Python Cheatsheet

1. Basic Syntax and Data Types

Showing Output to the User

```
print("Content that you want to print on screen")

var1 = "Shruti"
print("Hi, my name is:", var1)
```

Taking Input from the User

```
var1 = input("Enter your name: ")
print("My name is:", var1)

# Type casting for other data types
var2 = int(input("Enter an integer value: "))
print(var2)

var3 = float(input("Enter a float value: "))
print(var3)
```

Variables and Assignment

```
# Variable naming conventions
my_variable = 42  # Snake case recommended
camelCase = "Not preferred in Python"
CONSTANT_VALUE = 3.14159  # All uppercase for constants

# Multiple assignment
x, y, z = 1, 2, 3

# Type conversion
integer_value = int("123")
float_value = float(42)
string_value = str(3.14)
```

Type Checking

```
x = 42
print(type(x))  # <class 'int'>
isinstance(x, int)  # True
```

Data Types

Python supports several core data types:

1. **Numeric Types**
   - `int`: Whole numbers (e.g., 10, -5, 0)
   - `float`: Decimal numbers (e.g., 3.14, -0.5)
   - `complex`: Complex numbers (e.g., 3+4j)
2. **Sequence Types**
   - `list`: Mutable, ordered collection
   - `tuple`: Immutable, ordered collection
   - `range`: Sequence of numbers
3. **Text Type**
   - `str`: String, sequence of Unicode characters
4. **Mapping Type**
   - `dict`: Key-value pairs
5. **Set Types**
   - `set`: Unordered collection of unique elements
   - `frozenset`: Immutable version of set

Operators

Python provides a rich set of operators for manipulating data:

- **Arithmetic Operators:** `+`, `-`, `*`, `/`, `%` (modulo), `**` (exponentiation), `//` (floor division)
- **Comparison Operators:** `==` (equal to), `!=` (not equal to), `>`, `<`, `>=`, `<=`
- **Logical Operators:** `and`, `or`, `not`
- **Assignment Operators:** `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `//=`
- **Bitwise Operators:** `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (left shift), `>>` (right shift)
- **Membership Operators:** `in`, `not in`
- **Identity Operators:** `is`, `is not`

2. Control Flow

Control flow statements dictate the execution path of your program:

Conditional Statements

```
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition2 is True
else:
    # Code to execute if all conditions are False

# Example
a = 15
b = 20
c = 12
if (a > b and a > c):
    print(a, "is greatest")
elif (b > c and b > a):
    print(b, " is greatest")
else:
    print(c, "is greatest")
```

Loops

`for` loop: Iterates over a sequence (e.g., list, tuple, string).

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Example with range() function
for i in range(1, 101, 1):
    print(i)
```

`range` function: `range(start, stop, step)` generates a sequence of numbers.

```
# Display all even numbers between 1 to 100
for i in range(0, 101, 2):
    print(i)
```

`while` loop: Repeats a block of code as long as a condition is True.

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

# Example
i = 1
while (i <= 100):
    print(i)
    i = i + 1
```

Loop Control Statements

`break`: Terminates the loop prematurely.

```
for i in range(1, 101, 1):
    print(i, end=" ")
    if (i == 50):
        break
    else:
        print("Mississippi")
print("Thank you")
```

`continue`: Skips the current iteration and proceeds to the next.

```
for i in [2, 3, 4, 6, 8, 0]:
    if (i % 2 != 0):
        continue
    print(i)
```

`pass`: Acts as a placeholder, doing nothing.

Indentation: Python uses indentation to define code blocks. Consistent indentation is crucial for proper execution.

3. Functions

```
def greet(name, greeting="Hello"):
    """Docstring describing function purpose"""
    return f"{greeting}, {name}!"

# Default arguments
def power(base, exponent=2):
    return base ** exponent

# Variable-length arguments
def sum_all(*args):
    return sum(args)

# Keyword arguments
def user_profile(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

Lambda Functions

```
# Inline, anonymous functions
multiply = lambda x, y: x * y
print(multiply(3, 4))  # 12
```

4. Data Structures

Python has flexible data formats that help you organize and handle data easily.

Lists

```
my_list = [1, 2, 3, "apple", True]
my_list.append(4)  # Add an item
my_list.remove("apple")  # Remove an item

# List methods
my_list.index(2)  # Returns the index of the first occurrence of 2
my_list.extend([5, 6])  # Extends the list with elements from another
iterable
my_list.insert(1, "orange")  # Inserts "orange" at index 1
my_list.pop(3)  # Removes and returns the element at index 3
my_list.clear()  # Removes all elements from the list
my_list.count(2)  # Returns the number of times 2 appears in the list
my_list.reverse()  # Reverses the order of the list
my_list.sort()  # Sorts the list in ascending order
```

Tuples

```
my_tuple = (1, 2, 3, "apple", True)

# Tuple methods
my_tuple.count(2)  # Returns the number of times 2 appears in the tuple
my_tuple.index("apple")  # Returns the index of the first occurrence of "apple"
```

Dictionaries

```
my_dict = {"name": "Alice", "age": 30, "city": "New York"}
my_dict["age"] = 31  # Update a value
my_dict["country"] = "USA"  # Add a key-value pair

# Dictionary functions and methods
len(my_dict)  # Returns the number of key-value pairs
my_dict.clear()  # Removes all items from the dictionary
my_dict.get("name")  # Returns the value associated with the key "name"
my_dict.items()  # Returns a view object with key-value pairs
my_dict.keys()  # Returns a view object with the keys
my_dict.values()  # Returns a view object with the values
my_dict.update({"country": "Canada"})  # Updates the dictionary with new key-value pairs
```

Sets

```
my_set = {1, 2, 3, 3, 4}  # {1, 2, 3, 4}
my_set.add(5)  # Add an item
my_set.remove(3)  # Remove an item

# Set methods
my_set.discard(2)  # Removes 2 from the set if it exists
my_set.intersection({3, 4, 5})  # Returns a new set with elements common to both sets
my_set.issubset({1, 2, 3, 4, 5})  # Returns True if my_set is a subset of the other set
my_set.pop()  # Removes and returns an arbitrary element from the set
my_set.union({6, 7})  # Returns a new set with elements from both sets
```

5. Strings

Strings are sequences of characters and are immutable in Python:

- **String Literals:** Define strings using single (`'`), double (`"`), or triple (`'''` or `"""`) quotes.

- **String Formatting:**

    - **Old Style:** `%` operator
    - **New Style:** `.format()` method
    - **f-strings:** Introduced in Python 3.6
- **String Methods:** Python provides a plethora of built-in string methods for manipulation:

    - `lower()`, `upper()`
    - `strip()`, `lstrip()`, `rstrip()`
    - `split()`, `join()`
    - `replace()`
    - `find()`, `index()`
    - `isalnum()`, `isalpha()`, `isdecimal()`, `isdigit()`, `islower()`, `isspace()`, `isupper()`
    - and many more...
- **Escape Sequences:** Special characters used to represent certain formatting actions within strings.

```
print("\n")  # Newline
print("\\")  # Backslash
print("\'")  # Single quote
print("\t")  # Tab
print("\b")  # Backspace
print("\ooo")  # Octal value
print("\xhh")  # Hex value
print("\r")  # Carriage return
```

Indexing and Slicing: Access individual characters or substrings using indices.

```
my_string = "Hello, world!"
print(my_string[0])  # Output: H
print(my_string[1:5])  # Output: ello
```

6. File Handling

Reading and Writing

```python
# Reading
with open('file.txt', 'r') as file:
    content = file.read()
    lines = file.readlines()

# Writing
with open('output.txt', 'w') as file:
    file.write("Hello, world!")

# Appending
with open('log.txt', 'a') as file:
    file.write("New log entry\n")
```

7. Exception Handling

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
except Exception as e:
    print(f"An error occurred: {e}")
else:
    print("No exceptions")
finally:
    print("Always executed")

# Raising exceptions
if x < 0:
    raise ValueError("x must be non-negative")
```

8. Modules and Packages

```python
# Importing entire module
import math
```

```python
print(math.pi)

# Importing specific functions
from datetime import datetime, timedelta

# Aliasing
import numpy as np
```

9. Advanced Concepts

Decorators

```python
def logging_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

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

Generator Functions

```python
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Using generator
for num in fibonacci(10):
    print(num)
```

Lambda Functions

```python
add = lambda x, y: x + y
result = add(5, 3)  # result = 8
```

Context Managers

```python
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()

# Usage
with FileManager('example.txt', 'w') as f:
    f.write('Hello, world!')
```

List Comprehensions

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

10. Object-Oriented Programming

Here are the key OOP terms in Python:

1. Function
2. Arguments
3. Parameters
4. Methods
5. Attributes
6. Encapsulation
7. Abstraction
8. Composition
9. Inheritance
10. Aggregation
11. Override Methods
12. Polymorphism

Function

```python
# Function
def greet(name):
    print(f"Hello, {name}!")

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

Arguments and Parameters

```python
# Function with parameters
def greet(name, age):  # 'name', 'age' is a parameter
    print(f"Hello, {name}! You are {age} years old.")

# Calling
greet("Asad", 25) # 'Asad', 25 is an argument

#Output: Hello, Asad! You are 25 years old.
```

Methods and Attributes

Methods

```python
class Person:
    def greet(self, name):  # Method
        print(f"Hello, {name}!")

person = Person()
person.greet("Asad")  # Output: Hello, Asad!
```

Attributes

```python
class Vehicle:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        self.mileage = 0  # instance attribute


    num_wheels = 4  # class attribute


# Creating an object
car = Vehicle("Toyota", "Corolla", 2015)
print(car.brand, car.model, car.year)  # Output: Toyota Corolla 2015
print(car.num_wheels)  # Output: 4
```

Encapsulation

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # Private attribute
        self.__age = age  # Private attribute

    def get_name(self):  # Getter method
        return self.__name

    def get_age(self):  # Getter method
        return self.__age

person = Person("Asad", 25)
print(person.get_name())  # Output: Asad
print(person.get_age())  # Output: 25
```

Abstraction

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Abstracted data

    def deposit(self, amount):
        self.__balance += amount  # Abstracted control

    def get_balance(self):
        return self.__balance  # Abstracted interface
```

Composition

```
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()  # Composition

    def start_car(self):
        self.engine.start()  # Using contained object

my_car = Car()
my_car.start_car()  # Output: Engine started
```

Inheritance

```
class Animal:
    def sound(self): # Method
        print('Generic sound')

class Dog(Animal): # "Inheritance" Dog class inherit from Animal class
    def sound(self): # overriding the sound method
        print('Bark')

dog = Dog()
dog.sound()
```

Aggregation

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

class Library:
    def __init__(self):
        self.books = [] # aggregation

    def add_book(self, book):
        self.books.append(book)
        print(f'{book.title} by {book.author} added to the library')

book1 = Book('The Catcher in the Rye', 'J.D. Salinger')
library = Library()
library.add_book(book1)

book2 = Book('To Kill a Mockingbird', 'Harper Lee')
library.add_book(book2)
```

Override Methods

```
class Animal:
    def sound(self):
        print('Generic sound')

class Dog(Animal):
    def sound(self): # overriding the sound method
        print('Bark')

dog = Dog()
dog.sound()
```

Polymorphism

```
# Polymorphism -> Present the same interface for different data types.

class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self): # Polymorphic method
        return (f'area of the circle is {3.14 * self.radius ** 2}')

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self): # Polymorphic method
        return (f'area of square is {self.side ** 2}')

shapes = [Circle(5), Square(5)]
for shape in shapes:
    print(shape.area())

# Output: area of the circle is 78.5
         # area of square is 25
```

11. Standard Library Highlights

Collections Module

```
from collections import Counter, defaultdict, namedtuple

# Counting elements
word_counts = Counter(['apple', 'banana', 'apple'])

# Default dictionary
word_count = defaultdict(int)

# Named tuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
```

Datetime Module

```
from datetime import datetime, timedelta

now = datetime.now()
future_date = now + timedelta(days=30)
```

Random Module

```python
import random

# Random selection
random_item = random.choice([1, 2, 3, 4, 5])
random_sample = random.sample(range(100), 5)
```