Cell 1: Imports and Basic Python Concepts

This cell introduces Python modules, comments, and the concept of imports.

- **Modules**: Files containing reusable code.
- **Imports**: Bring functionality from other files/modules into your program.
- **Comments**: Notes in the code ignored by Python to explain functionality.

```
# PYTHON CONCEPT: Modules and Imports
# - Modules are Python files containing code we can reuse
# - 'import' lets us use code from other files/modules
# - These are built-in modules (come with Python)

import time      # For adding delays in our program
import random    # For generating random numbers and choic
import os        # For operating system related functions

# PYTHON CONCEPT: Comments
# - Lines starting with '#' are comments
# - Comments are notes that Python ignores when running
# - They help explain what the code does
```

Cell 2: Classes and Object-Oriented Programming

This cell introduces classes and object-oriented programming concepts.

- **Classes**: Templates for creating objects with properties and behaviors.
- **Methods**: Functions inside classes that define actions.
- **Instance Variables**: Variables unique to each object created from a class.
- **Docstrings**: Descriptions for classes and methods enclosed in triple quotes.

Explain code

You

Please explain this code.

Gemini

Okay, let's break down this part of the code:

```
def clear_screen():
    os.system('cls' if os.name

def visualize_world(world, age
    visualization = "+--" * le
    for i in range(len(world))
        if i == agent_position
            visualization += "
        else:
            visualization += f
    visualization += "\n+" + "
    return visualization
```

Use code with caution

This code snippet defines two functions: `clear_screen` and `visualize_world`.

1. **`clear_screen` function:**
   - **Purpose:** This function is designed to clear the output screen of the terminal or console where the code is running.
   - **How it works:**
     - It uses the `os.sy` function to execute a system command.
     - The command it executes depends on the operating system:
       - If the OS is Windows

```
class SimpleAgent:
    # PYTHON CONCEPT: Constructor Method
    def __init__(self, world_size=10):
        """
        PYTHON CONCEPT: Docstrings
        - Provide documentation for classes/functions
        """
        self.position = 0
        self.world_size = world_size
        self.score = 0

    def sense(self, world):
        try:
            if self.position + 1 < len(world):
                return world[self.position + 1]
            return None
        except IndexError:
            return None

    def decide(self, observation):
        if observation is None:
            return 'stop'
        if observation == 'X':
            return 'stop'
        elif observation == 'G':
            return 'move'
        elif observation == '_':
            return 'move'
        return 'stop'
```

Cell 3: Functions and List Operations

This cell explains Python functions, loops, and list operations.

- **Functions**: Reusable blocks of code.
- **Loops**: Repeat actions on lists or ranges.
- **List Operations**: Indexing, creation, and comprehension.

```
def create_world(size=10, num_obstacles=2):
    world = ['_'] * size
    obstacle_positions = random.sample(range(2, size-2),
    for pos in obstacle_positions:
        world[pos] = 'X'
```

(`os.name == 'nt'`), it executes `'cls'`, which is the Windows command to clear the screen.
    - If the OS is not Windows (e.g., Linux or macOS), it executes `'clear'`, the command used in those systems to clear the screen.

2. `visualize_world` **function:**

- **Purpose:** This function creates a text-based visual representation of the `world` (which is likely a list representing the environment) and the `agent_position` within that world.
- **How it works:**
    - It takes two arguments: `world` (the list representing the environment) and `agent_position` (an integer indicating the agent's location in the world).
    - It initializes an empty string `visualization` to store the visual representation.
    - It builds the visualization string step-by-step:
        - It creates the top border using `"+--"` repeated for

```
    while True:
        goal_pos = random.randint(size//2, size-2)
        if world[goal_pos] == '_':
            world[goal_pos] = 'G'
            break
    return world

test_world = create_world()
print("Example world:", test_world)
```

```
Example world: ['_', '_', '_', '_', '_', 'X', 'X', '
```

Cell 4: String Formatting and Visualization

This cell covers Python-specific string operations and visualization techniques.

- **String Multiplication**: Repeats a string.
- **Enumerate**: Retrieves index and value from a list.
- **f-strings**: Embed variables inside strings.

```
def clear_screen():
    os.system('cls' if os.name == 'nt' else 'clear')

def visualize_world(world, agent_position):
    visualization = "+--" * len(world) + "+\n|"
    for i in range(len(world)):
        if i == agent_position:
            visualization += "A |"
        else:
            visualization += f"{world[i]} |"
    visualization += "\n+" + "--" * len(world) + "+"
    return visualization
```

Cell 5: Main Simulation Loop

This cell contains the main logic for running the simulation.

- **Main Functionality**:
  - Agent senses the environment, decides, and acts.
  - Simulation runs for a set number of steps.
- **Python Features**:
  - the length of the `world` and adds a `"+"` and a newline character `"\n"` followed by a `"|"` to start the first row.
  - It iterates through each position in the `world` using a `for` loop.
  - If the current position (`i`) matches the `agent_position`, it adds `"A |"` to the visualization string, representing the agent.
  - If the current position is not the agent's position, it adds the element from the `world` list at that position followed by `" |"` to the visualization string.
  - Finally, it ad newline character and the bot border (similar to the top border) to complete the visualization.
  - It returns the `visualization` string.

- Uppercase strings, delays, and method chaining.

```
def run_simulation(steps=15, world_size=10, delay=1):
    world = create_world(world_size)
    agent = SimpleAgent(world_size)
    print("Starting simulation...".upper())
    time.sleep(2)
    for step in range(steps):
        clear_screen()
        observation = agent.sense(world)
        action = agent.decide(observation)
        print(f"\nStep {step + 1}/{steps}")
        print(visualize_world(world, agent.position))
        time.sleep(delay)
```

Cell 6: Program Entry Point and Error Handling

This cell runs the simulation if the script is executed directly.

- **Error Handling**:
  - Catching user interruptions and unexpected errors.
- **Constants**:
  - Defined values for the simulation parameters.

```
def clear_screen():
    os.system('cls' if os.name == 'nt' else 'clear')

def visualize_world(world, agent_position):
    visualization = "+--" * len(world) + "+\n|"
    for i in range(len(world)):
        if i == agent_position:
            visualization += "A |"
        else:
            visualization += f"{world[i]} |"
    visualization += "\n+" + "--" * len(world) + "+"
    return visualization
```

Cell 7: Testing Functions

In essence, this code provides a way to visually represent a simple environment with an agent in it and to clear the screen for updated visualizations during a simulation or game. I hope this explanation clarifies things for you! Let me know if you have any further questions.

▶ Sources

Rate this answer

This cell defines unit tests to validate individual components.

- **Assertions**: Validate expected behavior.
- **Testing Framework**: Basic assertions for key functionalities.

```
def run_tests():
    print("Running tests...")
    agent = SimpleAgent(10)
    assert agent.position == 0, "Starting position should
    world = create_world(5, 1)
    assert len(world) == 5, "World length should be 5"
    print("All tests passed! ✅")

run_tests()
```

```
Running tests...
All tests passed! ✅
```