PYTHON-DB IDLE development

*Breaking down a single line command input-*

1. **Command Type**:

Identify the type of command or operation the user intends to perform (e.g., assignment, function call, control flow statement, etc.).

1. **Tokens**:

Tokenize the input string to identify individual elements such as variable names, operators, literals, function names, etc.

Separate tokens based on whitespace, operators, parentheses, and other delimiters.

1. **Syntax Analysis**:

Analyze the syntax of the command to ensure it follows the correct syntax rules of the programming language (Python in this case).

Check for matching parentheses, proper indentation for blocks, correct use of operators, etc.

1. **Semantic Analysis**:

Perform semantic analysis to understand the meaning and intent of the command.

Check variable names for existence and scope, validate function calls and arguments, ensure proper usage of data structures and methods, etc.

1. **Execution Steps**:

Identify the sequence of steps required to execute the command successfully.

Resolve variable names to their corresponding values, evaluate expressions, handle control flow statements, etc.

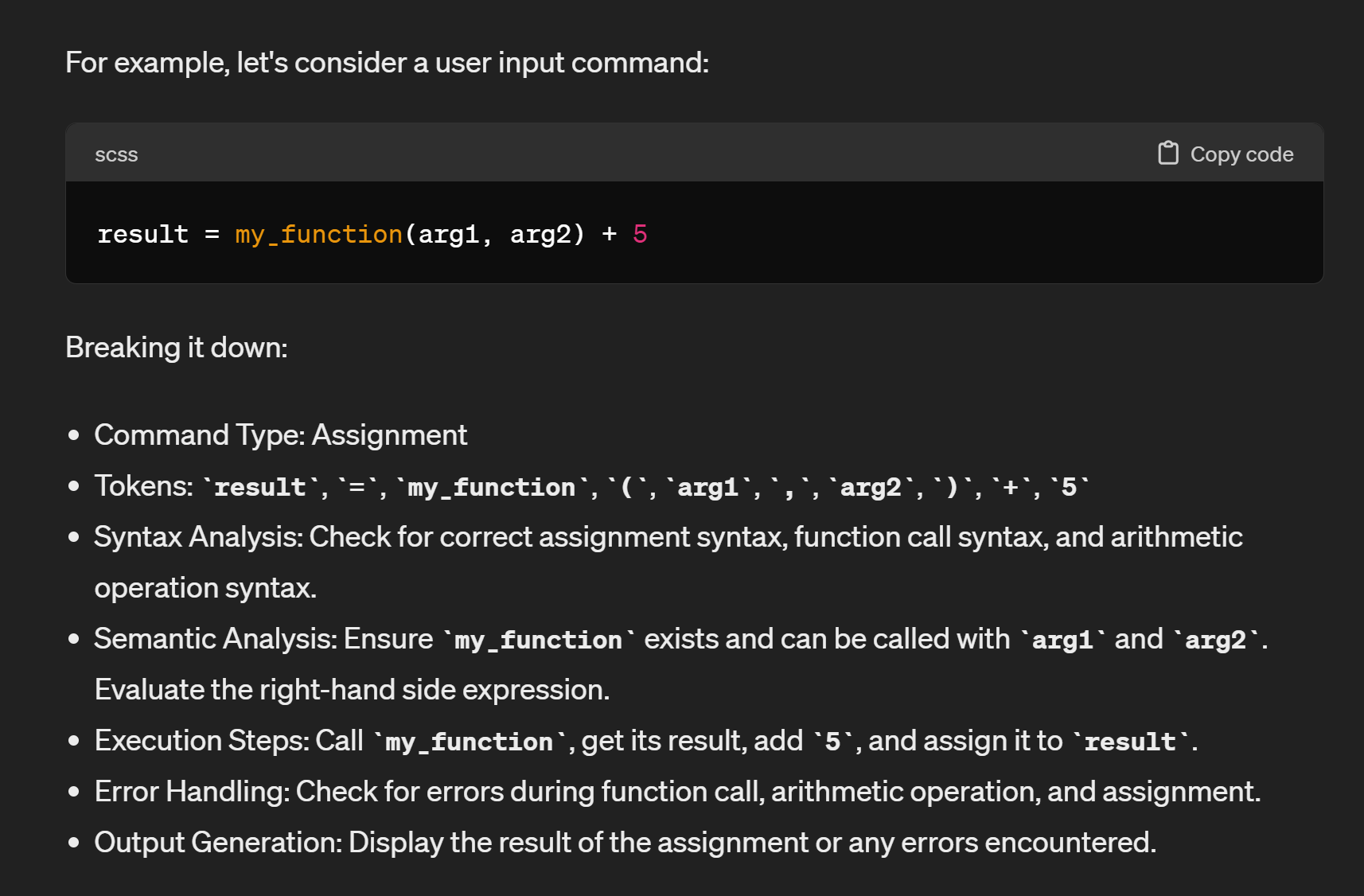
1. **Error Handling**:

Implement error handling mechanisms to catch and handle syntax errors, runtime errors, and logical errors in the user's command.

1. **Output Generation**:

Generate output or responses based on the execution result of the command.

Display errors, warnings, or successful execution messages to the user.

Ex: 

***Different type of command types that user can possibly give-***

1. **Assignment Command**:

- Example: `variable = value`

- Key Aspect: Look for the presence of the assignment operator `=`. Split the input into the left-hand side (variable) and right-hand side (value).

*Types of possible Assignment commands that user may give-*

# Assigning with input- Assigning values based on user input

# Ex. user\_input = input("Enter a value: ")

1. Slice assignment- Assigning values to the slice of the list

Ex. my\_list[1:3] = [10, 20]

1. Subscription Assignment- Assigning values to specific elements in dictionary & lists

Ex. my\_list[0] = 100

my\_dict["key"] = "value"

1. Augmented assignment- combining an arithmetic operation with assignment

Ex. total += amount

count -= 1

1. Function call assignment- Assigning the result of the function to the variable

Ex. result = calculate\_total(items)

length = len(name)

1. Arithmetic assignment- Using arithmetic operations in the assignment

Ex. a= b+23

1. **Mixture of all of these**-   
   Ex. a += len(list(b)) + add(2, 4) - 23

2. **Function Call Command**:

- Example: `function\_name(arg1, arg2)`

- Key Aspect: Look for the syntax of a function call with parentheses `()` and arguments separated by commas. Identify the function name and arguments.

A. Nested function calls- functions as arguments withing the other functions

Ex. total = add\_numbers(10, multiply\_numbers(5, 2))

3. **Arithmetic Operation Command**:

- Example: `result = x + y`

- Key Aspect: Look for arithmetic operators such as `+`, `-`, `\*`, `/`, `%`, `\*\*`(exponentiation) , `//`. Identify operands and operators to perform the operation.

- Augmented Assignment Operators such as `+=`, `-=`, `\*=`, `/=`, `//=`, `\*\*=`.

4**. Control Flow Statement**:

- Examples: `if`, `else`, `elif`, `for`, `while`

- Key Aspect: Look for keywords indicative of control flow statements. Check for proper syntax and indentation to determine the control flow structure.

A. Nesting of control statements.

B. if (***condition***)- condition can be an expression/ combination of TOKENS.

C. for x in (***iterable***):- iterable can be expression/ combination of TOKENS.

5. **List/Dictionary Operation Command**:

- Examples: `my\_list.append(item)`, `my\_dict[key] = value`

- Key Aspect: Look for operations specific to lists (`append`, `extend`, `pop`, etc.) or dictionaries (`get`, `update`, `pop`, etc.). Identify the target data structure and the operation being performed.

A. Accessing elements- `element = my\_list[index]`

B. Slicing- `subset = my\_list[start:end:step]`

C. Modifying lists-

* list.append(), list.sort(), list.insert(), list.extend(), list.pop() (i.e., returns an element), list.remove()… all the list built-in methods.

D. Counting and reversing- (list operations can be as *Tokens* too)

`a = list.count(‘element’)` .

6. **Input/Output Command**:

- Examples: `input("Enter a value: ")`,

`print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False).

- Key Aspect: Look for input/output functions such as `input()` and `print()`. Handle input prompts and display output messages accordingly.

7. **Import Statement (skippable\*)**:

- Example: `import module\_name`

- Key Aspect: Identify the `import` keyword followed by the module name. Handle importing modules and managing namespaces.

8. **Exception Handling (skippable\*)**:

- Examples: `try`, `except`, `finally`

- Key Aspect: Look for keywords related to exception handling. Identify try-except blocks and handle exceptions appropriately.

9. **Command Line Argument (skippable\*)**:

- Example: `python script.py arg1 arg2`

- Key Aspect: Identify command line arguments passed to a script or function. Parse and process command line arguments as needed.

10. **User-Defined Function Definition(Optional\*):**

- Example: `def my\_function(arg1, arg2):`

- Key Aspect: Look for the `def` keyword followed by the function name and parameter list. Handle defining user-defined functions and their parameters.

Identifying these key aspects can help categorize user input commands into different types, allowing the IDLE-like environment to parse, validate, and execute commands effectively based on their types.

Built-in functions ()

* **type(obj)**: Returns the type of the object obj. Implement a function that can determine and return the type of any object passed to it.
* **len(obj)**: Returns the length of the object obj. Implement a function that can calculate and return the length of lists, strings, dictionaries, etc.
* **int(value)**: Converts a value to an integer. Implement a function that can convert a string or float value to an integer.
* **float(value)**: Converts a value to a float. Implement a function that can convert a string or integer value to a float.
* **str(obj)**: Converts an object to a string. Implement a function that can convert different types of objects (integers, floats, lists, dictionaries, etc.) to strings.
* **input(prompt=None)**: Accepts user input from the console. Implement a function similar to input() that displays an optional prompt and returns the user's input.
* **print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)**: Prints objects to the console. Implement a function similar to print() that can handle printing of multiple objects with custom separator, end character, and file stream.
* **abs(x)**: Returns the absolute value of x. Implement a function that calculates and returns the absolute value of a number.
* **max(iterable, \*args, key=None, default=None)**: Returns the maximum element from an iterable or a sequence of arguments. Implement a function that finds and returns the maximum value from a list or other iterable.
* **min(iterable, \*args, key=None, default=None)**: Returns the minimum element from an iterable or a sequence of arguments. Implement a function that finds and returns the minimum value from a list or other iterable.
* **sum(iterable, start=0)**: Returns the sum of elements in an iterable, optionally starting with a specified value. Implement a function that calculates and returns the sum of elements in a list or other iterable.
* **round(number, ndigits=None)**: Rounds a number to a specified number of decimal digits. Implement a function that rounds a floating-point number to the specified number of decimal places.

Built-in list.methods()

* **append(element):** Adds an element to the end of the list.
* **extend(iterable)**: Extends the list by appending elements from the iterable.
* **insert(index, element)**: Inserts an element at the specified index in the list.
* **remove(element)**: Removes the first occurrence of the specified element from the

list.

* **pop(index=-1)**: Removes and returns the element at the specified index. If no index is specified, removes and returns the last element.
* **index(element, start=0, end=len(list))**: Returns the index of the first occurrence of the specified element in the list within the specified start and end indexes.
* **Count(element)**: Returns the number of occurrences of the specified element in the list.
* **sort(key=None, reverse=False):** Sorts the elements of the list in ascending order. Optional parameters key and reverse allow custom sorting and reverse sorting, respectively.
* **reverse()**: Reverses the order of elements in the list.
* **copy()**: Returns a shallow copy of the list.
* **clear()**: Removes all elements from the list.

Built-in dictionary.methods()

* **get(key, default=None)**: Returns the value associated with the specified key. If the key is not found, returns the default value (or None if not specified).
* **keys()**: Returns a view object that displays a list of all the keys in the dictionary.
* **values()**: Returns a view object that displays a list of all the values in the dictionary.
* **items()**: Returns a view object that displays a list of tuples containing key-value pairs in the dictionary.
* **update(other\_dict)**: Updates the dictionary with key-value pairs from another dictionary or an iterable of key-value pairs.
* **pop(key, default=None)**: Removes the specified key and returns its associated value. If the key is not found, returns the default value (or raises KeyError if not specified).
* **popitem()**: Removes and returns an arbitrary key-value pair from the dictionary as a tuple. Useful for LIFO (Last In, First Out) operations.
* **clear()**: Removes all items from the dictionary.
* **copy()**: Returns a shallow copy of the dictionary.
* **setdefault(key, default=None)**: Returns the value associated with the specified key. If the key is not found, inserts the key with the default value (or None if not specified) and returns the default value.
* **fromkeys(keys, value=None)**: Creates a new dictionary with keys from an iterable keys and values set to the specified value (or None if not specified) for each key.

Methods for Integer (`int`) Data Type(Optional\*)

- `**bit**\_**length**()`: Returns the number of bits required to represent the integer.

- `**to**\_**bytes**(length, byteorder)`: Converts the integer to a bytes object.

- `**from**\_**bytes**(bytes, byteorder)`: Converts bytes to an integer.

- `\_\_**add**\_\_(self, other)`: Adds two integers.

- `\_\_**sub**\_\_(self, other)`: Subtracts two integers.

- `\_\_**mul**\_\_(self, other)`: Multiplies two integers.

- `\_\_**truediv**\_\_(self, other)`: Performs true division.

- `\_\_**floordiv**\_\_(self, other)`: Performs floor division.

- `**\_\_mod\_\_**(self, other)`: Computes the modulus.

- `**\_\_pow\_\_**(self, exponent, modulus=None)`: Computes the power.

Methods for Boolean (`bool`) Data Type

- No specific methods are defined for the `bool` type as it represents True or False values directly.

Methods for Floating-Point (`float`) Data Type(Optional\*)

- `**is\_integer**()`: Checks if the float value is an integer.

- `**hex**()`: Returns a hexadecimal representation of the float.

- `**fromhex**(s)`: Converts a hexadecimal string to a float.

- `**as\_integer\_ratio**()`: Returns a tuple representing the float as a fraction.

- `**\_\_add\_\_**(self, other)`: Adds two float numbers.

- `**\_\_sub\_\_**(self, other)`: Subtracts two float numbers.

- `\_\_**mul**\_\_(self, other)`: Multiplies two float numbers.

- `\_\_**truediv**\_\_(self, other)`: Performs true division.

- `\_\_**floordiv**\_\_(self, other)`: Performs floor division.

- `\_\_**mod**\_\_(self, other)`: Computes the modulus.

- `\_\_**pow**\_\_(self, exponent)`: Computes the power.

Methods for String (`str`) Data Type

- `**capitalize**()`: Capitalizes the first character of the string.

- `**lower**()`: Converts the string to lowercase.

- `**upper**()`: Converts the string to uppercase.

- `**title**()`: Converts the string to title case.

- `**swapcase**()`: Swaps the case of characters in the string.

- `**strip**()`: Removes leading and trailing whitespace characters.

- `**lstrip**()`: Removes leading whitespace characters.

- `**rstrip**()`: Removes trailing whitespace characters.

- `**split**(sep=None, maxsplit=-1)`: Splits the string into a list of substrings.

- `**join**(iterable)`: Joins the elements of an iterable with the string as a separator.

- `**replace**(old, new, count=-1)`: Replaces occurrences of a substring.

- `**find**(sub, start=0, end=-1)`: Finds the first occurrence of a substring.

- `**count**(sub, start=0, end=-1)`: Counts occurrences of a substring.

- `**startswith**(prefix, start=0, end=-1)`: Checks if the string starts with a specified prefix.

- `**endswith**(suffix, start=0, end=-1)`: Checks if the string ends with a specified suffix.

2. Tokens

Tokens are the smallest units of a program that are meaningful to the compiler or interpreter. In Python, tokens are categorized into several types based on their functionality and purpose within the code. Here are the main types of tokens in Python:-

1. Identifiers:

- Identifiers are names given to variables, functions, classes, modules, etc.

- Rules for identifiers:

- Must start with a letter (a-z, A-Z) or an underscore (\_).

- Can be followed by letters, digits (0-9), or underscores.

- Cannot be a keyword or reserved word.

2. Keywords:

- Keywords are reserved words that have special meaning and are part of the Python language syntax.

- Examples: `if`, `else`, `for`, `while`, `def`, `class`, `import`, `True`, `False`, `None`, etc.

3. Literals:

- Literals represent fixed values in Python code.

- Types of literals:

- Numeric literals (integers, floats, complex numbers)

- String literals (enclosed in single quotes, double quotes, or triple quotes)

- Boolean literals (`True`, `False`)

- None literal (`None`)

4. Operators:

- Operators perform operations on operands and produce a result.

- Types of operators:

- Arithmetic operators (`+`, `-`, `\*`, `/`, `%`, `//`, `\*\*`)

- Comparison operators (`==`, `!=`, `>`, `<`, `>=`, `<=`)

- Assignment operators (`=`, `+=`, `-=`, `\*=`, `/=`, etc.)

- Logical operators (`and`, `or`, `not`)

- Bitwise operators (`&`, `|`, `^`, `<<`, `>>`, `~`)

- Membership operators (`in`, `not in`)

- Identity operators (`is`, `is not`)

5. Delimiters:

- Delimiters are characters used to mark the boundaries of code elements.

- Examples: parentheses `()`, brackets `[]`, braces `{}`, commas `,`, colons `:`, semicolons `;`, periods `.`

6. Whitespace:

- Whitespace characters include spaces, tabs, and newline characters.

- They are used for indentation and to separate tokens but are generally ignored by the interpreter.

7. Comments:

- Comments are not considered part of the code but provide explanations or annotations.

- Single-line comments start with `#`, and multi-line comments can be enclosed in triple quotes `'''`.

8. Special Symbols (Optional\*):

- Special symbols include symbols like `@` used in decorators, `\_` used in variable names, and `$` in some contexts.