Introduction to Artificial Intelligence — Week 2

Welcome to Week 1: a friendly, thorough introduction to Artificial Intelligence (AI). This lecture is designed for undergraduates and instructors preparing classroom material. We cover foundational definitions, historic roles of key thinkers, practical programming contrasts (with and without AI), core applications (games, NLP, expert systems, vision, robotics), and an extended focus on intelligent agents — structure, behavior, rationality, and agent program types. Each slide includes clear definitions, example code snippets, learning notes, and a real photographic image to ground abstract ideas.

Tip: Read slides sequentially. Each card builds on the previous one. Theme colors used for emphasis: #876cd4ff, #D783D8, #FF90A5, #FFB071, #14083A.

Foundations of AI — What is AI?

Definition: Artificial Intelligence is the field that studies how to build computer systems that perform tasks which, when done by humans, require intelligence. These tasks include learning, reasoning, perception, language understanding, and decision-making.

Core ideas: representation (how we describe knowledge), search and optimization (how we find solutions), learning (how systems adapt), and reasoning (how systems draw conclusions). Foundations mix theory from logic, probability, statistics, and algorithms with experimental practice.

Learning targets for this topic

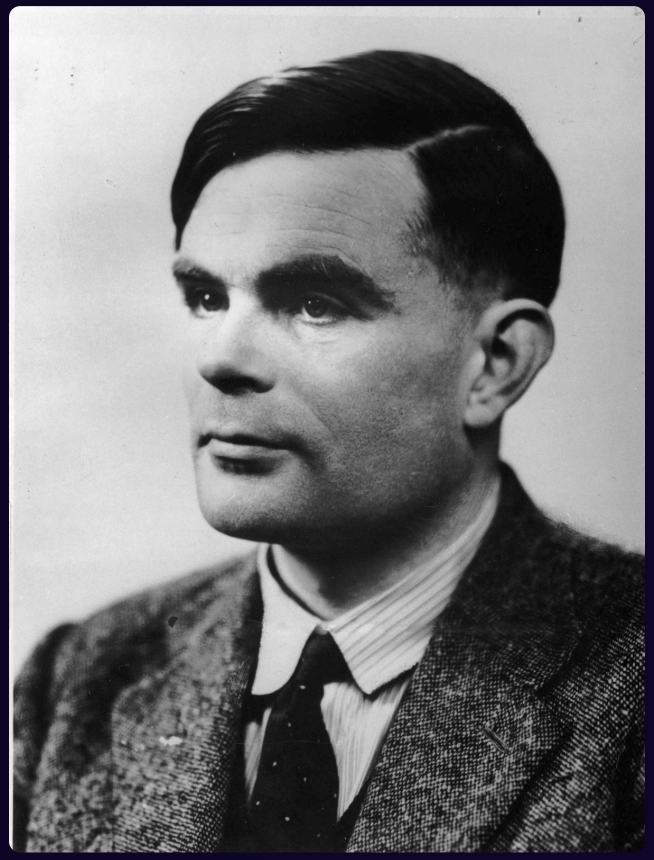
- Recognize AI tasks and everyday examples (spam filters, search, recommendations).
- Differentiate symbolic vs. statistical approaches.
- Understand the role of models, data, and evaluation metrics.

Code example — Simple search (Python)

```
def linear_search(arr, x): for i, v in enumerate(arr): if v == x: return i return -1 # Usage print(linear_search([3,1,4,1,5], 4)) # returns 2
```

Note: This small function shows algorithmic thinking — a foundation for building intelligent systems. Later, we replace simple algorithms with learned models when data-driven behavior is needed.

Role of Philosophers, Mathematicians, Psychologists, and Engineers



Interdisciplinary roots:

- **Philosophers**: Ask what intelligence and mind mean, propose thought experiments (e.g., Turing Test), and clarify ethical implications.
- **Mathematicians**: Provide formal models, logic, probability theory, and complexity analysis used to prove properties of algorithms.
- **Psychologists**: Study human learning and cognition; inspire computational models of memory, perception, and learning.
- **Computer Engineers**: Build hardware and software to implement AI systems at scale, optimizing speed, memory, and deployment.

Each discipline contributes questions, formalisms, and practical methods. Effective AI design borrows from all four to make robust, explainable systems.

Code example — Demonstrating a simple probabilistic idea (Bayes)

```
def bayes(p_h, p_e_given_h, p_e_given_not_h): p_not_h = 1 - p_h p_e = p_e_given_h * p_h + p_e_given_not_h * p_not_h return (p_e_given_h * p_h) / p_e # Example: disease probability update print(bayes(0.01, 0.9, 0.05))
```

Programming Without AI vs. With AI — What's Different?

Clear contrast:

1

Traditional Programming

Rule-based: Developers specify exact rules and logic. Input -> Program -> Output. Works well when behavior is fully understood and can be encoded.

Example: Sorting algorithms, parsers, calculators.

2

AI-driven Programming

Data-driven: Systems learn behavior from examples. Input + Data -> Model Training -> Inference. Useful when rules are hard to specify but examples are available.

Example: Image classifiers, language models, recommendation systems.

Code comparison — deterministic vs learned

```
# Deterministic rule def classify_age(age): return "adult" if age >= 18 else "minor" # Learned model (pseudo code) # features, labels = load_data() # model = train_model(features, labels) # prediction = model.predict(new_features)
```

Learning target: Understand when to write explicit logic and when to collect data and build models. Both techniques are part of the AI toolkit.

Applications of AI — Overview & Examples

AI is widely applied. Brief map:

Gaming

AI controls opponents, plans strategies, and adapts to player skill. Techniques: search (Minimax), reinforcement learning.

```
# Example: pseudocode for Minimax decision def minimax(state, depth, maximizing): if depth==0 or terminal(state): return evaluate(state) if maximizing: return max(minimax(child, depth-1, False) for child in children(state)) else: return min(minimax(child, depth-1, True) for child in children(state))
```

Natural Language Processing (NLP)

AI systems understand and generate language: translation, summarization, chatbots. Techniques: probabilistic models, neural networks, transformers.

```
# Simple tokenization example tokens = text.lower().split() # naive tokenizer
```

Expert Systems

Rule-based decision support (medical diagnosis, troubleshooting). Techniques: knowledge bases, inference engines.

```
# Rule example if fever and rash and travel_history: diagnosis = "possible infection"
```

Computer Vision

Image classification, object detection, segmentation. Techniques: convolutional neural networks (CNNs), transfer learning.

```
# Using a pretrained model (concept) model = load_pretrained_cnn() pred = model.predict(image)
```

Intelligent Robots

Robots combine perception, planning, and control to act in the physical world. Techniques: SLAM, motion planning, reinforcement learning.

```
# Pseudocode for a simple reactive controller if obstacle_ahead(): turn_left() else: move_forward()
```

Introduction to Artificial Agents — Definitions and Scope

Definition: An agent is anything that perceives its environment through sensors and acts upon that environment through actuators. An artificial agent is a program or machine designed to achieve goals by taking actions.

Scope: Agents can be simple (reactive thermostats) or complex (autonomous vehicles). We study their architectures, decision-making, and evaluation criteria like success rate, efficiency, and safety.

Learning objectives

- Explain agent basics: sensors, actuators, environment, and goals.
- Classify agent types by autonomy and reasoning capabilities.
- Relate agents to real-world systems: chatbots, robots, trading bots.

Code example — Minimal agent loop (Python)

class SimpleAgent: def __init__(self): pass def perceive(self, sensor_data): return sensor_data def act(self, decision): print("Acting:", decision) agent = SimpleAgent() while True: sensors = get_sensors() # placeholder decision = agent.perceive(sensors) agent.act(decision) break

Structure of an Agent — Components Explained

Key components:

1. Sensors

Gather raw data from the environment (cameras, microphones, temperature sensors, keyboard input).

2. Perception

Process sensor data into meaningful features (image preprocessing, speech-to-text).

3. Decision-making

Choose actions using rules, search, planning, or learned policies.

4. Actuators

Effectors that change the environment (motors, display outputs, network messages).

5. Memory & Learning

Store past experiences and improve future decisions (replay buffers, model updates).

Code example — simple perception function:

```
def preprocess_image(img): # convert to grayscale, resize, normalize gray = img.convert('L') small = gray.resize((64,64)) arr = np.array(small) / 255.0 return arr
```

How an Agent Should Act — Rationality, Goals, and Performance

Rationality: An agent is rational if it acts to maximize expected performance measured by a predefined performance metric given its knowledge and available resources. Rational behavior depends on:

- **Performance measure**: What counts as success (accuracy, reward, throughput).
- **Prior knowledge**: The agent's initial information about the environment.
- **Available actions**: What the agent can physically or digitally do.
- **Perceptions**: What the agent observes through sensors (partial or noisy).

Important concept: Bounded rationality — agents have limited time and computational power, so they often use heuristics or approximate methods rather than perfect optimization.

Code example — Expected utility selection (Python)

```
def choose_action(actions, expected_returns): # actions: ['a','b','c'], expected_returns: [1.2, 0.5, 0.9] return actions[int(np.argmax(expected_returns))]
```

Learning goal: Evaluate trade-offs between optimality and resource constraints. Design performance measures that align with real objectives and safety.

Concepts: Rationality vs Omniscience & Types of Agent Programs

Definitions:

- **Rationality**: Acting to achieve the best expected outcome given what the agent knows and can do.
- **Omniscience**: Hypothetical state of perfect knowledge about the world. Real agents are not omniscient; designs must handle uncertainty.

Types of agent programs (common taxonomy) — examples and code sketches:

1

Table-driven Agent

Maps every possible percept sequence to an action. Impractical for large spaces.

```
# impractical: lookup[percept_sequence] = action
```

2

Simple Reflex Agent

Selects actions based on current percept using condition-action rules. Fast, limited memory.

```
if percept == 'obstacle': action='turn'
```

3

Model-based Reflex Agent

Maintains an internal model of the world to handle partially observable environments.

```
state = update_state(state, percept)
```

4

Goal-based Agent

Chooses actions to achieve explicit goals using search or planning.

```
plan = search(start, goal)
```

5

Utility-based Agent

Chooses actions that maximize a utility function reflecting preferences among states.

```
action = argmax_a expected_utility(a)
```

6

Learning Agent

Improves performance over time through experience. Core components: critic, learning element, problem generator, performance element.

```
# RL outline: # agent observes state, picks action, receives reward, updates policy
```

Learning objective: Match the agent type to the problem constraints — observability, environment dynamics, and resource limits.

Putting It All Together — Study Guide, Next Steps, and Practical Exercises

Summary & study roadmap:

Week 1 Review

Definitions, history, multidisciplinary roles, programming contrasts, applications, and agents.

Hands-on Practice

Implement simple search, a reflex agent simulator, and a small supervised ML example (classification on tiny dataset).

3

Mini Project

Build a simulated agent (grid world) that navigates to a goal using either rule-based or learning approach; compare results.

4

Discussion Topics

Ethics of AI, limits of rationality, when to prefer rule-based vs. learned solutions, and reproducibility in experiments.

Practical exercises (detailed)

1. Implement a simple grid-world environment and code a reflex agent that avoids obstacles. Test and report failure cases.
2. Train a small classifier (e.g., scikit-learn) on a toy dataset (Iris or digits). Compare deterministic rules vs learned accuracy.
3. Read a short paper: "Computing Machinery and Intelligence" by Alan Turing. Write a one-page reflection linking philosophical questions to practical agent design.

Final code seed — supervised learning (Python, scikit-learn)

```
from sklearn.datasets import load_iris from sklearn.model_selection import train_test_split from sklearn.ensemble import RandomForestClassifier X,y = load_iris(return_X_y=True) X_train,X_test,y_train,y_test = train_test_split(X,y, test_size=0.3, random_state=42) clf = RandomForestClassifier() clf.fit(X_train, y_train) print("Test accuracy:", clf.score(X_test, y_test))
```

Closing note: This week establishes the language and mental models you'll use throughout the course. Keep experimenting, ask questions, and connect theoretical ideas to code and real images. For instructors: use the images and exercises as in-class demonstrations and assign the mini project for follow-up labs.