**Kubernetes With Python**

**Section 1: Introduction to Python and Kubernetes Integration**

**🔹 Why Automate Kubernetes with Python?**

Kubernetes is a declarative and API-driven container orchestration system. Everything you do via the CLI (kubectl) is ultimately just an API call to the Kubernetes **API server**.

🔑 Python is a powerful language for automation, scripting, and orchestration. It’s simple enough for quick scripts and strong enough for full-blown infrastructure tooling.

Using Python with Kubernetes offers:

| **Benefit** | **Description** |
| --- | --- |
| 🧠 Readability | Python’s syntax is clean and expressive, especially for YAML-like K8s manifests |
| 🧩 Modularity | You can organize K8s interactions as modules, classes, or REST-based microservices |
| ⚙️ Automation | Ideal for scripting repeatable deployments, updates, rollbacks, and inspections |
| 🔌 Extensibility | Python can integrate with APIs, DBs, notification systems, CI/CD platforms |
| 📡 Native Client SDK | The official Kubernetes Python client is mature and actively maintained |

**🔹 The Official Kubernetes Python Client SDK**

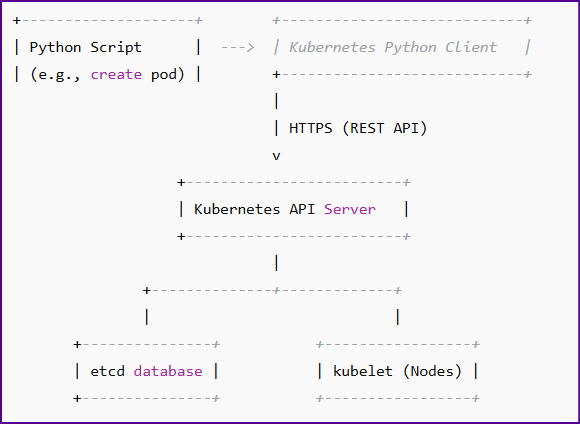
Install it using pip:

pip install kubernetes

This package gives you access to:

* Low-level and high-level interfaces to the **Kubernetes REST API**
* Auth and config loading from:
  + Local kubeconfig (~/.kube/config)
  + In-cluster config via ServiceAccount token
* Full access to CoreV1Api, AppsV1Api, BatchV1Api, and more
* Watch streams, exec sessions, logs, port forwarding

**🔹 Conceptual Architecture**



**🔹 How Kubernetes Python Client Works**

The kubernetes Python module wraps HTTP requests around Kubernetes resources using **OpenAPI specifications** and classes generated from them.

Example:

* A Pod in YAML is an instance of V1Pod class
* A Deployment is a V1Deployment object
* You interact via interfaces like client.CoreV1Api().create\_namespaced\_pod(...)

**🔹 Kubernetes APIs Exposed in Python**

| **API Class** | **Used For** |
| --- | --- |
| CoreV1Api | Pods, Services, Nodes, ConfigMaps |
| AppsV1Api | Deployments, StatefulSets, DaemonSets |
| BatchV1Api | Jobs, CronJobs |
| CustomObjectsApi | CRDs and Operators |
| RbacAuthorizationV1Api | Roles, RoleBindings |

**🔹 Authentication Options**

* 📁 **Kubeconfig**:
  + Used outside of the cluster
  + Loaded with config.load\_kube\_config()
  + Supports multiple contexts
* 🔐 **In-cluster ServiceAccount**:
  + Used inside Pods
  + Loaded with config.load\_incluster\_config()
  + Automatically uses /var/run/secrets/kubernetes.io/serviceaccount/token

**🔹 Sample Code Skeleton**

from kubernetes import client, config

config.load\_kube\_config() # or config.load\_incluster\_config()

v1 = client.CoreV1Api()

apps\_v1 = client.AppsV1Api()

# Use `v1` to deal with Pods, Nodes, Services

# Use `apps\_v1` to manage Deployments, StatefulSets

**Section 2: Deploying an Application and Exposing It via a LoadBalancer**

In this section, we’ll combine **two essential Kubernetes operations** done through Python:

1. Creating a **Deployment** (to run a replicated set of Pods)
2. Creating a **Service of type LoadBalancer** (to expose those Pods externally)

This is equivalent to the YAML-based process of:

kubectl apply -f deployment.yaml

kubectl apply -f service.yaml

But here, we’re doing everything via **Python SDK**, with each line **fully explained** in theory and internal logic.

**🧱 Full Combined Code (Clean & Corrected)**

from kubernetes import client, config

# Step 1: Load Kubernetes configuration

config.load\_kube\_config()

# Step 2: Create client APIs

apps\_v1 = client.AppsV1Api()

v1 = client.CoreV1Api()

# Step 3: Define Deployment

deployment = client.V1Deployment(

metadata=client.V1ObjectMeta(name="example-deployment"),

spec=client.V1DeploymentSpec(

replicas=2,

selector=client.V1LabelSelector(match\_labels={"app": "example"}),

template=client.V1PodTemplateSpec(

metadata=client.V1ObjectMeta(labels={"app": "example"}),

spec=client.V1PodSpec(containers=[

client.V1Container(

name="nginx",

image="nginx:1.17",

ports=[client.V1ContainerPort(container\_port=80)]

)

])

)

)

)

# Step 4: Create Deployment in "default" namespace

response = apps\_v1.create\_namespaced\_deployment(

namespace="default",

body=deployment

)

print(f"Deployment Created. Status: {response.metadata.name}")

# Step 5: Define LoadBalancer Service

service = client.V1Service(

metadata=client.V1ObjectMeta(name="example-service"),

spec=client.V1ServiceSpec(

selector={"app": "example"},

ports=[client.V1ServicePort(port=80, target\_port=80)],

type="LoadBalancer"

)

)

# Step 6: Create Service

response1 = v1.create\_namespaced\_service(

namespace="default",

body=service

)

print(f"Service Created. Name: {response1.metadata.name}")

**Explanation (Line-by-Line)**

**🔸 from kubernetes import client, config**

* **client** module: Contains all the Kubernetes API classes (like AppsV1Api, CoreV1Api, V1Deployment, etc.)
* **config** module: Loads cluster configuration from file or in-cluster environment.

**🔸 config.load\_kube\_config()**

* Loads the **Kubernetes configuration file** from the default path ~/.kube/config
* Reads cluster, context, user, token, and CA cert information
* Enables the Python client to **authenticate and talk** to the K8s cluster securely

**🔸 apps\_v1 = client.AppsV1Api()**

* This object gives us access to **Deployments, StatefulSets, DaemonSets**.
* Under the hood, it initializes a RESTClient with the proper base path (/apis/apps/v1)
* Deployment-related operations like create, patch, delete, and list use this interface

**🔸 v1 = client.CoreV1Api()**

* This gives access to **Core V1 resources**:
  + Pods
  + Services
  + ConfigMaps
  + Secrets
  + Nodes
* Internally talks to /api/v1

**Deployment Creation Breakdown**

**🔸 V1Deployment object**

Represents the YAML structure for a Deployment:

apiVersion: apps/v1

kind: Deployment

metadata:

name: example-deployment

spec:

replicas: 2

selector:

matchLabels:

app: example

template:

metadata:

labels:

app: example

spec:

containers:

- name: nginx

image: nginx:1.17

ports:

- containerPort: 80

**🔍 Deep Dive:**

* metadata=client.V1ObjectMeta(name="example-deployment")
  + This sets the name of the Deployment
  + It is used to track and manage the Deployment
* replicas=2
  + Indicates the **desired number of Pods**
  + Kubernetes will maintain exactly 2 Pods as long as the Deployment exists
* selector=client.V1LabelSelector(match\_labels={"app": "example"})
  + Selector that matches the labels inside the Pod template
  + Must match exactly to associate Pods with this Deployment
* template: Defines the Pod to create
  + Includes its own metadata and a spec
  + Uses labels={"app": "example"} to ensure selector can match
* containers: A list of containers inside the Pod
  + name="nginx": Container name
  + image="nginx:1.17": Docker image to use
  + ports=[...]: Exposes port 80 inside the container

**🔸 apps\_v1.create\_namespaced\_deployment(...)**

* This function sends an HTTP POST to:

POST /apis/apps/v1/namespaces/default/deployments

* It deploys the defined Deployment object in the default namespace
* Returns a V1Deployment object containing metadata, status, and spec

**🔸 print(f"Deployment Created. Status: {response.metadata.name}")**

* If creation is successful, prints:

Deployment Created. Status: example-deployment

**🌐 Service Creation Breakdown (LoadBalancer)**

apiVersion: v1

kind: Service

metadata:

name: example-service

spec:

selector:

app: example

ports:

- port: 80

targetPort: 80

type: LoadBalancer

**🔍 Deep Dive:**

* metadata=client.V1ObjectMeta(name="example-service")
  + Defines the Service name that can be referenced internally
* selector={"app": "example"}
  + Binds this Service to any Pod whose label includes app=example
  + Matches the Pods from the Deployment we created earlier
* ports=[client.V1ServicePort(port=80, target\_port=80)]
  + **port=80**: Port on which the Service listens (external)
  + **target\_port=80**: Port to forward to inside the container
* type="LoadBalancer"
  + Tells Kubernetes to provision an **external load balancer** (e.g., via AWS ELB, Azure LB)
  + It assigns a public IP address to the Service

**🔸 v1.create\_namespaced\_service(...)**

* Performs a POST call to:

POST /api/v1/namespaces/default/services

* Creates a Service resource in the default namespace

**🔸 print(f"Service Created. Name: {response1.metadata.name}")**

* After successful creation, prints:

Service Created. Name: example-service

**Architectural Impact**

By combining Deployment and LoadBalancer creation in one Python script, you achieve:

* 💡 **Declarative orchestration** via Python (not YAML)
* 🛠️ **Automation-friendly rollout** of web services
* 🌍 **Production-ready exposure** to internet traffic
* 🎛️ Scalable and repeatable deployment scripts

**Real-World Use Case**

Imagine you're building a Python CLI (devopsctl) for your team. One subcommand is:

devopsctl deploy-app nginx

Internally, this CLI will:

1. Build the Deployment definition in Python
2. Push it to the cluster
3. Attach a LoadBalancer service
4. Return the external IP

Now your infra team can deploy NGINX or any app **without touching YAML**.

**Section 3: Listing Pods in a Namespace**

**✅ Objective**

This section covers how to use Python to **query and list Pods** in a specific namespace using the official Kubernetes Python SDK.

This is the Pythonic equivalent of:

kubectl get pods -n default

We will deeply understand:

* What the API call actually does
* What data it returns
* How to use that information for real-world DevOps or internal tooling

**🧱 Full Code – Listing Pods**

from kubernetes import client, config

# Load kubeconfig from ~/.kube/config

config.load\_kube\_config()

# Instantiate the CoreV1 API

v1 = client.CoreV1Api()

# Define the namespace

namespace = "default"

print(f"Listing Pods in namespace: {namespace}")

# List all Pods in that namespace

pods = v1.list\_namespaced\_pod(namespace=namespace)

# Print each Pod's name

for pod in pods.items:

print(f"Pod Name: {pod.metadata.name}")

**Ultra Deep Explanation**

**🔸 from kubernetes import client, config**

* **client**: Contains all generated Kubernetes API classes.
* **config**: Used to load configuration files that contain cluster authentication and context.

**🔸 config.load\_kube\_config()**

* Reads your local Kubernetes configuration file from:

~/.kube/config

* It contains:
  + Cluster info
  + Authentication method
  + TLS certificates or tokens
  + Namespace context
* Enables your Python code to behave like kubectl under the hood

**🔸 v1 = client.CoreV1Api()**

* Initializes the Core V1 API.
* Provides access to:
* Pods
* Services
* Nodes
* Namespaces
* ConfigMaps
* Secrets

Under the hood, binds to:

/api/v1

**🔸 namespace = "default"**

* Specifies the Kubernetes namespace in which you want to list Pods
* You can change it to "kube-system", "webapps", "dev", etc., depending on your cluster setup

**🔸 v1.list\_namespaced\_pod(namespace=namespace)**

**This is the core method call:**

* Sends an HTTP request like:

GET /api/v1/namespaces/default/pods

* Returns a **V1PodList** object that contains:
  + A list of V1Pod objects (each representing a Pod)
  + Metadata about the list
  + Potential status conditions if something goes wrong

**🔸 for pod in pods.items:**

* pods.items is a list of V1Pod objects
* Each V1Pod includes:
  + metadata: name, labels, UID, creationTimestamp
  + spec: containers, volumes, tolerations
  + status: phase, conditions, IP address, container statuses

**🔸 print(f"Pod Name: {pod.metadata.name}")**

* Accesses the **name of the Pod** from the metadata section
* You can extend this to also print:

pod.status.phase # 'Running', 'Pending', 'Succeeded', 'Failed'

pod.status.pod\_ip # Internal cluster IP

pod.spec.node\_name # Node the Pod is scheduled to

**Sample Output**

Listing Pods in namespace: default

Pod Name: myapp-deployment-6f79c7bb96-7cg6s

Pod Name: myapp-deployment-6f79c7bb96-mt2q9

Pod Name: vault-0

Pod Name: noteapp-6b7b86744c-bzzqp

**📚 Real Use Cases**

| **Use Case** | **Description** |
| --- | --- |
| DevOps Dashboard | Display real-time Pods per namespace |
| CI/CD Debugging | Show last created Pods for a deploy pipeline |
| Health Audit | Filter Pods by status phase, e.g., Pending or CrashLoopBackOff |
| Log Management | Select Pods to fetch logs from using .read\_namespaced\_pod\_log() |
| Cleanup Automation | Find Pods matching specific labels and delete them automatically |

**🔐 RBAC Note**

To list Pods, your ServiceAccount (if running inside the cluster) must have this permission:

apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list", "watch"]

Bind this using a Role and RoleBinding.

**Extension Ideas**

Once you’ve listed Pods, you can:

**✅ Get logs of each Pod:**

log = v1.read\_namespaced\_pod\_log(name=pod.metadata.name, namespace="default")

**✅ Get Pod status:**

print(pod.status.phase)

**✅ Filter only Running Pods:**

if pod.status.phase == "Running":

print(f"Healthy Pod: {pod.metadata.name}")

**✅ Delete problematic Pods:**

if pod.status.phase == "Failed":

v1.delete\_namespaced\_pod(name=pod.metadata.name, namespace="default")

**✅ Summary of Section 3**

| **Task** | **Tool/Method** |
| --- | --- |
| List Pods | v1.list\_namespaced\_pod() |
| Print names | pod.metadata.name |
| Filter by phase | pod.status.phase |
| RBAC Needed | "pods" with "list" permission |
| Real Uses | Dashboarding, debugging, logs, cleanup |

**Section 4: Executing Commands in a Pod (Simulating kubectl exec with Python)**

**🎯 Objective**

This section will cover how to use the **Kubernetes Python SDK** to programmatically **execute a shell command inside a running Pod**, just like:

kubectl exec -it <pod-name> -- /bin/sh -c "echo Hello from the pod"

We’ll explain:

* How the stream module works
* When and why you use exec
* Security, RBAC, and lifecycle considerations
* Full line-by-line breakdown

**🧱 Full Code – Executing Command in a Pod**

from kubernetes import client, config, stream

# Load Kubernetes config

config.load\_kube\_config()

# Create API instance

v1 = client.CoreV1Api()

# Define the command to run inside the Pod

exec\_command = [

"/bin/sh",

"-c",

"echo Hello from the pod; sleep 5"

]

# Stream the exec command into the container

response = stream.stream(

v1.connect\_get\_namespaced\_pod\_exec,

name="example-deployment-9568bf58c-b5rhw",

namespace="default",

command=exec\_command,

stderr=True,

stdin=False,

stdout=True,

tty=False,

)

print(response)

**Explanation**

**🔸 from kubernetes import client, config, stream**

* stream is a submodule used to stream input/output (stdin, stdout, stderr) into a Pod
* stream.stream() wraps the exec WebSocket API exposed by Kubernetes

**🔸 config.load\_kube\_config()**

* Same as before: loads kubeconfig to authenticate with the cluster

**🔸 v1 = client.CoreV1Api()**

* Allows us to work with Pods and call connect\_get\_namespaced\_pod\_exec

**🔸 exec\_command = ["/bin/sh", "-c", "echo Hello from the pod; sleep 5"]**

**What does this do?**

* You are telling Kubernetes to **spawn a shell** (/bin/sh)
* Then **execute a single shell command** via -c
* This command will:
  + Print a message
  + Sleep for 5 seconds

**⚙️ Kubernetes Exec Internals**

**Under the hood:**

* Kubernetes API server opens a **WebSocket connection** to the Kubelet
* Kubelet opens a shell session inside the container
* stdin, stdout, stderr, and tty streams are managed as **multiplexed channels**

Python SDK uses the stream module to:

* Establish the WebSocket
* Send and receive byte data
* Close session on completion

**🔸 stream.stream(...)**

This is the main function to simulate exec via Python:

stream.stream(

func\_to\_call,

name="pod-name",

namespace="namespace",

command=["..."],

stderr=True,

stdin=False,

stdout=True,

tty=False,

)

**Arguments:**

| **Parameter** | **Explanation** |
| --- | --- |
| func\_to\_call | Usually connect\_get\_namespaced\_pod\_exec, defines the API to stream |
| name | Pod name |
| namespace | Namespace of the Pod |
| command | Shell command list to execute |
| stderr | Capture standard error |
| stdout | Capture standard output |
| stdin | Set to True if you want to type interactively |
| tty | Use a TTY (like interactive terminals) |

**🔸 Output**

Hello from the pod

The print will wait for the full output to complete (5 seconds in this case).

**📦 Real-World Use Cases**

| **Use Case** | **Description** |
| --- | --- |
| Debugging | Run custom shell commands inside containers |
| Health Checks | Verify runtime dependencies |
| Log Inspection | Check logs from within container filesystem |
| Script Trigger | Launch scripts already bundled inside container |
| Realtime Monitoring | Stream output like a background shell or tail logs |

**🔐 Security & RBAC**

**🔒 RBAC Permissions**

To execute commands inside Pods, your identity must have:

apiGroups: [""]

resources: ["pods/exec"]

verbs: ["create"]

This permission is **not given by default** even if you can get or list pods.

**⚠️ Security Caveats**

* **Pod exec = shell access** = equivalent to SSH
* Can be used to escalate privileges if:
  + Pod runs as root
  + Pod has hostPath mounts
* Always restrict access to pods/exec only to **trusted roles**

**🔧 Pro Tips**

* Use tty=True + stdin=True for interactive sessions (like Python shells)
* Use stdout=False to silence output
* Handle exceptions for Pods not in Running state
* Only exec into **containers with a shell** (e.g., busybox, alpine, ubuntu)

**💡 Advanced Extensions**

**📌 Execute inside a specific container:**

response = stream.stream(

v1.connect\_get\_namespaced\_pod\_exec,

name="mypod",

namespace="default",

container="my-container",

command=["/bin/bash", "-c", "ls -la"],

stderr=True,

stdin=False,

stdout=True,

tty=False

)

**📌 Automate with Pod discovery:**

* Find a running Pod dynamically by label:

pods = v1.list\_namespaced\_pod(namespace="default", label\_selector="app=example")

pod\_name = pods.items[0].metadata.name

Then exec into pod\_name.

**Section 5: Listing Nodes in a Kubernetes Cluster (Code 5)**

**🎯 Objective**

In this section, we'll learn how to use Python to **list all Kubernetes nodes** using the Kubernetes Python client. This is equivalent to:

kubectl get nodes

We'll explore:

* How node metadata is structured
* How this data can power real-world automation
* How DevOps teams can integrate this into dashboards, schedulers, and health monitors

**🧱 Full Code – List Nodes**

from kubernetes import client, config

# Load kubeconfig

config.load\_kube\_config()

# Instantiate CoreV1 API client

v1 = client.CoreV1Api()

# Fetch all nodes

nodes = v1.list\_node()

# Print node names

for node in nodes.items:

print(f"Node Name: {node.metadata.name}")

**Explanation (Line-by-Line)**

**🔸 from kubernetes import client, config**

* Standard SDK import:
  + client: API wrappers
  + config: Loads kubeconfig from ~/.kube/config

**🔸 config.load\_kube\_config()**

* Authenticates you against the cluster
* Reads the current context, user, and certificate

**🔸 v1 = client.CoreV1Api()**

* Instantiates the CoreV1 API client
* This class lets you work with:
  + Pods
  + Services
  + Namespaces
  + Nodes 🔥

**🔸 nodes = v1.list\_node()**

Sends an HTTP request:

GET /api/v1/nodes

Returns a V1NodeList object:

* .items: a list of V1Node objects

**🔸 for node in nodes.items:**

Iterates through the list of nodes. Each node is a full metadata + spec + status definition.

You can extract a wide range of info.

**🔸 print(f"Node Name: {node.metadata.name}")**

Prints the human-readable name of each node, typically:

ip-172-31-32-45.ap-south-1.compute.internal

This is the same as what you see in kubectl get nodes.

**📚 Real-World Node Metadata You Can Use**

**🧾 node.metadata.labels**

print(node.metadata.labels)

Labels such as:

* kubernetes.io/hostname
* node-role.kubernetes.io/control-plane
* topology.kubernetes.io/zone

These are used for **scheduling, affinity rules, and AZ-aware deployments.**

**📦 node.status.capacity**

print(node.status.capacity)

Returns resource capacity of the node:

json

CopyEdit

{

"cpu": "4",

"memory": "16383712Ki",

"pods": "110"

}

Useful for **cluster resource planning**.

**📊 node.status.allocatable**

Returns the *actual available* resources after reserving system overhead.

Compare it with capacity to find room for new workloads.

**📡 node.status.addresses**

for addr in node.status.addresses:

print(f"{addr.type}: {addr.address}")

Returns IPs and DNS names:

* InternalIP
* Hostname
* ExternalIP (if applicable)

**🔍 node.status.conditions**

List of conditions like:

* Ready
* MemoryPressure
* DiskPressure
* PIDPressure

Each condition is an object with:

type: "Ready"

status: "True"

reason: "KubeletReady"

Example logic to find unhealthy nodes:

python

CopyEdit

for condition in node.status.conditions:

if condition.type == "Ready" and condition.status != "True":

print(f"Node {node.metadata.name} is not ready!")

**🔐 RBAC Required**

To list nodes, your identity (user or ServiceAccount) needs:

apiGroups: [""]

resources: ["nodes"]

verbs: ["get", "list"]

**⚙️ Use Cases**

| **Use Case** | **Description** |
| --- | --- |
| Monitoring Dashboard | Display node health, capacity, readiness |
| Auto Scheduler | Make placement decisions based on zone, CPU, memory |
| Alerting | Trigger alerts for pressure conditions |
| Capacity Reporting | Track growth trends and plan horizontal scale |
| Cost Optimization | Shut down low-usage nodes or detect zombie nodes |

**🛠 Example Extension**

**✅ Print all zones (multi-AZ deployments):**

for node in v1.list\_node().items:

zone = node.metadata.labels.get("topology.kubernetes.io/zone", "unknown")

print(f"{node.metadata.name} is in zone: {zone}")

**📈 Sample Output**

Node Name: ip-172-31-45-201.ap-south-1.compute.internal

Node Name: ip-172-31-46-110.ap-south-1.compute.internal

Node Name: ip-172-31-48-89.ap-south-1.compute.internal

**🧠 Tips**

* Use field\_selector to filter by metadata.name
* Use label\_selector to get specific node pools
* Run this inside a Pod with limited scope to audit environments dynamically