Python Fundamentals Cheat Sheet

Introduction Python is a versatile, high-level programming language known for its readability and ease of use. This cheat sheet covers the fundamental concepts that are essential for anyone starting with Python.

1. Introduction

1.1 What is Python?

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace.

1.2 Setting Up Python Environment ¶

1. Download Python from python.org (https://www.python.org/).
2. Install Python following the instructions for your operating system.
3. Verify the installation by typing `python --version` in your command line.

1.3 Running Python Code

You can run Python code in several ways:

- **Interactive Mode**: Open a terminal and type `python` .
- **Script Mode**: Write your code in a `.py` file and run it using `python filename.py` .
- **Integrated Development Environment (IDE)**: Use an IDE like PyCharm, VSCode, or Jupyter Notebook for more features and better code management.

2. Variables and Data Types

2.1 Variables

Variables are containers for storing data values. In Python, variables are created when you assign a value to them.

In [36]:
```
# Creating variables
x = 10           # Integer
y = 20.5         # Float
name = 'Ashish'  # String
is_active = True # Boolean
```

2.2 Data Types

Python supports several data types including integers, floats, strings, and booleans.

In [37]:
```
# Data Types examples
x = 10
y = 20.5
name = 'Ashish'
is_active = True
```

2.3 Type Conversion

You can convert between different data types using functions like `int()`, `float()`, `str()`, etc.

In [38]:
```
# Converting between types
int_to_float = float(10)  # Output: 10.0
float_to_int = int(20.5)  # Output: 20
int_to_str = str(10)      # Output: '10'
str_to_int = int('10')    # Output: 10
```

3. Basic Operations

3.1 Arithmetic Operations

Python supports the following arithmetic operations:

In [39]:
```
a = 10
b = 3

print(a + b)  # Addition: 13
print(a - b)  # Subtraction: 7
print(a * b)  # Multiplication: 30
print(a / b)  # Division: 3.3333...
print(a % b)  # Modulus: 1
print(a ** b) # Exponentiation: 1000
```

```
13
7
30
3.3333333333333335
1
1000
```

3.2 Assignment Operations

In [40]:
```
x = 5
x += 3  # Equivalent to x = x + 3
x -= 2  # Equivalent to x = x - 2
x *= 4  # Equivalent to x = x * 4
x /= 2  # Equivalent to x = x / 2
```

3.3 Comparison Operations

In [41]:
```
a = 5
b = 10

print(a == b)  # Equal: False
print(a != b)  # Not equal: True
print(a > b)   # Greater than: False
print(a < b)   # Less than: True
```

```
False
True
False
True
```

3.4 Logical Operations

In [42]:
```
is_mentor = True
is_student = False

print(is_mentor and is_student) # Logical AND: False
print(is_mentor or is_student)  # Logical OR: True
print(not is_mentor)            # Logical NOT: False
```

```
False
True
False
```

4. Control Flow

4.1 If-Else Statements

Control flow statements in Python allow you to execute code based on conditions.

In [43]:
```
age = 18

if age >= 18:
    print('You are an adult.')
else:
    print('You are a minor.')
```

```
You are an adult.
```

4.2 For Loops

In [44]:
```
# Looping through a list
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)

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

```
apple
banana
cherry
0
1
2
3
4
```

4.3 While Loops

In [45]:
```
count = 0

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

```
0
1
2
3
4
```

4.4 Break and Continue

In [46]:
```
# Break statement
for i in range(10):
    if i == 5:
        break
    print(i)

# Continue statement
for i in range(10):
    if i == 5:
        continue
    print(i)
```

```
0
1
2
3
4
0
1
2
3
4
6
7
8
9
```

5. Data Structures

Python provides various data structures to store collections of data.

5.1 Lists

Lists are used to store multiple items in a single variable.

In [47]:
```python
# Creating a list
fruits = ['apple', 'banana', 'cherry']

# Accessing elements
print(fruits[0])  # Output: apple

# Adding elements
fruits.append('orange')

# Removing elements
fruits.remove('banana')

# List comprehension
squares = [x**2 for x in range(10)]
```

```
apple
```

5.2 Dictionaries

In [48]:
```python
# Creating a dictionary
person = {'name': 'Ashish', 'age': 25, 'city': 'Delhi'}

# Accessing values
print(person['name'])  # Output: Ashish

# Adding key-value pairs
person['email'] = 'ashish@example.com'

# Removing key-value pairs
del person['age']
```

```
Ashish
```

5.3 Tuples

In [49]:
```python
# Creating a tuple
coordinates = (10, 20)

# Accessing elements
print(coordinates[0])  # Output: 10

# Tuples are immutable
# coordinates[0] = 15  # This will raise an error
```

```
10
```

5.4 Sets

In [50]:
```
# Creating a set
unique_numbers = {1, 2, 3, 4, 5}

# Adding elements
unique_numbers.add(6)

# Removing elements
unique_numbers.remove(3)
```

6. Functions

Functions in Python are defined using the `def` keyword. They allow you to encapsulate code for reuse and better organization.

6.1 Defining and Calling Functions

In [51]:
```
# Defining a function
def greet(name):
    return f'Hello, {name}!'

# Calling a function
print(greet('Ashish'))  # Output: Hello, Ashish!
```

```
Hello, Ashish!
```

6.2 Lambda Functions

In [52]:
```
# Lambda function to add two numbers
add = lambda x, y: x + y
print(add(5, 3))  # Output: 8
```

```
8
```

6.3 Decorators

In [53]:
```
# Defining a simple decorator
def debug(func):
    def wrapper(*args, **kwargs):
        print(f'Calling {func.__name__} with {args} and {kwargs}')
        result = func(*args, **kwargs)
        print(f'{func.__name__} returned {result}')
        return result
    return wrapper

@debug
def multiply(a, b):
    return a * b

print(multiply(3, 4))  # Output: Calling multiply with (3, 4) and {
```

```
Calling multiply with (3, 4) and {}
multiply returned 12
12
```

7. File Handling

Python provides built-in functions to read from and write to files. This is essential for data persistence and manipulation.

7.1 Writing to a File

In [54]:
```
with open('file.txt', 'w') as file:
    file.write('Hello, world!')
```

7.2 Reading a File

In [55]:
```
with open('file.txt', 'r') as file:
    content = file.read()
    print(content)
```

```
Hello, world!
```

7.3 Appending to a File

In [56]:
```
with open('file.txt', 'a') as file:
    file.write('\nAppended text.')
```

8. Error Handling

Python uses try-except blocks to handle exceptions and errors gracefully. This ensures your program can handle unexpected situations without crashing.

8.1 Try-Except Block

In [57]:
```
try:
    x = 10 / 0
except ZeroDivisionError:
    print('Cannot divide by zero!')
```

```
Cannot divide by zero!
```

8.2 Custom Exception

In [58]:
```
class CustomError(Exception):
    pass

try:
    raise CustomError('This is a custom error')
except CustomError as e:
    print(e)  # Output: This is a custom error
```

```
This is a custom error
```

9. Libraries and Packages

Python's standard library and third-party packages extend its capabilities. Using libraries and packages allows you to leverage pre-written code for various tasks.

9.1 Importing Libraries

In [59]:
```
import math

# Using math library
print(math.sqrt(16))  # Output: 4.0
```

```
4.0
```

9.2 Installing Packages

In [60]:
```
# Using pip to install a package
!pip install numpy
```

```
Requirement already satisfied: numpy in /Users/ashishzangra/opt/an
aconda3/lib/python3.9/site-packages (1.21.5)
```

9.3 Using Packages

In [61]:
```
import numpy as np

# Creating an array
arr = np.array([1, 2, 3, 4])
print(arr)

# Basic array operations
print(arr + 5)  # Output: [ 6  7  8  9]
```

```
[1 2 3 4]
[6 7 8 9]
```

10. Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of 'objects', which can contain data and code.

10.1 Classes and Objects

In [62]:
```
# Defining a class
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return 'Woof!'

# Creating an object
my_dog = Dog('Buddy', 3)
print(my_dog.name)  # Output: Buddy
print(my_dog.bark())  # Output: Woof!
```

```
Buddy
Woof!
```

10.2 Methods

In [63]:
```python
# Adding methods to a class
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return 'Woof!'

    def get_age(self):
        return self.age

# Creating an object and using methods
my_dog = Dog('Buddy', 3)
print(my_dog.get_age())  # Output: 3
```

```
3
```

10.3 Inheritance

In [64]:
```python
# Inheritance in Python
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        return 'Some sound'

class Dog(Animal):
    def bark(self):
        return 'Woof!'

# Creating an object of the derived class
my_dog = Dog('Buddy')
print(my_dog.name)  # Output: Buddy
print(my_dog.make_sound())  # Output: Some sound
print(my_dog.bark())  # Output: Woof!
```

```
Buddy
Some sound
Woof!
```

10.4 Polymorphism

In [65]:
```python
# Polymorphism in Python
class Cat:
    def speak(self):
        return 'Meow'

class Dog:
    def speak(self):
        return 'Woof'

def make_animal_speak(animal):
    print(animal.speak())

my_cat = Cat()
my_dog = Dog()
make_animal_speak(my_cat)  # Output: Meow
make_animal_speak(my_dog)  # Output: Woof
```

```
Meow
Woof
```

11. Advanced Topics

11.1 List Comprehensions

List comprehensions provide a concise way to create lists.

In [66]:
```python
# List comprehension to create a list of squares
squares = [x**2 for x in range(10)]
print(squares)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

11.2 Generators

In [67]:
```python
# Generator function
def generate_numbers(n):
    for i in range(n):
        yield i

# Using the generator
gen = generate_numbers(5)
for num in gen:
    print(num)  # Output: 0 1 2 3 4
```

```
0
1
2
3
4
```

11.3 Context Managers

In [68]:
```python
# Using a context manager
with open('file.txt', 'w') as file:
    file.write('Hello, world!')
```

12. Surprise Elements

Interactive exercises can help reinforce your understanding of Python. Try these exercises to apply what you've learned.

12.1 Interactive Exercise:

Try creating a small program that calculates the factorial of a number provided by the user. This will help reinforce your understanding of loops and conditional statements.

In [69]:
```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

# Taking user input
num = int(input('Enter a number: '))
print(f'The factorial of {num} is {factorial(num)}')
```

```
Enter a number: 5
The factorial of 5 is 120
```

12.2 Data Science Exercise:

Here's a quick data science exercise. Calculate the mean of a list of numbers provided by the user.

In [70]:
```
def calculate_mean(numbers):
    return sum(numbers) / len(numbers)

# Taking user input
user_input = input('Enter numbers separated by spaces: ')
numbers = list(map(int, user_input.split()))
print(f'The mean of the numbers is {calculate_mean(numbers)}')
```

```
Enter numbers separated by spaces: 1 2 3
The mean of the numbers is 2.0
```