000
    },
    {"worker_type": "contractor", "id": 201, "name": "John Smith",
     "contact": {"email": "john.smith@contractor.net", "phone": "1-123-9999"},
"project": "Mobile App Development", "billing_rate": 150
    }
]

yaml_data=yaml.safe_dump_all(dict_list)
print(yaml_data)

type(yaml_data)
```

When the above code is executed, PyYAML parses and serializes the list of Python dictionaries to a multi-document YAML stream and displays below output:

```
contact:
  email: jane.doe@example.com
  phone: 1-234-5678
department: Engineering
id: 101
name: Jane Doe
salary: 120000
worker_type: employee
---
billing_rate: 150
contact:
  email: john.smith@contractor.net
  phone: 1-123-9999
id: 201
name: John Smith
project: Mobile App Development
worker_type: contractor
```


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 21 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
[50]: dict_list = [
          {"worker_type": "employee", "id": 101, "name": "Jane Doe",
           "contact": {"email": "jane.doe@example.com", "phone": "1-234-5678"}, "department": "Engineering", "salary": 120000
          },
          {"worker_type": "contractor", "id": 201, "name": "John Smith",
           "contact": {"email": "john.smith@contractor.net", "phone": "1-123-9999"}, "project": "Mobile App Development", "billing_rate": 150
          }
      ]

      yaml_data=yaml.safe_dump_all(dict_list)
      print(yaml_data)

      contact:
        email: jane.doe@example.com
        phone: 1-234-5678
      department: Engineering
      id: 101
      name: Jane Doe
      salary: 120000
      worker_type: employee
      ---
      billing_rate: 150
      contact:
        email: john.smith@contractor.net
        phone: 1-123-9999
      id: 201
      name: John Smith
      project: Mobile App Development
      worker_type: contractor

[51]: type(yaml_data)

[51]: str

[ ]:
```

## 8.12 Dump Formatted Documents

All PyYAML dump functions which are `dump()`, `safe_dump()`, `dump_all()` and `safe_dump_all()` accept optional arguments to control the output formatting. These arguments include:

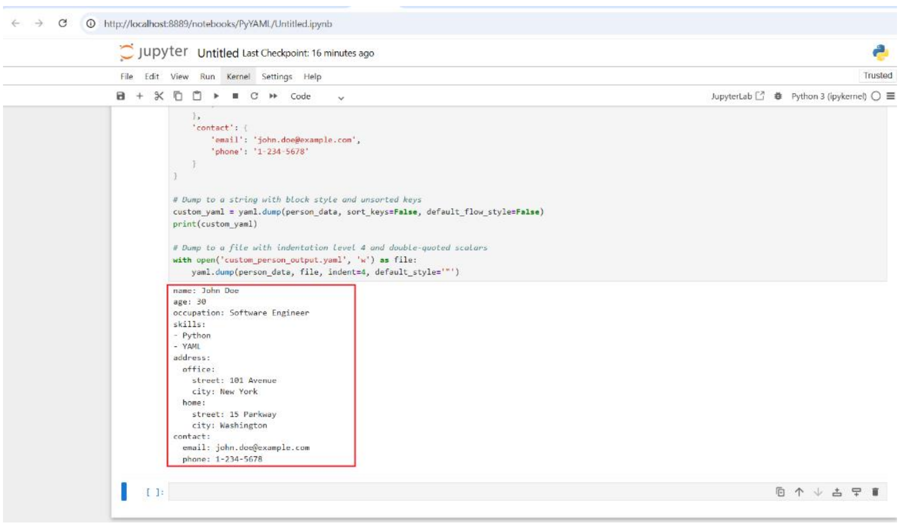
- `default_flow_style` which defines if nested collections (maps and sequences) to be displayed in Flow style or Block style. By default, this is set to `False`.

- `sort_keys` which determines if keys in mapping should be sorted alphabetically. By default, this is set to `True`.
- `indent` which specifies the number of spaces used for indentation in block style. This is an integer value which must be between 1 and 10. It's default value is `2`.
- `width` which specifies the maximum line width for the output. This value must be bigger than twice the indent.
- `explicit_start` which defines if Document Start marker (`---`) to be added to the output. By default, this is set to `False`.
- `explicit_end` which defines if Document End marker (…) to be added to the output. By default, this is set to `False`.
- `default_style` which defines the quotation style and accepts either `None` or `"'"` or `'"'` values.
- `allow_unicode` which controls the output when special characters are used. By default, this is set to `False` allowing to escape non-ASCII characters and when set to `True`, it outputs Unicode characters directly without escaping and double-quoting.
- `encoding` which specifies the encoding string (such as `'utf-8'`) to use when writing to a binary stream.
- `canonical` which defines if YAML output to be displayed in the canonical, strictly ordered form. By default, this is set to `False`.
- `line_break` which specifies a newline character and accepts either `'\r'` or `'\n'` or `'\r\n'` values.
- `tags` which accepts a dictionary value defining tag directives that maps custom tag handles to valid URI prefixes recognized by a YAML parser.
- `version` which accepts a tuple value with major and minor YAML version *(such as `(1, 2)` to specify version 1.2)*.

Run the following code to dump data with formatted output:

```
import yaml
person_data = {
    'name': 'John Doe',
    'age': 30,
    'occupation': 'Software Engineer',
    'skills': ['Python', 'YAML'],
    'address': {
        'office': {
            'street': '101 Avenue',
            'city': 'New York'
        },
        'home': {
            'street': '15 Parkway',
            'city': 'Washington'
        }
    },
    'contact': {
        'email': 'john.doe@example.com',
        'phone': '1-234-5678'
    }
}

# Dump to a string with block style and unsorted keys
custom_yaml = yaml.dump(person_data, sort_keys=False, default_flow_style=False)
print(custom_yaml)

# Dump to a file with indentation level 4 and double-quoted scalars
with open('custom_person_output.yaml', 'w') as file:
    yaml.dump(person_data, file, indent=4, default_style='"')
```

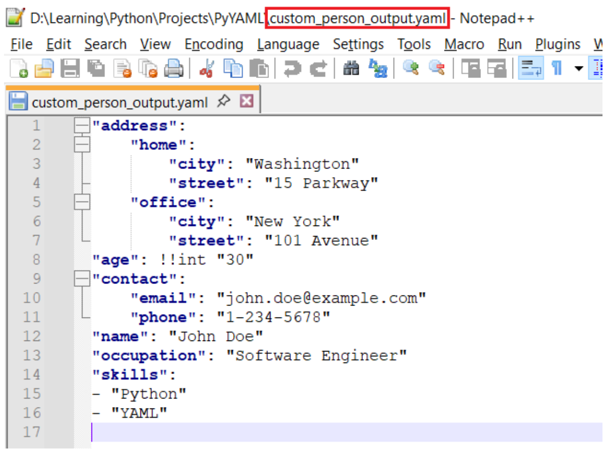
When the above code is executed, it displays a string output in block style with unsorted keys as below:



```
    },
    'contact': {
        'email': 'john.doe@example.com',
        'phone': '1-234-5678'
    }
}

# Dump to a string with block style and unsorted keys
custom_yaml = yaml.dump(person_data, sort_keys=False, default_flow_style=False)
print(custom_yaml)

# Dump to a file with indentation level 4 and double-quoted scalars
with open('custom_person_output.yaml', 'w') as file:
    yaml.dump(person_data, file, indent=4, default_style='"')
```

```
name: John Doe
age: 30
occupation: Software Engineer
skills:
- Python
- YAML
address:
  office:
    street: 101 Avenue
    city: New York
  home:
    street: 15 Parkway
    city: Washington
contact:
  email: john.doe@example.com
  phone: 1-234-5678
```

Also, creates a file named `custom_person_output.yaml` with 4 spaces indentation and double quoted scalars.


D:\Learning\Python\Projects\PyYAML\custom_person_output.yaml - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins W
custom_person_output.yaml

```
"address":
    "home":
        "city": "Washington"
        "street": "15 Parkway"
    "office":
        "city": "New York"
        "street": "101 Avenue"
"age": !!int "30"
"contact":
    "email": "john.doe@example.com"
    "phone": "1-234-5678"
"name": "John Doe"
"occupation": "Software Engineer"
"skills":
- "Python"
- "YAML"
```

## 8.13 Create Custom Tags

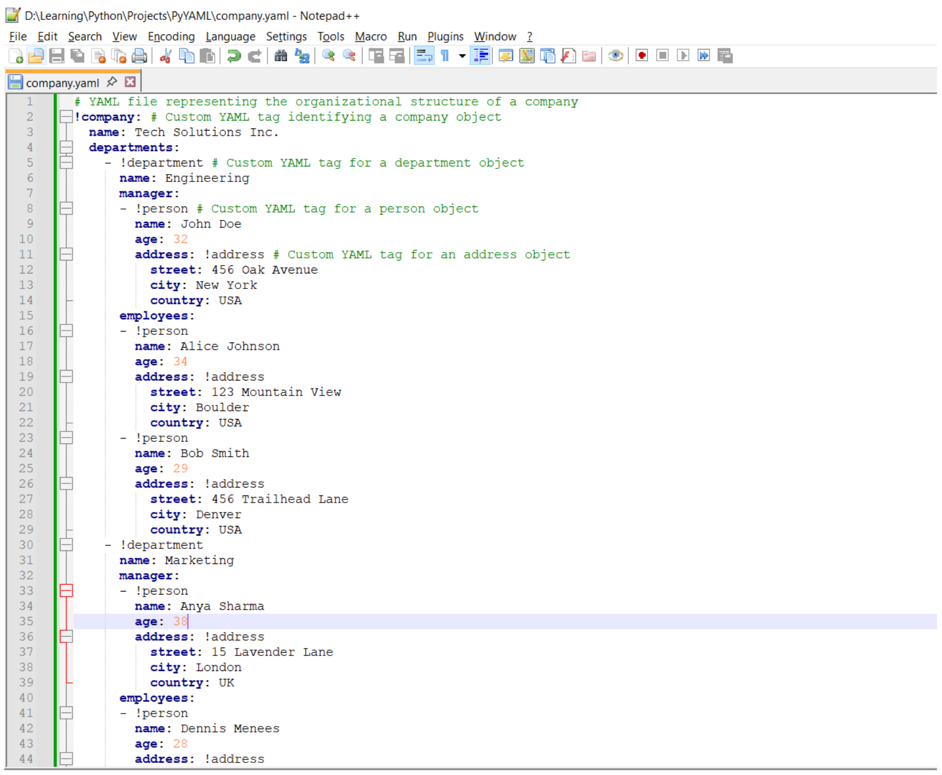
Creating and using custom tags in PyYAML involves various steps including:

- Defining custom data structure.
- Creating a Python class to represent the custom data structure.
- Implementing a **representer** for serializing the custom data structure to represent a specific Python object in YAML and a **constructor** for deserializing custom data structure to construct a YAML custom tag into a Python object.
- Registering the custom tag with PyYAML.

Consider a YAML example to define company information in a hierarchical structure, such as a company with departments, where each department has a manager and employees with their name, age and address. For this example, different custom tags such as `!company`, `!department`, `!person` and `!address` can be used and nested within one another.

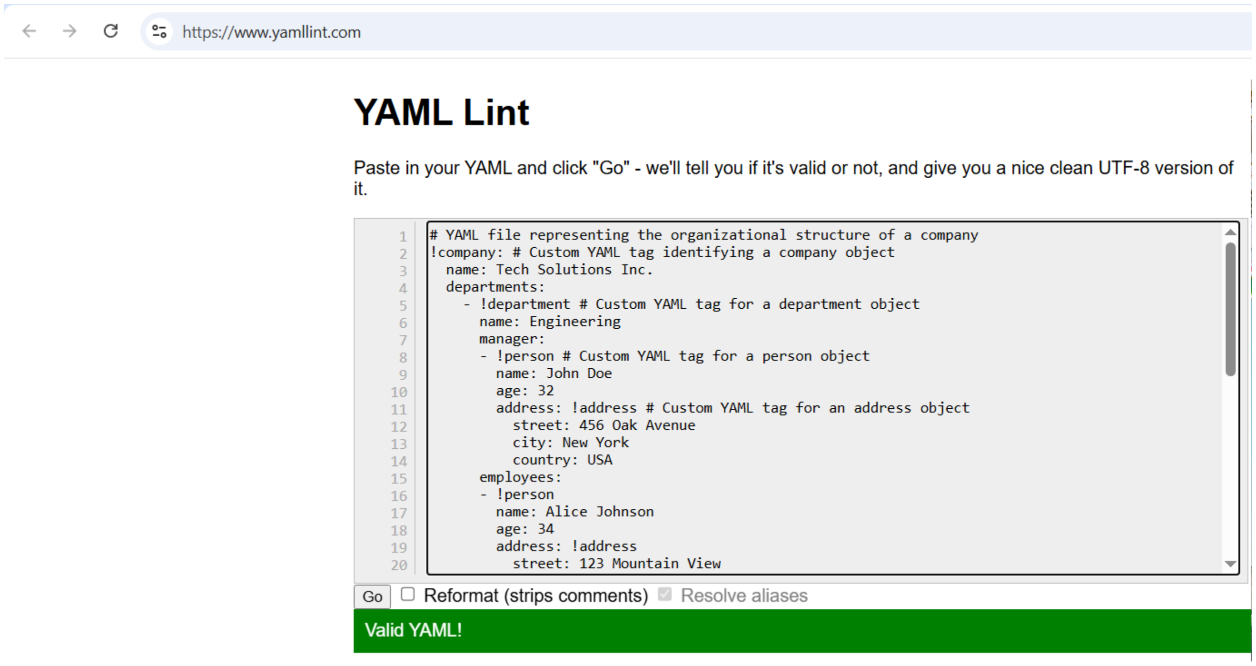
Create a YAML file named `company.yaml` with the following contents in the current directory:

```
# YAML file representing the organizational structure of a company
!company # Custom YAML tag identifying a company object
name: Tech Solutions Inc.
departments:
- !department # Custom YAML tag for a department object
  name: Engineering
  manager:
    !person # Custom YAML tag for a person object
    name: John Doe
    age: 32
    address: !address # Custom YAML tag for an address object
      street: 456 Oak Avenue
      city: New York
      country: USA
  employees:
  - !person
    name: Alice Johnson
    age: 34
    address: !address
      street: 123 Mountain View
      city: Boulder
      country: USA
  - !person
    name: Bob Smith
    age: 29
    address: !address
      street: 456 Trailhead Lane
      city: Denver
      country: USA
- !department
  name: Marketing
  manager:
    !person
    name: Anya Sharma
    age: 38
    address: !address
      street: 15 Lavender Lane
      city: London
      country: UK
  employees:
  - !person
    name: Dennis Menees
    age: 28
    address: !address
      street: Kings Square
      city: London
      country: UK
```


D:\Learning\Python\Projects\PyYAML\company.yaml - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
company.yaml

```yaml
# YAML file representing the organizational structure of a company
!company: # Custom YAML tag identifying a company object
  name: Tech Solutions Inc.
  departments:
    - !department # Custom YAML tag for a department object
      name: Engineering
      manager:
      - !person # Custom YAML tag for a person object
        name: John Doe
        age: 32
        address: !address # Custom YAML tag for an address object
          street: 456 Oak Avenue
          city: New York
          country: USA
      employees:
      - !person
        name: Alice Johnson
        age: 34
        address: !address
          street: 123 Mountain View
          city: Boulder
          country: USA
      - !person
        name: Bob Smith
        age: 29
        address: !address
          street: 456 Trailhead Lane
          city: Denver
          country: USA
    - !department
      name: Marketing
      manager:
      - !person
        name: Anya Sharma
        age: 38
        address: !address
          street: 15 Lavender Lane
          city: London
          country: UK
      employees:
      - !person
        name: Dennis Menees
        age: 28
        address: !address
```

**<u>Note:</u>** If needed, the above YAML file can be validated in [YAMLLint online tool](https://www.yamllint.com) to verify for any syntax issues.



YAML Lint

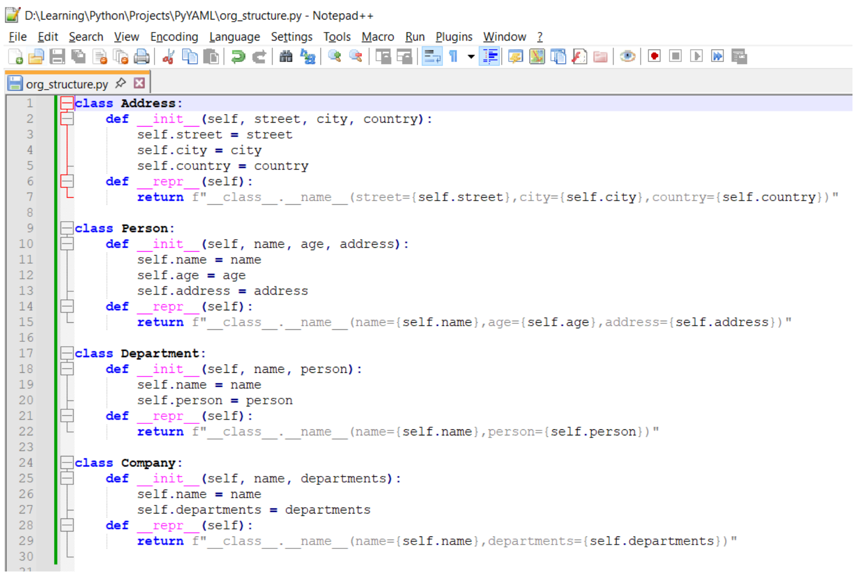
Paste in your YAML and click "Go" - we'll tell you if it's valid or not, and give you a nice clean UTF-8 version of it.

```yaml
# YAML file representing the organizational structure of a company
!company: # Custom YAML tag identifying a company object
  name: Tech Solutions Inc.
  departments:
    - !department # Custom YAML tag for a department object
      name: Engineering
      manager:
      - !person # Custom YAML tag for a person object
        name: John Doe
        age: 32
        address: !address # Custom YAML tag for an address object
          street: 456 Oak Avenue
          city: New York
          country: USA
      employees:
      - !person
        name: Alice Johnson
        age: 34
        address: !address
          street: 123 Mountain View
```

Valid YAML!

Next, create a Python file named `org_structure.py` and add the below code defining various classes, each representing custom data structure for company, department, person and address (corresponding to each custom tag defined in the above YAML file).

```
class Address:
    def __init__(self, street, city, country):
        self.street = street
        self.city = city
        self.country = country
    def __repr__(self):
        return
f"{__class__.__name__}(street={self.street},city={self.city},country={self.c
ountry})"

class Person:
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address
    def __repr__(self):
        return
f"{__class__.__name__}(name={self.name},age={self.age},address={self.address
})"

class Department:
    def __init__(self, name, manager, employees):
        self.name = name
        self.manager = manager
        self.employees = employees
    def __repr__(self):
        return
f"{__class__.__name__}(name={self.name},manager={self.manager},employees={se
lf.employees})"

class Company:
    def __init__(self, name, departments):
        self.name = name
        self.departments = departments
    def __repr__(self):
        return
f"{__class__.__name__}(name={self.name},departments={self.departments})"
```


D:\Learning\Python\Projects\PyYAML\org_structure.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
org_structure.py

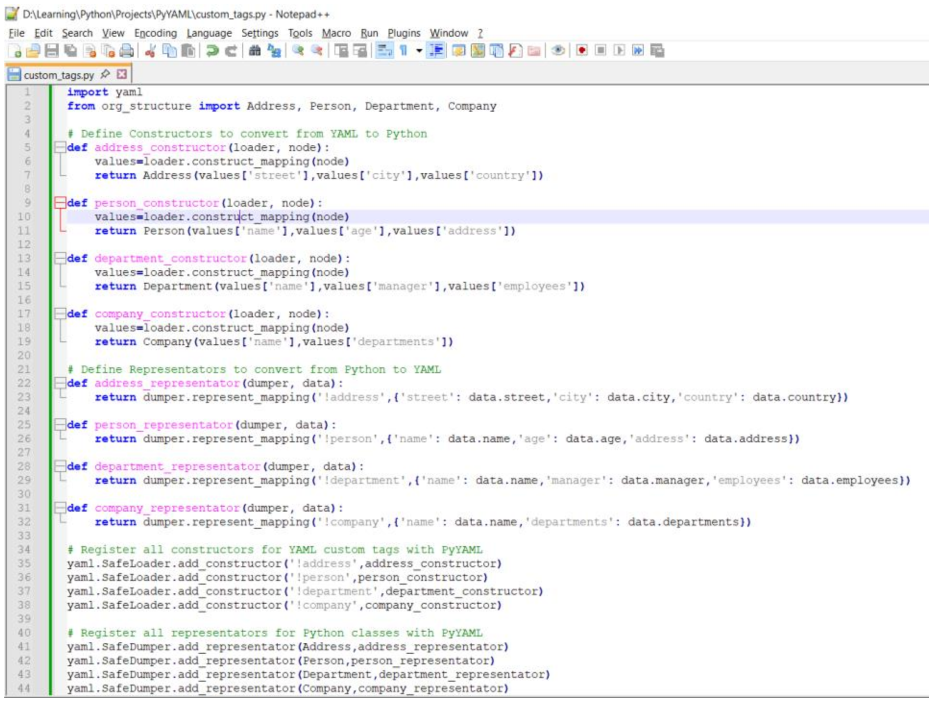
```python
class Address:
    def __init__(self, street, city, country):
        self.street = street
        self.city = city
        self.country = country
    def __repr__(self):
        return f"__class__.__name__(street={self.street},city={self.city},country={self.country})"

class Person:
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address
    def __repr__(self):
        return f"__class__.__name__(name={self.name},age={self.age},address={self.address})"

class Department:
    def __init__(self, name, person):
        self.name = name
        self.person = person
    def __repr__(self):
        return f"__class__.__name__(name={self.name},person={self.person})"

class Company:
    def __init__(self, name, departments):
        self.name = name
        self.departments = departments
    def __repr__(self):
        return f"__class__.__name__(name={self.name},departments={self.departments})"
```

Next, implement serialization and deserialization logic for the above custom data types and register the respective custom YAML tags in PyYAML. For this, create a new file named `custom_tags.py` in the current directory and add the following code:

```python
import yaml
from org_structure import Address, Person, Department, Company

# Define Constructors to convert from YAML to Python
def address_constructor(loader, node):
    values=loader.construct_mapping(node)
    return Address(values['street'],values['city'],values['country'])

def person_constructor(loader, node):
    values=loader.construct_mapping(node)
    return Person(values['name'],values['age'],values['address'])

def department_constructor(loader, node):
    values=loader.construct_mapping(node)
    return Department(values['name'],values['manager'],values['employees'])

def company_constructor(loader, node):
    values=loader.construct_mapping(node)
    return Company(values['name'],values['departments'])

# Define Representers to convert from Python to YAML
def address_representer(dumper, data):
    return dumper.represent_mapping('!address',{'street':
data.street,'city': data.city,'country': data.country})

def person_representer(dumper, data):
    return dumper.represent_mapping('!person',{'name': data.name,'age':
data.age,'address': data.address})
```

```
def department_representer(dumper, data):
    return dumper.represent_mapping('!department',{'name':
data.name,'manager': data.manager,'employees': data.employees})

def company_representer(dumper, data):
    return dumper.represent_mapping('!company',{'name':
data.name,'departments': data.departments})

# Register all constructors for YAML custom tags with PyYAML
yaml.SafeLoader.add_constructor('!address',address_constructor)
yaml.SafeLoader.add_constructor('!person',person_constructor)
yaml.SafeLoader.add_constructor('!department',department_constructor)
yaml.SafeLoader.add_constructor('!company',company_constructor)

# Register all representers for Python classes with PyYAML
yaml.SafeDumper.add_representer(Address,address_representer)
yaml.SafeDumper.add_representer(Person,person_representer)
yaml.SafeDumper.add_representer(Department,department_representer)
yaml.SafeDumper.add_representer(Company,company_representer)
```

Note that in the above code, `yaml.SafeLoader` and `yaml.SafeDumper` classes are specified to safely load and dump data with custom data structures using `safe_load()` and `safe_dump()` functions. If these classes are not explicitly specified, YAML by default uses `FullLoader` and `Dumper` classes in which case PyYAML will not be able to serialize/deserialize custom data structures using `safe_load()` and `safe_dump()` functions and can serialize/deserialize using other load and dump functions.


D:\Learning\Python\Projects\PyYAML\custom_tags.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
custom_tags.py

```
import yaml
from org_structure import Address, Person, Department, Company

# Define Constructors to convert from YAML to Python
def address_constructor(loader, node):
    values=loader.construct_mapping(node)
    return Address(values['street'],values['city'],values['country'])

def person_constructor(loader, node):
    values=loader.construct_mapping(node)
    return Person(values['name'],values['age'],values['address'])

def department_constructor(loader, node):
    values=loader.construct_mapping(node)
    return Department(values['name'],values['manager'],values['employees'])

def company_constructor(loader, node):
    values=loader.construct_mapping(node)
    return Company(values['name'],values['departments'])

# Define Representators to convert from Python to YAML
def address_representator(dumper, data):
    return dumper.represent_mapping('!address',{'street': data.street,'city': data.city,'country': data.country})

def person_representator(dumper, data):
    return dumper.represent_mapping('!person',{'name': data.name,'age': data.age,'address': data.address})

def department_representator(dumper, data):
    return dumper.represent_mapping('!department',{'name': data.name,'manager': data.manager,'employees': data.employees})

def company_representator(dumper, data):
    return dumper.represent_mapping('!company',{'name': data.name,'departments': data.departments})

# Register all constructors for YAML custom tags with PyYAML
yaml.SafeLoader.add_constructor('!address',address_constructor)
yaml.SafeLoader.add_constructor('!person',person_constructor)
yaml.SafeLoader.add_constructor('!department',department_constructor)
yaml.SafeLoader.add_constructor('!company',company_constructor)

# Register all representators for Python classes with PyYAML
yaml.SafeDumper.add_representator(Address,address_representator)
yaml.SafeDumper.add_representator(Person,person_representator)
yaml.SafeDumper.add_representator(Department,department_representator)
yaml.SafeDumper.add_representator(Company,company_representator)
```

Now, YAML file with company data can be safely loaded with custom data types to Python objects which can be safely dumped back to YAML.

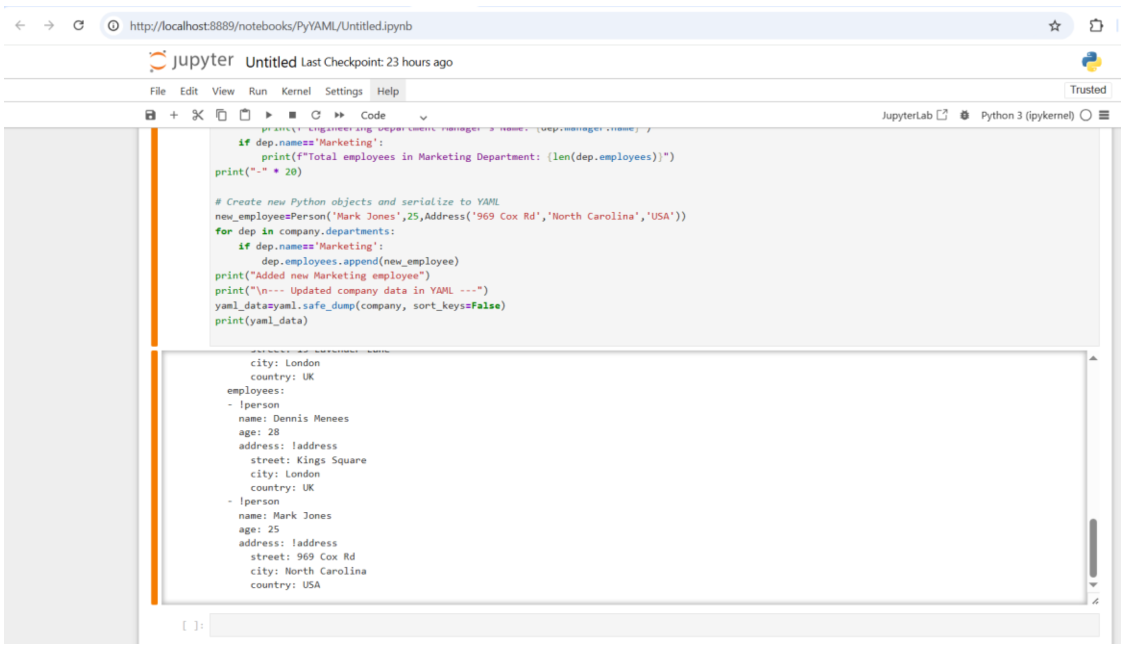
Run the following code in any Python interpreter such as Jupyter Notebook:

```python
from custom_tags import Address, Person, Department, Company

# Load YAML file and deserialize into Python objects
with open('company.yaml') as file:
    company=yaml.safe_load(file)

print("Loaded Python object from company.yaml:")
print(company)
print("-" * 20)
print(f"Company Name: {company.name}")

for dep in company.departments:
    if dep.name=='Engineering':
        print(f"Engineering Department Manager's Name: {dep.manager.name}")
    if dep.name=='Marketing':
        print(f"Total employees in Marketing Department: {len(dep.employees)}")
print("-" * 20)

# Create new Python objects and serialize to YAML
new_employee=Person('Mark Jones', 25, Address('969 Cox Rd', 'North Carolina', 'USA'))
for dep in company.departments:
    if dep.name=='Marketing':
        dep.employees.append(new_employee)
print("Added new Marketing employee")
print("\n--- Updated company data in YAML ---")
yaml_data=yaml.safe_dump(company, sort_keys=False)
print(yaml_data)
```

When the above code is executed, PyYAML parses `company.yaml` file and serializes data to a Python object with custom data structure which is displayed along with the company name and other specific data. It then instantiates the Person object with new employee data and adds to the existing company names and displays the updated output as below:

```
Loaded Python object from company.yaml:
Company(name=Tech Solutions
Inc.,departments=[Department(name=Engineering,manager=Person(name=John
Doe,age=32,address=Address(street=456 Oak Avenue,city=New
York,country=USA)),employees=[Person(name=Alice
Johnson,age=34,address=Address(street=123 Mountain View,city=Boulder,country=USA)),
Person(name=Bob Smith,age=29,address=Address(street=456 Trailhead
Lane,city=Denver,country=USA))]),
Department(name=Marketing,manager=Person(name=Anya
Sharma,age=38,address=Address(street=15 Lavender
Lane,city=London,country=UK)),employees=[Person(name=Dennis
Menees,age=28,address=Address(street=Kings Square,city=London,country=UK))])])
--------------------
Company Name: Tech Solutions Inc.
Engineering Department Manager's Name: John Doe
Total employees in Marketing Department: 1
--------------------
Added new Marketing employee

--- Updated company data in YAML ---
!company
name: Tech Solutions Inc.
departments:
- !department
  name: Engineering
  manager: !person
    name: John Doe
    age: 32
    address: !address
      street: 456 Oak Avenue
      city: New York
      country: USA
  employees:
  - !person
    name: Alice Johnson
    age: 34
    address: !address
      street: 123 Mountain View
      city: Boulder
      country: USA
  - !person
    name: Bob Smith
    age: 29
    address: !address
      street: 456 Trailhead Lane
      city: Denver
      country: USA
- !department
  name: Marketing
  manager: !person
    name: Anya Sharma
    age: 38
    address: !address
```

```
      street: 15 Lavender Lane
      city: London
      country: UK
  employees:
  - !person
    name: Dennis Menees
    age: 28
    address: !address
      street: Kings Square
      city: London
      country: UK
  - !person
    name: Mark Jones
    age: 25
    address: !address
      street: 969 Cox Rd
      city: North Carolina
      country: USA
```



```
    if dep.name=='Marketing':
        print(f"Total employees in Marketing Department: {len(dep.employees)}")
print("-" * 20)

# Create new Python objects and serialize to YAML
new_employee=Person('Mark Jones',25,Address('969 Cox Rd','North Carolina','USA'))
for dep in company.departments:
    if dep.name=='Marketing':
        dep.employees.append(new_employee)
print("Added new Marketing employee")
print("\n--- Updated company data in YAML ---")
yaml_data=yaml.safe_dump(company, sort_keys=False)
print(yaml_data)
```

```
      city: London
      country: UK
  employees:
  - !person
    name: Dennis Menees
    age: 28
    address: !address
      street: Kings Square
      city: London
      country: UK
  - !person
    name: Mark Jones
    age: 25
    address: !address
      street: 969 Cox Rd
      city: North Carolina
      country: USA
```

PyYAML has a flexibility of defining custom tags directly using `yaml.YAMLObject` sub class without the need of defining and registering constructors and representers explicitly *(using the `yaml.add_constructor()` and `yaml.add_representer()` functions)*. The `yaml.YAMLObject` class uses metaclass magic to register a constructor, which transforms a YAML node to a Python class instance, and a representer, which serializes a Python class instance to a YAML node.

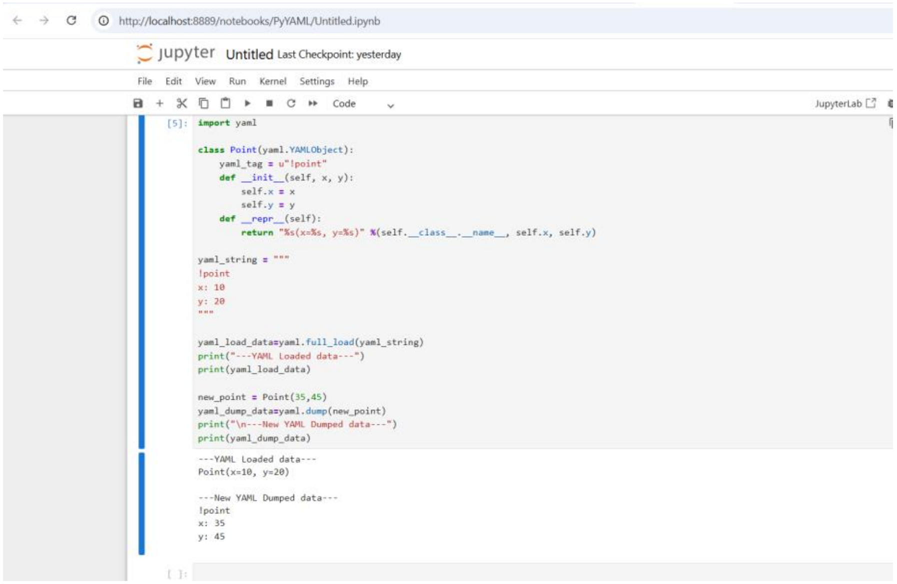
Run the following code to create a `Point` class which calls `yaml.YAMLObject` sub class and represent `x` and `y` values and serialize and deserialize YAML data:

```
import yaml

class Point(yaml.YAMLObject):
    yaml_tag = u"!point"
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return "%s(x=%s, y=%s)" %(self.__class__.__name__, self.x, self.y)

yaml_string = """
!point
x: 10
y: 20
"""

yaml_load_data=yaml.full_load(yaml_string)
print("---YAML Loaded data---")
print(yaml_load_data)

new_point = Point(35,45)
yaml_dump_data=yaml.dump(new_point)
print("\n---New YAML Dumped data---")
print(yaml_dump_data)
```

Once the above code is executed, it creates a `Point` type with `x` and `y` attributes and registers the custom tag `!point` (specified in `yaml_tag` attribute) with `Point` class that calls `yaml.YAMLObject` subclass which makes easy to deserialize YAML formatted data using `!point` custom tag to the Python custom object `Point` and serializes the Python `Point` object back to YAML structure with `!point` type and displays the following output:

```
---YAML Loaded data---
Point(x=10, y=20)

---New YAML Dumped data---
!point
x: 35
```

```
y: 45
```


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: yesterday
File Edit View Run Kernel Settings Help
Code
JupyterLab

```
[5]: import yaml

     class Point(yaml.YAMLObject):
         yaml_tag = u"!point"
         def __init__(self, x, y):
             self.x = x
             self.y = y
         def __repr__(self):
             return "%s(x=%s, y=%s)" %(self.__class__.__name__, self.x, self.y)

     yaml_string = """
     !point
     x: 10
     y: 20
     """

     yaml_load_data=yaml.full_load(yaml_string)
     print("---YAML Loaded data---")
     print(yaml_load_data)

     new_point = Point(35,45)
     yaml_dump_data=yaml.dump(new_point)
     print("\n---New YAML Dumped data---")
     print(yaml_dump_data)
```

```
---YAML Loaded data---
Point(x=10, y=20)

---New YAML Dumped data---
!point
x: 35
y: 45
```

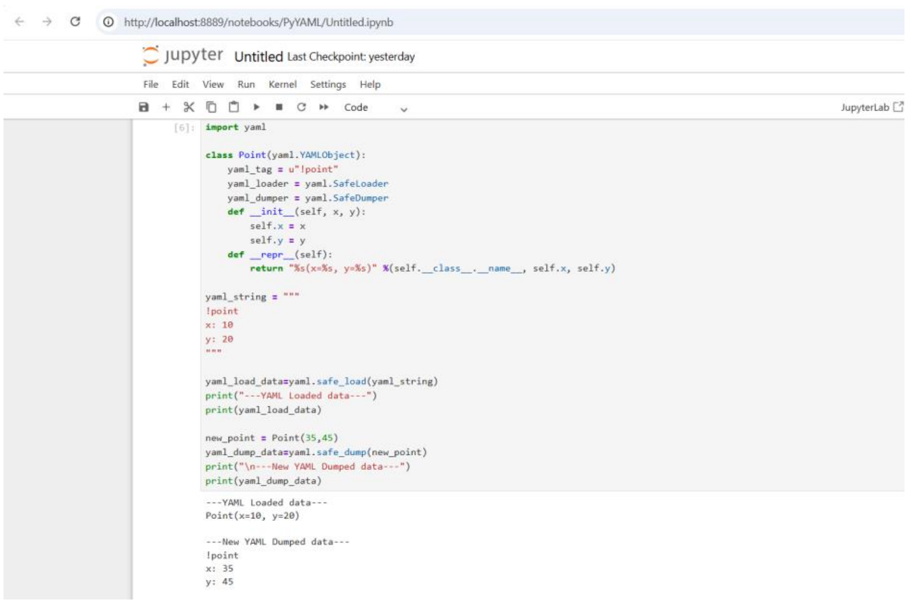
```
[ ]:
```

Note that in the above code, `dump()` and `full_load()` functions are used for YAML data serialization/deserialization to represent custom tags. In this case, if `safe_dump()` or `safe_load()` functions are used, it throws error since these safe functions calls `SafeDumper` and `SafeLoader` classes which by default knows only data represented in scalars (strings, integers, Booleans, nulls, timestamps) and collections (maps and sequences) but not custom tags. However, this default behavior can be overridden by explicitly specifying `yaml_loader` and `yaml_dumper` attributes in the class that calls `YAMLObject` sub class so that safe functions can identify custom tags and serializes/deserializes data safely.

Run the following code to safely convert the point data from YAML to Python and vice versa:

```
import yaml

class Point(yaml.YAMLObject):
    yaml_tag = u"!point"
    yaml_loader = yaml.SafeLoader
    yaml_dumper = yaml.SafeDumper
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return "%s(x=%s, y=%s)" %(self.__class__.__name__, self.x, self.y)
```

```
yaml_string = """
!point
x: 10
y: 20
"""

yaml_load_data=yaml.safe_load(yaml_string)
print("---YAML Loaded data---")
print(yaml_load_data)

new_point = Point(35,45)
yaml_dump_data=yaml.safe_dump(new_point)
print("\n---New YAML Dumped data---")
print(yaml_dump_data)
```


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: yesterday
File Edit View Run Kernel Settings Help
Code
JupyterLab
[6]:

```
import yaml

class Point(yaml.YAMLObject):
    yaml_tag = u"!point"
    yaml_loader = yaml.SafeLoader
    yaml_dumper = yaml.SafeDumper
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return "%s(x=%s, y=%s)" %(self.__class__.__name__, self.x, self.y)

yaml_string = """
!point
x: 10
y: 20
"""

yaml_load_data=yaml.safe_load(yaml_string)
print("---YAML Loaded data---")
print(yaml_load_data)

new_point = Point(35,45)
yaml_dump_data=yaml.safe_dump(new_point)
print("\n---New YAML Dumped data---")
print(yaml_dump_data)
```

```
---YAML Loaded data---
Point(x=10, y=20)

---New YAML Dumped data---
!point
x: 35
y: 45
```

## 8.14 Use Python Specific Tags

PyYAML uses specific tags to represent various Python-specific data types and objects within a YAML document. These tags allow for the serialization and deserialization of Python objects that do not have direct equivalents in standard YAML types. The YAML file or string using these tags can be parsed using `yaml.load()` or `yaml.unsafe_load()` or `yaml.full_load()` functions to construct Python objects correctly. However, these functions must be used cautiously on untrusted YAML files as these can execute arbitrary Python code.

Some common Python-specific tags in PyYAML include:

- `!!python/bool` to represent a Python `boolean` value (True or False).
- `!!python/int` to represent a Python `int` object.
- `!!python/long` to represent a Python `long` object (in Python 3).
- `!!python/float` to represent a Python `float` object.
- `!!python/str` to represent a Python `str` object (in Python 3).
- `!!python/bytes` to represent a Python `bytes` object (in Python 3).
- `!!python/Unicode` to represent a Python `unicode` object (in Python 2, equivalent to str in Python 3).
- `!!python/datetime` to represent a Python `datetime.datetime` object.
- `!!python/date` to represent a Python `datetime.date` object.
- `!!python/time` to represent a Python `datetime.time` object.
- `!!python/complex` to represent a Python `complex` object.
- `!!python/list` to represent a Python `list` object.
- `!!python/tuple` to represent a Python `tuple` object.
- `!!python/dict` to represent a Python `dict` object.
- `!!python/set` to represent a Python `set` object.

Below are some complex Python-specific tags that can be used to create or call Python objects. These tags are identified as most unsafe and can be parsed using `yaml.unsafe_load()` *(or* *`yaml.load(Loader=UnsafeLoader)`)* function only as these execute arbitrary code that can prone to security risks.

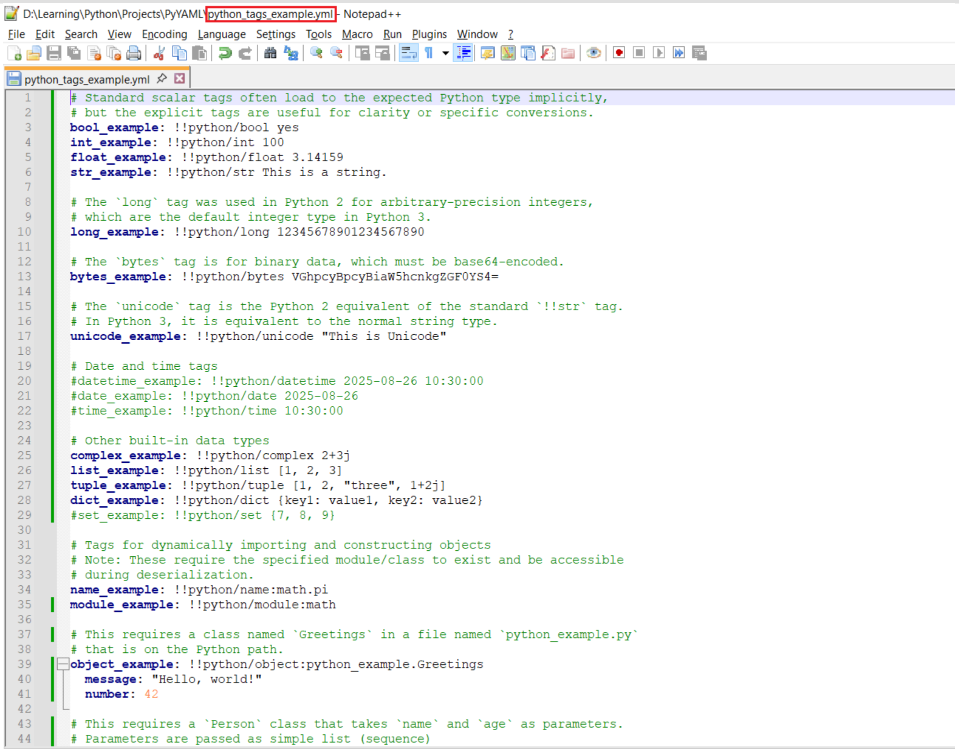
- `!!python/name:` is used to represent a specific, named object including a function, class, or variable within a module.
- `!!python/module:` is used to reference and import a specific Python module. Once the module is imported, other tags or commands could then interact with it.
- `!!python/object:` is a general tag used to create an uninitialized object and set its attributes. PyYAML attempts to construct the object by calling the class's `__new__` and `__setstate__` methods implicitly.
- `!!python/object/new:` is used to instantiate a specific Python class with arguments by calling the class's `__new__` method implicitly and `__init__` method explicitly.

- `!!python/object/apply`: is used to call a specific function in a Python module.

To demonstrate these tags, create a YAML file named `python_tags_example.yml` with the below content in the current directory:

```
# Standard scalar tags often load to the expected Python type implicitly,
# but the explicit tags are useful for clarity or specific conversions.
bool_example: !!python/bool yes
int_example: !!python/int 100
float_example: !!python/float 3.14159
str_example: !!python/str This is a string.

# The `long` tag was used in Python 2 for arbitrary-precision integers,
# which are the default integer type in Python 3.
long_example: !!python/long 12345678901234567890

# The `bytes` tag is for binary data, which must be base64-encoded.
bytes_example: !!python/bytes VGhpcyBpcyBiaW5hcnkgZGF0YS4=

# The `unicode` tag is the Python 2 equivalent of the standard `!!str` tag.
# In Python 3, it is equivalent to the normal string type.
unicode_example: !!python/unicode "This is Unicode"

# Date and time tags
#datetime_example: !!python/datetime 2025-08-26 10:30:00
#date_example: !!python/date 2025-08-26
#time_example: !!python/time 10:30:00

# Other built-in data types
complex_example: !!python/complex 2+3j
list_example: !!python/list [1, 2, 3]
tuple_example: !!python/tuple [1, 2, "three", 1+2j]
dict_example: !!python/dict {key1: value1, key2: value2}
#set_example: !!python/set {7, 8, 9}

# Tags for dynamically importing and constructing objects
# Note: These require the specified module/class to exist and be accessible
# during deserialization.
name_example: !!python/name:math.pi
module_example: !!python/module:math

# This requires a class named `Greetings` in a file named
`python_example.py`
# that is on the Python path.
object_example: !!python/object:python_example.Greetings
  message: "Hello, world!"
  number: 42

# This requires a `Person` class that takes `name` and `age` as parameters.
# Parameters are passed as simple list (sequence)
object_new_example: !!python/object/new:python_example.Person
  - "Jane Doe"
  - 40
```

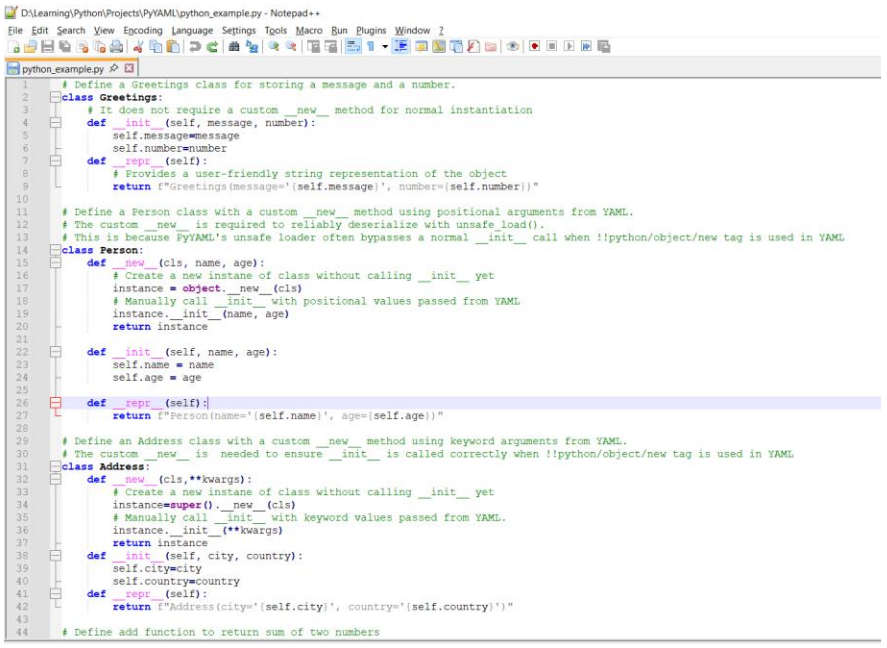
```
# This requires a `Address` class in `python_example` module, which takes
two key words.
# Another example of using !!python/object/new tag with specific key words
expected by `Address` class
object_new_example2: !!python/object/new:python_example.Address
  kwds:
    city: New York
    country: USA

# This requires a function named `add` in `python_example`, which takes two
arguments.
# This tag uses `apply` to call a function during deserialization.
object_apply_example: !!python/object/apply:python_example.add
  args: [100, 200]
```


D:\Learning\Python\Projects\PyYAML\python_tags_example.yml - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
python_tags_example.yml

```
# Standard scalar tags often load to the expected Python type implicitly,
# but the explicit tags are useful for clarity or specific conversions.
bool_example: !!python/bool yes
int_example: !!python/int 100
float_example: !!python/float 3.14159
str_example: !!python/str This is a string.

# The `long` tag was used in Python 2 for arbitrary-precision integers,
# which are the default integer type in Python 3.
long_example: !!python/long 12345678901234567890

# The `bytes` tag is for binary data, which must be base64-encoded.
bytes_example: !!python/bytes VGhpcyBpcyBiaW5hcnkgZGF0YS4=

# The `unicode` tag is the Python 2 equivalent of the standard `!!str` tag.
# In Python 3, it is equivalent to the normal string type.
unicode_example: !!python/unicode "This is Unicode"

# Date and time tags
#datetime_example: !!python/datetime 2025-08-26 10:30:00
#date_example: !!python/date 2025-08-26
#time_example: !!python/time 10:30:00

# Other built-in data types
complex_example: !!python/complex 2+3j
list_example: !!python/list [1, 2, 3]
tuple_example: !!python/tuple [1, 2, "three", 1+2j]
dict_example: !!python/dict {key1: value1, key2: value2}
#set_example: !!python/set {7, 8, 9}

# Tags for dynamically importing and constructing objects
# Note: These require the specified module/class to exist and be accessible
# during deserialization.
name_example: !!python/name:math.pi
module_example: !!python/module:math

# This requires a class named `Greetings` in a file named `python_example.py`
# that is on the Python path.
object_example: !!python/object:python_example.Greetings
  message: "Hello, world!"
  number: 42

# This requires a `Person` class that takes `name` and `age` as parameters.
# Parameters are passed as simple list (sequence)
```

Then create a Python file named `python_example.py` *(as the above YAML code expects* `pyton_example` ***module with necessary classes and functions)*** with the below content in the current directory:

```
# Define a Greetings class for storing a message and a number.
class Greetings:
    # It does not require a custom __new__ method for normal instantiation
    def __init__(self, message, number):
        self.message=message
        self.number=number
    def __repr__(self):
        # Provides a user-friendly string representation of the object
        return f"Greetings(message='{self.message}', number={self.number})"

# Define a Person class with a custom __new__ method using positional
arguments from YAML.
# The custom __new__ is required to reliably deserialize with unsafe_load().
# This is because PyYAML's unsafe loader often bypasses a normal __init__
call when !!python/object/new tag is used in YAML
class Person:
    def __new__(cls, name, age):
        # Create a new instane of class without calling __init__ yet
        instance = object.__new__(cls)
        # Manually call __init__ with positional values passed from YAML
        instance.__init__(name, age)
        return instance

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})"

# Define an Address class with a custom __new__ method using keyword
arguments from YAML.
# The custom __new__ is needed to ensure __init__ is called correctly when
!!python/object/new tag is used in YAML
class Address:
    def __new__(cls,**kwargs):
        # Create a new instane of class without calling __init__ yet
        instance=super().__new__(cls)
        # Manually call __init__ with keyword values passed from YAML.
        instance.__init__(**kwargs)
        return instance
    def __init__(self, city, country):
        self.city=city
        self.country=country
    def __repr__(self):
        return f"Address(city='{self.city}', country='{self.country}')"

# Define add function to return sum of two numbers
def add(x,y):
    return x+y
```


D:\Learning\Python\Projects\PyYAML\python_example.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
python_example.py

```
# Define a Greetings class for storing a message and a number.
class Greetings:
    # It does not require a custom __new__ method for normal instantiation
    def __init__(self, message, number):
        self.message=message
        self.number=number
    def __repr__(self):
        # Provides a user-friendly string representation of the object
        return f"Greetings(message='{self.message}', number={self.number})"

# Define a Person class with a custom __new__ method using positional arguments from YAML.
# The custom __new__ is required to reliably deserialize with unsafe_load().
# This is because PyYAML's unsafe loader often bypasses a normal __init__ call when !!python/object/new tag is used in YAML
class Person:
    def __new__(cls, name, age):
        # Create a new instane of class without calling __init__ yet
        instance = object.__new__(cls)
        # Manually call __init__ with positional values passed from YAML
        instance.__init__(name, age)
        return instance

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})"

# Define an Address class with a custom __new__ method using keyword arguments from YAML.
# The custom __new__ is  needed to ensure __init__ is called correctly when !!python/object/new tag is used in YAML
class Address:
    def __new__(cls,**kwargs):
        # Create a new instane of class without calling __init__ yet
        instance=super().__new__(cls)
        # Manually call __init__ with keyword values passed from YAML.
        instance.__init__(**kwargs)
        return instance
    def __init__(self, city, country):
        self.city=city
        self.country=country
    def __repr__(self):
        return f"Address(city='{self.city}', country='{self.country}')"

# Define add function to return sum of two numbers
```

Run the following code in any Python interpreter such as Jupyter Notebook to parse the YAML file:

```
import yaml
from python_example import Greetings, Person, Address

with open('python_tags_example.yml', 'r') as file:
    data = yaml.unsafe_load(file)

print("--- Loaded Data ---")
for key, value in data.items():
    print(f"{key}: {value} (type: {type(value)})")
```

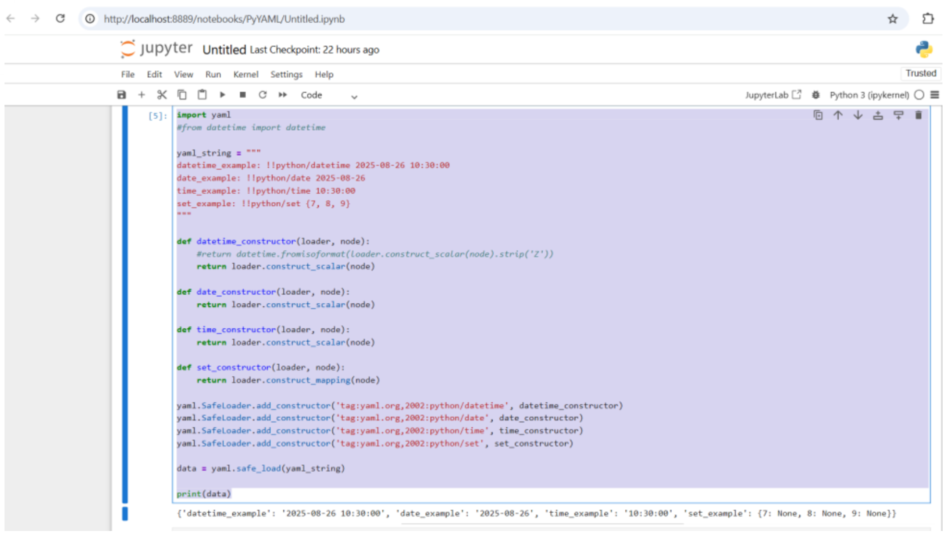
Once the above code is executed, it parses the YAML file and displays the loaded data:

```
--- Loaded Data ---
bool_example: True (type: <class 'bool'>)
int_example: 100 (type: <class 'int'>)
float_example: 3.14159 (type: <class 'float'>)
str_example: This is a string. (type: <class 'str'>)
long_example: 12345678901234567890 (type: <class 'int'>)
bytes_example: b'This is binary data.' (type: <class 'bytes'>)
unicode_example: This is Unicode (type: <class 'str'>)
complex_example: (2+3j) (type: <class 'complex'>)
list_example: [1, 2, 3] (type: <class 'list'>)
tuple_example: (1, 2, 'three', '1+2j') (type: <class 'tuple'>)
dict_example: {'key1': 'value1', 'key2': 'value2'} (type: <class 'dict'>)
name_example: 3.141592653589793 (type: <class 'float'>)
module_example: <module 'math' (built-in)> (type: <class 'module'>)
```

```
object_example: Greetings(message='Hello, world!', number=42) (type: <class
'python_example.Greetings'>)
object_new_example: Person(name='Jane Doe', age=40) (type: <class
'python_example.Person'>)
object_new_example2: Address(city='New York', country='USA') (type: <class
'python_example.Address'>)
object_apply_example: 300 (type: <class 'int'>)
```

Often, PyYAML may not be able to parse datetime values with Python specific tags using `yaml.full_load()` or `yaml.unsafe_load()` in which case it is recommended to create custom constructors and register tags with `SafeLoader` to load using `safe_load()`.

```
import yaml
#from datetime import datetime

yaml_string = """
datetime_example: !!python/datetime 2025-08-26 10:30:00
date_example: !!python/date 2025-08-26
time_example: !!python/time 10:30:00
set_example: !!python/set {7, 8, 9}
"""

def datetime_constructor(loader, node):
    #return datetime.fromisoformat(loader.construct_scalar(node).strip('Z'))
    return loader.construct_scalar(node)

def date_constructor(loader, node):
    return loader.construct_scalar(node)

def time_constructor(loader, node):
    return loader.construct_scalar(node)

def set_constructor(loader, node):
    return loader.construct_mapping(node)

yaml.SafeLoader.add_constructor('tag:yaml.org,2002:python/datetime',
datetime_constructor)
yaml.SafeLoader.add_constructor('tag:yaml.org,2002:python/date', date_constructor)
yaml.SafeLoader.add_constructor('tag:yaml.org,2002:python/time', time_constructor)
yaml.SafeLoader.add_constructor('tag:yaml.org,2002:python/set', set_constructor)

data = yaml.safe_load(yaml_string)

print(data)
```


http://localhost:8889/notebooks/PyYAML/Untitled.ipynb
jupyter Untitled Last Checkpoint: 22 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
[5]: import yaml
     #from datetime import datetime

     yaml_string = """
     datetime_example: !!python/datetime 2025-08-26 10:30:00
     date_example: !!python/date 2025-08-26
     time_example: !!python/time 10:30:00
     set_example: !!python/set {7, 8, 9}
     """

     def datetime_constructor(loader, node):
         #return datetime.fromisoformat(loader.construct_scalar(node).strip('Z'))
         return loader.construct_scalar(node)

     def date_constructor(loader, node):
         return loader.construct_scalar(node)

     def time_constructor(loader, node):
         return loader.construct_scalar(node)

     def set_constructor(loader, node):
         return loader.construct_mapping(node)

     yaml.SafeLoader.add_constructor('tag:yaml.org,2002:python/datetime', datetime_constructor)
     yaml.SafeLoader.add_constructor('tag:yaml.org,2002:python/date', date_constructor)
     yaml.SafeLoader.add_constructor('tag:yaml.org,2002:python/time', time_constructor)
     yaml.SafeLoader.add_constructor('tag:yaml.org,2002:python/set', set_constructor)

     data = yaml.safe_load(yaml_string)

     print(data)

     {'datetime_example': '2025-08-26 10:30:00', 'date_example': '2025-08-26', 'time_example': '10:30:00', 'set_example': {7: None, 8: None, 9: None}}
```

## 8.15 Low-Level Functions

PyYAML provides a low-level event-based API with classes and functions for parsing and emitting YAML, similar to SAX for XML. This API allows to get fine-grained control over the YAML serialization process and is typically used for advanced tasks, such as creating custom loaders, adding special tags, handling streaming data, managing complex data manipulation tasks that are not possible with simple YAML load and dump operations.

Parsing a YAML document (converting YAML to Python objects) is a multi-step process involving several low-level components:

- **Scanning**: The `yaml.scan()` function takes a stream (a YAML string or file) as input to scan and produces a sequence of tokens such as `StreamStartToken`, `StreamEndToken`, `DocumentStartToken`, `DocumentEndToken`, `KeyToken`, `ValueToken`, etc. These tokens represent the fundamental components of the YAML document, such as indentation, keys, and values. Each token has a unique meaning and commonly takes `start_mark` and `end_mark` attributes to specify where it starts and where it ends, including the exact line and column number, as well as the offset from the beginning of the document which can be taken as an advantage to manipulate the YAML file.
- **Parsing**: The `yaml.parse()` function takes the sequence of tokens from the scanner *(when a YAML `stream` is passed to `parse()`, it scans internally to fetch sequence of tokens)* and produces a sequence of parsing events such as `StreamStartEvent`, `StreamEndEvent`, `DocumentStartEvent`, `DocumentEndEvent`,

`AliasEvent`, `ScalarEvent`, etc. These events provide more structured information about the document's hierarchy.

- **Composing**: The `yaml.compose()` function uses the parsing events *(when a YAML `stream` is passed to `compose()`, it scans and parses internally)* to build a representation graph with nodes such as *`MappingNode`*, *`SequenceNode` and `ScalarNode`,* which is a tree-like data structure that represents the logical structure of the YAML document. This is often the starting point for custom loaders. The `yaml.compose_all()` function uses the parsing events and returns a sequence of representation graphs corresponding to the multiple documents in the stream.
- **Constructing**: The loader classes (*`Loader`*, *`SafeLoader`*, *`FullLoader`* etc.) take the representation graph and construct the final Python objects (dictionaries, lists, strings).

The process for emitting a YAML document (converting Python objects to YAML) involves the following components:

- **Serializing**: The `yaml.serialize()` or `yaml.serialize_all()` function takes a representation graph (the logical structure of the Python object) or a sequence of representation graphs and serializes into a stream.
- **Emitting**: The `yaml.emit()` function takes the sequence of parsing events and writes them to a stream, producing the final YAML document.
- **Representing**: The dumper classes (*`Dumper`*, *`SafeDumper`*, etc.) represent Python objects as a representation graph before serialization.

Run the following code to see how these functions work:

```
import yaml

yaml_string = """
name: Alice
age: 30
"""

print(f"---YAML String--- {yaml_string}")

print("\n" + "="*40)
# Step 1: Scan the input stream into tokens
tokens = yaml.scan(yaml_string)
print("---Produced Tokens after scanning YAML---")
for token in tokens:
    print(token)
```

```
# Step 2: Parse the tokens into events
events = yaml.parse(yaml_string)
print("\n---Produced Events after parsing---")
for event in events:
    print(event)

# Step 3: Compose the events into a representation graph
node = yaml.compose(yaml_string)
print("\n---Produced Representation graph after composing---")
print(node)

# Step 4: Load into Python objects
loaded_yaml = yaml.safe_load(yaml_string)
print("\n---Produced Python object after loading---")
print(loaded_yaml)

print("\n" + "="*40)
# Serialize a representation graph
print("\n---Produced stream after serializing---")
print(yaml.serialize(yaml.compose(yaml_string)))

# Emit into stream
print("\n---Produced stream after emitting---")
print(yaml.emit(yaml.parse(yaml_string)))

# Direct dump
print("\n---Produced stream after dumping directly---")
print(yaml.dump(yaml.safe_load(yaml_string)))
```

Once the above code is executed, it displays the following output:

```
name: Alice
age: 30

========================================
---Produced Tokens after scanning YAML---
StreamStartToken(encoding=None)
BlockMappingStartToken()
KeyToken()
ScalarToken(plain=True, style=None, value='name')
ValueToken()
ScalarToken(plain=True, style=None, value='Alice')
KeyToken()
ScalarToken(plain=True, style=None, value='age')
ValueToken()
ScalarToken(plain=True, style=None, value='30')
BlockEndToken()
StreamEndToken()

---Produced Events after parsing---
```

```
StreamStartEvent()
DocumentStartEvent()
MappingStartEvent(anchor=None, tag=None, implicit=True)
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='name')
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='Alice')
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='age')
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='30')
MappingEndEvent()
DocumentEndEvent()
StreamEndEvent()

---Produced Representation graph after composing---
MappingNode(tag='tag:yaml.org,2002:map',
value=[(ScalarNode(tag='tag:yaml.org,2002:str', value='name'),
ScalarNode(tag='tag:yaml.org,2002:str', value='Alice')),
(ScalarNode(tag='tag:yaml.org,2002:str', value='age'),
ScalarNode(tag='tag:yaml.org,2002:int', value='30'))])

---Produced Python object after loading---
{'name': 'Alice', 'age': 30}

========================================

---Produced stream after serializing---
name: Alice
age: 30


---Produced stream after emitting---
name: Alice
age: 30


---Produced stream after dumping directly---
age: 30
name: Alice
```

In the above code, `yaml.emit()` function took the list of events generated by `yaml.parse()` function. Instead, a list of events can be generated manually corresponding to the Python data and passed to `yaml.emit()` function to display a YAML formatted output.

For example, run the following code:

```
import sys
      from yaml import (
    emit,
    StreamStartEvent,
    StreamEndEvent,
    DocumentStartEvent,
    DocumentEndEvent,
    MappingStartEvent,
    MappingEndEvent,
    SequenceStartEvent,
    SequenceEndEvent,
    ScalarEvent,
)

# Python object with a list of dictionaries
python_data = [
    {"name": "John Doe", "occupation": "gardener"},
    {"name": "Lucy Black", "occupation": "teacher"},
]

# The emit function takes a generator that yields events
# Define a function to produce the necessary events
def my_emitter_events(data):
    yield StreamStartEvent()
    yield DocumentStartEvent()

    # Start the root sequence
    yield SequenceStartEvent(anchor=None, tag=None, implicit=True, flow_style=None)

    for item in data:
        # For each dictionary in the list, start a mapping
        yield MappingStartEvent(anchor=None, tag=None, implicit=True,
flow_style=None)

        for key, value in item.items():
            # Emit key-value pairs as scalar events
            yield ScalarEvent(
                anchor=None,
                tag=None,
                implicit=(True, False),
                value=str(key),
                style=None,
            )
            yield ScalarEvent(
                anchor=None,
                tag=None,
                implicit=(True, False),
                value=str(value),
                style=None,
            )

        # End the mapping for the current dictionary
        yield MappingEndEvent()

    # End the root sequence
    yield SequenceEndEvent()
    yield DocumentEndEvent()
    yield StreamEndEvent()


# Create a list to capture the output events
events_list = list(my_emitter_events(python_data))
```

```
print("--- Generated Events ---")
for event in events_list:
    print(event)
print("\n" + "=" * 25 + "\n")

# Emit to write to a stream
print("--- Emitted YAML ---")
stream = sys.stdout
emit(my_emitter_events(python_data), stream)
```

The above code displays the following output:

```
--- Generated Events ---
StreamStartEvent()
DocumentStartEvent()
SequenceStartEvent(anchor=None, tag=None, implicit=True)
MappingStartEvent(anchor=None, tag=None, implicit=True)
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='name')
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='John Doe')
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='occupation')
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='gardener')
MappingEndEvent()
MappingStartEvent(anchor=None, tag=None, implicit=True)
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='name')
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='Lucy Black')
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='occupation')
ScalarEvent(anchor=None, tag=None, implicit=(True, False), value='teacher')
MappingEndEvent()
SequenceEndEvent()
DocumentEndEvent()
StreamEndEvent()

=========================

--- Emitted YAML ---
- name: John Doe
  occupation: gardener
- name: Lucy Black
  occupation: teacher
```

These YAML low level functions have key advantages in real-time applications:

- **Custom data processing:** The most significant advantage is the ability to write custom logic for how YAML data is handled. Instead of letting a library automatically convert the entire document to a standard data structure like a dictionary, it is possible to process the data as a stream of events (such as `DocumentStart`, `Scalar`, `SequenceEnd`) and write custom handlers for each event type.

- **Reduced memory usage:** Processing YAML as a stream of events is ideal for very large files, as it avoids loading the entire document into memory at once. This prevents memory issues in performance-critical or memory-constrained applications.
- **Real-time data handling:** In event-driven systems like those used in IoT or log processing, low-level functions allow to react to data as it arrives. An application can begin processing a document or stream as soon as the first event is parsed, rather than waiting for the entire file to be loaded.
- **Building custom tools and libraries**: Using low-level event-based API, custom tools can be created that inspect or modify YAML documents in ways that high-level functions don't allow. This includes writing linters, validators, and formatters that work directly on the parsed node structure or event stream.
- **Security and sanitization**: With manual control on the deserialization process, one can prevent code execution attacks and other vulnerabilities that can occur with high-level functions like `yaml.load()` when parsing untrusted files.
- **Specialized formatting:** Standard dump functions provide limited control over the output style. Low-level emitters enable to manually construct the output with precise control over indentation, line breaks, and other formatting details that are important for certain applications or human readability.
- **Generating valid documents with advanced features**: Low-level APIs are necessary for generating complex YAML documents that use anchors, aliases, and custom tags, ensuring they are correctly constructed and resolved without relying on an intermediate object model.

# 9 YAML USE CASES

YAML is widely used in software development, infrastructure automation, and API management. It's human-readable syntax makes it a preferred format for configuration files, data serialization, and Infrastructure as Code (IaC).

**1. Configuration Files:**

YAML is widely used for configuration in applications like **Docker Compose**, **Kubernetes** and CI/CD pipelines. It's ease of understanding makes it straightforward for anyone to pick up Docker YAML set-up files and understand what is happening.

The basic example of Docker Compose file (`docker-compose.yaml`) looks like:

```
version: '3'
services:
  web:
    image: nginx
    ports:
      - "80:80"
    environment:
      - NGINX_HOST=localhost
      - NGINX_PORT=80
```

**2. Data Serialization:**

YAML is used to serialize data for APIs and configuration management tools by converting complex data structures into a human-readable format and easily parsed by machines.

For example, an API request body formatted in YAML looks like:

```
user:
  id: 123
  name: "John Doe"
  email: "johndoe@example.com"
  active: true
```

**3. Infrastructure as code (IaC):**

Configuration management tools like **Kubernetes** and **Ansible** leverage YAML to define system states, automate processes, and ensure consistency across environments.

Kubernetes utilizes YAML to define resources such as pods, services, and deployments, enabling automated orchestration of containerized applications.

Here’s an example of a Kubernetes Pod configuration:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: app-container
      image: my-app:latest
      ports:
        - containerPort: 8080
```

The basic example of Kubernetes Deployment YAML file looks like:

```
apiversion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app:nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

In Ansible, YAML is used to write playbooks that define system states, tasks, and dependencies, ensuring that infrastructure components are configured consistently.

A sample YAML file for an Ansible playbook can be structured as:

```
---
- name: Configure Web Server
  hosts: webservers
  become: true
  vars:
    http_port: 80
    max_clients: 200

  tasks:
    - name: Install Apache Web Server
      ansible.builtin.package:
        name: httpd
        state: present

    - name: Ensure Apache service is running and enabled
      ansible.builtin.service:
        name: httpd
        state: started
        enabled: true

    - name: Copy custom index.html
      ansible.builtin.copy:
        src: files/index.html
        dest: /var/www/html/index.html
        mode: '0644'

    - name: Configure Apache with template
      ansible.builtin.template:
        src: templates/httpd.conf.j2
        dest: /etc/httpd/conf/httpd.conf
        mode: '0644'
      notify: Restart Apache

  handlers:
    - name: Restart Apache
      ansible.builtin.service:
        name: httpd
        state: restarted
```

**4. API Documentation:**

API services like **OpenAPI** and **Swagger** use YAML to define endpoints and data structures in an easy-to-read way. YAML is used to outline API methods, request parameters, response formats, and authentication methods. OpenAPI specifications use YAML to document RESTful APIs. This allows to provide a clear blueprint for generating client SDKs, interactive API documentation, and automated testing. This structured format ensures consistency across API implementations.

For example, an OpenAPI specification can be defined as below in YAML:

```
openapi: 3.0.0
info:
  title: User API
  version: "1.0"
paths:
  /users:
    get:
      summary: Retrieve a list of users
      responses:
        "200":
          description: Successful response
```

# 10 YAML BEST PRACTICES

For easy readability, maintainability, and securing YAML files, it is recommended to follow best practices focusing on consistent formatting, structural clarity, and secure handling of data.

Here are some of the best practices to follow when writing a clean YAML code:

- **Consistent indentation** – Always use spaces (not tabs) and be consistent with the number of spaces used throughout the file (usually, this should be two or four spaces).
- **Descriptive key names** – To make the configuration smooth and easy to understand, the key names used throughout the file should be meaningful.
- **Case Sensitive** – Since YAML is case sensitive, be careful while using key names. For example, YAML considers `name:` and `Name:` as two different keys.
- **Colons** – Ensure that each key-value pair is properly separated by a colon followed by a space to avoid misconfigurations.
- **Structure complex data** – Use hyphen (-) to represent a list of data while the general representation is a mapping with key-value pairs.
- **Proper syntax** – Ensure colons (**:**) and hyphens (**-**) are followed by a space to maintain readability and correctness.
- **Trailing spaces** – Remove any trailing spaces after the scalar value in a mapping or sequence, as they can cause issues in parsing the YAML file.
- **Comments** – Add comments using the **#** character to explain configurations, making YAML files easier to understand.
- **Data Types** – Use appropriate data types (strings, numbers, booleans) for values. Always use lowercase `true` or `false` values for booleans.
- **Quotes for strings** – Use single or double quotes for strings containing special characters or reserved words or starts with a number to avoid misinterpretation.
- **Chomp modifiers** – Use chomp modifiers for multi-line SQL.
- **Blank Lines** - Use blank lines to separate logical sections for improved visual organization.

- **Validations** – Validate YAML files regularly using YAML linters or validators to check for syntax errors and adherence to specific schema requirements.
- **Short files** – Though everything can be configured in a single YAML file, consider breaking configurations into multiple files to speed up the debugging process.
- **Clear sensitive data** – Remove sensitive data from configuration files after deployment if it is no longer needed for run-time operation.