Python

General Python Cheatsheet

Basics

Variables and Data Types

Variables

Variables are used to store data values.

Data Types

Python has several built-in data types, including:

```
x = 5        # Integer
y = 3.14     # Float
z = "Hello"  # String
w = True     # Boolean
v = [1,2,3]  # List
t = (1,2,3)  # Tuple
s = {1,2,3}  # Set
d = {'a':1, 'b':2} # Dictionary
```

- `int` : Integer numbers.
- `float` : Floating-point numbers.
- `str` : Strings (text).
- `bool` : Boolean (True or False).
- `list` : Ordered, mutable sequence of items.
- `tuple` : Ordered, immutable sequence of items.
- `set` : Unordered collection of unique items.
- `dict` : Collection of key-value pairs.

Casting

Data types can be explicitly converted using casting:

```
x = int("5")   # x is 5
y = float(2)   # y is 2.0
z = str(3.14) # z is "3.14"
```

Slicing Strings and Lists

Slicing

Slicing is used to extract a portion of a string or list.

```
my_string = "Python"
my_list = [10, 20, 30, 40, 50]
```

- `[start:end]` - Items from start up to (but not including) end.
- `[start:]` - Items from start to the end of the sequence.
- `[:end]` - Items from the beginning to end.
- `[start:end:step]` - Items from start to end with step increment.

Examples

```
print(my_string[1:4])  # "yth"
print(my_list[2:])     # [30, 40, 50]
print(my_string[:3])  # "Pyt"
print(my_list[::2])    # [10, 30, 50]
print(my_string[::-1]) # "nohtyP" (reverse)
```

If Else Statements

If-Else

Conditional statements are used to execute different code blocks based on conditions.

```
x = 10
y = 5
```

- `if condition:` - Executes if condition is true.
- `elif condition:` - Executes if the previous if/elif conditions are false and the current condition is true.
- `else:` - Executes if all previous conditions are false.

Examples

```
if x > y:
    print("x is greater than y")
elif x < y:
    print("x is less than y")
else:
    print("x is equal to y")
```

One-Line If Statement

```
print("x is greater") if x > y else print("x is smaller")
```

Loops

For Loop

Iterates over a sequence (list, tuple, string) or other iterable objects.

```
my_list = [1, 2, 3]
```

While Loop

Executes a block of code as long as a condition is true.

```
x = 0
```

Examples

```
for item in my_list:
    print(item)

while x < 3:
    print(x)
    x += 1
```

Break and Continue

- `break`: Terminates the loop.
- `continue`: Skips the rest of the current iteration and proceeds to the next iteration.

```
for i in range(5):
    if i == 3:
        break  # Exit loop when i is 3
    print(i)
```

```
for i in range(5):
    if i == 3:
        continue  # Skip when i is 3
    print(i)
```

Functions

Functions

Functions are blocks of reusable code.

```
def greet(name):
    return f"Hello, {name}!"
```

- `def` keyword is used to define a function.
- `return` statement returns a value from the function.

Examples

```
print(greet("Alice"))  # Output: Hello, Alice!
```

Lambda Functions

Anonymous functions defined using the `lambda` keyword.

```
add = lambda x, y: x + y
print(add(5, 3))  # Output: 8
```

File Handling

File Operations

- `open(filename, mode)`: Opens a file.
- `mode`: `'r'` (read), `'w'` (write), `'a'` (append), `'b'` (binary), `'+'` (updating).

```
# Writing to a file
file = open("my_file.txt", "w")
file.write("Hello, file!")
file.close()
```

- `read()`: Reads the entire file.
- `readline()`: Reads a single line.
- `readlines()`: Reads all lines into a list.
- `write(string)`: Writes a string to the file.
- `close()`: Closes the file.

Examples

```
# Reading from a file
file = open("my_file.txt", "r")
content = file.read()
print(content)
file.close()
```

Using `with` Statement

Automatically closes the file after the block is executed.

```
with open("my_file.txt", "r") as file:
    content = file.read()
    print(content)
```

Arithmetic Operations

Basic Operations

Python supports standard arithmetic operations:

```
x = 10
y = 3
```

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `//`: Floor Division (integer division)
- `%`: Modulo (remainder)
- `**`: Exponentiation

Examples

```
print(x + y)  # 13
print(x - y)  # 7
print(x * y)  # 30
print(x / y)  # 3.333...
print(x // y) # 3
print(x % y)  # 1
print(x ** y) # 1000
```

Plus-Equals Operator

Plus-Equals (+=)

A shorthand operator that combines addition and assignment.

```
x = 5
```

- `x += y` is equivalent to `x = x + y`

Examples

```
x += 3
print(x)  # Output: 8
```

It also works with other arithmetic operations:

```
x -= 2  # x = x - 2
x *= 4  # x = x * 4
x /= 2  # x = x / 2
```

f-Strings (Python 3.6+)

f-Strings

A convenient way to embed expressions inside string literals for formatting.

```
name = "Alice"
age = 30
```

- f-strings start with an `f` or `F` before the opening quote.
- Expressions are placed inside curly braces `{}`.

Examples

```
print(f"My name is {name} and I am {age} years old.")
# Output: My name is Alice and I am 30 years old.
```

Formatting Options

```
pi = 3.1415926535
print(f"Pi is approximately {pi:.2f}")
# Output: Pi is approximately 3.14
```

Python Data Types & Casting Cheat Sheet

Strings

Definition: Immutable sequence of characters.

Creation:

```
s = "Hello, world!"
s = 'Hello, world!'
s = str(123)  # Convert number to string
```

Common Operations:

- **Concatenation:** `s1 + s2`
- **Length:** `len(s)`
- **Slicing:** `s[start:end:step]`
- **Case Conversion:** `s.upper()`, `s.lower()`, `s.capitalize()`
- **Stripping Whitespace:** `s.strip()`, `s.lstrip()`, `s.rstrip()`
- **Finding Substrings:** `s.find(sub)`, `s.index(sub)`
- **Replacing Substrings:** `s.replace(old, new)`
- **Splitting:** `s.split(sep)`
- **Joining:** `sep.join(list_of_strings)`
- **Formatting:** `f"Value: {variable}"`, `"Value: {}".format(variable)`

Examples:

```
string1 = "Python"
string2 = " is fun!"
combined = string1 + string2  # Concatenation
print(combined)  # Output: Python is fun!

text = "  Trim me  "
print(text.strip())  # Output: Trim me

message = "hello world"
print(message[0:5]) # Output: hello
```

Numbers

Integers (int)

Whole numbers, positive or negative, without decimals.

```
x = 10
y = -5
```

Floating-Point Numbers (float)

Numbers with a decimal point.

```
x = 3.14
y = -2.5
```

Complex Numbers (complex)

Numbers with a real and imaginary part (j).

```
z = 2 + 3j
```

Operators:

`+, -, *, /, // (floor division), %, ** (exponentiation)`

Examples:

```
a = 10
b = 3
print(a + b)  # Output: 13
print(a / b)  # Output: 3.3333333333333335
print(a // b) # Output: 3
print(a % b)  # Output: 1
print(a ** b) # Output: 1000
```

Booleans

Definition:

Represents truth values: `True` or `False` (case-sensitive).

Boolean Operators:

`and`, `or`, `not`

Truthiness:

Most values are `True`. Values that evaluate to `False` include: `False`, `None`, `0`, empty strings (`""`), empty lists (`[]`), empty tuples (`()`), empty dictionaries (`{}`), empty sets (`set()`).

Comparison Operators:

`==` (equal), `!=` (not equal), `>` (greater than), `<` (less than), `>=` (greater than or equal to), `<=` (less than or equal to)

Examples:

```
x = 5
y = 10
print(x > y)  # Output: False
print(x < y and x > 0)  # Output: True
print(not x == y)  # Output: True
```

Lists

Definition: Ordered, mutable (changeable) sequence of items.

Creation:

```
my_list = [1, 2, 3, 'a', 'b']
my_list = list((1, 2, 3))  # Convert tuple to list
```

Common Operations:

- **Accessing Elements:** `my_list[index]`
- **Slicing:** `my_list[start:end:step]`
- **Length:** `len(my_list)`
- **Adding Elements:** `my_list.append(item)`, `my_list.insert(index, item)`, `my_list.extend(another_list)`
- **Removing Elements:** `my_list.remove(item)`, `my_list.pop(index)`, `del my_list[index]`
- **Searching:** `item in my_list`
- **Sorting:** `my_list.sort()`, `sorted(my_list)`
- **Reversing:** `my_list.reverse()`

Examples:

```
numbers = [1, 2, 3, 4, 5]
print(numbers[0])  # Output: 1
numbers.append(6)
print(numbers)  # Output: [1, 2, 3, 4, 5, 6]
numbers.remove(3)
print(numbers)  # Output: [1, 2, 4, 5, 6]
print(len(numbers)) # Output: 5
```

Tuples

Definition:

Ordered, immutable (unchangeable) sequence of items.

Creation:

```
my_tuple = (1, 2, 3, 'a', 'b')
my_tuple = tuple([1, 2, 3])  # Convert list to tuple
```

Common Operations:

- **Accessing Elements:** `my_tuple[index]`
- **Slicing:** `my_tuple[start:end:step]`
- **Length:** `len(my_tuple)`
- **Counting:** `my_tuple.count(item)`
- **Finding Index:** `my_tuple.index(item)`

Immutability:

Tuples cannot be modified after creation. You can't add, remove, or change elements.

Examples:

```
point = (10, 20)
print(point[0])  # Output: 10
print(len(point)) # Output: 2
```

Sets

Definition: Unordered collection of unique items.

Creation:

```
my_set = {1, 2, 3, 4, 5}
my_set = set([1, 2, 3])  # Convert list to set
```

Common Operations:

- **Adding Elements:** `my_set.add(item)`
- **Removing Elements:** `my_set.remove(item)`, `my_set.discard(item)`
- **Membership Testing:** `item in my_set`
- **Set Operations:** `union`, `intersection`, `difference`, `symmetric_difference`

Set Operations:

- **Union:** `set1 | set2` or `set1.union(set2)` (All elements in both sets)
- **Intersection:** `set1 & set2` or `set1.intersection(set2)` (Common elements)
- **Difference:** `set1 - set2` or `set1.difference(set2)` (Elements in set1 but not in set2)
- **Symmetric Difference:** `set1 ^ set2` or `set1.symmetric_difference(set2)` (Elements in either set, but not both)

Examples:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1 | set2)  # Output: {1, 2, 3, 4, 5}
print(set1 & set2)  # Output: {3}
print(set1 - set2)  # Output: {1, 2}
```

Dictionaries

Definition:

Unordered collection of key-value pairs. Keys must be unique and immutable.

Creation:

```
my_dict = {'name': 'Alice', 'age': 30}
my_dict = dict(name='Bob', age=25)
```

Common Operations:

- **Accessing Values:** `my_dict[key]`
- **Adding/Updating:** `my_dict[key] = value`
- **Removing:** `del my_dict[key]`, `my_dict.pop(key)`
- **Checking Key Existence:** `key in my_dict`
- **Getting Keys:** `my_dict.keys()`
- **Getting Values:** `my_dict.values()`
- **Getting Items:** `my_dict.items()`

Examples:

```
person = {'name': 'Alice', 'age': 30}
print(person['name'])  # Output: Alice
person['city'] = 'New York'
print(person)  # Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}
del person['age']
print(person)  # Output: {'name': 'Alice', 'city': 'New York'}
```

Casting

Definition: Converting a value from one data type to another.

Functions:

- `int(x)`: Converts `x` to an integer.
- `float(x)`: Converts `x` to a floating-point number.
- `str(x)`: Converts `x` to a string.
- `bool(x)`: Converts `x` to a boolean.
- `list(x)`: Converts `x` to a list.
- `tuple(x)`: Converts `x` to a tuple.
- `set(x)`: Converts `x` to a set.
- `dict(x)`: Converts `x` to a dictionary (where `x` is a sequence of key-value pairs).

Examples:

```
x = "10"
y = int(x)  # y is now 10 (integer)
z = float(x) # z is now 10.0 (float)

a = 1
b = bool(a) # b is True

list1 = [(1, 'a'), (2, 'b')]
dict1 = dict(list1)
print(dict1) # Output: {1: 'a', 2: 'b'}
```

Python Cheat Sheet: Heaps, Stacks, and Queues

Heaps (Priority Queues)

What is a Heap?

A heap is a tree-based data structure that satisfies the heap property: In a *min-heap*, the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root. In a *max-heap*, the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

`heapq` Module

Python's `heapq` module provides an implementation of the heap queue algorithm (also known as the priority queue algorithm).

```
import heapq
```

`heapify(iterable)`

Transforms a list into a heap, in-place, in linear time.

```
heapq.heapify(x) # Transform list x into a heap
```

`heappush(heap, item)`

Pushes the `item` onto the `heap`, maintaining the heap invariant.

```
heapq.heappush(heap, item)
```

`heappop(heap)`

Pops and returns the smallest item from the `heap`, maintaining the heap invariant. If the heap is empty, it raises an `IndexError`.

```
smallest = heapq.heappop(heap)
```

`heappushpop(heap, item)`

Pushes `item` on the `heap`, then pops and returns the smallest item from the `heap`. More efficient than `heappush()` followed by `heappop()`.

```
smallest = heapq.heappushpop(heap, item)
```

`heapreplace(heap, item)`

Pops and returns the smallest item from the `heap`, and then pushes the new `item`. More efficient than `heappop()` followed by `heappush()`.

```
smallest = heapq.heapreplace(heap, item)
```

Example: Creating and using a min-heap

```
import heapq

my_list = [4, 1, 7, 3, 8, 5]
heapq.heapify(my_list)
print(my_list)  # Output: [1, 3, 5, 4, 8, 7]

smallest = heapq.heappop(my_list)
print(smallest)  # Output: 1
print(my_list)  # Output: [3, 4, 5, 7, 8]

heapq.heappush(my_list, 2)
print(my_list)  # Output: [2, 3, 5, 7, 8, 4]
```

Finding the `n` largest/smallest elements

`heapq` offers functions to find the `n` largest or smallest elements without fully sorting the data.

```
import heapq

numbers = [1, 4, 2, 10, 8, 5, 7]

largest_3 = heapq.nlargest(3, numbers)
smallest_3 = heapq.nsmallest(3, numbers)

print(largest_3)  # Output: [10, 8, 7]
print(smallest_3) # Output: [1, 2, 4]
```

Max-Heap Implementation

Python's `heapq` is a min-heap. To simulate a max-heap, insert the negative of each value. When popping, negate the value again.

```
import heapq

numbers = [1, 4, 2, 10, 8, 5, 7]

max_heap = [-x for x in numbers]
heapq.heapify(max_heap)

largest = -heapq.heappop(max_heap)
print(largest) # Output: 10
```

Complexity Analysis

Heap (using `heapq`)

- `heapify(iterable)` — O(n)
- `heappush(heap, item)` — O(log n)
- `heappop(heap)` — O(log n)
- `heappushpop(heap, item)` — O(log n)
- `heapreplace(heap, item)` — O(log n)

Stack (using `list`)

- `append(item)` (push) — O(1) (amortized)
- `pop()` — O(1)

Queue (using `collections.deque`)

- `append(item)` (enqueue) — O(1)
- `popleft()` (dequeue) — O(1)

Queues (FIFO)

What is a Queue?

A queue is a linear data structure that follows the First-In, First-Out (FIFO) principle. The first element added to the queue is the first element to be removed.

Implementation with `collections.deque`

Python's `collections.deque` (double-ended queue) is the preferred way to implement a queue because it provides efficient `append()` and `popleft()` operations.

```
from collections import deque

queue = deque()
```

Enqueue (append)

Adds an element to the rear of the queue.

```
queue.append(item)
```

Dequeue (popleft)

Removes and returns the front element from the queue. If the queue is empty, it raises an `IndexError`.

```
front_element = queue.popleft()
```

Peek (front)

Returns the front element of the queue without removing it. Check for emptiness before peeking.

```
def peek(queue):
    if not queue:
        return None
    return queue[0]
```

isEmpty

Checks if the queue is empty.

```
def is_empty(queue):
    return len(queue) == 0
```

Size

Returns the number of elements in the queue.

```
def size(queue):
    return len(queue)
```

Example: Using a queue

```
from collections import deque

queue = deque()

queue.append(10)
queue.append(20)
queue.append(30)

print(peek(queue))        # Output: 10
print(queue.popleft())  # Output: 10
print(is_empty(queue))    # Output: False
print(size(queue))      # Output: 2
```

Applications

Queues are used in many algorithms and scenarios, such as:

- Breadth-First Search (BFS)
- Task scheduling
- Handling requests in web servers

`queue.Queue`

The `queue.Queue` class is useful for thread-safe queue operations.

```
import queue

q = queue.Queue()
q.put(item)
item = q.get()
```

Thread-safe Queues

`queue.Queue` Class

The `queue.Queue` class from the `queue` module is designed for thread-safe queue operations, ensuring that multiple threads can safely access and modify the queue.

```
import queue

q = queue.Queue(maxsize=0) # maxsize=0 means infinite queue size
```

`put(item, block=True, timeout=None)`

Places `item` into the queue. If `block` is true and the queue is full, it waits until a free slot is available. If `timeout` is a positive number, it blocks at most `timeout` seconds and raises `queue.Full` exception if no free slot was available within that time.

```
q.put(item)
```

`get(block=True, timeout=None)`

Removes and returns an item from the queue. If `block` is true and the queue is empty, it waits until an item is available. If `timeout` is a positive number, it blocks at most `timeout` seconds and raises `queue.Empty` exception if no item was available within that time.

```
item = q.get()
```

`task_done()`

Indicates that a formerly enqueued task is complete. Used by queue consumers. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

```
q.task_done()
```

`join()`

Blocks until all items in the queue have been gotten and processed. The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate that the thread has finished working on the item.

```
q.join()
```

Example: Using `queue.Queue` in a multithreaded scenario

```
import queue
import threading
import time

def worker(q):
    while True:
        item = q.get()
        if item is None:
            break
        print(f'Processing: {item}')
        time.sleep(1)
        print(f'Finished: {item}')
        q.task_done()

q = queue.Queue()
threads = []
for i in range(3):
    t = threading.Thread(target=worker, args=(q,))
    t.start()
    threads.append(t)

for item in range(10):
    q.put(item)

# block until all tasks are done
q.join()

# stop workers
for i in range(3):
    q.put(None)
for t in threads:
    t.join()
```

Why use `queue.Queue` for thread safety?

`queue.Queue` provides built-in locking mechanisms that prevent race conditions when multiple threads try to access the queue simultaneously. Without such mechanisms, data corruption and unpredictable behavior can occur.

Alternatives to `queue.Queue`

While `queue.Queue` is the standard for thread-safe queues, other libraries like `multiprocessing.Queue` (for inter-process communication) and `asyncio.Queue` (for asynchronous programming) offer similar functionalities tailored to their respective environments.

Stacks (LIFO)

What is a Stack?

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. The last element added to the stack is the first element to be removed.

Implementation with Lists

Python lists can be easily used as stacks. The `append()` method adds an element to the top, and the `pop()` method removes the top element.

```
stack = []
```

Push (append)

Adds an element to the top of the stack.

```
stack.append(item)
```

Pop

Removes and returns the top element from the stack. If the stack is empty, it raises an `IndexError`.

```
top_element = stack.pop()
```

Peek (top)

Returns the top element of the stack without removing it. It's a good practice to check if the stack is empty before peeking.

```
def peek(stack):
    if not stack:
        return None
    return stack[-1]
```

isEmpty

Checks if the stack is empty.

```
def is_empty(stack):
    return len(stack) == 0
```

Size

Returns the number of elements in the stack.

```
def size(stack):
    return len(stack)
```

Example: Using a stack

```
stack = []

stack.append(10)
stack.append(20)
stack.append(30)

print(peek(stack))   # Output: 30
print(stack.pop())  # Output: 30
print(is_empty(stack)) # Output: False
print(size(stack))    # Output: 2
```

Applications

Stacks are used in many algorithms, such as:

- Function call stack
- Expression evaluation
- Backtracking algorithms

Using `collections.deque`

For optimized stack operations, particularly in multi-threaded environments, use `collections.deque`:

```
from collections import deque

stack = deque()
stack.append(item)
stack.pop()
```

Use cases and applications

Heaps:

- **Priority Queues:** Managing tasks with different priorities.
- **Heap Sort:** Efficient sorting algorithm.
- **Graph Algorithms:** Dijkstra's shortest path algorithm, Prim's minimum spanning tree algorithm.
- **Median Maintenance:** Dynamically finding the median of a stream of numbers.

Stacks:

- **Function Call Stack:** Managing function calls and returns.
- **Expression Evaluation:** Evaluating arithmetic expressions.
- **Backtracking Algorithms:** Solving problems by exploring possible solutions.
- **Undo/Redo Functionality:** Implementing undo/redo operations in applications.
- **Depth-First Search (DFS):** Traversing trees and graphs.

Queues:

- **Breadth-First Search (BFS):** Traversing trees and graphs.
- **Task Scheduling:** Managing tasks in a specific order.
- **Web Server Request Handling:** Processing incoming requests.
- **Print Queue:** Managing print jobs.
- **Asynchronous Data Transfer:** Handling data transfer between processes or systems.

Real-world examples

- **Heaps:** Task schedulers in operating systems.
- **Stacks:** Browser history (back button).
- **Queues:** Customer service call centers.

Python Strings

Array-like Operations

`list()`	Creates a list from an iterable. `list('abc')  # Output: ['a', 'b', 'c']`
`tuple()`	Creates a tuple from an iterable. `tuple([1, 2, 3])  # Output: (1, 2, 3)`
`len()`	Returns the length (number of items) of an object. `len('hello')  # Output: 5`
`[]` (indexing)	Access elements by index (0-based). `'hello'[0]  # Output: 'h'`
`[:]` (slicing)	Extract a sub-sequence. `'hello'[1:4]  # Output: 'ell'`
`in` operator	Check if an element is present. `'h' in 'hello'  # Output: True`
`+` (concatenation)	Concatenate sequences. `'hello' + ' world'  # Output: 'hello world'`

Looping Through Strings

Iterating over each character in a string:

```
for char in 'Python':
    print(char)
# Output:
# P
# y
# t
# h
# o
# n
```

Using `enumerate` to get both index and character:

```
for index, char in enumerate('Python'):
    print(f'Character at index {index}
is {char}')
# Output:
# Character at index 0 is P
# Character at index 1 is y
# Character at index 2 is t
# Character at index 3 is h
# Character at index 4 is o
# Character at index 5 is n
```

Looping through a string in reverse:

```
string = 'Python'
for char in reversed(string):
    print(char)
# Output:
# n
# o
# h
# t
# y
# P
```

String Slicing

`string[start:end]`	Extracts a portion of the string from `start` (inclusive) to `end` (exclusive). `'Python'[1:4]  # Output: 'yth'`
`string[start:]`	Extracts from `start` to the end of the string. `'Python'[2:]  # Output: 'thon'`
`string[:end]`	Extracts from the beginning to `end` (exclusive). `'Python'[:3]  # Output: 'Pyt'`
`string[start:end:step]`	Extracts a portion with a specified step. `'Python'[::2]  # Output: 'Pto'` `'Python'[::-1] # Output: 'nohtyP' (reverse)`
Negative Indices	Indices can be negative, starting from the end (-1). `'Python'[-1]  # Output: 'n'` `'Python'[-2:] # Output: 'on'`

String Length and Repetition

`len(string)`	Returns the number of characters in the string. `len('Python')  # Output: 6`
`string * n`	Repeats the string `n` times. `'Py' * 3  # Output: 'PyPyPy'`

String Membership

`substring in string`	Checks if `substring` is present in `string`. `'yth' in 'Python'  # Output: True` `'abc' in 'Python'  # Output: False`
`substring not in string`	Checks if `substring` is not present in `string`. `'abc' not in 'Python'  # Output: True`

String Concatenation

`string1 + string2`	Concatenates two strings. `'Hello' + ' ' + 'World'  # Output: 'Hello World'`
`.join(iterable)`	Joins elements of an iterable into a string. `'-'.join(['a', 'b', 'c'])  # Output: 'a-b-c'`

`f-strings`	String interpolation using f-strings (Python 3.6+).
	`name = 'Alice'`
	`age = 30`
	`f'My name is {name} and I am {age} years old.'`
	`# Output: 'My name is Alice and I am 30 years old.'`

String Formatting

Using `.format()` method:

```
'Hello, {}! You are {} years old.'.format('World', 25)
# Output: 'Hello, World! You are 25 years old.'
```

Formatting with named placeholders:

```
'Hello, {name}! You are {age} years old.'.format(name='World', age=25)
# Output: 'Hello, World! You are 25 years old.'
```

Using f-strings (Python 3.6+):

```
name = 'World'
age = 25
f'Hello, {name}! You are {age} years old.'
# Output: 'Hello, World! You are 25 years old.'
```

String Input

`input()`	Reads a line from input, converts it to a string, and returns it.
	`name = input('Enter your name: ')`
	`print('Hello, ' + name + '!')`

String `join()` Method

The `join()` method is used to concatenate elements of an iterable (like a list or tuple) into a single string.

```
words = ['Python', 'is', 'awesome']
separator = ' '
result = separator.join(words)
print(result)  # Output: Python is awesome
```

Joining with different separators:

```
numbers = ['1', '2', '3']
comma_separated = ', '.join(numbers)
print(comma_separated)  # Output: 1, 2, 3
```

Joining characters of a string (less common):

```
word = 'Python'
spaced_word = ' '.join(word)
print(spaced_word)  # Output: P y t h o n
```

String `endswith()` Method

`string.endswith(suffix)`	Checks if the string ends with the specified `suffix`.
	`filename = 'document.pdf'`
	`filename.endswith('.pdf')  # Output: True`
	`filename.endswith('.txt')  # Output: False`

`string.endswith(suffix, start, end)`	Checks if the string ends with the specified `suffix` within the specified `start` and `end` positions.
	`text = 'Python is great.'`
	`text.endswith('great.', 8) # Output: True`

Checking with a tuple of suffixes:	You can pass a tuple of suffixes to check if the string ends with any of them.
	`filename = 'image.jpg'`
	`filename.endswith(('.jpg', '.png', '.gif'))  # Output: True`

Python F-Strings Cheat Sheet

Introduction to F-Strings

F-strings (formatted string literals) are a powerful and convenient way to embed expressions inside string literals for formatting.

Introduced in Python 3.6, they provide a concise and readable syntax.

They are faster than both `%`-formatting and `.format()` string formatting methods.

To create an f-string, prefix the string with the letter `f` or `F`.

Expressions inside the string are enclosed in curly braces `{}`.

Example:

```
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

Basic Usage

Variable substitution	`name = "Bob"`
	`print(f"Hello, {name}!")`
	`# Output: Hello, Bob!`

Evaluating expressions	`x = 10`
	`y = 5`
	`print(f"The sum of x and y is {x + y}.")  # Output: The sum of x and y is 15.`

Calling functions	`def greet(name):`
	`    return f"Hello, {name}!"`
	
	`print(f"{greet('Charlie')}")  # Output: Hello, Charlie!`

F-Strings Fill and Align

`>`	Right alignment
	`num = 42`
	`print(f"{num:>4}")  # Output: 42`

`<`	Left alignment
	`text = "hello"`
	`print(f"{text:<10}")  # Output: hello`

`^`	Center alignment
	`word = "Python"`
	`print(f"{word:^10}")  # Output:   Python`

`=`	Forces the padding to be placed after the sign but before the digits.
	`number = -123`
	`print(f"{number:=10}")  # Output: -      123`

Fill character	Uses the specified character to fill the remaining space.
	`number = 5`
	`print(f"{number:0>4}") # Output: 0005`

F-Strings Type

`d`	Integer type
	`num = 42`
	`print(f"{num:d}")  # Output: 42`

`f`	Float type
	`pi = 3.14159`
	`print(f"{pi:.2f}")  # Output: 3.14`

`s`	String type
	`text = "hello"`
	`print(f"{text:s}")  # Output: hello`

`b`	Binary type
	`number = 10`
	`print(f"{number:b}") # Output: 1010`

`o`	Octal type
	`number = 10`
	`print(f"{number:o}") # Output: 12`

`x`	Hexadecimal type (lowercase)
	`number = 255`
	`print(f"{number:x}") # Output: ff`

`X`	Hexadecimal type (uppercase)
	`number = 255`
	`print(f"{number:X}") # Output: FF`

`e`	Scientific notation (lowercase)
	`number = 1000`
	`print(f"{number:e}") # Output: 1.000000e+03`

`E`	Scientific notation (uppercase)
	`number = 1000`
	`print(f"{number:E}") # Output: 1.000000E+03`

F-Strings Other Formatting Options

Precision for floats	`pi = 3.1415926535` `print(f"{pi:.3f}")  # Output: 3.142`
Thousands separator	`number = 1234567` `print(f"{number:,}")  # Output: 1,234,567`

Percentage formatting	`ratio = 0.75`
	`print(f"{ratio:.2%}")`
	`# Output: 75.00%`

Date formatting	`import datetime`
	`today = datetime.datetime.now()`
	`print(f"{today:%Y-%m-%d %H:%M:%S}")`
	`# Output (example): 2024-01-01 12:00:00`

F-Strings Sign

`+`

Indicates that a sign should be used for both positive as well as negative numbers.

```
num1 = 10
num2 = -10
print(f"{num1:+d}")  # Output: +10
print(f"{num2:+d}")  # Output: -10
```

` ` (space)

Indicates that a space should be used for positive numbers and a minus sign for negative numbers.

```
num1 = 10
num2 = -10
print(f"{num1: d}")  # Output: 10
print(f"{num2: d}")  # Output: -10
```

`-`

Indicates that only negative numbers should have a sign (default behavior).

```
num1 = 10
num2 = -10
print(f"{num1:-d}")  # Output: 10
print(f"{num2:-d}")  # Output: -10
```

Escaping Characters in F-Strings

To include a literal curly brace in an f-string, double it: `{{` or `}}`.

Example:

```
print(f"{{This is not a variable}}")  # Output: {This is not a variable}
```

Backslashes cannot be directly included inside the expression part of f-strings.

However, you can use a variable to hold the backslash or use other escaping methods.

Example:

```
newline = "\n"
print(f"This is a new line{newline}")
```

Limitations of F-Strings

F-strings are evaluated at runtime, so they don't support comments inside the expression part.

F-strings cannot be used directly to specify the format string in methods like `str.format()` or `%` formatting.

They are most suitable for simple and readable string formatting tasks.

Python Lists Cheat Sheet

Defining Lists

Lists are ordered, mutable collections of items.

Creating an empty list:

```
my_list = []
my_list = list()
```

Creating a list with initial values:

```
my_list = [1, 2, 3]
my_list = ['a', 'b', 'c']
my_list = [1, 'hello', 3.4]
```

Lists can contain mixed data types:

```
mixed_list = [1, "Hello", 3.4, True]
```

Nested Lists:

```
nested_list = [ [1, 2], [3, 4] ]
```

Generating Lists

Using `range()`:

```
numbers = list(range(5)) # [0, 1, 2, 3, 4]
numbers = list(range(2, 7)) # [2, 3, 4, 5, 6]
numbers = list(range(0, 10, 2)) # [0, 2, 4, 6, 8]
```

List Comprehensions:

```
squares = [x**2 for x in range(10)] # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
even_squares = [x**2 for x in range(10) if x % 2 == 0] # [0, 4, 16, 36, 64]
```

Appending and Extending Lists

`append(item)`

Adds an item to the end of the list.

```
my_list = [1, 2, 3]
my_list.append(4) # my_list is now [1, 2, 3, 4]
```

`extend(iterable)`

Extends the list by appending elements from an iterable.

```
my_list = [1, 2, 3]
my_list.extend([4, 5, 6]) # my_list is now [1, 2, 3, 4, 5, 6]
```

`insert(index, item)`

Inserts an item at a given position.

```
my_list = [1, 2, 3]
my_list.insert(1, 'hello') # my_list is now [1, 'hello', 2, 3]
```

List Slicing

Slicing allows you to extract portions of a list.

`list[start:end:step]`

- `start` : The index to start the slice (inclusive). If omitted, defaults to 0.
- `end` : The index to end the slice (exclusive). If omitted, defaults to the length of the list.
- `step` : The increment between each index. If omitted, defaults to 1.

Examples:

```
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

my_list[2:5]  # [2, 3, 4]
my_list[:3]   # [0, 1, 2]
my_list[5:]   # [5, 6, 7, 8, 9]
my_list[:]    # [0, 1, 2, 3, 4, 5, 6, 7,
8, 9] (a copy of the list)
my_list[::2]  # [0, 2, 4, 6, 8]
my_list[1::2] # [1, 3, 5, 7, 9]
my_list[::-1] # [9, 8, 7, 6, 5, 4, 3, 2,
1, 0] (reverses the list)
```

Removing Items

`remove(item)`	Removes the first occurrence of a value. `my_list = [1, 2, 3, 2]` `my_list.remove(2) # my_list is now [1, 3, 2]`
`pop(index)`	Removes and returns the item at index. If index is not specified, removes and returns the last item. `my_list = [1, 2, 3]` `item = my_list.pop(1) # item is 2, my_list is now [1, 3]` `item = my_list.pop()  # item is 3, my_list is now [1]`
`del list[index]` or `del list[start:end]`	Deletes an item at a specific index or a slice of the list. `my_list = [1, 2, 3, 4]` `del my_list[1] # my_list is now [1, 3, 4]` `del my_list[1:3] # my_list is now [1]`
`clear()`	Removes all items from the list. `my_list = [1, 2, 3]` `my_list.clear() # my_list is now []`

Accessing Elements

Elements in a list are accessed using their index. Indexing starts at 0.

```
my_list = ['a', 'b', 'c']
first_element = my_list[0] #
first_element is 'a'
second_element = my_list[1] #
second_element is 'b'
```

Negative Indexing: Access elements from the end of the list.

```
my_list = ['a', 'b', 'c']
last_element = my_list[-1] #
last_element is 'c'
second_last = my_list[-2]  # second_last
is 'b'
```

Concatenating Lists

Using the `+` operator:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined_list = list1 + list2  #
combined_list is [1, 2, 3, 4, 5, 6]
```

Using `extend()` method:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)  # list1 is now [1,
2, 3, 4, 5, 6]
```

Sorting and Reversing

`sort()`

Sorts the list in place (modifies the original list).

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6]
my_list.sort() # my_list is now [1, 1,
2, 3, 4, 5, 6, 9]
my_list.sort(reverse=True) # my_list is
now [9, 6, 5, 4, 3, 2, 1, 1]
```

`sorted()`

Returns a new sorted list (does not modify the original list).

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6]
sorted_list = sorted(my_list) #
sorted_list is [1, 1, 2, 3, 4, 5, 6, 9],
my_list is unchanged
```

`reverse()`

Reverses the list in place.

```
my_list = [1, 2, 3, 4, 5]
my_list.reverse() # my_list is now [5,
4, 3, 2, 1]
```

`reversed()`

Returns an iterator that accesses the given sequence in the reverse order.

```
my_list = [1, 2, 3, 4, 5]
reversed_list = list(reversed(my_list))
# reversed_list is [5, 4, 3, 2, 1],
my_list is unchanged
```

Counting Elements

`count(item)` : Returns the number of times item appears in the list.

```
my_list = [1, 2, 2, 3, 2, 4]
count = my_list.count(2)  # count is 3
```

Repeating Lists

Using the `*` operator:

```
my_list = [1, 2, 3]
repeated_list = my_list * 3  #
repeated_list is [1, 2, 3, 1, 2, 3, 1,
2, 3]
```

Python Flow Control

Basic `if` Statement

The `if` statement is the fundamental control structure in Python. It allows you to execute a block of code only if a certain condition is true.

```
if condition:
    # Code to execute if the condition is true
    statement1
    statement2
```

Example:

```
x = 10
if x > 5:
    print("x is greater than 5")
```

Here, the condition `x > 5` is checked. If it's true, the indented block of code (the `print` statement) is executed.

The indentation is crucial in Python. It defines the scope of the code block associated with the `if` statement. Consistent indentation (usually 4 spaces) is essential for correct execution.

Conditions can involve any comparison operators:

- `==` (equal to)
- `!=` (not equal to)
- `>` (greater than)
- `<` (less than)
- `>=` (greater than or equal to)
- `<=` (less than or equal to)

`if...else` Statement

The `if...else` statement provides an alternative block of code to execute if the `if` condition is false.

```
if condition:
    # Code to execute if the condition is true
    statement1
else:
    # Code to execute if the condition is false
    statement2
```

Example:

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

In this case, since `x` is not greater than 5, the code inside the `else` block is executed.

`if...elif...else` Statement

The `if...elif...else` statement allows you to check multiple conditions in sequence. `elif` is short for "else if".

```
if condition1:
    # Code to execute if condition1 is true
    statement1
elif condition2:
    # Code to execute if condition1 is false and condition2 is true
    statement2
else:
    # Code to execute if all conditions are false
    statement3
```

Example:

```
x = 5
if x > 5:
    print("x is greater than 5")
elif x < 5:
    print("x is less than 5")
else:
    print("x is equal to 5")
```

Here, the conditions are checked in order. If `x` is greater than 5, the first block is executed. If `x` is less than 5, the second block is executed. Otherwise, the `else` block is executed.

You can have multiple `elif` blocks to handle different scenarios.

One-Line `if` Statements (Conditional Expressions)

Python allows you to write simple `if...else` statements in a single line using conditional expressions.

```
value_if_true if condition else value_if_false
```

Example:

```
x = 10
y = 20 if x > 5 else 30
print(y)  # Output: 20
```

This is equivalent to:

```
x = 10
if x > 5:
    y = 20
else:
    y = 30
print(y)
```

One-line `if` statements are best suited for simple conditions and assignments. Avoid using them for complex logic to maintain readability.

Nested `if` Statements

You can nest `if` statements inside other `if` statements to create more complex decision-making structures.

```
if condition1:
    if condition2:
        # Code to execute if both condition1 and condition2 are true
        statement1
    else:
        # Code to execute if condition1 is true but condition2 is false
        statement2
else:
    # Code to execute if condition1 is false
    statement3
```

Example:

```
x = 10
y = 5
if x > 5:
    if y > 2:
        print("x is greater than 5 and y is greater than 2")
    else:
        print("x is greater than 5 but y is not greater than 2")
else:
    print("x is not greater than 5")
```

Nesting can make code harder to read, so use it judiciously and consider alternative approaches like logical operators (`and`, `or`) when possible.

Using Logical Operators in `if` Statements

Logical operators (`and` , `or` , `not`) can combine multiple conditions in an `if` statement.

- `and` : Returns `True` if both conditions are true.
- `or` : Returns `True` if at least one condition is true.
- `not` : Returns the opposite of the condition's value.

Examples:

```
x = 10
y = 5

if x > 5 and y > 2:
    print("Both conditions are true")

if x > 5 or y < 2:
    print("At least one condition is
true")

if not x < 5:
    print("x is not less than 5")
```

Using logical operators can simplify complex nested `if` statements and improve readability.

Truthiness of Values

In Python, certain values are considered "truthy" (evaluate to `True` in a boolean context) and others are "falsy" (evaluate to `False`).

Falsy Values:

- `False`
- `None`
- `0` (zero of any numeric type)
- Empty sequences (e.g., `''` , `[]` , `()` , `{}`)

Truthy Values:

- All other values are considered truthy.

Example:

```
my_list = []

if my_list:
    print("List is not empty")
else:
    print("List is empty") # This will
be printed
```

You can directly use variables in `if` conditions based on their truthiness, without explicitly comparing them to `True` or `False` .

Python Loops Cheat Sheet

Basic For Loop

The basic `for` loop iterates through each item in a sequence (like a list, tuple, or string).

Syntax:

```
for item in sequence:
    # Code to execute for each item
```

Example:

```
my_list = [1, 2, 3, 4, 5]
for number in my_list:
    print(number)
```

Iterating through a string:

```
my_string = "Python"
for char in my_string:
    print(char)
```

Iterating through a tuple:

```
my_tuple = (10, 20, 30)
for item in my_tuple:
    print(item)
```

For loop with conditional statement:

```
my_list = [1, 2, 3, 4, 5]
for number in my_list:
    if number % 2 == 0:
        print(f"{number} is even")
    else:
        print(f"{number} is odd")
```

Using `_` as a variable name when the value is not needed:

```
for _ in range(5):
    print("Hello")
```

For Loop with Index

To access both the item and its index, use the `enumerate()` function.

Syntax:

```
for index, item in enumerate(sequence):
    # Code to execute with index and
item
```

Example:

```
my_list = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(my_list):
    print(f"Index: {index}, Fruit:
{fruit}")
```

Starting the index from a different number (e.g., 1):

```
my_list = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(my_list,
start=1):
    print(f"Position: {index}, Fruit:
{fruit}")
```

Using enumerate with a conditional statement:

```
my_list = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(my_list):
    if index % 2 == 0:
        print(f"Even Index: {index},
Fruit: {fruit}")
```

While Loop

The `while` loop executes a block of code as long as a condition is true.

Syntax:

```
while condition:
    # Code to execute while the
condition is true
```

Example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

While loop with `else`:

```
count = 0
while count < 5:
    print(count)
    count += 1
else:
    print("Loop finished")
```

Using `break` to exit the loop:

```
count = 0
while True:
    print(count)
    count += 1
    if count >= 5:
        break
```

Using `continue` to skip to the next iteration:

```
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:
        continue
    print(count)
```

Break Statement

The `break` statement is used to exit a loop prematurely.

Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Break nested loop:

```
for i in range(3):
    for j in range(3):
        if i == 1 and j == 1:
            break # Breaks only the
inner loop
        print(f"i={i}, j={j}")
```

Continue Statement

The `continue` statement skips the rest of the current iteration and proceeds to the next iteration of the loop.

Example:

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Continue nested loop:

```
for i in range(3):
    for j in range(3):
        if i == 1 and j == 1:
            continue # Skips only the
inner loop iteration
        print(f"i={i}, j={j}")
```

Range Function

The `range()` function generates a sequence of numbers.

Syntax:

```
range(start, stop, step)
```

- `start`: The starting number (inclusive, default is 0).
- `stop`: The ending number (exclusive).
- `step`: The increment between numbers (default is 1).

Example:

```
for i in range(5):
    print(i)
```

Using `start` and `stop`:

```
for i in range(2, 7):
    print(i)
```

Using `start`, `stop`, and `step`:

```
for i in range(0, 10, 2):
    print(i)
```

Iterating in reverse:

```
for i in range(5, 0, -1):
    print(i)
```

Looping with Zip

The `zip()` function allows you to iterate over multiple sequences in parallel.

Syntax:

```
for item1, item2, ... in zip(sequence1,
sequence2, ...):
    # Code to execute with items from
each sequence
```

Example:

```
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 28]

for name, age in zip(names, ages):
    print(f"{name} is {age} years old.")
```

Zip with different length lists:

```
list1 = [1, 2, 3]
list2 = ['a', 'b']
for item1, item2 in zip(list1, list2):
    print(item1, item2)
```

For/Else Construct

The `else` block in a `for` loop is executed if the loop completes normally (i.e., without encountering a `break` statement).

Syntax:

```
for item in sequence:
    # Code to execute
else:
    # Code to execute if the loop
completes without break
```

Example:

```
my_list = [1, 2, 3, 4, 5]
for number in my_list:
    if number == 6:
        print("Found 6!")
        break
else:
    print("6 not found in the list.")
```

For/else example with break:

```
my_list = [1, 2, 3, 4, 5]
for number in my_list:
    if number == 3:
        print("Found 3!")
        break
else:
    print("This will not be printed.")
```

Python Functions Cheat Sheet

Basic Function Definition

A function is a block of code that performs a specific task. It's reusable and makes code more organized.

Syntax:

```
def function_name(parameters):
    """Docstring: Describes the function"""
    # Function body (code to be executed)
    return [expression] # Optional return statement
```

Example:

```
def greet(name):
    """This function greets the person passed in as a parameter."""
    print(f"Hello, {name}!")

# Calling the function
greet("Alice")  # Output: Hello, Alice!
```

Key Components:

- `def`: Keyword indicating the start of a function definition.
- `function_name`: A unique identifier for the function.
- `parameters`: Input values the function accepts (optional).
- `Docstring`: A string literal providing documentation (optional but recommended).
- `Function body`: The code block to be executed.
- `return`: Optional statement to return a value from the function.

Return Statement

The `return` statement exits a function and optionally returns a value to the caller.

Syntax:

```
return [expression]
```

If no expression is provided, `return` returns `None`.

```
def no_return():
    print("This function doesn't return anything.")
    return

result = no_return()
print(result)  # Output: None
```

Returning a Value:

```
def add(x, y):
    """Returns the sum of x and y."""
    return x + y

sum_result = add(5, 3)
print(sum_result)  # Output: 8
```

Positional Arguments

Positional arguments are passed to a function based on their order. The order matters!

Example:

```
def describe_person(name, age, city):
    """Describes a person's name, age, and city."""
    print(f"Name: {name}, Age: {age}, City: {city}")

describe_person("Bob", 30, "New York")
# Name: Bob, Age: 30, City: New York
```

If the arguments are not provided in the correct order, the result might be unexpected:

```
describe_person(30, "New York", "Bob")
# Name: 30, Age: New York, City: Bob
```

Keyword Arguments

Keyword arguments are passed to a function with the parameter name explicitly specified. Order doesn't matter!

Syntax:

```
function_name(parameter1=value1, parameter2=value2)
```

Example:

```
def describe_person(name, age, city):
    """Describes a person's name, age, and city."""
    print(f"Name: {name}, Age: {age}, City: {city}")

describe_person(age=30, city="New York", name="Bob")  # Name: Bob, Age: 30, City: New York
```

You can mix positional and keyword arguments, but positional arguments must come first.

```
describe_person("Bob", age=30, city="New York")  # Valid
# describe_person(name="Bob", 30, "New York")  # Invalid: positional argument after keyword argument
```

Returning Multiple Values

Python functions can return multiple values as a tuple.

Example:

```
def get_name_and_age():
    """Returns a name and an age."""
    name = "Charlie"
    age = 25
    return name, age

person_data = get_name_and_age()
print(person_data)  # Output: ('Charlie', 25)

#Unpacking the tuple
name, age = get_name_and_age()
print(f"Name: {name}, Age: {age}")  # Output: Name: Charlie, Age: 25
```

The returned values are packed into a tuple. You can then unpack the tuple into separate variables.

Default Argument Values

You can specify default values for function parameters. If the caller doesn't provide a value for that parameter, the default is used.

Syntax:

```
def
function_name(parameter=default_value):
    # Function body
```

Example:

```
def greet(name="Guest"):
    """Greets a person, or Guest if no
name is provided."""
    print(f"Hello, {name}!")

greet()  # Output: Hello, Guest!
greet("David")  # Output: Hello, David!
```

Default values are evaluated only once, at the point of function definition. Be careful when using mutable default arguments (like lists or dictionaries).

```
def append_to_list(value, my_list=[]):
    my_list.append(value)
    return my_list

print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [1,
2] (Unexpected!)
```

To avoid this, use `None` as the default and create a new list inside the function if the argument is `None`.

```
def append_to_list(value, my_list=None):
    if my_list is None:
        my_list = []
    my_list.append(value)
    return my_list

print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [2]
(Correct!)
```

Anonymous Functions (Lambda Expressions)

Lambda expressions are small, anonymous functions defined using the `lambda` keyword. They can have any number of arguments but only one expression.

Syntax:

```
lambda arguments: expression
```

Example:

```
# A lambda function that adds two
numbers
add = lambda x, y: x + y

result = add(3, 5)
print(result)  # Output: 8
```

Lambda functions are often used with functions like `map()`, `filter()`, and `sorted()`.

```
numbers = [1, 2, 3, 4, 5]

# Square each number using map and a
lambda function
squared_numbers = list(map(lambda x:
x**2, numbers))
print(squared_numbers)  # Output: [1, 4,
9, 16, 25]

# Filter even numbers using filter and a
lambda function
even_numbers = list(filter(lambda x: x %
2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]
```

Variable Scope

The scope of a variable refers to the region of the code where the variable is accessible.

- **Local Scope:** Variables defined inside a function have local scope. They are only accessible within that function.
- **Global Scope:** Variables defined outside of any function have global scope. They are accessible from anywhere in the code.

Example:

```
global_var = 10  # Global variable

def my_function():
    local_var = 5  # Local variable
    print(f"Inside function: global_var
= {global_var}")
    print(f"Inside function: local_var =
{local_var}")

my_function()
print(f"Outside function: global_var =
{global_var}")
# print(f"Outside function: local_var =
{local_var}")  # Error: local_var is not
defined
```

To modify a global variable from within a function, you need to use the `global` keyword.

```
global_var = 10

def modify_global():
    global global_var
    global_var = 20

modify_global()
print(f"Global variable after
modification: {global_var}")  # Output:
Global variable after modification: 20
```

Python Modules Cheat Sheet

Importing Modules

The most basic way to access code from another module is to do a simple import statement.

```
import module_name
```

This imports the module as a whole, and you need to use the module name to access its contents.

Accessing functions/attributes after importing a module.

```
import math

x = math.sqrt(25)  # Accessing the sqrt function
print(x)
```

Importing with an alias (renaming the module).

```
import module_name as alias
```

This allows you to use a shorter or more descriptive name for the module in your code.

```
import numpy as np

arr = np.array([1, 2, 3]) # Using the alias 'np'
print(arr)
```

Importing a submodule

```
import package.module

#Example
import os.path

file_exists = os.path.exists('my_file.txt')
print(file_exists)
```

Importing Specific Items From a Module

You can import specific functions or attributes from a module using the `from ... import ...` syntax.

```
from module_name import item_name
```

This imports only the specified item, making it directly accessible without needing to use the module name.

Importing multiple items from a module.

```
from module_name import item1, item2, item3
```

This can make your code more concise when you only need a few specific items from a module.

```
from math import sqrt, pi

radius = 5
area = pi * sqrt(radius)
print(area)
```

Importing an item with an alias.

```
from module_name import item_name as alias
```

This is useful for avoiding naming conflicts or using a more descriptive name for the imported item.

```
from datetime import datetime as dt

now = dt.now()
print(now)
```

Importing Everything From a Module

You can import all names from a module's symbol table using the `from ... import *` syntax.

```
from module_name import *
```

Warning: This is generally discouraged as it can lead to naming conflicts and make your code harder to understand.

Example of importing all from a module (use with caution!).

```
from math import *

x = sqrt(25)  # Directly using sqrt
print(x)
```

It is recommended to explicitly import the names you need to avoid potential issues.

Module Search Path

When you import a module, Python searches for it in a specific order:

1. The current directory.
2. `PYTHONPATH` environment variable (if set).
3. Installation-dependent default directory.

You can view the module search path using the `sys` module.

```
import sys
print(sys.path)
```

This will print a list of directories where Python looks for modules.

Modifying the module search path (use with caution!).

```
import sys
sys.path.append('/path/to/your/module')
```

This allows you to import modules from custom locations, but should be done carefully to avoid conflicts.

Functions and Attributes

`dir(module)` — Returns a list of valid attributes for that object.

```
import math
print(dir(math))
```

`__name__` — Every Python module has a special attribute called `__name__`. When the module is run as the main program, `__name__` is set to `'__main__'`. Otherwise, it is set to the module's name.

```
# my_module.py
print(__name__)

# When run as a script:
# python my_module.py  # Output: __main__

# When imported:
# import my_module  # Output: my_module
```

`help(module.function)` — Display the documentation for a function.

```
import math
help(math.sqrt)
```

Packages

A package is a way of organizing related modules into a directory hierarchy. Each package directory contains a special file, `__init__.py`, which can be empty or contain initialization code for the package.

```
my_package/
    __init__.py
    module1.py
    module2.py
```

Importing from a package.

```
import my_package.module1

my_package.module1.my_function()
```

Or:

```
from my_package import module1

module1.my_function()
```

Using `__init__.py` to define package-level imports.

```
# my_package/__init__.py
from .module1 import my_function
```

Then:

```
from my_package import my_function

my_function()
```

Python Classes & Inheritance Cheat Sheet

Defining Classes

A class is a blueprint for creating objects (instances). It defines attributes (data) and methods (behavior).

```
class MyClass:
    # Class attributes (variables)
    class_variable = "Shared among all
instances"

    # Constructor (initializer)
    def __init__(self,
instance_variable):
        # Instance attributes
        self.instance_variable =
instance_variable

    # Method
    def my_method(self):
        return f"Instance variable:
{self.instance_variable}"
```

Example:

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return "Woof!"

# Creating an instance of the Dog class
milo = Dog("Milo", 3)

print(milo.name)    # Output: Milo
print(milo.age)     # Output: 3
print(milo.species) # Output: Canis
familiaris
print(milo.bark())  # Output: Woof!
```

Constructors (__init__)

The `__init__` method is a special method called the constructor. It's automatically called when an object is created from the class.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is
{self.name} and I am {self.age} years
old.")

# Creating an object of the Person class
person1 = Person("Alice", 30)
person1.greet()  # Output: Hello, my
name is Alice and I am 30 years old.
```

Key points about `__init__`:
* It must have at least one parameter, `self`, which refers to the instance being created.
* It can have other parameters to receive initial values for the instance's attributes.
* It's used to initialize the object's state.

Methods

Methods are functions defined within a class that operate on the object's data.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width +
self.height)

rect = Rectangle(5, 10)
print(rect.area())      # Output: 50
print(rect.perimeter()) # Output: 30
```

Key points about methods:

- They must have `self` as the first parameter, which refers to the instance.
- They can access and modify the object's attributes.
- They define the object's behavior.

Class Variables

Class variables are variables that are shared by all instances of a class.

```
class Circle:
    pi = 3.14159  # Class variable

    def __init__(self, radius):
        self.radius = radius  # Instance
variable

    def area(self):
        return Circle.pi *
self.radius**2  # Accessing class
variable

circle1 = Circle(5)
circle2 = Circle(10)

print(circle1.area()) # Output: 78.53975
print(circle2.area()) # Output: 314.159
```

Key points about class variables:

- They are defined outside of the `__init__` method.
- They are accessed using the class name (e.g., `ClassName.variable_name`).
- Changes to a class variable affect all instances of the class.

The `super()` Function

The `super()` function is used to call a method from a parent class.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Generic animal sound"

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Calling the parent's constructor
        self.breed = breed

    def speak(self):
        return "Woof!"

dog = Dog("Buddy", "Golden Retriever")
print(dog.name)  # Output: Buddy
print(dog.breed) # Output: Golden Retriever
print(dog.speak()) # Output: Woof!
```

Key points about `super()`:

- It simplifies calling methods from parent classes, especially in multiple inheritance scenarios.
- It automatically passes the instance (`self`) to the parent's method.

The `__repr__()` Method

The `__repr__` method is a special method used to represent an object as a string. It's used for debugging and logging.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

p = Point(2, 3)
print(p)        # Output: Point(x=2, y=3)
print(repr(p))   # Output: Point(x=2, y=3)
```

Key points about `__repr__`:

- It should return a string that can be used to recreate the object.
- If `__str__` is not defined, `__repr__` is used when `str()` is called on the object.

User-Defined Exceptions

You can create your own exception classes by inheriting from the built-in `Exception` class.

```
class CustomError(Exception):
    """Base class for other exceptions"""
    pass

class InputError(CustomError):
    """Exception raised for errors in the input."""
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

try:
    raise InputError("demo", "Invalid input!")
except InputError as e:
    print(f"Error: {e.message}")
```

Key points about user-defined exceptions:

- They allow you to handle specific error conditions in your code.
- They make your code more readable and maintainable.

Polymorphism

Polymorphism means "many forms". In object-oriented programming, it refers to the ability of a single interface to represent different types.

```
class Cat:
    def speak(self):
        return "Meow!"

class Dog:
    def speak(self):
        return "Woof!"


def animal_sound(animal):
    return animal.speak()

cat = Cat()
dog = Dog()

print(animal_sound(cat)) # Output: Meow!
print(animal_sound(dog)) # Output: Woof!
```

Key points about polymorphism:

- It allows you to write code that can work with objects of different classes in a uniform way.
- It promotes code reusability and flexibility.

Overriding

Overriding is the ability of a subclass to provide a specific implementation of a method that is already defined in its superclass.

```
class Animal:
    def speak(self):
        return "Generic animal sound"

class Dog(Animal):
    def speak(self):
        return "Woof!"

animal = Animal()
dog = Dog()

print(animal.speak()) # Output: Generic animal sound
print(dog.speak())    # Output: Woof!
```

Key points about overriding:

- The method in the subclass must have the same name, parameters, and return type as the method in the superclass.
- It allows you to customize the behavior of a subclass without modifying the superclass.

Inheritance

Inheritance is a mechanism by which a new class (subclass) can be created from an existing class (superclass), inheriting its attributes and methods.

```
class Vehicle:
    def __init__(self, model, color):
        self.model = model
        self.color = color

    def description(self):
        return f"{self.color}
{self.model}"

class Car(Vehicle):
    def __init__(self, model, color,
num_doors):
        super().__init__(model, color)
        self.num_doors = num_doors

    def description(self):
        return f"
{super().description()},
{self.num_doors} doors"

my_car = Car("Sedan", "Red", 4)
print(my_car.description()) # Output:
Red Sedan, 4 doors
```

Key points about inheritance:

- It promotes code reusability by allowing you to reuse existing code.
- It creates a hierarchy of classes, making your code more organized and maintainable.
- Subclasses can add new attributes and methods or override existing ones.

Python File Handling Cheat Sheet

Opening Files

`open(filename, mode)`

Opens a file. `filename` is the file path (string). `mode` specifies the file opening mode (string).

Common Modes:

- `'r'`: Read (default). Opens the file for reading. Returns error if the file does not exist.
- `'w'`: Write. Opens the file for writing. Creates a new file if it does not exist or overwrites the file if it exists.
- `'a'`: Append. Opens the file for appending. Creates a new file if it does not exist.
- `'x'`: Create. Creates a new file. Returns an error if the file exists.
- `'b'`: Binary. Opens the file in binary mode.
- `'t'`: Text (default). Opens the file in text mode.
- `'+'`: Update. Opens the file for updating (reading and writing).

Example (Read):

```python
f = open('my_file.txt', 'r')
content = f.read()
f.close()
print(content)
```

Example (Write):

```python
f = open('my_file.txt', 'w')
f.write('Hello, world!')
f.close()
```

Reading Files

`f.read(size)`

Reads at most `size` characters/bytes from the file. If `size` is omitted or negative, reads the entire file.

`f.readline()`

Reads a single line from the file (including the newline character).

`f.readlines()`

Reads all lines from the file and returns them as a list of strings.

Example (Read lines):

```python
f = open('my_file.txt', 'r')
for line in f:
    print(line.strip())
f.close()
```

Writing to Files

`f.write(string)`

Writes the contents of `string` to the file. Returns the number of characters written.

`f.writelines(list_of_strings)`

Writes a list of strings to the file.

Example (Write lines):

```python
f = open('my_file.txt', 'w')
lines = ['Line 1\n', 'Line 2\n']
f.writelines(lines)
f.close()
```

Closing Files

`f.close()`

Closes the file. It is important to close files to free up system resources and ensure that all data is written to disk.

`with open(filename, mode) as f:`

Using the `with` statement automatically closes the file, even if errors occur. This is the recommended approach.

Example (with statement):

```python
with open('my_file.txt', 'r') as f:
    content = f.read()
    print(content)
# File is automatically closed here
```

File Object Attributes

`f.closed`	Returns `True` if the file is closed, `False` otherwise.
`f.mode`	Returns the file opening mode.
`f.name`	Returns the name of the file.

Deleting Files

`import os`
`os.remove(filename)`

Deletes the file specified by `filename`. You must first import the `os` module.

Example:

```python
import os

file_path = 'my_file.txt'

if os.path.exists(file_path):
    os.remove(file_path)
    print(f'{file_path} deleted successfully!')
else:
    print(f'{file_path} does not exist!')
```

Checking if a File Exists

`import os`
`os.path.exists(filename)`

Checks if the file specified by `filename` exists. Returns `True` if it exists, `False` otherwise.

Example:

```python
import os

file_path = 'my_file.txt'

if os.path.exists(file_path):
    print(f'{file_path} exists!')
else:
    print(f'{file_path} does not exist!')
```

Deleting Folders (Directories)

`import os`
`os.rmdir(dirname)`

Deletes the directory specified by `dirname`. The directory must be empty.

`shutil.rmtree(dirname)` - To delete a directory and all its contents, use `shutil.rmtree()`.

Example (Empty Directory):

```
import os

dir_path = 'my_directory'

if os.path.exists(dir_path):
    os.rmdir(dir_path)
    print(f'{dir_path} deleted
successfully!')
else:
    print(f'{dir_path} does not exist!')
```

Example (Non-Empty Directory):

```
import shutil
import os

dir_path = 'my_directory'

if os.path.exists(dir_path):
    shutil.rmtree(dir_path)
    print(f'{dir_path} and its contents
deleted successfully!')
else:
    print(f'{dir_path} does not exist!')
```

Handling Exceptions

`try: # Code that might raise an exception except FileNotFoundError: # Handle the exception finally: # Code that always executes`

Use `try...except` blocks to handle file-related exceptions, such as `FileNotFoundError` or `IOError`. The `finally` block is executed regardless of whether an exception occurred.

Example:

```
try:
    f = open('nonexistent_file.txt',
'r')
    content = f.read()
    print(content)
except FileNotFoundError:
    print('File not found!')
finally:
    try:
        f.close()
    except NameError:
        pass # File was never opened
```

Python Miscellaneous

Comments

Single-line comments start with `#`.

```
# This is a single-line comment
```

Multi-line comments (docstrings) are enclosed in triple quotes (`'''` or `"""`).

```
'''
This is a multi-line comment.
It can span multiple lines.
'''

"""
This is also a multi-line comment.
"""
```

Docstrings are used to document functions, classes, modules, and methods.

```
def my_function():
    """This is a docstring for
my_function."""
    pass
```

Accessing docstrings: Use `help(object)` or `object.__doc__`.

```
help(my_function)
print(my_function.__doc__)
```

Comments should be clear and concise, explaining the purpose of the code.

Generators

Generators are functions that use the `yield` keyword to return a sequence of values lazily.

Instead of returning all values at once, generators produce them one at a time, saving memory.

Example:

```
def my_generator(n):
    for i in range(n):
        yield i

# Usage
for num in my_generator(5):
    print(num)
```

Generator expressions are a concise way to create generators using a syntax similar to list comprehensions.

```
my_generator = (i for i in range(5))

for num in my_generator:
    print(num)
```

Generators are iterable, meaning you can use them in `for` loops or with functions like `next()`.

The `next()` function retrieves the next item from the generator. When the generator is exhausted, it raises `StopIteration`.

```
g = my_generator(3)
print(next(g))
print(next(g))
print(next(g))
# print(next(g))  # Raises StopIteration
```

Generator to List

You can convert a generator to a list using the `list()` constructor. This will consume the entire generator and store the results in a list.

Example:

```
def my_generator(n):
    for i in range(n):
        yield i

my_list = list(my_generator(5))
print(my_list)
```

Converting a generator to a list requires storing all elements in memory, so be mindful of memory usage for large generators.

Alternative: Process the generator's output iteratively instead of converting it to a list if memory is a concern.

Combining `list()` with generator expressions:

```
my_list = list(i * 2 for i in range(5))
print(my_list)
```

Handle Exceptions

Use `try` and `except` blocks to handle exceptions.

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError as e:
    # Code to handle the exception
    print(f"Error: {e}")
```

You can handle multiple exception types in a single `try` block.

```
try:
    # Code that may raise multiple
exceptions
    value = int(input("Enter a number:
"))
    result = 10 / value
except ZeroDivisionError as e:
    print(f"Division error: {e}")
except ValueError as e:
    print(f"Value error: {e}")
```

The `else` block executes if no exceptions are raised in the `try` block.

```
try:
    result = 10 / 2
except ZeroDivisionError as e:
    print(f"Error: {e}")
else:
    print(f"Result: {result}")
```

The `finally` block always executes, regardless of whether an exception was raised or not.

```
try:
    f = open("myfile.txt", "r")
    content = f.read()
except FileNotFoundError as e:
    print(f"File not found: {e}")
finally:
    if 'f' in locals():
        f.close()
```

Raise exceptions using the `raise` keyword.

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be
negative")
    print("Age is valid")

try:
    validate_age(-1)
except ValueError as e:
    print(f"Error: {e}")
```

List Comprehensions

List comprehensions provide a concise way to create lists based on existing iterables.

Syntax: `[expression for item in iterable if condition]`

Example: Creating a list of squares.

```
squares = [x**2 for x in range(10)]
print(squares)
```

Example: Filtering even numbers.

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)
```

List comprehensions can also be used with nested loops.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]
print(flattened)
```

List comprehensions are more readable and often faster than traditional loops for creating lists.

Lambda Functions

Lambda functions are anonymous, small, and inline functions defined using the `lambda` keyword.

Syntax: `lambda arguments: expression`

Example: A lambda function that adds two numbers.

```
add = lambda x, y: x + y
print(add(5, 3))
```

Lambda functions are often used with functions like `map()`, `filter()`, and `sorted()`.

Example: Using `map()` with a lambda function to square each element in a list.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)
```

Example: Using `filter()` with a lambda function to filter even numbers.

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

Lambda functions can only contain a single expression and cannot include statements or annotations.