7. Title: Extra - F-strings formatting Cheat Sheet

1. Expressions Inside Braces

You can perform operations or call functions directly within the braces of an F-string.

```python
name = "Alice"
age = 30
print(f"{name} is {age + 1} years old next year.")
```

2. Calling Functions

Call functions directly within the F-string braces for dynamic string formatting.

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

print(f"{greet('Bob')}")
```

3. Format Specifiers

Use format specifiers for controlling the format of the output, such as number of decimals.

```python
price = 49.99
print(f"The price is {price:.2f} dollars.")
```

4. Dictionary Values

Access dictionary values directly in an F-string.

```python
person = {'name': 'Alice', 'age': 30}
print(f"{person['name']} is {person['age']} years old.")
```

5. Date Formatting

Format dates using the `datetime` module directly within F-strings.

```python
from datetime import datetime
now = datetime.now()
print(f"The current time is {now:%Y-%m-%d %H:%M}.")
```

6. Multiline F-strings

Create multiline strings without losing the F-string functionality.

```python
name = "Alice"
profession = "developer"
bio = (f"Name: {name}\n"
       f"Profession: {profession}")
print(bio)
```

7. Conditional Expressions

Embed conditional expressions within F-strings for dynamic content.

```python
age = 18
status = "adult" if age &gt;= 18 else "minor"
print(f"You are an {status}.")
```

8. Nesting F-strings

Nest F-strings for complex formatting scenarios.

```python
from math import pi
precision = 2
print(f"Pi rounded to {precision} decimal places is {pi:.{precision}f}.")
```

9. Using Braces

To include literal braces in your string, double them.

```python
print(f"{Hello}")
```

10. Debugging with F-strings (Python 3.8+)

Include the variable name with its value for easier debugging.

```python
name = "Alice"
print(f"{name=}")
```

11. Alignment and Padding

You can align text and add padding directly within an F-string.

```python
name = "Alice"
# Right align with padding
print(f"{name:&gt;10}")
# Left align with padding
print(f"{name:&lt;10}")
# Center align with padding
print(f"{name:^10}")
```

12. Thousands Separator

Format large numbers with commas as thousands separators for better readability.

```python
number = 1000000
print(f"{number:,}")
```

13. Dynamic Format Specifiers

Dynamically specify format options using variables.

Python

```
width = 10
precision = 4
number = 12.34567
print(f"result: {number:{width}.{precision}}")
```

14. Using Objects

Directly access object attributes and methods.

Python

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greet(self):
        return f"Hello, my name is {self.name}."

person = Person("Alice", 30)
print(f"{person.greet()} I am {person.age} years old.")
```

15. Binary, Hexadecimal, and Octal Formatting

Convert numbers into binary, hexadecimal, or octal formats.

Python

```
number = 255
print(f"Binary: {number:b}, Hex: {number:x}, Octal: {number:o}")
```

16. Combining F-strings with Dictionaries for Data Unpacking

Unpack and format data from dictionaries elegantly.

Python

```
data = {"name": "Bob", "age": 25}
print(f"{data['name']} is {data['age']} years old.")
```

17. Multiline Expressions

Use parentheses for longer expressions that span multiple lines.

Python
```
x = 10
y = 5
print(f"Sum: {(x + y)}\n"
      f"Difference: {(x - y)}\n"
      f"Product: {(x * y)}")
```

18. Using Lambdas

Inline lambda functions for quick transformations or calculations.

Python
```
print(f"Square of 5: {(lambda x: x*x)(5)}")
```

19. Escaping Curly Braces

Use double curly braces to escape them in F-strings.

Python
```
print(f"{{{{double braces}}}} are used to escape in F-strings.")
```

20. Complex Numbers Formatting

Format complex numbers by accessing their real and imaginary parts.

Python
```
complex_number = 3 + 4j
print(f"The real part is {complex_number.real} and the imaginary part is {complex_number.imag}.")
```

21. Formatting with Percentages

Easily format numbers as percentages.

Python
```
completion = 0.756
```

```
print(f"Task completed: {completion:.2%}")
```

22. Safe String Interpolation with `str.format()`

While not a direct feature of F-strings, combining them with `str.format()` can offer a safer way to dynamically insert values that aren't known until runtime, thus avoiding code injection risks.

Python

```
user_input = "{name} is {age} years old."
# Safe interpolation
print(f"{user_input}".format(name="Alice", age=30))
```

23. Reusing Calculated Values in F-strings

By using a lambda function, you can calculate values once and reuse them within the same F-string.

Python

```
result = (lambda x: (x, x**2))(3)
print(f"The number is {result[0]} and its square is {result[1]}.")
```

24. Formatting Paths and Files

F-strings can be particularly useful for dynamically generating file paths or URLs.

Python

```
filename = "report"
date = "2023-01-01"
print(f"./data/{filename}_{date}.txt")
```

25. Embedding Expressions for Object Instantiation

Instantiate objects directly within an F-string.

Python

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"Point({self.x}, {self.y})"
```

```
print(f"New point: {Point(1, 2)}")
```

26. Formatting Exceptions

Use F-strings to format exception messages for clearer error reporting.

Python

```
try:
    # Potentially problematic code
    x = 1 / 0
except Exception as e:
    print(f"An error occurred: {e}")
```

27. Dynamic Padding with Variables

Control padding dynamically by using variables within the format specifier.

Python

```
name = "Alice"
pad = 20
print(f"{name:&gt;{pad}}")
```

28. Using F-strings with Maps and Filters

F-strings can be used inside `map()` and `filter()` for more readable code.

Python

```
names = ["Alice", "Bob", "Charlie"]
upper_names = map(lambda x: f"{x.upper()}", names)
print(list(upper_names))
```

29. Custom Object Formatting

Define how your custom objects should be formatted within F-strings by implementing the `__format__` method.

Python

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

```
        if format_spec == 'upper':
            return self.name.upper()
        return self.name

person = Person("Alice")
print(f"{person:upper}")
```

30. Inline Comments

While not a feature of F-strings themselves, you can use comments to clarify complex expressions within them (though this requires breaking the expression into multiple steps).

Python

```
# Intended for readability, not direct execution within F-strings
number = 123456.789
# Format number with commas and 2 decimal places
formatted_number = f"{number:,.2f}"  # Comment: Format as a string with commas
print(formatted_number)
```