#_ Comprehensive Python CheatSheet

1. Basic Operations

- Print to console: `print("Hello, World!")`
- Get user input: `name = input("Enter your name: ")`
- String concatenation: `full_name = first_name + " " + last_name`
- String formatting (f-string): `print(f"Hello, {name}!")`
- String formatting (.format): `print("Hello, {}!".format(name))`
- String formatting (%): `print("Hello, %s!" % name)`
- Type conversion (to int): `age = int("25")`
- Type conversion (to float): `price = float("19.99")`
- Type conversion (to string): `age_str = str(25)`
- Check type: `type(variable)`
- Get length: `len(sequence)`
- Get memory address: `id(object)`
- Delete variable: `del variable_name`
- Check if object is instance of class: `isinstance(object, class)`
- Get all attributes of an object: `dir(object)`

2. Numbers and Math

- Addition: `result = 5 + 3`
- Subtraction: `result = 10 - 4`
- Multiplication: `result = 6 * 7`
- Division: `result = 15 / 3`
- Integer division: `result = 17 // 3`
- Modulus: `remainder = 17 % 3`
- Exponentiation: `result = 2 ** 3`
- Absolute value: `abs(-5)`
- Round number: `round(3.7)`
- Round to specific decimal places: `round(3.14159, 2)`
- Floor division: `import math; math.floor(5.7)`
- Ceiling division: `import math; math.ceil(5.2)`
- Square root: `import math; math.sqrt(16)`
- Calculate pi: `import math; math.pi`
- Calculate e: `import math; math.e`
- Logarithm (base e): `import math; math.log(10)`
- Logarithm (base 10): `import math; math.log10(100)`

- Cosine: `import math; math.cos(math.pi)`
- Tangent: `import math; math.tan(math.pi/4)`
- Degrees to radians: `import math; math.radians(180)`
- Radians to degrees: `import math; math.degrees(math.pi)`
- Generate random number: `import random; random.random()`
- Generate random integer: `import random; random.randint(1, 10)`
- Choose random element: `import random; random.choice([1, 2, 3, 4, 5])`

3. Strings

- Create string: `s = "Hello, World!"`
- Multiline string: `s = """This is a multiline string"""`
- Raw string: `s = r"C:\Users\John"`
- String repetition: `"Hello" * 3`
- String indexing: `first_char = s[0]`
- String slicing: `substring = s[1:5]`
- Reverse string: `reversed_string = s[::-1]`
- Convert to uppercase: `s.upper()`
- Convert to lowercase: `s.lower()`
- Capitalize string: `s.capitalize()`
- Title case: `s.title()`
- Swap case: `s.swapcase()`
- Strip whitespace: `s.strip()`
- Left strip: `s.lstrip()`
- Right strip: `s.rstrip()`
- Replace substring: `s.replace("old", "new")`
- Split string: `parts = s.split(",")`
- Join strings: `",".join(["a", "b", "c"])`
- Check if string starts with: `s.startswith("Hello")`
- Check if string ends with: `s.endswith("World!")`
- Find substring: `index = s.find("World")`
- Count occurrences: `count = s.count("l")`
- Check if string is alphanumeric: `s.isalnum()`
- Check if string is alphabetic: `s.isalpha()`
- Check if string is digit: `s.isdigit()`
- Check if string is lowercase: `s.islower()`
- Check if string is uppercase: `s.isupper()`
- Check if string is title case: `s.istitle()`
- Check if string is whitespace: `s.isspace()`

4. Lists

- Create list: `lst = [1, 2, 3, 4, 5]`
- Create list with range: `lst = list(range(1, 6))`
- Access element: `element = lst[0]`
- Slice list: `subset = lst[1:4]`
- Append to list: `lst.append(6)`
- Extend list: `lst.extend([7, 8, 9])`
- Insert at index: `lst.insert(0, 0)`
- Remove by value: `lst.remove(3)`
- Remove by index: `lst.pop(2)`
- Clear list: `lst.clear()`
- Index of element: `index = lst.index(4)`
- Count occurrences: `count = lst.count(2)`
- Sort list: `lst.sort()`
- Sort list in reverse: `lst.sort(reverse=True)`
- Reverse list: `lst.reverse()`
- Copy list: `new_lst = lst.copy()`
- Shallow copy: `import copy; new_lst = copy.copy(lst)`
- Deep copy: `import copy; new_lst = copy.deepcopy(lst)`
- List comprehension: `squares = [x**2 for x in range(10)]`
- Filter with list comprehension: `evens = [x for x in range(10) if x % 2 == 0]`
- Nested list comprehension: `matrix = [[i*j for j in range(5)] for i in range(5)]`
- Flatten nested list: `flattened = [item for sublist in nested_list for item in sublist]`
- Zip lists: `zipped = list(zip(list1, list2))`
- Unzip lists: `unzipped = list(zip(*zipped))`
- Check if element in list: `5 in lst`
- Get max value: `max(lst)`
- Get min value: `min(lst)`
- Sum of list: `sum(lst)`
- Join list of strings: `" ".join(lst)`
- Create list of lists: `matrix = [[0 for _ in range(5)] for _ in range(5)]`

5. Tuples

- Create tuple with single element: `t = (1,)`
- Access element: `element = t[0]`
- Slice tuple: `subset = t[1:3]`
- Concatenate tuples: `new_t = t + (4, 5, 6)`
- Repeat tuple: `repeated_t = t * 3`
- Count occurrences: `count = t.count(2)`
- Index of element: `index = t.index(3)`
- Check if element in tuple: `2 in t`
- Unpack tuple: `a, b, c = t`
- Swap values using tuple: `a, b = b, a`
- Convert list to tuple: `t = tuple([1, 2, 3])`
- Convert tuple to list: `lst = list(t)`
- Create tuple of tuples: `matrix = ((1, 2, 3), (4, 5, 6), (7, 8, 9))`
- Named tuple: `from collections import namedtuple; Point = namedtuple('Point', ['x', 'y']); p = Point(1, 2)`

6. Sets

- Create set: `s = {1, 2, 3}`
- Create set from list: `s = set([1, 2, 3, 3, 2, 1])`
- Add element: `s.add(4)`
- Update set: `s.update([4, 5, 6])`
- Remove element: `s.remove(2)`
- Remove element if present: `s.discard(5)`
- Pop random element: `element = s.pop()`
- Clear set: `s.clear()`
- Union of sets: `union = s1 | s2`
- Intersection of sets: `intersection = s1 & s2`
- Difference of sets: `difference = s1 - s2`
- Symmetric difference: `sym_diff = s1 ^ s2`
- Check if subset: `is_subset = s1.issubset(s2)`
- Check if superset: `is_superset = s1.issuperset(s2)`
- Check if disjoint: `is_disjoint = s1.isdisjoint(s2)`
- Frozen set (immutable): `fs = frozenset([1, 2, 3])`

7. Dictionaries

- Create dictionary: `d = {"key": "value"}`
- Create dictionary with dict(): `d = dict(key="value")`
- Access value: `value = d["key"]`

- Else clause in while loop: `while condition: do_something() else: condition_is_false()`
- Pass statement: `if condition: pass`
- Match-case (Python 3.10+): `match value: case pattern: do_something()`
- Loop over multiple lists: `for item1, item2 in zip(list1, list2): do_something()`
- Nested loops: `for i in range(3): for j in range(3): print(i, j)`
- List comprehension with if: `[x for x in range(10) if x % 2 == 0]`
- Dictionary comprehension with if: `{x: x**2 for x in range(5) if x % 2 == 0}`

9. Functions

- Define function: `def func_name(param): return result`
- Function with default parameter: `def greet(name="World"): print(f"Hello, {name}!")`
- Function with multiple parameters: `def func(param1, param2, param3): pass`
- Function with variable arguments: `def sum_all(*args): return sum(args)`
- Function with keyword arguments: `def print_info(**kwargs): for k, v in kwargs.items(): print(f"{k}: {v}")`
- Lambda function: `square = lambda x: x**2`
- Return multiple values: `def func(): return 1, 2, 3`
- Nested function: `def outer(): def inner(): pass; return inner`
- Closure: `def outer(x): def inner(y): return x + y; return inner`
- Decorator: `def decorator(func): def wrapper(*args, **kwargs): return func(*args, **kwargs); return wrapper`
- Apply decorator: `@decorator def function(): pass`
- Partial function: `from functools import partial; add_five = partial(add, 5)`
- Recursive function: `def factorial(n): return 1 if n == 0 else n * factorial(n-1)`
- Generator function: `def gen(): yield item`
- Asynchronous function: `async def async_func(): await asyncio.sleep(1)`

10. Classes and Object-Oriented Programming

- Define class: `class ClassName: pass`
- Create instance: `obj = ClassName()`
- Define constructor: `def __init__(self, param): self.param = param`

- Define method: `def method_name(self): pass`
- Define class method: `@classmethod def class_method(cls): pass`
- Define static method: `@staticmethod def static_method(): pass`
- Inheritance: `class ChildClass(ParentClass): pass`
- Multiple inheritance: `class ChildClass(Parent1, Parent2): pass`
- Call superclass method: `super().method_name()`
- Property decorator: `@property def prop_name(self): return self._prop`
- Setter decorator: `@prop_name.setter def prop_name(self, value): self._prop = value`
- Abstract base class: `from abc import ABC, abstractmethod; class AbstractClass(ABC): @abstractmethod def abstract_method(self): pass`
- Dataclass (Python 3.7+): `from dataclasses import dataclass; @dataclass class Point: x: float; y: float`
- Method overriding: `def method_name(self): # Override parent method`
- Private attribute: `self.__private_attr = value`
- Name mangling: `obj._ClassName__private_attr`
- Duck typing: `if hasattr(obj, 'method_name'): obj.method_name()`
- Context manager class: `class ContextManager: def __enter__(self): pass; def __exit__(self, exc_type, exc_value, traceback): pass`
- Metaclass: `class Meta(type): pass; class MyClass(metaclass=Meta): pass`

11. Exceptions and Error Handling

- Try-except block: `try: do_something() except Exception as e: handle_error(e)`
- Try-except-else block: `try: result = do_something() except Exception: handle_error() else: use_result(result)`
- Try-except-finally block: `try: do_something() except Exception: handle_error() finally: cleanup()`
- Catch multiple exceptions: `try: do_something() except (TypeError, ValueError) as e: handle_error(e)`
- Raise exception: `raise ValueError("Invalid input")`
- Raise from: `raise ValueError("Invalid input") from original_error`
- Assert statement: `assert condition, "Error message"`
- Custom exception: `class CustomError(Exception): pass`
- Handle all exceptions: `try: do_something() except Exception as e: handle_any_error(e)`
- Re-raise exception: `try: do_something() except Exception as e:`

- **Exception chaining:** `try: do_something() except Exception as e: raise RuntimeError("Operation failed") from e`

12. File I/O and Context Managers

- **Open file:** `with open("file.txt", "r") as f: content = f.read()`
- **Write to file:** `with open("file.txt", "w") as f: f.write("content")`
- **Append to file:** `with open("file.txt", "a") as f: f.write("new content")`
- **Read lines from file:** `with open("file.txt", "r") as f: lines = f.readlines()`
- **Write lines to file:** `with open("file.txt", "w") as f: f.writelines(lines)`
- **Read file line by line:** `with open("file.txt", "r") as f: for line in f: print(line)`
- **Check if file exists:** `import os; os.path.exists("file.txt")`
- **Get file size:** `import os; os.path.getsize("file.txt")`
- **Delete file:** `import os; os.remove("file.txt")`
- **Rename file:** `import os; os.rename("old_name.txt", "new_name.txt")`
- **Create directory:** `import os; os.mkdir("new_directory")`
- **Change directory:** `import os; os.chdir("path/to/directory")`
- **Get current working directory:** `import os; cwd = os.getcwd()`
- **List directory contents:** `import os; files = os.listdir("directory")`
- **Walk directory tree:** `import os; for root, dirs, files in os.walk("directory"): print(root, dirs, files)`

- Access value with default: `value = d.get("key", "default")`
- Add/update key-value pair: `d["new_key"] = "new_value"`
- Update dictionary: `d.update({"key1": "value1", "key2": "value2"})`
- Remove key-value pair: `del d["key"]`
- Remove and return value: `value = d.pop("key")`
- Remove and return last item: `item = d.popitem()`
- Get keys: `keys = d.keys()`
- Get values: `values = d.values()`
- Get key-value pairs: `items = d.items()`
- Clear dictionary: `d.clear()`
- Copy dictionary: `new_d = d.copy()`
- Deep copy dictionary: `import copy; new_d = copy.deepcopy(d)`
- Check if key in dictionary: `"key" in d`
- Dictionary comprehension: `squares = {x: x**2 for x in range(5)}`
- Merge dictionaries (Python 3.5+): `merged = {**dict1, **dict2}`
- Get value of nested dictionary: `value = d['outer_key']['inner_key']`
- Default dictionary: `from collections import defaultdict; dd = defaultdict(int)`
- Ordered dictionary: `from collections import OrderedDict; od = OrderedDict()`
- Counter dictionary: `from collections import Counter; c = Counter(['a', 'b', 'c', 'a', 'b', 'a'])`

8. Control Flow

- If statement: `if condition: do_something()`
- If-else statement: `if condition: do_something() else: do_other_thing()`
- If-elif-else statement: `if condition1: do_something() elif condition2: do_other_thing() else: do_default()`
- Ternary operator: `result = x if condition else y`
- For loop: `for item in iterable: do_something()`
- For loop with index: `for index, item in enumerate(iterable): do_something()`
- For loop with range: `for i in range(5): do_something()`
- While loop: `while condition: do_something()`
- Break from loop: `if condition: break`
- Continue to next iteration: `if condition: continue`
- Else clause in for loop: `for item in iterable: do_something() else: no_break_occurred()`