**Different ways to schedule tasks in Python**

[[Gaurav Kumar](https://gaurav-adarshi.medium.com/?source=post_page-----45e03d5411ee--------------------------------)](https://gaurav-adarshi.medium.com/?source=post_page-----45e03d5411ee--------------------------------)

[Gaurav Kumar](https://gaurav-adarshi.medium.com/?source=post_page-----45e03d5411ee--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff319c9e06fc1&operation=register&redirect=https%3A%2F%2Fgaurav-adarshi.medium.com%2Fdifferent-ways-to-schedule-tasks-in-python-45e03d5411ee&user=Gaurav+Kumar&userId=f319c9e06fc1&source=post_page-f319c9e06fc1----45e03d5411ee---------------------post_header-----------)

8 min read

·

Dec 8

67

3

**6 Ways to Schedule and Execute Jobs**

Scheduling tasks is a vital aspect of many modern applications, whether it involves regularly checking APIs or databases, monitoring the health of systems, or implementing auto-scaling functionalities. Even auto-scaling systems like Kubernetes and Apache Mesos rely on periodic checks to ensure the smooth operation of deployed applications.

To maintain a separation between task execution and core business logic, independent execution queues such as Redis queues are often employed. In this article, we’ll explore various methods for scheduling and running Python jobs through simple tutorials. These methods include basic loops, threaded loops for concurrency, the Schedule Library for task scheduling, Python Crontab for time-based scheduling, and RQ Scheduler for leveraging decoupled queues.

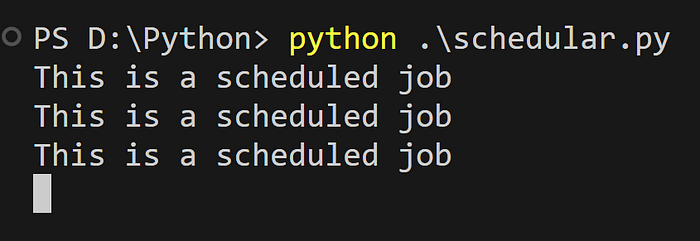
**Method 1: The Simple Loop**

Setting up a job using simple loops is a direct and simple method. It entails running a continuous while loop that intermittently triggers a specific function. While there are more efficient approaches available, this straightforward method gets the job done.

To introduce time intervals between function calls, you can utilize the sleep function from Python’s built-in time module. Despite being less favored due to its relatively lower visual appeal and readability compared to other methods, scheduling Python jobs with simple loops involves using the time module. This enables the creation of a loop that executes a designated function at predetermined time intervals. Below is an example code snippet:

import time  
  
def scheduled\_job():  
 print("This is a scheduled job")  
  
while True:  
 scheduled\_job()  
 time.sleep(10) # Run the job every 10 seconds

**Output:**



In the provided example, the scheduled\_job function is set to run every 10 seconds by employing a basic loop and the time.sleep() method. Feel free to customize the time interval based on your specific needs.

However, it’s important to note a common issue associated with this method. It is not ideal for tasks that are extensive or intricate, as it has the potential to obstruct the main thread and cause your application to become unresponsive.

**Method 2: Simple Threaded Loops**

To address the issue of blocking, consider employing simple threaded loops using Python’s threading module. This technique resembles simple loops, but instead of relying on a single loop, you generate a new thread for each task you wish to execute.

Here’s a concise breakdown of the process:

1. Specify the function that you intend to execute periodically.
2. Employ the threading module to craft a thread for the designated function.
3. Launch the thread and establish a time delay, indicating how frequently the function should be executed. This delay is set using the sleep function from the time module.

import threading  
import time  
  
def job():  
 print("This is a scheduled job")  
  
def run\_threaded(job\_func):  
 job\_thread = threading.Thread(target=job\_func)  
 job\_thread.start()  
  
while True:  
 run\_threaded(job)  
 time.sleep(60)

In this instance, we first define the job function that we wish to execute every 60 seconds. Subsequently, we create a run\_threaded function, which is responsible for generating a new thread dedicated to the job function and initiating it.

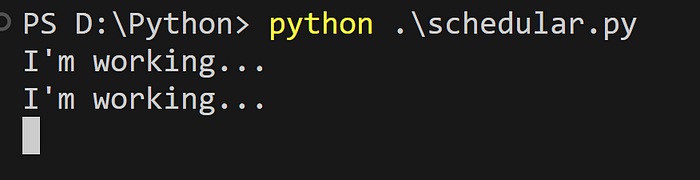
To ensure the periodic execution of the job function, we employ an infinite loop that continually calls the run\_threaded function. To introduce a time gap of 60 seconds between each call, we utilize the sleep function. This setup ensures the job function runs every minute.

It’s essential to note a common challenge associated with this approach: once a thread is started and in progress, the primary program or thread cannot modify its behavior. If there’s a need for the thread to respond to specific conditions or events, additional resources or logic may be required in the program to monitor these situations and take appropriate actions accordingly.

**Method 3: A Schedule Library**

Although while loops may lack visual appeal, this issue can be resolved by utilizing a scheduling library.

import schedule  
import time  
  
def job():  
 print("I'm working...")  
  
schedule.every(10).seconds.do(job)  
  
while True:  
 schedule.run\_pending()  
 time.sleep(1)



In this example, we define a function called “job” that we aim to execute every 10 seconds. The scheduling is handled by using `schedule.every(10).seconds.do(job)` to set up the function to run at 10-second intervals. The code structure becomes cleaner and more readable compared to using a while loop. Additionally, modifying the code to schedule multiple jobs or adjusting the schedule for an existing job is more straightforward.

However, a prevalent issue with the Python `schedule` library is that it may cease running if the process or program employing the library exits or terminates. This could lead to missed job executions or incomplete schedules. To circumvent this problem, it is advisable to run the `schedule` library in a separate thread or process that persists even if the main program concludes.

**Method 4: Python Crontab**

A Python crontab serves as a mechanism for automating tasks at specified intervals, employing the syntax of the UNIX cron utility. Cron, a time-based job scheduler native to Unix-like operating systems, inspired the Python crontab package. This Python package provides a user-friendly interface for creating schedules reminiscent of cron, enabling the execution of Python code.

Python crontab facilitates the precise specification of intervals, times of day, days of the week, or months of the year when a particular script or program should run. Scheduling is achieved by articulating a set of rules using the crontab syntax, which precisely outlines when a task should be carried out.

Consider the following example that illustrates the crontab syntax:

┌───────────── minute (0–59)  
│ ┌───────────── hour (0–23)  
│ │ ┌───────────── day of the month (1–31)  
│ │ │ ┌───────────── month (1–12)  
│ │ │ │ ┌───────────── day of the week (0–6) (Sunday to Saturday;  
│ │ │ │ │ 7 is also Sunday on some systems)  
│ │ │ │ │  
│ │ │ │ │  
\* \* \* \* \* command to execute

In this depiction, the first five columns outline the time intervals for executing the command, while the last column specifies the command to be executed at the designated intervals.

To use Python crontab, begin by installing it via pip. Once installed, create a crontab object and utilize its schedule method to add a task scheduled at specific intervals.

Here’s an example code snippet that schedules a Python script to run every day at 6 AM:

from crontab import CronTab  
  
# Create a new crontab object  
cron = CronTab(user='username')  
  
# Add a new cron job to run the script every day at 6 AM  
job = cron.new(command='python /path/to/script.py')  
job.setall('0 6 \*')  
  
# Write the job to the user's crontab  
cron.write()

This code establishes a new crontab object, adds a cron job for running the specified Python script daily at 6 AM, and writes the job to the user’s crontab. It uses the `setall` method to define the time interval in crontab syntax.

**Common Problem:**  
It’s important to note that the `write()` method must be manually executed to save the schedules in Python-Crontab, as the library lacks an auto-save feature.

**Method 5: Rq Scheulder**

When dealing with tasks that cannot be immediately performed, the creation of a queue becomes essential. Organizing these tasks based on a queue system, such as Last In, First Out (LIFO) or First In, First Out (FIFO), is crucial. Python-rq, a tool leveraging Redis as a broker, facilitates job queuing. Information about a new job is stored in a hash map, encompassing details like creation time, enqueue time, origin, data, and description.

To execute these queued jobs, a program, known as a worker, is employed. Workers have an entry in the Redis cache and are responsible for dequeuing jobs and updating their status in Redis. While jobs can be queued as needed, rq-scheduler becomes indispensable for scheduling them.

from datetime import datetime, timedelta  
from redis import Redis  
from rq\_scheduler import Scheduler  
  
# Create a connection to Redis  
redis\_conn = Redis(host='localhost', port=6379)  
  
# Create a scheduler object  
scheduler = Scheduler(connection=redis\_conn)  
  
# Define the job function  
def my\_job():  
 print("Hello, world!")  
  
# Schedule the job to run every minute  
scheduler.schedule(  
 scheduled\_time=datetime.utcnow(), # Start immediately  
 func=my\_job,  
 interval=timedelta(minutes=1)  
)

In this illustration, we initiate a connection to Redis and instantiate a Scheduler object using that connection. We then define a straightforward job function that prints “Hello, world!” and schedule it to run every minute using the scheduler.schedule() method. The datetime.utcnow() function ensures the job starts immediately.

Common Problem: RQ scheduling necessitates a distinct worker process to execute the jobs, which may not be feasible for smaller applications or systems with limited resources. Additionally, as jobs are executed by separate worker processes, managing resources and ensuring optimal performance under heavy loads may require additional effort.

**How to Use Python with Redwood RunMyJobs**

Redwood RunMyJobs is a comprehensive Python job scheduling and workflow management system designed to execute tasks seamlessly across diverse platforms.

To integrate Python with RunMyJobs, adhere to these general steps:

**1. Install the RunMyJobs Python Client Library:**  
Begin by installing the RunMyJobs Python client library. The installation can be effortlessly accomplished using pip with the command: `pip install runmyjobs`.

**2. Create a Python Script for Your Job:**  
Develop a Python script that outlines your job. This script should encapsulate the desired code to execute along with any essential dependencies.

**3. Upload Script and Dependencies to RunMyJobs:**  
Utilize the runmyjobs command-line tool, bundled with the RunMyJobs Python client library, to upload your Python script and any required dependencies to the RunMyJobs platform.

**4. Define a Job in RunMyJobs:**  
Establish a job definition within RunMyJobs that references your Python script. This definition should encompass any necessary configuration settings, such as the specific environment in which your script should operate.

**5. Submit Your Job to RunMyJobs:**  
Submit your job to RunMyJobs utilizing the runmyjobs command-line tool.

**6. Monitor Job Status in RunMyJobs:**  
Keep tabs on the progress of your job within RunMyJobs. This can be achieved either through the runmyjobs command-line tool or the user-friendly RunMyJobs web interface.

**7. Optional: Install Schedule Library in Python:**  
If needed, you can employ the `pip install schedule` command to install the schedule library in Python.

Upon job completion, retrieve the output from RunMyJobs and employ it according to your requirements. These steps provide a general framework and may necessitate adjustments based on the specifics of your use case and environment.

**Conclusion**

Python serves as a versatile programming language, well-suited for a broad range of applications such as web development, data science, machine learning, and scientific computing.

Python provides diverse approaches for task scheduling, including simple loops, threaded loops, the Schedule library, Python crontab, and RQ Scheduler. While simple loops may not be ideal for lengthy or intricate tasks, threaded loops offer a solution to blocking issues at the expense of additional resources. The Schedule library, although visually appealing and easily modifiable, may halt when the program exits. Python crontab stands out as an excellent choice for scheduling tasks at specific intervals.

Selecting the most suitable Python scheduling method hinges on factors like task complexity, time intervals, and the level of monitoring required. For automation and streamlined monitoring of Python workflows and job dependencies, consider leveraging tools like Redwood RunMyJobs.

Follow me for more such articles and Happy Learning !!!