Advanced Python Automation Techniques

Parallel Processing and Multithreading

Parallel processing in Python enables developers to harness the full power of modern multi-core processors, dramatically improving performance for computationally intensive tasks. These techniques are essential for creating high-performance automation solutions.

Multiprocessing for CPU-Bound Tasks

The `multiprocessing` module provides a powerful framework for parallel execution that bypasses Python's Global Interpreter Lock (GIL):

```
 import multiprocessing
import time
import os

def cpu_intensive_task(data_chunk):
    """Process a chunk of data with CPU-intensive operations"""
    result = []
    for item in data_chunk:
        # Simulate complex computation
        processed = item ** 2
        for _ in range(1000000):
            processed = (processed * 1.01) // 1
        result.append(processed)
    return result

def parallel_processing_example(data, num_processes=None):
    """Process data in parallel using multiple CPU cores"""
    if num_processes is None:
        # Use all available cores by default
        num_processes = multiprocessing.cpu_count()

    # Split data into chunks for each process
    chunk_size = len(data) // num_processes
    chunks = [data[i:i + chunk_size] for i in range(0, len(data), chunk_size)]

    # Create a pool of worker processes
    start_time = time.time()
    with multiprocessing.Pool(processes=num_processes) as pool:
        # Map each chunk to a separate process
        results = pool.map(cpu_intensive_task, chunks)
```

```
    # Flatten results from all processes
    final_results = [item for sublist in results for item in sublist]

    end_time = time.time()
    print(f"Processed {len(data)} items using {num_processes} processes")
    print(f"Time taken: {end_time - start_time:.2f} seconds")
    print(f"Process IDs used: {os.getpid()}")

    return final_results
```

Process-Based Concurrency with Advanced Patterns

For more sophisticated parallel processing needs, we can implement advanced patterns:

```
from multiprocessing import Process, Queue, Lock, Value, Array
import time

class AdvancedParallelProcessor:
    def __init__(self, num_workers=4):
        self.num_workers = num_workers
        self.task_queue = Queue()
        self.result_queue = Queue()
        self.lock = Lock()
        self.counter = Value('i', 0)  # Shared counter
        self.workers = []

    def worker_process(self, worker_id):
        """Worker process that processes tasks from the queue"""
        while True:
            try:
                # Get task with timeout to allow checking for termination
                task = self.task_queue.get(timeout=0.5)

                # Check for termination signal
                if task is None:
                    break

                # Process the task
                task_id, func, args = task
                try:
                    result = func(*args)
                    # Safely increment the counter
                    with self.lock:
                        self.counter.value += 1
                    # Put result in result queue
                    self.result_queue.put((task_id, result, None))
                except Exception as e:
                    # Handle exceptions and report them
                    self.result_queue.put((task_id, None, str(e)))
```

```
            except Queue.Empty:
                # Queue is empty but worker continues running
                continue

        print(f"Worker {worker_id} shutting down")

    def start(self):
        """Start the worker processes"""
        for i in range(self.num_workers):
            p = Process(target=self.worker_process, args=(i,))
            p.daemon = True  # Daemon processes exit when main process exits
            p.start()
            self.workers.append(p)

    def submit_task(self, task_id, func, *args):
        """Submit a task to be processed by a worker"""
        self.task_queue.put((task_id, func, args))

    def get_results(self, block=True, timeout=None):
        """Get results from the result queue"""
        try:
            return self.result_queue.get(block=block, timeout=timeout)
        except Queue.Empty:
            return None

    def shutdown(self):
        """Shutdown the worker processes"""
        # Send termination signal to all workers
        for _ in range(self.num_workers):
            self.task_queue.put(None)

        # Wait for all workers to finish
        for p in self.workers:
            p.join()

        print(f"All workers shut down. Processed {self.counter.value} tasks.")
```

Advanced System Automation

System automation involves creating scripts that can monitor, manage, and respond to system events without human intervention.

Modular Script Design with Libraries

Creating modular, reusable automation scripts improves maintainability and scalability:

```python
 # utils.py - Reusable utility functions
import logging
import os
import datetime
import subprocess
import shutil

def setup_logging(log_file=None, level=logging.INFO):
    """Set up logging configuration"""
    if log_file:
        logging.basicConfig(
            filename=log_file,
            level=level,
            format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
            datefmt='%Y-%m-%d %H:%M:%S'
        )
    else:
        logging.basicConfig(
            level=level,
            format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
            datefmt='%Y-%m-%d %H:%M:%S'
        )
    return logging.getLogger()

def check_disk_space(path, threshold_gb=10):
    """Check if available disk space is below threshold"""
    stats = shutil.disk_usage(path)
    available_gb = stats.free / (1024**3)
    return available_gb < threshold_gb, available_gb

def archive_logs(source_dir, archive_dir, days_old=7):
    """Archive log files older than specified days"""
    if not os.path.exists(archive_dir):
        os.makedirs(archive_dir)

    current_time = datetime.datetime.now()
    cutoff_time = current_time - datetime.timedelta(days=days_old)

    archived_files = []
    for filename in os.listdir(source_dir):
        filepath = os.path.join(source_dir, filename)
        if os.path.isfile(filepath) and filename.endswith('.log'):
            file_time = datetime.datetime.fromtimestamp(os.path.getmtime(filepath))
            if file_time < cutoff_time:
                archive_path = os.path.join(archive_dir, filename)
                shutil.move(filepath, archive_path)
                archived_files.append(filename)

    return archived_files

def run_command(command, timeout=60):
```

```
    """Run a shell command with timeout and return output"""
    try:
        result = subprocess.run(
            command,
            shell=True,
            check=True,
            capture_output=True,
            text=True,
            timeout=timeout
        )
        return True, result.stdout
    except subprocess.CalledProcessError as e:
        return False, e.stderr
    except subprocess.TimeoutExpired:
        return False, f"Command timed out after {timeout} seconds"
```

Real-Time File Monitoring and Event Response

Python can be used to create sophisticated file monitoring systems:

```
 import time
import os
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
import subprocess
import logging

class AdvancedFileMonitor(FileSystemEventHandler):
    def __init__(self, watch_dir, patterns=None, ignore_patterns=None,
logger=None):
        self.watch_dir = watch_dir
        self.patterns = patterns or ["*"]
        self.ignore_patterns = ignore_patterns or []
        self.logger = logger or logging.getLogger(__name__)
        self.observer = None

    def on_modified(self, event):
        if not event.is_directory:
            self.logger.info(f"File modified: {event.src_path}")
            self._handle_file_event(event.src_path, "modified")

    def on_created(self, event):
        if not event.is_directory:
            self.logger.info(f"File created: {event.src_path}")
            self._handle_file_event(event.src_path, "created")

    def on_deleted(self, event):
        if not event.is_directory:
            self.logger.info(f"File deleted: {event.src_path}")
            self._handle_file_event(event.src_path, "deleted")
```

```
    def on_moved(self, event):
        if not event.is_directory:
            self.logger.info(f"File moved: {event.src_path} -> {event.dest_path}")
            self._handle_file_event(event.src_path, "moved", dest_path=event.dest_path)

    def _handle_file_event(self, file_path, event_type, **kwargs):
        """Handle file events based on file type and event type"""
        filename = os.path.basename(file_path)

        # Example: Run specific actions for configuration files
        if filename.endswith('.conf') or filename.endswith('.cfg'):
            if event_type in ('modified', 'created'):
                self.logger.info(f"Configuration file {filename} changed, validating...")
                self._validate_config_file(file_path)

        # Example: Process log files
        elif filename.endswith('.log'):
            if event_type == 'created' and os.path.getsize(file_path) > 10*1024*1024:  # 
10MB
                self.logger.warning(f"Large log file created: {filename}")

        # Example: Handle executable files
        elif os.access(file_path, os.X_OK) and event_type == 'created':
            self.logger.warning(f"Executable file created: {filename}")

    def _validate_config_file(self, file_path):
        """Validate a configuration file"""
        # Example implementation - customize based on config format
        try:
            with open(file_path, 'r') as f:
                content = f.read()

            # Simple validation - check for balanced brackets
            if content.count('{') != content.count('}'):
                self.logger.error(f"Config file {file_path} has unbalanced brackets")
                # Could trigger an alert or notification here
            else:
                self.logger.info(f"Config file {file_path} validated successfully")

        except Exception as e:
            self.logger.error(f"Error validating config file {file_path}: {str(e)}")

    def start(self):
        """Start monitoring the directory"""
        self.observer = Observer()
        self.observer.schedule(self, self.watch_dir, recursive=True)
        self.observer.start()
        self.logger.info(f"Started monitoring directory: {self.watch_dir}")

        try:
            while True:
```

```
            time.sleep(1)
        except KeyboardInterrupt:
            self.stop()

    def stop(self):
        """Stop monitoring"""
        if self.observer:
            self.observer.stop()
            self.observer.join()
            self.logger.info("Stopped file monitoring")
```

API Integration and Web Automation

Modern automation often involves integrating with web services and APIs:

```
 import requests
import json
import time
import hmac
import hashlib
import base64
from datetime import datetime
import asyncio
import aiohttp

class AdvancedAPIClient:
    def __init__(self, base_url, api_key=None, api_secret=None, timeout=30):
        self.base_url = base_url.rstrip('/')
        self.api_key = api_key
        self.api_secret = api_secret
        self.timeout = timeout
        self.session = requests.Session()
        self.rate_limit_remaining = None
        self.rate_limit_reset = None

    def _generate_signature(self, method, endpoint, params=None, data=None):
        """Generate HMAC signature for API authentication"""
        if not self.api_secret:
            return None

        timestamp = int(time.time() * 1000)
        payload = f"{timestamp}{method.upper()}{endpoint}"

        if params:
            sorted_params = "&".join(f"{k}={v}" for k, v in sorted(params.items()))
            payload += f"?{sorted_params}"

        if data:
            if isinstance(data, dict):
```

```
            payload += json.dumps(data, separators=(',', ':'))
        else:
            payload += data

        signature = hmac.new(
            self.api_secret.encode(),
            payload.encode(),
            hashlib.sha256
        ).hexdigest()

        return {
            'X-API-Key': self.api_key,
            'X-API-Signature': signature,
            'X-API-Timestamp': str(timestamp)
        }

    def _handle_response(self, response):
        """Handle API response, including rate limiting"""
        # Check for rate limit headers
        if 'X-RateLimit-Remaining' in response.headers:
            self.rate_limit_remaining = int(response.headers['X-RateLimit-Remaining'])

        if 'X-RateLimit-Reset' in response.headers:
            self.rate_limit_reset = int(response.headers['X-RateLimit-Reset'])

        # Handle rate limiting
        if response.status_code == 429:  # Too Many Requests
            reset_time = self.rate_limit_reset or (time.time() + 60)
            wait_time = max(0, reset_time - time.time())
            print(f"Rate limited. Waiting {wait_time:.2f} seconds")
            time.sleep(wait_time)
            return None

        # Parse response
        try:
            if response.content:
                return response.json()
            return None
        except ValueError:
            return response.text

    def request(self, method, endpoint, params=None, data=None, headers=None,
retry=3):
        """Make an API request with retries and error handling"""
        url = f"{self.base_url}/{endpoint.lstrip('/')}"
        method = method.upper()

        # Prepare headers
        request_headers = {}
        if headers:
            request_headers.update(headers)
```

```python
        # Add authentication if available
        auth_headers = self._generate_signature(method, endpoint, params, data)
        if auth_headers:
            request_headers.update(auth_headers)

        # Prepare request
        request_kwargs = {
            'method': method,
            'url': url,
            'headers': request_headers,
            'timeout': self.timeout
        }

        if params:
            request_kwargs['params'] = params

        if data:
            if isinstance(data, dict):
                request_kwargs['json'] = data
                request_headers['Content-Type'] = 'application/json'
            else:
                request_kwargs['data'] = data

        # Execute request with retries
        for attempt in range(retry + 1):
            try:
                response = self.session.request(**request_kwargs)

                if response.status_code >= 200 and response.status_code < 300:
                    return self._handle_response(response)

                if response.status_code == 429 and attempt < retry:
                    # Rate limited, handled in _handle_response
                    result = self._handle_response(response)
                    if result is not None:
                        return result
                    continue

                # Handle other errors
                error_msg = f"API error: {response.status_code}"
                try:
                    error_data = response.json()
                    if 'message' in error_data:
                        error_msg = f"{error_msg} - {error_data['message']}"
                except:
                    if response.text:
                        error_msg = f"{error_msg} - {response.text[:100]}"

                if attempt < retry and response.status_code >= 500:
                    # Server error, retry after backoff
                    wait_time = 2 ** attempt
                    print(f"Server error, retrying in {wait_time} seconds...")
```

```
                time.sleep(wait_time)
                continue

            raise Exception(error_msg)

        except (requests.RequestException, ConnectionError) as e:
            if attempt < retry:
                wait_time = 2 ** attempt
                print(f"Connection error: {str(e)}, retrying in {wait_time} seconds...")
                time.sleep(wait_time)
                continue
            raise

    return None

# Async methods for high-performance API interaction
async def async_request_batch(self, requests_data, max_concurrent=5):
    """Execute multiple API requests asynchronously"""
    async def _single_request(session, req_data):
        method = req_data['method'].upper()
        endpoint = req_data['endpoint']
        params = req_data.get('params')
        data = req_data.get('data')
        req_id = req_data.get('id', endpoint)

        url = f"{self.base_url}/{endpoint.lstrip('/')}"

        # Prepare headers
        headers = req_data.get('headers', {})
        auth_headers = self._generate_signature(method, endpoint, params, data)
        if auth_headers:
            headers.update(auth_headers)

        try:
            if method == 'GET':
                async with session.get(url, params=params, headers=headers) as
response:
                    response_data = await response.json()
                    return {'id': req_id, 'success': True, 'data': response_data}
            elif method == 'POST':
                async with session.post(url, params=params, json=data,
headers=headers) as response:
                    response_data = await response.json()
                    return {'id': req_id, 'success': True, 'data': response_data}
            # Add other methods as needed
        except Exception as e:
            return {'id': req_id, 'success': False, 'error': str(e)}

    async with aiohttp.ClientSession() as session:
        # Create a semaphore to limit concurrent requests
        semaphore = asyncio.Semaphore(max_concurrent)
```

```
        async def _bounded_request(req_data):
            async with semaphore:
                return await _single_request(session, req_data)

        # Execute all requests with concurrency limit
        tasks = [_bounded_request(req) for req in requests_data]
        return await asyncio.gather(*tasks)
```

Sources

- Advanced Shell & Python Scripting: Automation Techniques in System Administration
- Tutorial: Parallel Programming with multiprocessing in Python
- Parallel execution of Python automation: methods and example
- Python automation: 9 scripts to automate critical workflows