Basics Cheat Sheet:

1. Basic Syntax:

- Python uses indentation to indicate blocks of code.
- Comments start with `#`.
- Statements end with a newline character, no need for semicolons.

2. Types of Data Types:

- Integer (`int`): Whole numbers.
- Float (`float`): Numbers with decimal points.
- String (`str`): Sequence of characters.
- Boolean (`bool`): True or False.
- List (`list`): Ordered collection of items.
- Tuple (`tuple`): Immutable ordered collection of items.
- Dictionary (`dict`): Key-value pairs.

3. Declaration & Initialization:

- Variables are declared with a name and assigned a value using `=`:

```
x = 10  # Integer
y = 3.14  # Float
name = "Alice"  # String
is_valid = True  # Boolean
my_list = [1, 2, 3]  # List
my_tuple = (4, 5, 6)  # Tuple
my_dict = {"key": "value"}  # Dictionary
```

4. Getting Input from User:

- Use the `input()` function to get user input:

```
name = input("Enter your name: ")
print("Hello,", name)
```

5. Basic Built-in Functions:

- `id()`: Returns the memory address of an object.
- `type()`: Returns the data type of an object.
- `len()`: Returns the length of an object.
- `print()`: Prints the specified content to the console.

Example:

```
# Get user input
age = input("Enter your age: ")

# Check data type
print("Data type of age variable:", type(age))

# Convert age to integer
age = int(age)

# Print age
print("Your age is:", age)

# Find memory address
print("Memory address of age variable:", id(age))

# Find data type
print("Data type of age variable:", type(age))

# Find size (length of string)
print("Number of characters in age variable:", len(str(age)))
```

Explanation:

- The program prompts the user to enter their age.
- It then checks the data type of the input using `type()` and prints it.
- The input is converted to an integer using `int()`.
- The program prints the age.
- Memory address of the age variable is obtained using `id()`.
- Data type of the age variable is again printed to confirm the conversion.
- Finally, the length of the age (converted to a string) is printed using `len()`.

Python Operators Cheat Sheet:

1. Arithmetic Operators:

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `%`: Modulus (remainder)
- `**`: Exponentiation (power)
- `//`: Floor division (quotient)

2. Comparison Operators:

- `==`: Equal to
- `!=`: Not equal to
- `<`: Less than
- `>`: Greater than
- `<=`: Less than or equal to
- `>=`: Greater than or equal to

3. Assignment Operators:

- `=`: Assign value
- `+=`: Add and assign
- `-=`: Subtract and assign
- `*=`: Multiply and assign
- `/=`: Divide and assign
- `%=`: Modulus and assign
- `**=`: Exponentiation and assign
- `//=`: Floor division and assign

4. Logical Operators:

- `and`: Logical AND
- `or`: Logical OR
- `not`: Logical NOT

5. Bitwise Operators:

- `&`: Bitwise AND
- `|`: Bitwise OR
- `^`: Bitwise XOR
- `~`: Bitwise NOT
- `<<`: Left shift
- `>>`: Right shift

6. Membership Operators:

- `in`: True if value is found in sequence
- `not in`: True if value is not found in sequence

7. Identity Operators:

- `is`: True if both variables are the same object

- `is not`: True if both variables are not the same object

Example:

```
# Arithmetic Operators
x = 10
y = 3
print("Addition:", x + y)
print("Subtraction:", x - y)
print("Multiplication:", x * y)
print("Division:", x / y)
print("Modulus:", x % y)
print("Exponentiation:", x ** y)
print("Floor Division:", x // y)

# Comparison Operators
print("Equal to:", x == y)
print("Not equal to:", x != y)
print("Less than:", x < y)
print("Greater than:", x > y)
print("Less than or equal to:", x <= y)
print("Greater than or equal to:", x >= y)

# Logical Operators
a = True
b = False
print("Logical AND:", a and b)
print("Logical OR:", a or b)
print("Logical NOT:", not a)

# Bitwise Operators
p = 5  # 0101
q = 3  # 0011
print("Bitwise AND:", p & q)   # Output: 1 (0001)
print("Bitwise OR:", p | q)    # Output: 7 (0111)
print("Bitwise XOR:", p ^ q)   # Output: 6 (0110)
print("Bitwise NOT:", ~p)      # Output: -6 (bitwise negation)
print("Left Shift:", p << 1)   # Output: 10 (1010)
print("Right Shift:", p >> 1)  # Output: 2 (0010)

# Membership Operators
my_list = [1, 2, 3, 4, 5]
print("Membership Check:", 3 in my_list)  # Output: True

# Identity Operators
x = 10
y = 10
print("Identity Check:", x is y)  # Output: True
```

Explanation:

- Each section demonstrates the usage of different operators with simple examples.
- Arithmetic operators perform mathematical operations on operands.
- Comparison operators compare two values and return a boolean result.
- Logical operators perform logical operations on boolean values.
- Bitwise operators perform operations on binary representations of integers.
- Membership operators check for membership within sequences like lists, tuples, etc.
- Identity operators compare the memory locations of two objects.

Python Conditional Statements Cheat Sheet:

1. If Statement:

- Executes a block of code if a specified condition is true.

2. If-Else Statement:

- Executes one block of code if the condition is true, and another if false.

3. If-Elif-Else Statement:

- Allows for multiple conditions to be checked.

4. Nested If Statement:

- An if statement inside another if statement.

Example:

```
# Example: Grade Calculator

# Get input from user
score = float(input("Enter your score: "))

# Determine grade
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
    grade = 'F'

# Output result
if grade == 'F':
    print("Sorry, you failed.")
else:
    print("Congratulations! Your grade is:", grade)
```

Explanation:

- The program prompts the user to enter their score.
- Based on the score, it determines the grade using conditional statements.
- If the grade is 'F', it prints a message indicating failure; otherwise, it congratulates the user and prints the grade.
- This example illustrates the use of if, elif, and else statements to handle different conditions.

The `match-case` statement was introduced in Python 3.10 as a more flexible and readable replacement for the `switch-case` pattern. Here's a cheat sheet explaining how to use it along with an example:

Python `match-case` Statement Cheat Sheet:

1. **Syntax:**

```
match expression:
    case pattern1:
        # code block
    case pattern2:
        # code block
    ...
    case patternN:
        # code block
    case _:
        # default code block
```

2. **Usage:**

- `match` keyword is followed by an expression to be matched.
- `case` keyword is used to define different patterns to match against the expression.
- Patterns can include literals, variables, sequences, etc.
- The _ pattern serves as the default case.

Example:

```
# Example: Grade Calculator using match-case statement

# Define function to calculate grade
def calculate_grade(score):
    return match score:
        case x if x >= 90:
            return 'A'
        case x if x >= 80:
            return 'B'
        case x if x >= 70:
            return 'C'
        case x if x >= 60:
            return 'D'
        case _:
            return 'F'

# Get input from user
score = float(input("Enter your score: "))

# Calculate grade
grade = calculate_grade(score)

# Output result
match grade:
    case 'F':
        print("Sorry, you failed.")
    case _:
        print(f"Congratulations! Your grade is: {grade}")
```

Explanation:

- We define a function `calculate_grade()` that takes a score as input and returns the corresponding grade using the `match-case` statement.
- The function `calculate_grade()` uses `match` to match the input score against different patterns representing grade ranges.
- The `case` statements include conditions for each grade range (A, B, C, D) and a default case (_) for failing grades.
- We get the score from the user and calculate the grade using the `calculate_grade()` function.
- Finally, we use another `match-case` statement to print the result, congratulating the user if they passed or informing them if they failed.

Python Loops Cheat Sheet:

1. While Loop:

- Repeats a block of code as long as a specified condition is true.

```
while condition:
    # code block
```

2. For Loop:

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

```
for item in iterable:
    # code block
```

3. Break Statement:

- Terminates the loop prematurely when a certain condition is met.

```
for item in iterable:
    if condition:
        break
```

4. Continue Statement:

- Skips the rest of the current iteration and continues to the next iteration of the loop.

```
for item in iterable:
    if condition:
        continue
```

5. Do-While Loop (Emulation):

- Not directly supported in Python, but can be emulated using a while loop with an initial condition that is always true and then using `break` to exit the loop when necessary.

```
while True:
    # code block
    if condition:
        break
```

Basic Example:

```
# While Loop
num = 1
while num <= 5:
    print(num)
    num += 1

# For Loop
my_list = [1, 2, 3, 4, 5]
for num in my_list:
    print(num)

# Break Statement
for num in my_list:
    if num == 3:
        break
    print(num)
```

```
# Continue Statement
for num in my_list:
    if num == 3:
        continue
    print(num)
```

Advanced Example (Do-While Emulation):

```
# Emulating Do-While Loop
num = 0
while True:
    num += 1
    print(num)
    if num == 5:
        break
```

Explanation:

- While loop iterates as long as the condition is true.
- For loop iterates over each item in the iterable.
- Break statement terminates the loop prematurely.
- Continue statement skips the current iteration and proceeds to the next.
- The advanced example emulates a do-while loop by using a while loop with an always true condition and breaking when the desired condition is met.

here are multiple examples demonstrating the usage of `for` loop with `zip()`, `enumerate()`, and `range()` functions:

Using `zip()` Function:

```
# Example 1: Iterate over two lists simultaneously
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
for item1, item2 in zip(list1, list2):
    print(item1, item2)

# Example 2: Iterate over multiple lists of different lengths
list1 = ['a', 'b', 'c']
list2 = [1, 2]
for item1, item2 in zip(list1, list2):
    print(item1, item2)
```

Using `enumerate()` Function:

```
# Example 1: Iterate over a list with index
my_list = ['a', 'b', 'c']
for index, item in enumerate(my_list):
    print(index, item)
```

```
# Example 2: Specify start index
my_list = ['a', 'b', 'c']
for index, item in enumerate(my_list, start=1):
    print(index, item)
```

Using `range()` Function:

```
# Example 1: Iterate over a range of numbers
for num in range(5):
    print(num)

# Example 2: Iterate over a range with start and end
for num in range(1, 6):
    print(num)

# Example 3: Iterate over a range with step
for num in range(1, 10, 2):
    print(num)
```

Explanation:

- `zip()` function takes iterables (like lists) as arguments and returns an iterator that produces tuples containing the elements from each iterable.
- `enumerate()` function adds a counter to an iterable and returns it as an enumerate object. It allows iteration over both the index and the item in the iterable.
- `range()` function generates a sequence of numbers. It can be used in `for` loops to iterate a specified number of times or over a range of values.

The `pass` statement in Python is a null operation. It doesn't do anything when executed. It acts as a placeholder to maintain the syntactic structure of the code. It's often used as a placeholder for code that will be added later or in situations where no action is required.

Example:

```
x = 10
if x > 5:
    pass  # No action needed for now, code will be added later
else:
    print("x is not greater than 5")
```

In this example, if `x` is greater than 5, the `pass` statement does nothing, and the code moves to the next block. If `x` is not greater than 5, it prints a message.

The `pass` statement is particularly useful in situations like defining empty classes or functions that need to be implemented later. It allows the code to run without any errors until the implementation is completed.

Python Functions & Modules Cheat Sheet:

1. Defining Functions:

- Use the `def` keyword followed by the function name and parameters in parentheses.
- Indent the function body.
- Use the `return` statement to return a value (optional).

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

2. Calling Functions:

- Simply use the function name followed by parentheses and arguments (if any).

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

3. Default Arguments:

- Specify default values for function parameters.

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

4. Keyword Arguments:

- Pass arguments by specifying the parameter name.

```
message = greet(name="Bob")
print(message)  # Output: Hello, Bob!
```

5. Arbitrary Arguments:

- Accept any number of arguments using `*args`.

```
def add(*args):
    return sum(args)
```

6. Docstrings:

- Provide documentation for functions using triple quotes.

```
def greet(name):
    """
    Greets the user by name.

    Args:
        name (str): Name of the user.

    Returns:
        str: Greeting message.
    """
    return f"Hello, {name}!"
```

7. Modules:

- A Python file containing definitions and statements.
- Import modules using `import` statement.

```
import math
```

8. Module Aliases:

- Use aliases for imported modules using the `as` keyword.

```
import math as m
```

9. Import Specific Items:

- Import only specific items from a module.

```
from math import pi
```

10. Import All Items: - Import all items from a module (not recommended to avoid name clashes).

```
from math import *
```

Example Covering Basic and Advanced Concepts:

```
# Example: Calculation module

import math

def add(x, y):
    """Addition"""
    return x + y

def subtract(x, y):
    """Subtraction"""
    return x - y
```

```
def multiply(x, y):
    """Multiplication"""
    return x * y

def divide(x, y):
    """Division"""
    if y == 0:
        return "Cannot divide by zero"
    else:
        return x / y

def circle_area(radius):
    """Calculate area of a circle"""
    return math.pi * radius**2

def quadratic_eqn(a, b, c):
    """Solve quadratic equation"""
    discriminant = b**2 - 4*a*c
    if discriminant < 0:
        return "No real roots"
    elif discriminant == 0:
        return -b / (2*a)
    else:
        root1 = (-b + math.sqrt(discriminant)) / (2*a)
        root2 = (-b - math.sqrt(discriminant)) / (2*a)
        return root1, root2

# Test the functions
print("Addition:", add(5, 3))
print("Subtraction:", subtract(5, 3))
print("Multiplication:", multiply(5, 3))
print("Division:", divide(5, 3))
print("Circle Area:", circle_area(5))
print("Quadratic Roots:", quadratic_eqn(1, -3, 2))
```

Explanation:

- This example defines a module with various functions for basic arithmetic operations, area calculation, and solving quadratic equations.
- Functions include examples of default arguments, docstrings, and advanced mathematical calculations.
- The module is imported and functions are called to demonstrate their usage.

Python String Cheat Sheet:

1. String Basics:

- A string is a sequence of characters, enclosed in single, double, or triple quotes.

```
my_string = "Hello, World!"
```

2. String Slicing:

- Extract parts of a string using indices or slicing notation.

```
my_string = "Hello, World!"
print(my_string[0])        # Output: H
print(my_string[7:12])     # Output: World
```

3. Modify Strings:

- Strings are immutable, but you can create new strings by modifying existing ones.

```
my_string = "Hello, World!"
modified_string = my_string.replace("World", "Python")
```

4. String Concatenation:

- Combine strings using the `+` operator.

```
string1 = "Hello"
string2 = "World"
concatenated_string = string1 + ", " + string2 + "!"
```

5. String Formatting:

- Use `%` or `format()` method for string formatting.
- F-strings (`f""`) are a more modern and convenient way for string formatting in Python 3.6+.

```
name = "Alice"
age = 30
message = "My name is %s and I am %d years old." % (name, age)
```

or

```
name = "Alice"
age = 30
message = "My name is {} and I am {} years old.".format(name, age)
```

or

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

6. Escape Character:

- Use `\` to escape special characters in strings.

```
my_string = "This is a \"quote\" inside a string."
```

7. Raw String (`r""`):

- A raw string is prefixed with `r` or `R`.
- Treats backslashes (`\`) as literal characters.

```
path = r"C:\Users\Username\Desktop\file.txt"
```

8. F-String (`f""`):

- An f-string is prefixed with `f` or `F`.
- Allows for string interpolation, making it easy to embed expressions within strings.

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

Example Covering Basic and Advanced Concepts:

```
# Example: String Manipulation

# String Basics
my_string = "Hello, World!"

# String Slicing
print(my_string[0])        # Output: H
print(my_string[7:12])     # Output: World

# Modify Strings
modified_string = my_string.replace("World", "Python")

# String Concatenation
```

```python
string1 = "Hello"
string2 = "World"
concatenated_string = string1 + ", " + string2 + "!"

# String Formatting
name = "Alice"
age = 30
message = f"My name is {name} and I am {age} years old."

# Escape Character
my_string = "This is a \"quote\" inside a string."

# Raw String
path = r"C:\Users\Username\Desktop\file.txt"

# F-String
name = "Alice"
age = 30
message = f"My name is {name} and I am {age} years old."
```

Explanation:

- This example covers basic and advanced string concepts including string basics, slicing, modification, concatenation, formatting, escape characters, raw strings, and f-strings.
- Each concept is demonstrated with an example to illustrate its usage and output.

1. capitalize():

```python
text = "hello, world!"
result = text.capitalize()
print(result)  # Output: Hello, world!
```

2. casefold():

```python
text = "HELLO, World!"
result = text.casefold()
print(result)  # Output: hello, world!
```

3. lower():

```python
text = "HELLO, World!"
result = text.lower()
print(result)  # Output: hello, world!
```

4. swapcase():

```python
text = "Hello, World!"
```

```
result = text.swapcase()
print(result)  # Output: hELLO, wORLD!
```

5. title():

```
text = "hello, world!"
result = text.title()
print(result)  # Output: Hello, World!
```

6. upper():

```
text = "hello, world!"
result = text.upper()
print(result)  # Output: HELLO, WORLD!
```

7. center(width, fillchar):

```
text = "hello"
result = text.center(10, "*")
print(result)  # Output: **hello***
```

8. ljust(width[, fillchar]):

```
text = "hello"
result = text.ljust(10, "*")
print(result)  # Output: hello*****
```

9. rjust(width[, fillchar]):

```
text = "hello"
result = text.rjust(10, "*")
print(result)  # Output: *****hello
```

10. expandtabs(tabsize=8):

```
text = "hello\tworld"
result = text.expandtabs(tabsize=4)
print(result)  # Output: hello   world
```

11. zfill(width):

```
text = "42"
result = text.zfill(5)
print(result)  # Output: 00042
```

12. lstrip():

```
text = "   hello, world!   "
result = text.lstrip()
print(result)  # Output: hello, world!
```

13. rstrip():

```
text = "   hello, world!   "
result = text.rstrip()
print(result)  # Output:    hello, world!
```

14. strip():

```
text = "   hello, world!   "
result = text.strip()
print(result)  # Output: hello, world!
```

15. rsplit():

```
text = "apple, banana, cherry"
result = text.rsplit(", ")
print(result)  # Output: ['apple', 'banana', 'cherry']
```

16. split():

```
text = "apple, banana, cherry"
result = text.split(", ")
print(result)  # Output: ['apple', 'banana', 'cherry']
```

17. splitlines():

```
text = "Hello\nWorld\n"
result = text.splitlines()
print(result)  # Output: ['Hello', 'World']
```

18. partition():

```
text = "apple, banana, cherry"
result = text.partition(", ")
print(result)  # Output: ('apple', ', ', 'banana, cherry')
```

19. rpartition():

```
text = "apple, banana, cherry"
result = text.rpartition(", ")
print(result)  # Output: ('apple, banana', ', ', 'cherry')
```

20. join():

```
my_list = ["apple", "banana", "cherry"]
result = ", ".join(my_list)
```

```
print(result)  # Output: apple, banana, cherry
```

21. removeprefix(prefix):

```
text = "hello, world!"
result = text.removeprefix("hello, ")
print(result)  # Output: world!
```

22. removesuffix(suffix):

```
text = "hello, world!"
result = text.removesuffix(", world!")
print(result)  # Output: hello
```

23. isalnum():

```
text = "abc123"
result = text.isalnum()
print(result)  # Output: True
```

24. isalpha():

```
text = "abc"
result = text.isalpha()
print(result)  # Output: True
```

25. isdigit():

```
text = "123"
result = text.isdigit()
print(result)  # Output: True
```

26. islower():

```
text = "hello"
result = text.islower()
print(result)  # Output: True
```

27. isnumeric():

```
text = "123"
result = text.isnumeric()
print(result)  # Output: True
```

28. isspace():

```
text = "   "
result = text.isspace()
print(result)  # Output: True
```

29. istitle():

```
text = "Hello, World!"
result = text.istitle()
print(result)  # Output: True
```

30. isupper():

```
text = "HELLO"
result = text.isupper()
print(result)  # Output: True
```

31. isascii():

```
text = "hello"
result = text.isascii()
print(result)  # Output: True
```

32. isdecimal():

```
text = "123"
result = text.isdecimal()
print(result)  # Output: True
```

33. isidentifier():

```
text = "hello_world"
result = text.isidentifier()
print(result)  # Output: True
```

34. isprintable():

```
text = "hello\nworld"
result = text.isprintable()
print(result)  # Output: False
```

35. count(sub, beg, end):

```
text = "apple, banana, cherry"
result = text.count("a")
print(result)  # Output: 3
```

36. find(sub, beg, end):

```
text = "apple, banana, cherry"
result = text.find("banana")
print(result)  # Output: 7
```

37. index(sub, beg, end):

```
text = "apple, banana, cherry"
result = text.index("banana")
```

```
print(result)  # Output: 7
```

38. replace(old, new [, max]):

```
text = "apple, banana, cherry"
result = text.replace("banana", "orange")
print(result)  # Output: apple, orange, cherry
```

39. rfind(sub, beg, end):

```
text = "apple, banana, banana, cherry"
result = text.rfind("banana")
print(result)  # Output: 13
```

40. rindex(sub, beg, end):

```
text = "apple, banana, banana, cherry"
result = text.rindex("banana")
print(result)  # Output: 13
```

41. startswith(sub, beg, end):

```
text = "apple, banana, cherry"
result = text.startswith("apple")
print(result)  # Output: True
```

42. endswith(suffix, beg, end):

```
text = "apple, banana, cherry"
result = text.endswith("cherry")
print(result)  # Output: True
```

43. translate():

```
text = "hello"
translation_table = text.maketrans("aeiou", "12345")
result = text.translate(translation_table)
print(result)  # Output: h2ll4
```

Python List Cheat Sheet:

1. Python List:

- A list is a collection of items, ordered and changeable, allowing duplicate elements.
- Lists are defined by enclosing items in square brackets `[ ]`.

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

2. Access List Items:

- Elements in a list are accessed by their index, starting from 0.
- Negative indexing starts from the end of the list (-1 refers to the last item).

```
first_item = my_list[0]
last_item = my_list[-1]
```

3. Change List Items:

- You can change the value of specific items by referring to their index.

```
python

my_list[0] = "one"
```

4. Add List Items:

- Use `append()`, `insert()`, or concatenation to add items to a list.

```
my_list.append(6)
my_list.insert(0, "zero")
```

5. Remove List Items:

- Use `remove()`, `pop()`, or `del` to remove items from a list.

```
my_list.remove(2)
my_list.pop(0)
del my_list[0]
```

6. Loop List:

- Iterate through the elements of a list using `for` loop.

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

7. List Comprehension:

- A concise way to create lists in Python.
- Allows for transforming or filtering elements of an existing list.

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

8. Sort List:

- Use `sort()` to sort the list in ascending order.
- Use `reverse=True` for descending order.

```
my_list.sort()
```

9. Copy List:

- Create a copy of a list using `copy()` method or slicing.

```
new_list = my_list.copy()
```

10. Join List: - Use `join()` method to concatenate elements of a list into a single string.

```
my_list = ["Hello", "world"]
result = " ".join(my_list)
```

11. List Methods: - `append()`: Add an element to the end of the list. - `clear()`: Remove all elements from the list. - `count()`: Return the number of occurrences of a value in the list. - `extend()`: Add elements from another list to the end of the list. - `index()`: Return the index of the first occurrence of a value. - `pop()`: Remove and return the last element of the list. - `remove()`: Remove the first occurrence of a value from the list. - `reverse()`: Reverse the order of the list. - `sort()`: Sort the list in ascending order.

Example Covering Basic and Advanced Concepts:

```
# Example: List Manipulation

# Python List
my_list = [1, 2, 3, 4, 5]
```

```
# Access List Items
first_item = my_list[0]
last_item = my_list[-1]

# Change List Items
my_list[0] = "one"

# Add List Items
my_list.append(6)
my_list.insert(0, "zero")

# Remove List Items
my_list.remove(2)
my_list.pop(0)
del my_list[0]

# Loop List
for item in my_list:
    print(item)

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

# Sort List
my_list.sort()

# Copy List
new_list = my_list.copy()

# Join List
my_list = ["Hello", "world"]
result = " ".join(my_list)
```

Explanation:

- This example demonstrates various list operations including accessing, changing, adding, and removing items.
- It covers list manipulation techniques like loop iteration, list comprehension, sorting, copying, and joining lists.
- Each concept is illustrated with examples to showcase its usage and output.

Python Dictionaries Cheat Sheet

What are Dictionaries?

- Dictionaries in Python are unordered collections of items.
- Each item in a dictionary is a key-value pair.
- Dictionaries are mutable, meaning they can be changed after creation.

- They are enclosed in curly braces `{}` and each item is separated by a comma.

Basic Dictionary Operations:

1. **Access Dictionary:**

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

2. **Change Dictionary:**

```
my_dict['age'] = 35
print(my_dict)  # Output: {'name': 'John', 'age': 35, 'city': 'New York'}
```

3. **Add to Dictionary:**

```
my_dict['job'] = 'Engineer'
print(my_dict)  # Output: {'name': 'John', 'age': 35, 'city': 'New York', 'job':
'Engineer'}
```

4. **Remove from Dictionary:**

```
del my_dict['city']
print(my_dict)  # Output: {'name': 'John', 'age': 35, 'job': 'Engineer'}
```

5. **Dictionary View Objects:**

    - `keys()`: Returns a view of all the keys in the dictionary.
    - `values()`: Returns a view of all the values in the dictionary.
    - `items()`: Returns a view of all the key-value pairs in the dictionary.

6. **Loop through Dictionary:**

```
for key, value in my_dict.items():
    print(key, value)
# Output:
# name John
# age 35
# job Engineer
```

7. **Copy Dictionary:**

```
new_dict = my_dict.copy()
```

8. **Nested Dictionary:**

```
nested_dict = {'person1': {'name': 'John', 'age': 30}, 'person2': {'name': 'Alice',
'age': 25}}
print(nested_dict['person1']['name'])  # Output: John
```

Dictionary Methods:

- `clear()`: Removes all items from the dictionary.
- `get(key)`: Returns the value associated with the specified key.
- `pop(key)`: Removes the item with the specified key and returns its value.
- `popitem()`: Removes and returns an arbitrary item (key, value) pair from the dictionary.
- `update(other_dict)`: Updates the dictionary with the key-value pairs from the specified dictionary.
- `setdefault(key, default_value)`: Returns the value of the specified key. If the key does not exist, inserts the key with the specified default value.
- `keys()`: Returns a view of all the keys in the dictionary.
- `values()`: Returns a view of all the values in the dictionary.
- `items()`: Returns a view of all the key-value pairs in the dictionary.

Examples:

```
# Create a dictionary
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

# Access dictionary
print(my_dict['age'])  # Output: 30

# Change dictionary
my_dict['age'] = 35
print(my_dict)  # Output: {'name': 'John', 'age': 35, 'city': 'New York'}

# Add to dictionary
my_dict['job'] = 'Engineer'
print(my_dict)  # Output: {'name': 'John', 'age': 35, 'city': 'New York', 'job': 
'Engineer'}

# Remove from dictionary
del my_dict['city']
print(my_dict)  # Output: {'name': 'John', 'age': 35, 'job': 'Engineer'}

# Loop through dictionary
for key, value in my_dict.items():
    print(key, value)
# Output:
# name John
# age 35
# job Engineer

# Copy dictionary
new_dict = my_dict.copy()

# Nested dictionary
nested_dict = {'person1': {'name': 'John', 'age': 30}, 'person2': {'name': 'Alice', 
'age': 25}}
print(nested_dict['person1']['name'])  # Output: John
```

This cheat sheet covers basic and advanced concepts of Python dictionaries, including creation, manipulation, iteration, and methods available for dictionaries.

Python Tuples Cheat Sheet

What are Tuples?

- Tuples are ordered collections of items, similar to lists.
- However, tuples are immutable, meaning they cannot be changed after creation.
- Tuples are enclosed in parentheses `()` and each item is separated by a comma.

Creating Tuples:

```
# Empty tuple
empty_tuple = ()
# Tuple with elements
my_tuple = (1, 2, 3, 4, 5)
# Tuple with mixed data types
mixed_tuple = (1, 'hello', 3.5, True)
```

Accessing Tuple Items:

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple[2])  # Output: 3
```

Updating Tuples:

```
# Tuples are immutable, so you cannot update individual elements.
# You can create a new tuple with updated values.
my_tuple = (1, 2, 3)
new_tuple = my_tuple + (4,)
print(new_tuple)  # Output: (1, 2, 3, 4)
```

Unpacking Tuples:

```
my_tuple = (1, 2, 3)
a, b, c = my_tuple
print(a, b, c)  # Output: 1 2 3
```

Looping through Tuples:

```
my_tuple = (1, 2, 3)
for item in my_tuple:
    print(item)
# Output:
# 1
# 2
# 3
```

Joining Tuples:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
joined_tuple = tuple1 + tuple2
print(joined_tuple)  # Output: (1, 2, 3, 4, 5, 6)
```

Tuple Methods:

```
my_tuple = (1, 2, 3, 2, 4)

# count(value): Returns the number of occurrences of the specified value in the tuple.
print(my_tuple.count(2))  # Output: 2

# index(value): Returns the index of the first occurrence of the specified value in the tuple.
print(my_tuple.index(3))  # Output: 2
```

More Examples:

```
# Length of a tuple
print(len(my_tuple))  # Output: 5

# Check if an item exists in the tuple
print(2 in my_tuple)  # Output: True

# Nested tuples
nested_tuple = ((1, 2), (3, 4), (5, 6))
print(nested_tuple[1][0])  # Output: 3
```

Tuple Slicing:

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple[1:4])  # Output: (2, 3, 4)
```

Tuple Packing and Unpacking:

```
# Tuple Packing
packed_tuple = 1, 2, 3
print(packed_tuple)  # Output: (1, 2, 3)

# Tuple Unpacking
x, y, z = packed_tuple
print(x, y, z)  # Output: 1 2 3
```

Tuple Comparison:

```
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 4)
print(tuple1 == tuple2)  # Output: False
```

Tuple Concatenation and Repetition:

```
tuple1 = (1, 2)
tuple2 = (3, 4)
concatenated_tuple = tuple1 + tuple2
print(concatenated_tuple)  # Output: (1, 2, 3, 4)

repeated_tuple = tuple1 * 3
print(repeated_tuple)  # Output: (1, 2, 1, 2, 1, 2)
```

Python Setes Cheat Sheet

Accessing set items: Sets are unordered, so you can't access items by index like in lists or tuples. Instead, you can check for membership using the `in` keyword.

```
my_set = {1, 2, 3, 4, 5}
print(3 in my_set)  # Output: True
```

1. **Adding items to sets**: You can add items to a set using the `add()` method for single items and `update()` method to add multiple items.

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}

my_set.update([5, 6])
print(my_set)  # Output: {1, 2, 3, 4, 5, 6}
```

2. **Removing items from sets**: Items can be removed from sets using the `remove()` or `discard()` method. `remove()` will raise an error if the item is not found, whereas `discard()` will not.

```
my_set = {1, 2, 3, 4}
my_set.remove(3)
print(my_set)  # Output: {1, 2, 4}

my_set.discard(5)  # No error raised
```

3. **Looping through sets**: You can loop through sets using a for loop.

```
my_set = {1, 2, 3}
for item in my_set:
    print(item)
```

4. **Joining sets**: You can use the `union()` method or the `|` operator to join two or more sets together.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = set1.union(set2)
# Or using the | operator
set4 = set1 | set2
print(set3)  # Output: {1, 2, 3, 4, 5}
```

5. **Copying sets**: You can use the `copy()` method to create a shallow copy of a set.

```
original_set = {1, 2, 3}
copied_set = original_set.copy()
```

6. **Set operators**: Python provides various operators to perform set operations like union, intersection, difference, and symmetric difference.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Union
print(set1 | set2)  # Output: {1, 2, 3, 4, 5}

# Intersection
print(set1 & set2)  # Output: {3}

# Difference
print(set1 - set2)  # Output: {1, 2}

# Symmetric Difference
print(set1 ^ set2)  # Output: {1, 2, 4, 5}
```

7. **Set methods**: Python provides several methods for set manipulation, including `intersection()`, `difference()`, `symmetric_difference()`, `issubset()`, `issuperset()`, and others.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Intersection
print(set1.intersection(set2))  # Output: {3}

# Difference
print(set1.difference(set2))  # Output: {1, 2}

# Symmetric Difference
print(set1.symmetric_difference(set2))  # Output: {1, 2, 4, 5}

# Subset
print(set1.issubset(set2))  # Output: False
```

```
# Superset
print(set1.issuperset(set2))  # Output: False
```