


# Complete Python Utility Methods Cheat Sheet

## What Are Utility Methods?

**Utility methods** are small, reusable helper functions that make your life easier! Think of them as **tools** in a **Swiss Army knife**  - each one has a specific purpose and you use them whenever needed.

### Key Characteristics:

- Usually `@staticmethod` (don't rely on object state)
  - Perform common, repetitive tasks
  - Help avoid code duplication
  - Make your code cleaner and more readable
- 

## 1. String Utilities

### Basic String Operations

```
python
```

```
@staticmethod
```

```
def to_camel_case(s: str) -> str:  
    """Convert snake_case to CamelCase"""  
    return ''.join(word.capitalize() for word in s.split('_'))  
    # "hello_world" → "HelloWorld"
```

```
@staticmethod
```

```
def to_snake_case(s: str) -> str:  
    """Convert CamelCase to snake_case"""  
    import re  
    return re.sub(r'(?<!^)(?=[A-Z])', '_', s).lower()  
    # "HelloWorld" → "hello_world"
```

```
@staticmethod
```

```
def to_kebab_case(s: str) -> str:  
    """Convert to kebab-case"""  
    import re  
    return re.sub(r'(?<!^)(?=[A-Z])', '-', s).lower().replace('_', '-')  
    # "HelloWorld" → "hello-world"
```

```
@staticmethod
```

```
def slugify(s: str) -> str:  
    """Convert string to URL-friendly slug"""  
    import re  
    s = s.lower().strip()  
    s = re.sub(r'^\w\s-', '', s)  
    s = re.sub(r'[-\s]+' , '-', s)  
    return s.strip('-')  
    # "Hello World!" → "hello-world"
```

## String Validation & Analysis

```
python
```

```
@staticmethod
```

```
def is_palindrome(s: str) -> bool:
```

```
    """Check if string is a palindrome"""
```

```
    clean = ''.join(c.lower() for c in s if c.isalnum())
```

```
    return clean == clean[::-1]
```

```
    # "A man a plan a canal Panama" → True
```

```
@staticmethod
```

```
def count_words(s: str) -> int:
```

```
    """Count words in string"""
```

```
    return len(s.split())
```

```
@staticmethod
```

```
def count_vowels(s: str) -> int:
```

```
    """Count vowels in string"""
```

```
    return sum(1 for c in s.lower() if c in 'aeiou')
```

```
@staticmethod
```

```
def reverse_words(s: str) -> str:
```

```
    """Reverse order of words"""
```

```
    return ' '.join(s.split()[::-1])
```

```
    # "Hello World" → "World Hello"
```

```
@staticmethod
```

```
def title_case(s: str) -> str:
```

```
    """Convert to proper title case"""
```

```
    return ' '.join(word.capitalize() for word in s.split())
```

```
    # "hello world" → "Hello World"
```

```
@staticmethod
```

```
def truncate(s: str, length: int, suffix: str = "...") -> str:
```

```
    """Truncate string with suffix"""
```

```
    if len(s) <= length:
```

```
        return s
```

```
    return s[:length - len(suffix)] + suffix
```

```
    # truncate("Hello World", 8) → "Hello..."
```

## Advanced String Processing

python

@staticmethod

```
def extract_numbers(s: str) -> list:
    """Extract all numbers from string"""
    import re
    return [float(x) if '.' in x else int(x) for x in re.findall(r'-?\d+\.?\d*', s)]
# "I have 5 apples and 3.5 oranges" → [5, 3.5]
```

@staticmethod

```
def extract_emails(s: str) -> list:
    """Extract email addresses from string"""
    import re
    pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
    return re.findall(pattern, s)
```

@staticmethod

```
def extract_urls(s: str) -> list:
    """Extract URLs from string"""
    import re
    pattern = r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[*\(\)\,\:]|(?%[0-9a-fA-F][0-9a-fA-F]))+'
    return re.findall(pattern, s)
```

@staticmethod

```
def mask_sensitive_data(s: str, mask_char: str = "*", visible_chars: int = 4) -> str:
    """Mask sensitive data showing only last few characters"""
    if len(s) <= visible_chars:
        return mask_char * len(s)
    return mask_char * (len(s) - visible_chars) + s[-visible_chars:]
# mask_sensitive_data("1234567890", visible_chars=4) → "*****7890"
```

## 2. Math & Number Utilities

### Basic Math Operations

python

```
@staticmethod
```

```
def clamp(num: float, low: float, high: float) -> float:
```

```
    """Force number to stay between two values"""
```

```
    return max(low, min(num, high))
```

```
    # clamp(15, 0, 10) → 10
```

```
@staticmethod
```

```
def is_prime(n: int) -> bool:
```

```
    """Check if number is prime"""
```

```
    if n < 2:
```

```
        return False
```

```
    for i in range(2, int(n ** 0.5) + 1):
```

```
        if n % i == 0:
```

```
            return False
```

```
    return True
```

```
@staticmethod
```

```
def factorial(n: int) -> int:
```

```
    """Calculate factorial"""
```

```
    if n <= 1:
```

```
        return 1
```

```
    result = 1
```

```
    for i in range(2, n + 1):
```

```
        result *= i
```

```
    return result
```

```
@staticmethod
```

```
def gcd(a: int, b: int) -> int:
```

```
    """Greatest Common Divisor"""
```

```
    while b:
```

```
        a, b = b, a % b
```

```
    return a
```

```
@staticmethod
```

```
def lcm(a: int, b: int) -> int:
```

```
    """Least Common Multiple"""
```

```
    return abs(a * b) // gcd(a, b)
```

## Number Conversions & Formatting

```
python
```

@staticmethod

```
def to_binary(n: int) -> str:
    """Convert integer to binary string"""
    return bin(n)[2:] # Remove '0b' prefix
```

@staticmethod

```
def to_hex(n: int) -> str:
    """Convert integer to hexadecimal string"""
    return hex(n)[2:] # Remove '0x' prefix
```

@staticmethod

```
def from_binary(binary_str: str) -> int:
    """Convert binary string to integer"""
    return int(binary_str, 2)
```

@staticmethod

```
def format_number(n: float, decimals: int = 2) -> str:
    """Format number with thousands separator"""
    return f"{n:,.{decimals}f}"
# format_number(1234567.89) -> "1,234,567.89"
```

@staticmethod

```
def bytes_to_human(bytes_num: int) -> str:
    """Convert bytes to human readable format"""
    for unit in ['B', 'KB', 'MB', 'GB', 'TB']:
        if bytes_num < 1024:
            return f"{bytes_num:.1f} {unit}"
        bytes_num /= 1024
    return f"{bytes_num:.1f} PB"
# bytes_to_human(1048576) -> "1.0 MB"
```

@staticmethod

```
def percentage(part: float, whole: float) -> float:
    """Calculate percentage"""
    return (part / whole) * 100 if whole != 0 else 0
# percentage(25, 100) -> 25.0
```

## Statistical Operations

python

```
@staticmethod
def average(numbers: list) -> float:
    """Calculate average of numbers"""
    return sum(numbers) / len(numbers) if numbers else 0

@staticmethod
def median(numbers: list) -> float:
    """Calculate median of numbers"""
    sorted_nums = sorted(numbers)
    n = len(sorted_nums)
    if n % 2 == 0:
        return (sorted_nums[n//2 - 1] + sorted_nums[n//2]) / 2
    return sorted_nums[n//2]

@staticmethod
def mode(numbers: list):
    """Find most common number"""
    from collections import Counter
    count = Counter(numbers)
    return count.most_common(1)[0][0] if count else None

@staticmethod
def range_normalize(numbers: list) -> list:
    """Normalize numbers to 0-1 range"""
    min_val, max_val = min(numbers), max(numbers)
    if min_val == max_val:
        return [0] * len(numbers)
    return [(x - min_val) / (max_val - min_val) for x in numbers]
```

### 3. 📁 File & Path Utilities

#### File Operations

```
python
```

```
@staticmethod
```

```
def read_file(path: str, encoding: str = 'utf-8') -> str:
```

```
    """Read entire file content"""
```

```
    try:
```

```
        with open(path, 'r', encoding=encoding) as f:
```

```
            return f.read()
```

```
    except FileNotFoundError:
```

```
        return ""
```

```
@staticmethod
```

```
def write_file(path: str, content: str, encoding: str = 'utf-8') -> bool:
```

```
    """Write content to file"""
```

```
    try:
```

```
        with open(path, 'w', encoding=encoding) as f:
```

```
            f.write(content)
```

```
        return True
```

```
    except Exception:
```

```
        return False
```

```
@staticmethod
```

```
def append_file(path: str, content: str, encoding: str = 'utf-8') -> bool:
```

```
    """Append content to file"""
```

```
    try:
```

```
        with open(path, 'a', encoding=encoding) as f:
```

```
            f.write(content)
```

```
        return True
```

```
    except Exception:
```

```
        return False
```

```
@staticmethod
```

```
def file_exists(path: str) -> bool:
```

```
    """Check if file exists"""
```

```
    from pathlib import Path
```

```
    return Path(path).exists()
```

```
@staticmethod
```

```
def get_file_size(path: str) -> int:
```

```
    """Get file size in bytes"""
```

```
    from pathlib import Path
```

```
    return Path(path).stat().st_size if Path(path).exists() else 0
```

```
@staticmethod
```

```
def get_file_extension(path: str) -> str:
```



```
"""Get file extension"""
```

```
from pathlib import Path
```

```
return Path(path).suffix.lower()
```

```
# get_file_extension("document.pdf") → ".pdf"
```

## Advanced File Operations

```
python
```

```
@staticmethod
```

```
def copy_file(src: str, dst: str) -> bool:
```

```
    """Copy file from source to destination"""
```

```
    import shutil
```

```
    try:
```

```
        shutil.copy2(src, dst)
```

```
        return True
```

```
    except Exception:
```

```
        return False
```

```
@staticmethod
```

```
def move_file(src: str, dst: str) -> bool:
```

```
    """Move file from source to destination"""
```

```
    import shutil
```

```
    try:
```

```
        shutil.move(src, dst)
```

```
        return True
```

```
    except Exception:
```

```
        return False
```

```
@staticmethod
```

```
def delete_file(path: str) -> bool:
```

```
    """Delete file"""
```

```
    import os
```

```
    try:
```

```
        os.remove(path)
```

```
        return True
```

```
    except Exception:
```

```
        return False
```

```
@staticmethod
```

```
def create_directory(path: str) -> bool:
```

```
    """Create directory if it doesn't exist"""
```

```
    from pathlib import Path
```

```
    try:
```

```
        Path(path).mkdir(parents=True, exist_ok=True)
```

```
        return True
```

```
    except Exception:
```

```
        return False
```

```
@staticmethod
```

```
def list_files(directory: str, extension: str = None) -> list:
```

```
    """List files in directory, optionally filter by extension"""
```

```
from pathlib import Path
path = Path(directory)
if not path.exists():
    return []

if extension:
    return [f.name for f in path.iterdir() if f.is_file() and f.suffix.lower() == extension.lower()]
return [f.name for f in path.iterdir() if f.is_file()]

@staticmethod
def get_directory_size(path: str) -> int:
    """Get total size of directory in bytes"""
    from pathlib import Path
    total = 0
    for file_path in Path(path).rglob('*'):
        if file_path.is_file():
            total += file_path.stat().st_size
    return total
```

---

## 4. Date & Time Utilities

### Basic Date Operations

```
python
```

@staticmethod

```
def now_str(format_str: str = "%Y-%m-%d %H:%M:%S") -> str:
    """Get current timestamp as formatted string"""
    from datetime import datetime
    return datetime.now().strftime(format_str)
```

@staticmethod

```
def days_between(date1: str, date2: str, date_format: str = "%Y-%m-%d") -> int:
    """Calculate days between two dates"""
    from datetime import datetime
    d1 = datetime.strptime(date1, date_format)
    d2 = datetime.strptime(date2, date_format)
    return abs((d2 - d1).days)
```

@staticmethod

```
def add_days(date_str: str, days: int, date_format: str = "%Y-%m-%d") -> str:
    """Add days to a date"""
    from datetime import datetime, timedelta
    date_obj = datetime.strptime(date_str, date_format)
    new_date = date_obj + timedelta(days=days)
    return new_date.strftime(date_format)
```

@staticmethod

```
def is_weekend(date_str: str, date_format: str = "%Y-%m-%d") -> bool:
    """Check if date is weekend (Saturday or Sunday)"""
    from datetime import datetime
    date_obj = datetime.strptime(date_str, date_format)
    return date_obj.weekday() >= 5 # 5=Saturday, 6=Sunday
```

@staticmethod

```
def get_age(birth_date: str, date_format: str = "%Y-%m-%d") -> int:
    """Calculate age from birth date"""
    from datetime import datetime
    birth = datetime.strptime(birth_date, date_format)
    today = datetime.now()
    return today.year - birth.year - ((today.month, today.day) < (birth.month, birth.day))
```

## Advanced Date Operations

python

@staticmethod

```
def to_utc(local_time: str, timezone: str, time_format: str = "%Y-%m-%d %H:%M:%S") -> str:
    """Convert local time to UTC"""
    from datetime import datetime
    import pytz # pip install pytz

    local_tz = pytz.timezone(timezone)
    local_dt = datetime.strptime(local_time, time_format)
    local_dt = local_tz.localize(local_dt)
    utc_dt = local_dt.astimezone(pytz.UTC)
    return utc_dt.strftime(time_format)
```

@staticmethod

```
def time_ago(timestamp: str, time_format: str = "%Y-%m-%d %H:%M:%S") -> str:
    """Get human readable time ago string"""
    from datetime import datetime

    past = datetime.strptime(timestamp, time_format)
    now = datetime.now()
    diff = now - past

    if diff.days > 0:
        return f"{diff.days} day{'s' if diff.days != 1 else ''} ago"
    elif diff.seconds > 3600:
        hours = diff.seconds // 3600
        return f"{hours} hour{'s' if hours != 1 else ''} ago"
    elif diff.seconds > 60:
        minutes = diff.seconds // 60
        return f"{minutes} minute{'s' if minutes != 1 else ''} ago"
    else:
        return "Just now"
```

@staticmethod

```
def get_quarter(date_str: str, date_format: str = "%Y-%m-%d") -> int:
    """Get quarter of year (1-4) for given date"""
    from datetime import datetime

    date_obj = datetime.strptime(date_str, date_format)
    return (date_obj.month - 1) // 3 + 1
```

@staticmethod

```
def is_leap_year(year: int) -> bool:
    """Check if year is leap year"""
    return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
```

```
@staticmethod
```

```
def get_week_number(date_str: str, date_format: str = "%Y-%m-%d") -> int:
```

```
    """Get ISO week number for date"""
```

```
    from datetime import datetime
```

```
    date_obj = datetime.strptime(date_str, date_format)
```

```
    return date_obj.isocalendar()[1]
```

## 5. Collection Utilities (Lists, Dicts, Sets)

### List Operations

```
python
```

@staticmethod

```
def flatten_list(nested_list: list) -> list:
    """Flatten nested list structure"""
    result = []
    for item in nested_list:
        if isinstance(item, list):
            result.extend(flatten_list(item))
        else:
            result.append(item)
    return result
# flatten_list([[1,2],[3,[4,5]]]) → [1,2,3,4,5]
```

@staticmethod

```
def chunk_list(lst: list, size: int) -> list:
    """Split list into chunks of specified size"""
    return [lst[i:i + size] for i in range(0, len(lst), size)]
# chunk_list([1,2,3,4,5,6], 2) → [[1,2],[3,4],[5,6]]
```

@staticmethod

```
def unique(seq: list) -> list:
    """Remove duplicates while preserving order"""
    seen = set()
    return [x for x in seq if not (x in seen or seen.add(x))]
# unique([1,2,2,3,1,4]) → [1,2,3,4]
```

@staticmethod

```
def group_by(lst: list, key_func) -> dict:
    """Group list items by key function"""
    from collections import defaultdict
    groups = defaultdict(list)
    for item in lst:
        groups[key_func(item)].append(item)
    return dict(groups)
# group_by(['apple','banana','apricot'], lambda x: x[0]) → {'a': ['apple','apricot'], 'b': ['banana']}
```

@staticmethod

```
def find_duplicates(lst: list) -> list:
    """Find duplicate items in list"""
    from collections import Counter
    counts = Counter(lst)
    return [item for item, count in counts.items() if count > 1]
```

@staticmethod

```
def rotate_list(lst: list, n: int) -> list:
    """Rotate list by n positions"""
    if not lst:
        return lst
    n = n % len(lst)
    return lst[n:] + lst[:n]
# rotate_list([1,2,3,4,5], 2) → [3,4,5,1,2]
```

## Dictionary Operations

python



@staticmethod

```
def safe_get(d: dict, key: str, default=None):  
    """Safely get nested dictionary value using dot notation"""  
    keys = key.split('.')  
    value = d  
    for k in keys:  
        if isinstance(value, dict) and k in value:  
            value = value[k]  
        else:  
            return default  
    return value  
# safe_get({'a': {'b': {'c': 5}}}, 'a.b.c') → 5
```

@staticmethod

```
def merge_dicts(*dicts) -> dict:  
    """Merge multiple dictionaries"""  
    result = {}  
    for d in dicts:  
        result.update(d)  
    return result
```

@staticmethod

```
def invert_dict(d: dict) -> dict:  
    """Invert dictionary (keys become values, values become keys)"""  
    return {v: k for k, v in d.items()}
```

@staticmethod

```
def filter_dict(d: dict, condition) -> dict:  
    """Filter dictionary by condition function"""  
    return {k: v for k, v in d.items() if condition(k, v)}  
# filter_dict({'a': 1, 'b': 2, 'c': 3}, lambda k, v: v > 1) → {'b': 2, 'c': 3}
```

@staticmethod

```
def dict_to_object(d: dict):  
    """Convert dictionary to object with dot notation access"""  
    class DictObj:  
        def __init__(self, dictionary):  
            for key, value in dictionary.items():  
                if isinstance(value, dict):  
                    setattr(self, key, DictObj(value))  
                else:  
                    setattr(self, key, value)  
    return DictObj(d)
```

```
@staticmethod
def flatten_dict(d: dict, separator: str = '.') -> dict:
    """Flatten nested dictionary"""
    def _flatten(obj, parent_key=""):
        items = []
        for k, v in obj.items():
            new_key = f"{parent_key}{separator}{k}" if parent_key else k
            if isinstance(v, dict):
                items.extend(_flatten(v, new_key).items())
            else:
                items.append((new_key, v))
        return dict(items)
    return _flatten(d)
# flatten_dict({'a': {'b': 1, 'c': 2}}) -> {'a.b': 1, 'a.c': 2}
```

## 6. Validation Utilities

### Email & URL Validation

python

```
@staticmethod
```

```
def is_email(email: str) -> bool:
```

```
    """Validate email format"""
```

```
    import re
```

```
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
```

```
    return re.match(pattern, email) is not None
```

```
@staticmethod
```

```
def is_url(url: str) -> bool:
```

```
    """Validate URL format"""
```

```
    import re
```

```
    pattern = r'^https?://(?:[-\w.])+(?:\:[0-9]+)?(?:/(?:[\w/_]*\?(?:\:[\w&=%.])*)?(?:\#(?:[\w.])*)?)?$'
```

```
    return re.match(pattern, url) is not None
```

```
@staticmethod
```

```
def is_phone_number(phone: str) -> bool:
```

```
    """Validate phone number (basic US format)"""
```

```
    import re
```

```
    pattern = r'^\+?1?[-\s]?(\([0-9]{3}\)|[-\s]?([0-9]{3})[-\s]?([0-9]{4}))$'
```

```
    return re.match(pattern, phone) is not None
```

```
@staticmethod
```

```
def is_ip_address(ip: str) -> bool:
```

```
    """Validate IP address (IPv4)"""
```

```
    parts = ip.split('.')
```

```
    if len(parts) != 4:
```

```
        return False
```

```
    try:
```

```
        return all(0 <= int(part) <= 255 for part in parts)
```

```
    except ValueError:
```

```
        return False
```

## Data Type Validation

```
python
```

```
@staticmethod
```

```
def is_number(s: str) -> bool:
```

```
    """Check if string can be converted to number"""
```

```
    try:
```

```
        float(s)
```

```
        return True
```

```
    except ValueError:
```

```
        return False
```

```
@staticmethod
```

```
def is_integer(s: str) -> bool:
```

```
    """Check if string is valid integer"""
```

```
    try:
```

```
        int(s)
```

```
        return True
```

```
    except ValueError:
```

```
        return False
```

```
@staticmethod
```

```
def is_positive_number(s: str) -> bool:
```

```
    """Check if string is positive number"""
```

```
    try:
```

```
        return float(s) > 0
```

```
    except ValueError:
```

```
        return False
```

```
@staticmethod
```

```
def is_json(s: str) -> bool:
```

```
    """Check if string is valid JSON"""
```

```
    import json
```

```
    try:
```

```
        json.loads(s)
```

```
        return True
```

```
    except ValueError:
```

```
        return False
```

```
@staticmethod
```

```
def is_date(s: str, date_format: str = "%Y-%m-%d") -> bool:
```

```
    """Check if string is valid date"""
```

```
    from datetime import datetime
```

```
    try:
```

```
        datetime.strptime(s, date_format)
```

```
        return True
```

```
except ValueError:
```

```
    return False
```

```
@staticmethod
```

```
def is_strong_password(password: str) -> tuple:
```

```
    """Validate password strength (returns bool, message)"""
```

```
    if len(password) < 8:
```

```
        return False, "Password must be at least 8 characters"
```

```
    has_upper = any(c.isupper() for c in password)
```

```
    has_lower = any(c.islower() for c in password)
```

```
    has_digit = any(c.isdigit() for c in password)
```

```
    has_special = any(c in '!@#$$%^&*()_+==[]{}|;,:.<.>?' for c in password)
```

```
    if not all([has_upper, has_lower, has_digit, has_special]):
```

```
        return False, "Password must contain uppercase, lowercase, digit, and special character"
```

```
    return True, "Password is strong"
```

---

## 7. Function & Performance Utilities

### Decorators & Higher-Order Functions

```
python
```

@staticmethod

```
def retry(func, max_attempts: int = 3, delay: float = 1):
```

```
    """Retry function on failure"""
```

```
    import time
```

```
    def wrapper(*args, **kwargs):
```

```
        for attempt in range(max_attempts):
```

```
            try:
```

```
                return func(*args, **kwargs)
```

```
            except Exception as e:
```

```
                if attempt == max_attempts - 1:
```

```
                    raise e
```

```
                time.sleep(delay)
```

```
        return None
```

```
    return wrapper
```

@staticmethod

```
def memoize(func):
```

```
    """Cache function results"""
```

```
    cache = {}
```

```
    def wrapper(*args, **kwargs):
```

```
        key = str(args) + str(sorted(kwargs.items()))
```

```
        if key not in cache:
```

```
            cache[key] = func(*args, **kwargs)
```

```
        return cache[key]
```

```
    return wrapper
```

@staticmethod

```
def timing_decorator(func):
```

```
    """Measure function execution time"""
```

```
    import time
```

```
    def wrapper(*args, **kwargs):
```

```
        start = time.time()
```

```
        result = func(*args, **kwargs)
```

```
        end = time.time()
```

```
        print(f"{func.__name__} took {end - start:.4f} seconds")
```

```
        return result
```

```
    return wrapper
```

@staticmethod

```
def rate_limit(calls_per_second: float):
```

```
    """Rate limit function calls"""
```

```
    import time
```

```
    min_interval = 1.0 / calls_per_second
```

```
last_called = [0.0]
```

```
def decorator(func):  
    def wrapper(*args, **kwargs):  
        elapsed = time.time() - last_called[0]  
        left_to_wait = min_interval - elapsed  
        if left_to_wait > 0:  
            time.sleep(left_to_wait)  
        ret = func(*args, **kwargs)  
        last_called[0] = time.time()  
        return ret  
    return wrapper  
return decorator
```

## Performance Monitoring

python

@staticmethod

def measure\_memory\_usage(func):

"""Measure memory usage of function"""

import tracemalloc

def wrapper(\*args, \*\*kwargs):

tracemalloc.start()

result = func(\*args, \*\*kwargs)

current, peak = tracemalloc.get\_traced\_memory()

tracemalloc.stop()

print(f"Memory usage: Current={current/1024/1024:.2f}MB, Peak={peak/1024/1024:.2f}MB")

return result

return wrapper

@staticmethod

def profile\_function(func):

"""Profile function performance"""

import cProfile

import pstats

def wrapper(\*args, \*\*kwargs):

profiler = cProfile.Profile()

profiler.enable()

result = func(\*args, \*\*kwargs)

profiler.disable()

stats = pstats.Stats(profiler)

stats.sort\_stats('cumulative')

stats.print\_stats(10) # Top 10 functions

return result

return wrapper

@staticmethod

def benchmark(func, iterations: int = 1000):

"""Benchmark function performance"""

import time

times = []

for \_ in range(iterations):

start = time.perf\_counter()

func()

end = time.perf\_counter()

times.append(end - start)

avg\_time = sum(times) / len(times)

min\_time = min(times)

max\_time = max(times)



```
return {  
    'avg_time': avg_time,  
    'min_time': min_time,  
    'max_time': max_time,  
    'total_time': sum(times),  
    'iterations': iterations  
}
```

---

## 8. Network & API Utilities

### HTTP Requests

python

```
@staticmethod
```

```
def make_request(url: str, method: str = 'GET', headers: dict = None, data: dict = None, timeout: int = 30):
```

```
    """Make HTTP request with error handling"""
```

```
    import requests
```

```
    try:
```

```
        response = requests.request(
```

```
            method=method.upper(),
```

```
            url=url,
```

```
            headers=headers or {},
```

```
            json=data,
```

```
            timeout=timeout
```

```
        )
```

```
        response.raise_for_status()
```

```
        return {
```

```
            'success': True,
```

```
            'data': response.json() if response.content else None,
```

```
            'status_code': response.status_code
```

```
        }
```

```
    except requests.RequestException as e:
```

```
        return {
```

```
            'success': False,
```

```
            'error': str(e),
```

```
            'status_code': getattr(e.response, 'status_code', None)
```

```
        }
```

```
@staticmethod
```

```
def download_file(url: str, filename: str) -> bool:
```

```
    """Download file from URL"""
```

```
    import requests
```

```
    try:
```

```
        response = requests.get(url, stream=True)
```

```
        response.raise_for_status()
```

```
        with open(filename, 'wb') as f:
```

```
            for chunk in response.iter_content(chunk_size=8192):
```

```
                f.write(chunk)
```

```
        return True
```

```
    except Exception:
```

```
        return False
```

```
@staticmethod
```

```
def get_public_ip() -> str:
```

```
    """Get public IP address"""
```

```
    import requests
```

```

try:
    response = requests.get('https://api.ipify.org?format=json', timeout=5)
    return response.json()['ip']
except Exception:
    return "Unknown"

```

```

@staticmethod
def ping_host(host: str, timeout: int = 5) -> bool:
    """Check if host is reachable"""
    import socket
    try:
        socket.create_connection((host, 80), timeout)
        return True
    except OSError:
        return False

```

## 9. Security & Hashing Utilities

### Password & Hashing

python

```

@staticmethod
def hash_password(password: str) -> str:
    """Hash password using bcrypt"""
    import bcrypt
    return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')

@staticmethod
def verify_password(password: str, hashed: str) -> bool:
    """Verify password against hash"""
    import bcrypt
    return bcrypt.checkpw(password.encode('utf-8'), hashed.encode('utf-8'))

@staticmethod
def generate_random_string(length: int = 32, include_symbols: bool = True) -> str:
    """Generate cryptographically secure random string"""
    import secrets

```