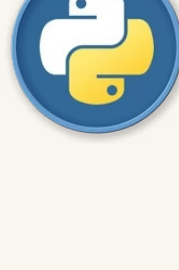


Python Cheatsheet



Essential syntax, structures, and patterns for modern Python development

This cheatsheet provides a quick reference to fundamental Python concepts, syntax, and advanced features, ideal for both beginners and experienced developers.

Core Syntax Operators, data types, variables	Functions & Control Definitions, parameters, flow control	Data Structures Lists, dicts, sets, tuples
Advanced Patterns OOP, decorators, comprehensions	File & Package Management I/O operations, virtual environments	

Basics: Operators & Data Types

Operator Precedence

Operators determine how values are combined. Precedence dictates the order of operations.

```
** > % // * > - +  
  
2 + 3 * 6 # 20 (multiplication before addition)  
(2 + 3) * 6 # 30 (parentheses override precedence)  
2 ** 8 # 256 (exponentiation)
```

Core Data Types

Python's fundamental data types define the kind of values variables can hold.

```
age = 25 # int (integer, whole number)  
price = 19.99 # float (floating-point number, decimal)  
name = "Alice" # str (string, sequence of characters)  
is_student = True # bool (boolean, True or False)  
scores = [85, 92, 78] # list (ordered, mutable collection)  
person = {'name': 'Bob'} # dict (dictionary, key-value pairs)
```

Functions: Definition & Lambda

Functions allow you to encapsulate reusable blocks of code.

```
# Define a function to print a greeting  
def say_hello(name):  
    print(f'Hello {name}')  
  
say_hello('Carlos') # Call the function with an argument  
  
# Functions can return values  
def sum_two_numbers(a, b):  
    return a + b # Returns the sum  
  
# Lambda functions are small, anonymous functions for simple expressions.  
add = lambda x, y: x + y # Defines a lambda function that takes x, y and returns their sum  
add(5, 3) # 8
```

Lists & Tuples

Lists (Mutable)

Lists are ordered collections of items that can be changed after creation.

```
furniture = ['table', 'chair']  
furniture[0] # 'table' (access item by index)  
furniture[1:3] # slice (get a sub-list)  
furniture.append('bed') # Add item to end  
furniture.remove('chair') # Remove specific item  
furniture.sort() # Sort items in place
```

Tuples (Immutable)

Tuples are ordered collections similar to lists, but they cannot be changed after creation.

```
coords = (10, 20)  
coords[0] # 10 (access item by index)  
  
# Cannot modify once created  
# coords[0] = 15 # Error! (attempting to change an immutable tuple)
```

Dictionaries: Key-Value Pairs

Dictionaries store data in unordered key-value pairs, allowing efficient lookup by key.

```
my_cat = {  
    'size': 'fat',  
    'color': 'gray',  
    'disposition': 'loud'  
}  
  
# Add a new key-value pair or modify an existing one  
my_cat['age_years'] = 2  
  
# Iterate over key-value pairs in the dictionary  
for key, value in my_cat.items():  
    print(f'{key}: {value}')  
  
# Safe retrieval: Use .get() to avoid KeyError if the key doesn't exist, providing a default value instead.  
my_cat.get('breed', 'unknown')
```

Sets: Unique Collections

Sets are unordered collections of unique items, useful for membership testing and eliminating duplicates.

Create Sets

Sets can be created from a list or by directly listing elements.

```
s = {1, 2, 3} # Directly create a set  
s = set([1, 2, 3]) # Convert a list to a set
```

Set Operations

Perform mathematical set operations like union, intersection, and difference.

```
s1 = {1, 2, 3}  
s2 = {3, 4, 5}  
  
s1.union(s2) # {1,2,3,4,5} (all unique elements from both sets)  
s1.intersection(s2) # {3} (elements common to both sets)  
s1.difference(s2) # {1,2} (elements in s1 but not in s2)
```

Control Flow

Control flow statements determine the order in which code instructions are executed.

1	2	3
Conditionals Use 'if', 'elif' (else if), and 'else' for decision-making logic. <pre>if name == 'Debora': print('Hi!') # Executes if name is 'Debora' elif name == 'George': print('Hello!') # Executes if name is 'George' else: print('Who?') # Executes if neither of the above conditions are met</pre>	For Loops Iterate over a sequence (like a list, tuple, or string) or other iterable objects. <pre>pets = ['Bella', 'Milo'] for pet in pets: # Loop through each item in the 'pets' list print(pet)</pre>	While Loops Repeats a block of code as long as a specified condition is true. <pre>count = 0 while count < 5: # Loop continues while count is less than 5 print('Hello') count += 1 # Increment count to eventually stop the loop</pre>

Comprehensions: Elegant Iteration

Comprehensions offer a concise way to create lists, sets, or dictionaries based on existing iterables.

```
# List comprehension: Creates a new list from an existing one, optionally with filtering.  
names = ['Charles', 'Susan', 'Patrick']  
new_list = [n for n in names if n.startswith('C')] # Creates ['Charles']  
  
# Set comprehension: Creates a new set, ensuring unique elements.  
s_upper = {s for s in ["abc", "def"]} # {"ABC", "DEF"} (converts strings to uppercase)  
  
# Dictionary comprehension: Creates a new dictionary from an iterable, often transforming key-value pairs.  
c = {'name': 'Pooka', 'age': 5}  
{v: k for k, v in c.items()} # Swaps keys and values: {'Pooka': 'name', 5: 'age'}
```

String Formatting: Modern Approach

F-Strings (Python 3.6+)

F-strings (formatted string literals) provide a readable and efficient way to embed expressions inside string literals.

```
name = 'Elizabeth'  
f'Hello {name}!' # Embeds variable directly  
  
a = 5  
b = 10  
f'Sum is {a + b}' # Embeds an expression
```

Format Numbers

F-strings also allow for flexible number formatting.

```
amount = 10000000  
f'{amount:,.2f}' # '10,000,000' (add comma thousands separator)  
  
pi = 3.1415926  
f'{pi:.2f}' # '3.14' (format to two decimal places)
```

String Manipulation

Python offers various methods to work with and modify strings.

```
# Slicing: Extracts parts of a string using start, end, and step indices.  
spam = 'Hello world!'  
spam[0:5] # 'Hello' (characters from index 0 up to (but not including) 5)  
spam[::-1] # '!dlrow olleH' (reverses the string)  
  
# String Methods: Common operations like changing case.  
greet = 'Hello world!'  
greet.upper() # 'HELLO WORLD!' (converts to uppercase)  
greet.lower() # 'hello world!' (converts to lowercase)  
greet.title() # 'Hello World!' (capitalizes the first letter of each word)  
  
# Join & Split: Combine or break apart strings based on delimiters.  
, '.join(['cats', 'rats']) # 'cats, rats' (joins elements of a list with a comma and space)  
'My name is Simon'.split() # ['My', 'name', 'is', 'Simon'] (splits string into a list of words)
```

Regular Expressions

Regular expressions (regex) are powerful tools for pattern matching and text parsing.

```
import re  
  
# Compile pattern for better performance if used multiple times  
phone_regex = re.compile(r'\d\d\d\d-\d\d\d\d-\d\d\d\d\d\d')  
mo = phone_regex.search('My number is 415-555-4242.') # Search for the pattern in a string  
mo.group() # '415-555-4242' (returns the matched string)  
  
# Groups: Use parentheses in the regex to capture specific parts of the match.  
phone_regex = re.compile(r'(\d\d\d\d)-(\d\d\d\d-\d\d\d\d\d\d)')  
mo.group(1) # '415' (first group)  
mo.group(2) # '555-4242' (second group)  
  
# Find all: Returns a list of all non-overlapping matches.  
phone_regex.findall('Cell: 415-555-9999 Work: 212-555-0000') # Returns a list of tuples for grouped matches
```

Exception Handling

Handle runtime errors gracefully to prevent program crashes and provide meaningful feedback.

Try-Except

Use 'try', 'except', and 'finally' blocks to catch and manage exceptions.

```
try:  
    result = 10 / 0 # This will raise a ZeroDivisionError  
except ZeroDivisionError: # Catch specific error type  
    print('Cannot divide by 0')  
finally: # Always executes, regardless of exception  
    print('Cleanup')
```

Custom Exceptions

Define your own exception classes for specific error conditions in your application.

```
class MyException(Exception): # Inherit from base  
    Exception class  
    pass  
  
raise MyException('Custom error') # Raise an instance of your custom exception
```

File Operations

Interact with the file system to read from or write to files.

```
# Read file: Open a file for reading and get its content. The 'with' statement ensures the file is closed automatically.  
with open('file.txt') as f:  
    content = f.read()  
  
# Read line by line: Iterate over each line in a file.  
with open('file.txt') as f:  
    for line in f:  
        print(line, end='') # 'end=' prevents extra newlines  
  
# Write file: Open a file for writing. If the file exists, its content is truncated (deleted).  
with open('output.txt', 'w') as f:  
    f.write('Hello world!\n')  
  
# Append: Open a file for appending. New content is added to the end of the file.  
with open('output.txt', 'a') as f:  
    f.write('Additional line')
```

Path Operations

The 'pathlib' module offers an object-oriented approach to filesystem paths, making operations cleaner and more robust.

```
from pathlib import Path  
  
# Path joining: Concatenate path components securely, handling separators automatically.  
print(Path('usr') / 'bin' / 'spam')  
  
# Current directory: Get the path to the current working directory.  
Path.cwd()  
  
# Create directories: Make new directories, optionally creating parent directories if they don't exist.  
(Path.cwd() / 'new' / 'folder').mkdir(parents=True)  
  
# Check existence: Verify if a path exists and if it's a file or directory.  
Path('file.py').exists() # True if file exists  
Path('file.py').is_file() # True if it's a file  
Path('V').is_dir() # True if it's a directory
```

Decorators: Enhance Functions

Decorators are a powerful way to modify or enhance functions or methods without changing their source code.

```
import functools # Used to preserve the original function's metadata  
  
def your_decorator(func):  
    @functools.wraps(func) # Ensures wrapped function retains its original name, docstring, etc.  
    def wrapper(*args, **kwargs): # Wrapper function that will be executed instead of the original  
        print("Before function") # Code to execute before the original function  
        result = func(*args, **kwargs) # Call the original function  
        print("After function") # Code to execute after the original function  
        return result  
    return wrapper  
  
@your_decorator # Apply the decorator to the 'foo' function  
def foo():  
    print("Hello World!")  
  
foo() # Calling foo() now runs the wrapper logic around it  
# Expected output:  
# Before function  
# Hello World!  
# After function
```

*Args & **Kwargs: Flexible Parameters

Use '*args' and '**kwargs' to allow functions to accept an arbitrary number of positional and keyword arguments, respectively.

```
def some_function(*args, **kwargs): # *args collects positional arguments into a tuple, **kwargs collects keyword arguments into a dictionary  
    print(f'Arguments: {args}')  
    print(f'Keywords: {kwargs}')  
  
some_function('arg1', 'arg2', key1='val1', key2='val2')  
  
# Expected output:  
# Arguments: ('arg1', 'arg2')  
# Keywords: {'key1': 'val1', 'key2': 'val2'}
```

OOP: Core Concepts

Object-Oriented Programming (OOP) structures code using objects that contain data and methods. Key concepts include:

01	02	03
Encapsulation Bundling data (attributes) and methods that operate on the data within a single unit (class). It restricts direct access to some of an object's components.	Inheritance Allows a class (child/subclass) to inherit attributes and methods from another class (parent/superclass), promoting code reusability.	Polyorphism Means "many forms." It allows objects of different classes to be treated as objects of a common base class, responding to the same method call in different ways.
<pre>class MyClass: def __init__(self): self._protected = 10 # Convention for protected member self._private = 20 # Name mangling for private member</pre>	<pre>class Animal: # Parent class def speak(self): pass # Placeholder method class Dog(Animal): # Child class inheriting from Animal def speak(self): # Overrides parent's speak method print("Woof!")</pre>	<pre>class Shape: # Base class def area(self): pass class Rectangle(Shape): # Subclass def area(self): # Specific implementation of area for Rectangle return w * h</pre>

Dataclasses: Simplified Classes

Dataclasses provide a decorator to automatically generate common methods ('__init__', '__repr__', '__eq__', ...) for classes primarily used to store data.

Basic Dataclass

Define data-holding classes with minimal boilerplate.

```
from dataclasses import dataclass  
  
@dataclass # Decorator to create a dataclass  
class Number:  
    val: int # Type-hinted field  
  
obj = Number(2)  
obj.val # 2
```

With Defaults

Assign default values to fields directly in the class definition.

```
@dataclass  
class Product:  
    name: str  
    count: int = 0 # Field with default value  
    price: float = 0.0 # Another field with default  
  
obj = Product("Python") # 'count' and 'price' use their defaults  
obj.count # 0
```

JSON & YAML

These are common data serialization formats for configuration files and data exchange.

```
import json  
  
# Read: Load JSON data from a file into a Python object (dictionary/list).  
with open('data.json', 'r') as f:  
    content = json.load(f)  
  
# Write: Dump Python data (dictionary/list) to a JSON formatted file.  
data = {"name": "Joe", "age": 20}  
with open('out.json', 'w') as f:  
    json.dump(data, f, indent=2) # 'indent' makes the output human-readable
```

```
from ruamel.yaml import YAML  
  
yaml = YAML() # Initialize YAML parser  
with open('config.yaml') as f:  
    data = yaml.load(f) # Load YAML data from a file
```

Virtual Environments

Virtual environments isolate Python project dependencies, preventing conflicts between different projects.

virtualenv A tool to create isolated Python environments. <pre>pip install virtualenv # Install virtualenv mkvirtualenv HelloWorld # Create a new environment workon HelloWorld # Activate the environment deactivate # Deactivate the environment</pre>	Poetry A dependency management and packaging tool for Python. <pre>pip install poetry # Install Poetry poetry new my-project # Create a new project with a virtual environment poetry add pendulum # Add a dependency poetry install # Install dependencies</pre>	UV (Fast) A modern, fast Python package installer and resolver. <pre>curl -LsSf https://astral.sh/uv/install.sh sh # Install UV uv init my-project # Initialize a project with UV uv add requests # Add a package uv run python script.py # Run a script within the UV environment</pre>
---	--	---

Main Entry Point

The 'if __name__ == "__main__":' block ensures that certain code only runs when the script is executed directly, not when imported as a module.

```
def add(a, b):  
    return a + b  
  
if __name__ == "__main__":  
    # This code block will only execute when the script is run directly (e.g., python my_script.py)  
    # It will not run if this file is imported into another Python script.  
    result = add(3, 5)  
    print(result)
```

Built-in Functions: Quick Reference

Python provides a rich set of built-in functions for common tasks.

- abs()** - Returns the absolute value of a number.
- all()** - Returns True if all elements in an iterable are true.
- any()** - Returns True if any element in an iterable is true.
- enumerate()** - Adds a counter to an iterable, returning it as an enumerate object.
- filter()** - Constructs an iterator from elements of an iterable for which a function returns true.
- len()** - Returns the number of items in an object.
- map()** - Applies a given function to each item of an iterable and returns a map object.
- max()** / **min()** - Returns the largest/smallest item in an iterable or between two or more arguments.
- range()** - Generates an immutable sequence of numbers.
- sorted()** - Returns a new sorted list from the items in an iterable.
- sum()** - Sums the items of an iterable.
- zip()** - Combines multiple iterables into a single iterator of tuples.

```
# Examples of built-in functions  
len([1, 2, 3]) # 3 (length of the list)  
max([1, 5, 3]) # 5 (maximum value in the list)  
sorted([3, 1, 2]) # [1, 2, 3] (returns a new sorted list)  
list(zip([1, 2], ['a', 'b'])) # [(1, 'a'), (2, 'b')] (combines elements pairwise)
```

Debugging Essentials

Tools and techniques to identify and resolve issues in your code.

Exceptions & Assertions

Exceptions signal errors, while assertions check for conditions that must be true at a certain point in code.

```
# Raise exception: Explicitly trigger an error.  
raise Exception('Error message')  
  
# Assertion: Verify a condition. If false, it raises an AssertionError.  
status = 'open'  
assert status == 'open', 'Must be open' # The message is shown if assertion fails
```

Logging

Logging provides a way to track events that happen when some software runs, offering more flexibility than print statements for debugging and monitoring.

```
import logging  
  
# Configure basic logging to show debug messages  
logging.basicConfig(  
    level=logging.DEBUG, # Set the logging level (DEBUG, INFO, WARNING, ERROR, CRITICAL)  
    format='%(asctime)s - %(levelname)s - %(message)s' # Define output format  
)  
logging.debug('Program started') # Log a debug message
```

Reference: This cheatsheet covers Python 3.6+ syntax and modern best practices for efficient development.

