Tips & Tricks

AI Basics

Python Optimization Cheat Sheet

```
## Fast Execution
from timeit import *
from cprofile import *
```

#Python

/sumitkhanna

Python

Follow me on

Sumit Khanna for more updates

Python Optimization Guide

Optimizing Python code for faster execution is crucial for developers to ensure their applications run efficiently. This guide provides comprehensive tips and examples to help you optimize your Python code, focusing on various functions and techniques that can enhance performance.

Cheat Sheet Table

Feature/Function	Brief Explanation
`timeit`	Measures the execution time of small code snippets
`cProfile`	Provides a detailed report on the time spent in each function
`lru_cache`	Caches the results of function calls to avoid redundant calculations
`multiprocessing`	Enables parallel execution of tasks using multiple processes
`numba`	Just-in-time compiler that translates Python functions to optimized machine code
`cython`	Optimizes Python code by compiling it into C extensions
`numpy`	Provides efficient array operations and mathematical functions
`pandas`	Optimizes data manipulation and analysis
`memory_profiler`	Measures memory usage of Python code
`psutil`	Provides utilities for system and process management
`asyncio`	Supports asynchronous programming for concurrent execution
`concurrent.futures`	High-level interface for asynchronously executing callables
`pickle`	Serializes and deserializes Python objects
`joblib`	Efficiently handles serialization and parallelism for large data
`jit`	Just-in-time compilation for performance enhancement
`vectorization`	Converts operations to work on entire arrays for efficiency

Feature/Function	Brief Explanation
`threading`	Enables concurrent execution using threads
`contextlib`	Provides utilities for common tasks in context management
`setdefault`	Optimizes dictionary operations
`itertools`	Efficiently handles iterators for looping constructs
`functools.reduce`	Applies a function cumulatively to the items of an iterable
`operator`	Provides efficient functions corresponding to Python operators
`bisect`	Maintains a list in sorted order efficiently
`collections.deque`	Provides a double-ended queue for efficient appends and pops
`heapq`	Implements a heap queue for efficient priority queue operations
`gc`	Provides an interface to the garbage collector
`blist`	Provides a list-like type with better performance for specific operations
`psutil`	Provides utilities for system and process management
`bottleneck`	Fast NumPy array functions
`sortedcontainers`	Efficiently maintains sorted collections
`cachetools`	Provides extensible memoizing collections and decorators
`aiohttp`	Asynchronous HTTP client/server for asyncio
`uvloop`	Ultra fast implementation of asyncio event loop
`aiomultiprocess`	Enables the use of multiprocessing with asyncio
`faulthandler`	Dumps Python tracebacks explicitly on a crash
`trace`	Tracks statement execution
`tracemalloc`	Traces memory allocations
`py-heat`	Visualizes code hotspots
`py-spy`	Sampling profiler for Python programs
`snakeviz`	Visualizes profile statistics
`line_profiler`	Profiles individual lines of Python code
`bottleneck`	Optimizes bottlenecks in numerical computations
`pytables`	Efficiently handles large datasets
`numexpr`	Fast numerical expression evaluator for NumPy

Feature/Function	Brief Explanation
`xarray`	N-D labeled arrays and datasets
`sparse`	Provides sparse matrices and operations
`zarr`	Chunked, compressed, N-dimensional arrays
`dask`	Parallel computing with task scheduling
`ray`	Scales Python applications with simple, flexible parallel and distributed execution

---

1. `timeit`

The `timeit` module is used to measure the execution time of small code snippets. It avoids some of the common pitfalls of measuring execution time with `time.time()`.

Parameters:

- `stmt`: The code statement to be timed (default: `'pass'`).
- `setup`: The setup code that will be executed once before the `stmt` (default: `'pass'`).
- `timer`: The timer function to use (default: `time.perf_counter`).

Return Value:

- Returns the execution time in seconds.

Usage:

```
import timeit

# Example to time the execution of a simple list comprehension
time_taken = timeit.timeit('sum([i for i in range(1000)])', number=1000)
print(f'Time taken: {time_taken}')
```

2. `cProfile`

The `cProfile` module provides a detailed report on the time spent in each function call in a Python program.

Parameters:

- `filename`: The name of the file where the profile results will be stored (default: `None`).

Return Value:

- Returns a `Profile` object containing the profiling results.

Usage:

```python
import cProfile

def example_function():
    sum([i for i in range(1000)])

# Profile the example_function
cProfile.run('example_function()')
```

3. `lru_cache`

The `lru_cache` decorator from the `functools` module caches the results of function calls to avoid redundant calculations, improving performance for functions with expensive or repetitive computations.

Parameters:

- `maxsize`: The maximum size of the cache (default: `128`). If `None`, the cache can grow without bound.
- `typed`: If `True`, arguments of different types will be cached separately (default: `False`).

Return Value:

- Returns a decorated function with caching capability.

Usage:

```python
from functools import lru_cache

@lru_cache(maxsize=100)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
```

4. `multiprocessing`

The `multiprocessing` module enables parallel execution of tasks using multiple processes, leveraging multiple CPU cores for improved performance.

Parameters:

- `target`: The target function to be executed by the process.
- `args`: The arguments to be passed to the target function.

Return Value:

- Returns a `Process` object representing the process.

Usage:

```python
from multiprocessing import Process

def print_numbers():
    for i in range(10):
        print(i)

# Create and start a new process
p = Process(target=print_numbers)
p.start()
p.join()  # Wait for the process to finish
```

5. `numba`

The `numba` module is a just-in-time (JIT) compiler that translates Python functions to optimized machine code at runtime using the LLVM compiler infrastructure.

Parameters:

- `signature`: Specifies the types of the input arguments and return value.

Return Value:

- Returns a compiled version of the function for faster execution.

Usage:

```python
from numba import jit

@jit
def sum_of_squares(n):
    return sum(i**2 for i in range(n))

print(sum_of_squares(10000))
```

6. `cython`

The `cython` module optimizes Python code by compiling it into C extensions, resulting in significant performance improvements for computationally intensive tasks.

Parameters:

- None (uses a special syntax to indicate types and compile-time optimizations).

Return Value:

- Returns a compiled version of the Python code.

Usage:

```
# Save this code in a file named example.pyx
def sum_of_squares(n):
    cdef int i
    cdef int result = 0
    for i in range(n):
        result += i * i
    return result

# Then compile it using: cythonize -i example.pyx
```

7. `numpy`

The `numpy` module provides efficient array operations and mathematical functions, offering significant performance improvements over standard Python lists.

Parameters:

- `array`: A multi-dimensional array of homogeneous data.

Return Value:

- Returns an ndarray object with optimized array operations.

Usage:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(np.sum(arr))
```

8. `pandas`

The `pandas` module optimizes data manipulation and analysis with efficient DataFrame operations.

Parameters:

- `data`: Data to be stored in the DataFrame.
- `columns`: Column names (optional).

Return Value:

- Returns a DataFrame object with optimized data manipulation methods.

Usage:

```
import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)
print(df.describe())
```

9. `memory_profiler`

The `memory_profiler` module measures the memory usage of Python code, helping identify memory leaks and optimize memory usage.

Parameters:

- `filename`: The name of the file where memory usage results will be stored (default: `None`).

Return Value:

- Returns a decorated function that tracks memory usage.

Usage:

```
from memory_profiler import profile

@profile
def example_function():
    a = [i for i in range(100000)]
    return a

example_function()
```

10. `psutil`

The `psutil` module provides utilities for system and process management, allowing monitoring and control of system resources.

Parameters:

- None (provides functions and classes for system and process management).

Return Value:

- Returns various system and process information.

Usage:

```
import psutil

# Get system memory information
memory_info = psutil.virtual_memory()
print(memory_info)
```

11. `asyncio`

The `asyncio` module supports asynchronous programming, allowing concurrent execution of tasks without the need for multi-threading or multiprocessing.

Parameters:

- None (provides functions and classes for asynchronous programming).

Return Value:

- Returns an event loop for managing asynchronous tasks.

Usage:

```
import asyncio

async def say_hello():
    await asyncio.sleep(1)
    print("Hello, world!")

# Run the async function
asyncio.run(say_hello())
```

12. `concurrent.futures`

The `concurrent.futures` module provides a high-level interface for asynchronously executing callables using threads or processes.

Parameters:

- `max_workers`: The maximum number of worker threads or processes (optional).

Return Value:

- Returns a `Future` object representing the execution of the callable.

Usage:

```python
from concurrent.futures import ThreadPoolExecutor

def print_number(number):
    print(number)

# Create a ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=2) as executor:
    executor.submit(print_number, 1)
    executor.submit(print_number, 2)
```

13. `pickle`

The `pickle` module serializes and deserializes Python objects, allowing them to be saved to a file or transmitted over a network.

Parameters:

- `obj`: The Python object to be serialized.

Return Value:

- Returns a byte stream representing the serialized object.

Usage:

```python
import pickle

data = {'key': 'value'}

# Serialize the data
with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)

# Deserialize the data
with open('data.pkl', 'rb') as file:
    loaded_data = pickle.load(file)

print(loaded_data)
```

14. `joblib`

The `joblib` module efficiently handles serialization and parallelism for large data, providing a faster alternative to `pickle`.

Parameters:

- `obj`: The Python object to be serialized.
- `n_jobs`: The number of parallel jobs to run (default: `None`).

Return Value:

- Returns a serialized byte stream or parallel execution results.

Usage:

```
from joblib import Parallel, delayed

def square(n):
    return n * n

# Parallel execution
results = Parallel(n_jobs=2)(delayed(square)(i) for i in range(10))
print(results)
```

15. `jit`

The `jit` (Just-In-Time) compilation technique, often used with libraries like `numba`, translates Python code into machine code at runtime for performance enhancement.

Parameters:

- `signature`: Specifies the types of the input arguments and return value (optional).

Return Value:

- Returns a compiled version of the function for faster execution.

Usage:

```
from numba import jit

@jit
def multiply(a, b):
    return a * b

print(multiply(10, 20))
```

16. `vectorization`

Vectorization converts operations to work on entire arrays instead of individual elements, significantly improving performance.

Parameters:

- `array`: The input array for the operation.

Return Value:

- Returns the result of the vectorized operation.

Usage:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr * 2)  # Vectorized multiplication
```

17. `threading`

The `threading` module enables concurrent execution using threads, allowing multiple tasks to run simultaneously.

Parameters:

- `target`: The target function to be executed by the thread.
- `args`: The arguments to be passed to the target function.

Return Value:

- Returns a `Thread` object representing the thread.

Usage:

```
import threading

def print_hello():
    print("Hello, world!")

# Create and start a new thread
t = threading.Thread(target=print_hello)
t.start()
t.join()  # Wait for the thread to finish
```

18. `contextlib`

The `contextlib` module provides utilities for common tasks in context management, simplifying the use of context managers.

Parameters:

- None (provides functions and classes for context management).

Return Value:

- Returns a context manager for the specified task.

Usage:

```python
from contextlib import contextmanager

@contextmanager
def open_file(name):
    file = open(name, 'w')
    yield file
    file.close()

with open_file('example.txt') as f:
    f.write('Hello, world!')
```

19. `setdefault`

The `setdefault` method optimizes dictionary operations by setting a default value if the key is not already present.

Parameters:

- `key`: The key to be checked in the dictionary.
- `default`: The default value to be set if the key is not present.

Return Value:

- Returns the value associated with the key.

Usage:

```python
data = {'a': 1}
value = data.setdefault('b', 2)
print(data)
```

20. `itertools`

The `itertools` module efficiently handles iterators for looping constructs, providing a set of fast, memory-efficient tools.

Parameters:

- Varies depending on the function used (e.g., `combinations`, `permutations`, `product`).

Return Value:

- Returns an iterator that generates the specified sequence.

Usage:

```python
import itertools

# Generate combinations of a list
combinations = itertools.combinations([1, 2, 3], 2)
print(list(combinations))
```

21. `functools.reduce`

The `functools.reduce` function applies a function cumulatively to the items of an iterable, reducing the iterable to a single value.

Parameters:

- `function`: The function to apply.
- `iterable`: The iterable to reduce.
- `initializer`: The initial value (optional).

Return Value:

- Returns the reduced value.

Usage:

```python
from functools import reduce

numbers = [1, 2, 3, 4]
result = reduce(lambda x, y: x + y, numbers)
print(result)
```

22. `operator`

The `operator` module provides efficient functions corresponding to Python operators, enabling concise and faster code.

Parameters:

- Varies depending on the operator function used (e.g., `add`, `mul`, `itemgetter`).

Return Value:

- Returns the result of the operator function.

Usage:

```python
import operator

result = operator.add(1, 2)
print(result)
```

23. `bisect`

The `bisect` module maintains a list in sorted order efficiently, providing functions for inserting and finding elements.

Parameters:

- `list`: The sorted list to insert/find elements.
- `item`: The item to insert/find.

Return Value:

- Returns the index of the item or the insertion point.

Usage:

```
import bisect

sorted_list = [1, 2, 4, 5]
bisect.insort(sorted_list, 3)
print(sorted_list)
```

24. `collections.deque`

The `collections.deque` module provides a double-ended queue for efficient appends and pops from both ends.

Parameters:

- `iterable`: An optional iterable to initialize the deque.

Return Value:

- Returns a `deque` object.

Usage:

```
from collections import deque

dq = deque([1, 2, 3])
dq.appendleft(0)
dq.append(4)
print(dq)
```

25. `heapq`

The `heapq` module implements a heap queue for efficient priority queue operations, providing functions for heap operations.

Parameters:

- `list`: The list to be heapified.
- `item`: The item to be pushed/popped.

Return Value:

- Returns the heapified list or the popped item.

Usage:

```
import heapq

heap = [3, 1, 4, 1, 5]
heapq.heapify(heap)
heapq.heappush(heap, 2)
print(heapq.heappop(heap))
```

26. `gc`

The `gc` module provides an interface to the garbage collector, allowing for manual garbage collection and inspection.

Parameters:

- None (provides functions for garbage collection control).

Return Value:

- Returns various garbage collection statistics and control.

Usage:

```
import gc

# Manually trigger garbage collection
gc.collect()
```

27. `blist`

The `blist` module provides a list-like type with better performance for specific operations, such as slicing and insertion.

Parameters:

- `iterable`: An optional iterable to initialize the blist.

Return Value:

- Returns a `blist` object.

Usage:

```
from blist import blist

bl = blist([1, 2, 3])
bl.insert(1, 1.5)
print(bl)
```

28. `psutil`

The `psutil` module provides utilities for system and process management, allowing monitoring and control of system resources.

Parameters:

- None (provides functions and classes for system and process management).

Return Value:

- Returns various system and process information.

Usage:

```
import psutil

# Get CPU usage information
cpu_info = psutil.cpu_percent(interval=1)
print(cpu_info)
```

29. `bottleneck`

The `bottleneck` module provides fast NumPy array functions, optimizing performance for numerical computations.

Parameters:

- `array`: The input array for the operation.

Return Value:

- Returns the result of the optimized operation.

Usage:

```
import bottleneck as bn
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(bn.nanmean(arr))
```

30. `sortedcontainers`

The `sortedcontainers` module efficiently maintains sorted collections, providing sorted list, dict, and set types.

Parameters:

- `iterable`: An optional iterable to initialize the sorted container.

Return Value:

- Returns a sorted container object.

Usage:

```
from sortedcontainers import SortedList

sl = SortedList([3, 1, 4, 1, 5])
sl.add(2)
print(sl)
```

31. `cachetools`

The `cachetools` module provides extensible memoizing collections and decorators, optimizing performance by caching results.

Parameters:

- `maxsize`: The maximum size of the cache.

Return Value:

- Returns a decorated function or a cache object.

Usage:

```python
from cachetools import cached, LRUCache

cache = LRUCache(maxsize=100)

@cached(cache)
def compute(x):
    return x * x

print(compute(2))
```

32. `aiohttp`

The `aiohttp` module provides an asynchronous HTTP client/server for asyncio, enabling efficient HTTP communication.

Parameters:

- `url`: The URL to send the request to.
- `method`: The HTTP method to use (default: `'GET'`).

Return Value:

- Returns a response object.

Usage:

```python
import aiohttp
import asyncio

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://python.org')
        print(html)

asyncio.run(main())
```

33. `uvloop`

The `uvloop` module provides an ultra-fast implementation of the asyncio event loop, significantly improving the performance of asynchronous applications.

Parameters:

- None (provides an event loop implementation).

Return Value:

- Returns an event loop object.

Usage:

```
import asyncio
import uvloop

asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())

async def say_hello():
    await asyncio.sleep(1)
    print("Hello, world!")

asyncio.run(say_hello())
```

34. `aiomultiprocess`

The `aiomultiprocess` module enables the use of multiprocessing with asyncio, allowing parallel execution of tasks in an asynchronous environment.

Parameters:

- `target`: The target function to be executed by the process.
- `args`: The arguments to be passed to the target function.

Return Value:

- Returns a `Process` object representing the process.

Usage:

```
import asyncio
from aiomultiprocess import Pool

async def square(x):
    return x * x

async def main():
    async with Pool() as pool:
        results = await pool.map(square, range(10))
        print(results)

asyncio.run(main())
```

35. `faulthandler`

The `faulthandler` module dumps Python tracebacks explicitly on a crash, helping identify the cause of crashes in Python programs.

Parameters:

- None (provides functions for fault handling).

Return Value:

- Returns None.

Usage:

```
import faulthandler

# Enable fault handler
faulthandler.enable()
```

36. `trace`

The `trace` module tracks statement execution, providing detailed information about which lines of code were executed.

Parameters:

- `count`: Whether to count the number of times each line is executed (default: `True`).
- `trace`: Whether to print a trace of each line executed (default: `True`).

Return Value:

- Returns a `Trace` object.

Usage:

```
import trace

tracer = trace.Trace(count=True, trace=True)
tracer.run('example_function()')
```

37. `tracemalloc`

The `tracemalloc` module traces memory allocations, helping identify memory usage patterns and leaks in Python programs.

Parameters:

- None (provides functions for memory tracing).

Return Value:

- Returns memory allocation statistics.

Usage:

```
import tracemalloc

# Start tracing memory allocations
tracemalloc.start()

# Your code here

# Get memory allocation statistics
snapshot = tracemalloc.take_snapshot()
print(snapshot.statistics('lineno'))
```

38. `py-heat`

The `py-heat` module visualizes code hotspots, helping identify the most time-consuming parts of the code for optimization.

Parameters:

- `filename`: The name of the file to analyze.

Return Value:

- Returns a heatmap visualization.

Usage:

```
import pyheat

# Generate a heatmap for the specified file
pyheat.create_heatmap('example.py')
```

39. `py-spy`

The `py-spy` module is a sampling profiler for Python programs, providing real-time profiling data without significant overhead.

Parameters:

- `pid`: The process ID of the Python program to profile.

Return Value:

- Returns profiling data.

Usage:

```
py-spy top --pid 12345
```

40. `snakeviz`

The `snakeviz` module visualizes profile statistics, providing an interactive browser-based interface for exploring profiling results.

Parameters:

- `filename`: The name of the file containing the profile results.

Return Value:

- Returns an interactive visualization.

Usage:

```
snakeviz profile_results.prof
```

41. `line_profiler`

The `line_profiler` module profiles individual lines of Python code, providing detailed information about execution time for each line.

Parameters:

- `func`: The function to be profiled.

Return Value:

- Returns a decorated function with line-by-line profiling.

Usage:

```
from line_profiler import LineProfiler

def example_function():
    sum([i for i in range(1000)])

profiler = LineProfiler()
profiler.add_function(example_function)
profiler.run('example_function()')
profiler.print_stats()
```

42. `bottleneck`

The `bottleneck` module optimizes bottlenecks in numerical computations, providing fast implementations of common NumPy functions.

Parameters:

- `array`: The input array for the operation.

Return Value:

- Returns the result of the optimized operation.

Usage:

```
import bottleneck as bn
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(bn.nanmean(arr))
```

43. `pytables`

The `pytables` module efficiently handles large datasets, providing a hierarchical database format optimized for fast access.

Parameters:

- `filename`: The name of the file to store the dataset.
- `mode`: The mode to open the file (default: `'a'`).

Return Value:

- Returns a `File` object representing the HDF5 file.

Usage:

```
import tables as tb

# Create a new HDF5 file
file = tb.open_file('example.h5', mode='w')
# Define a new table
table = file.create_table('/', 'example_table', {'value': tb.Int32Col()})
# Insert data into the table
table.row['value'] = 42
table.row.append()
file.close()
```

44. `numexpr`

The `numexpr` module provides a fast numerical expression evaluator for NumPy, optimizing performance for large arrays.

Parameters:

- `expression`: The numerical expression to evaluate.
- `local_dict`: A dictionary of local variables to use in the expression.

Return Value:

- Returns the result of the evaluated expression.

Usage:

```
import numexpr as ne
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
result = ne.evaluate('arr * 2 + 1')
print(result)
```

45. `xarray`

The `xarray` module provides N-D labeled arrays and datasets, optimizing performance for multi-dimensional data.

Parameters:

- `data`: The data to be stored in the xarray.
- `dims`: The dimension names (optional).
- `coords`: The coordinates for each dimension (optional).

Return Value:

- Returns a `DataArray` or `Dataset` object.

Usage:

```
import xarray as xr

data = xr.DataArray([[1, 2, 3], [4, 5, 6]], dims=['x', 'y'])
print(data)
```

46. `sparse`

The `sparse` module provides sparse matrices and operations, optimizing performance for large datasets with many zero elements.

Parameters:

- `data`: The data to be stored in the sparse matrix.
- `dims`: The dimensions of the matrix (optional).

Return Value:

- Returns a `COO` or `DOK` sparse matrix.

Usage:

```
import sparse

data = sparse.COO.from_numpy(np.array([[1, 0, 0], [0, 0, 2]]))
print(data)
```

47. `zarr`

The `zarr` module provides chunked, compressed, N-dimensional arrays, optimizing performance for large datasets.

Parameters:

- `shape`: The shape of the array.
- `chunks`: The chunk size (optional).

Return Value:

- Returns a `zarr` array object.

Usage:

```
import zarr

arr = zarr.zeros((1000, 1000), chunks=(100, 100))
print(arr)
```

48. `dask`

The `dask` module provides parallel computing with task scheduling, optimizing performance for large datasets and computations.

Parameters:

- `data`: The data to be processed.
- `chunks`: The chunk size for parallel processing.

Return Value:

- Returns a `dask` array, DataFrame, or Delayed object.

Usage:

```
import dask.array as da

arr = da.random.random((10000, 10000), chunks=(1000, 1000))
print(arr.mean().compute())
```

49. `ray`

The `ray` module scales Python applications with simple, flexible parallel and distributed execution, optimizing performance for complex workflows.

Parameters:

- `num_cpus`: The number of CPUs to use (optional).

Return Value:

- Returns a `ray` object for distributed computing.

Usage:

```
import ray

ray.init()

@ray.remote
def compute(x):
    return x * x

results = ray.get([compute.remote(i) for i in range(10)])
print(results)
```

50. `mlpack`

The `mlpack` module is a fast, flexible machine learning library, providing a wide range of machine learning algorithms optimized for performance.

Parameters:

- Varies depending on the algorithm used (e.g., `KMeans`, `LogisticRegression`).

Return Value:

- Returns the result of the machine learning algorithm.

Usage:

```
import mlpack

# Perform k-means clustering
result = mlpack.kmeans(input=dataset, clusters=3)
print(result)
```

Follow me on

Sumit Khanna for more updates