AI Alchemy with Python: Sets, Functions, and Modularity

Welcome, Future AI Wizards!

This module is your friendly guide to understanding fundamental Python concepts that will form the bedrock of your journey into the exciting world of Artificial Intelligence. We'll focus on sets, built-in functions, and the power of modularity in your code. Let's dive in!

Figure 1: Buckle up, buckaroos!

From Sunlight to Supercomputers: An Analogy for Problem Solving

Imagine a world without electricity. A world where the only source of light is the sun. Life would be very different, right? Our activities would be limited to

daylight hours.

Now, imagine someone inventing the candle. This is progress! We can now have light even after sunset, extending our day.

But a candle isn't perfect. It's smoky, needs constant attention, and doesn't provide a lot of light. So, we invent the light bulb! Brighter, cleaner, and more efficient.

This progression from sunlight to candles to light bulbs illustrates a key principle in problem-solving and in AI: **evolution**. We start with a basic solution, identify its limitations, and then innovate to create something better. This continuous cycle of improvement is at the heart of AI development. From simple command-line interfaces, to mainframes, to cloud computing and serverless architectures, we build upon previous solutions to develop better solutions in the field of AI.

Figure 2: Evolution of problem-solving

Working with Sets

Let's talk about sets in Python. Think of a set as a collection of unique items. Imagine you have two bags of marbles.

- Bag A: Contains marbles with numbers 1, 2, and 3.
- Bag B: Contains marbles with numbers 3, 4, and 5.

A set only keeps unique values, so if there are 2 '1's in Bag A, the set will take only one '1'.

Now, what if you want to combine these bags? That's where set operations come in.

```
# Defining our sets
set_a = {1, 2, 3}
set_b = {3, 4, 5}
```

1. Union

The union of two sets combines all the elements from both sets, removing any duplicates. It's like emptying both bags into a new, larger bag.

```
# Using the union method
set_union = set_a.union(set_b)
print(set_union) # Output: {1, 2, 3, 4, 5}

# Another way to achieve the same result is using the "|" operator
set_union_2 = set_a | set_b
print(set_union_2) # Output: {1, 2, 3, 4, 5}
```

2. Intersection

The intersection of two sets finds the elements that are common to both. It's like finding the marbles that are present in *both* Bag A and Bag B.

```
# Using the intersection method
set_intersection = set_a.intersection(set_b)
print(set_intersection) # Output: {3}

# Another way to achieve the same result is using the "&" operator
set_intersection_2 = set_a & set_b
print(set_intersection_2) # Output: {3}
```

3. Difference

The difference between two sets finds the elements that are present in the first set but *not* in the second set. It's like finding the marbles that are in Bag A but not in Bag B.

```
# Using the difference method
set_difference = set_a.difference(set_b)
print(set_difference) # Output: {1, 2}
```

Handy Built-in Functions

Python comes with many built-in functions that can save you time and effort. Let's look at a few useful ones.

1. `abs()`: Absolute Value

This function returns the absolute value of a number. Think of it as the distance from zero, regardless of whether the number is positive or negative.

```
number = -10
absolute_value = abs(number)
print(absolute_value) # Output: 10
```

2. `pow()`: Power

This function calculates the power of a number. For example, 2 raised to the power of 4 (2^4).

```
base = 2
exponent = 4
result = pow(base, exponent)
print(result) # Output: 16
```

3. `round()`: Rounding Numbers

This function rounds a number to a specified number of decimal places.

```
pi_value = 3.14159
rounded_pi = round(pi_value, 2) # Round to 2 decimal places
print(rounded_pi) # Output: 3.14
```

4. `divmod()`: Division with Remainder

This function performs division and returns both the quotient and the remainder.

```
numerator = 10
denominator = 3
quotient, remainder = divmod(numerator, denominator)
print(quotient, remainder) # Output: 3 1
print(type((quotient, remainder))) # Output: <class 'tuple'>
```

Notice that `divmod()` returns the result as a *tuple*. A tuple is an immutable data structure that's used to store related data together. A key learning is that function designers choose the right datatypes to store the information in; in this example, Python has intelligently given the result in a tuple so that it is immutable, and we are prevented from changing the results of an operation.

5. `max()`, `min()`, and `sum()`: List Operations

These functions are useful for working with lists of numbers.

- `max()`: Returns the largest number in the list.
- `min()`: Returns the smallest number in the list.

- `sum()`: Returns the sum of all the numbers in the list.

```
numbers = [1, 5, 2, 8, 3]
maximum = max(numbers)
minimum = min(numbers)
total = sum(numbers)

print(maximum) # Output: 8
print(minimum) # Output: 1
print(total)   # Output: 19
```

6. `sqrt()` and `factorial()` from the `math` Module

Before you can use these functions, you need to import the `math` module.

```
import math

number = 16
square_root = math.sqrt(number)
print(square_root)

number = 4
factorial = math.factorial(number)
print(factorial)
```

The lesson here is, some basic math functions like `sum`, `min`, `max` are built into Python. However, for more complex mathematical functions like square root and factorial, you need to import the `math` module.

Working with Strings

Strings are sequences of characters. Python provides many built-in functions for manipulating strings.

```
text = "  Hello Tuesday  "
```

1. `len()`: String Length

This function returns the number of characters in a string, including spaces.

```
string_length = len(text)
print(string_length) # Output: 17
```

2. `upper()` and `lower()`: Case Conversion

These functions convert a string to uppercase or lowercase, respectively.

```
uppercase_text = text.upper()
lowercase_text = text.lower()

print(uppercase_text) # Output:   HELLO TUESDAY
print(lowercase_text) # Output:   hello tuesday
```

3. `replace()`: Replacing Substrings

This function replaces a specified substring with another substring.

```
new_text = text.replace("Tuesday", "Holiday")
print(new_text) # Output:   Hello Holiday
```

4. `strip()`: Removing Whitespace

This function removes leading and trailing whitespace from a string.

```
stripped_text = text.strip()
print(stripped_text) # Output: Hello Tuesday
```

5. `split()`: Splitting Strings

This function splits a string into a list of substrings based on a specified delimiter.

```
words = stripped_text.split(" ")
print(words) # Output: ['Hello', 'Tuesday']

# Another Example
text = "one,two,three"
numbers = text.split(",")
print(numbers) # Output: ['one', 'two', 'three']
```

6. `startswith()`: Checking String Start

This function checks if a string starts with a specified prefix.

```
starts_with_hello = stripped_text.startswith("Hello")
print(starts_with_hello) # Output: True
```

7. `find()`: Finding Substrings

This function finds the first occurrence of a substring within a string and returns its index. If the substring is not found, it returns -1.

```
index = stripped_text.find("Tue")
print(index) # Output: 6

index = stripped_text.find("Wed")
print(index) # Output: -1
```

Writing Modular Code with Functions

As your programs grow, it's essential to organize your code into reusable blocks. That's where functions come in.

A function is a named block of code that performs a specific task. It can take inputs (arguments) and return an output (return value).

```python
def string_length(text):
    count = 0
    for char in text:
        count += 1
    return count

text = "Hello Python"
length = string_length(text)
print(length) # Output: 12
```

1. Default Parameters

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

```python
def greet(name="Guest"):
    print("Hello, " + name)

greet("Alice") # Output: Hello, Alice
greet()        # Output: Hello, Guest
```

Inline Execution with Lambda Functions

Lambda functions are small, anonymous functions that can be defined inline. They are often used for short, simple operations.

```python
cube = lambda x: x * x * x
print(cube(3)) # Output: 27

number = lambda x: x * 10
print(number(3)) # Output: 30
```

Congratulations! You've taken your first steps towards mastering Python for AI. You've learned about sets, built-in functions, strings, functions, and lambda expressions. Keep practicing and exploring, and you'll be well on your way to becoming an AI expert!

Figure 3: Keep exploring AI with Python