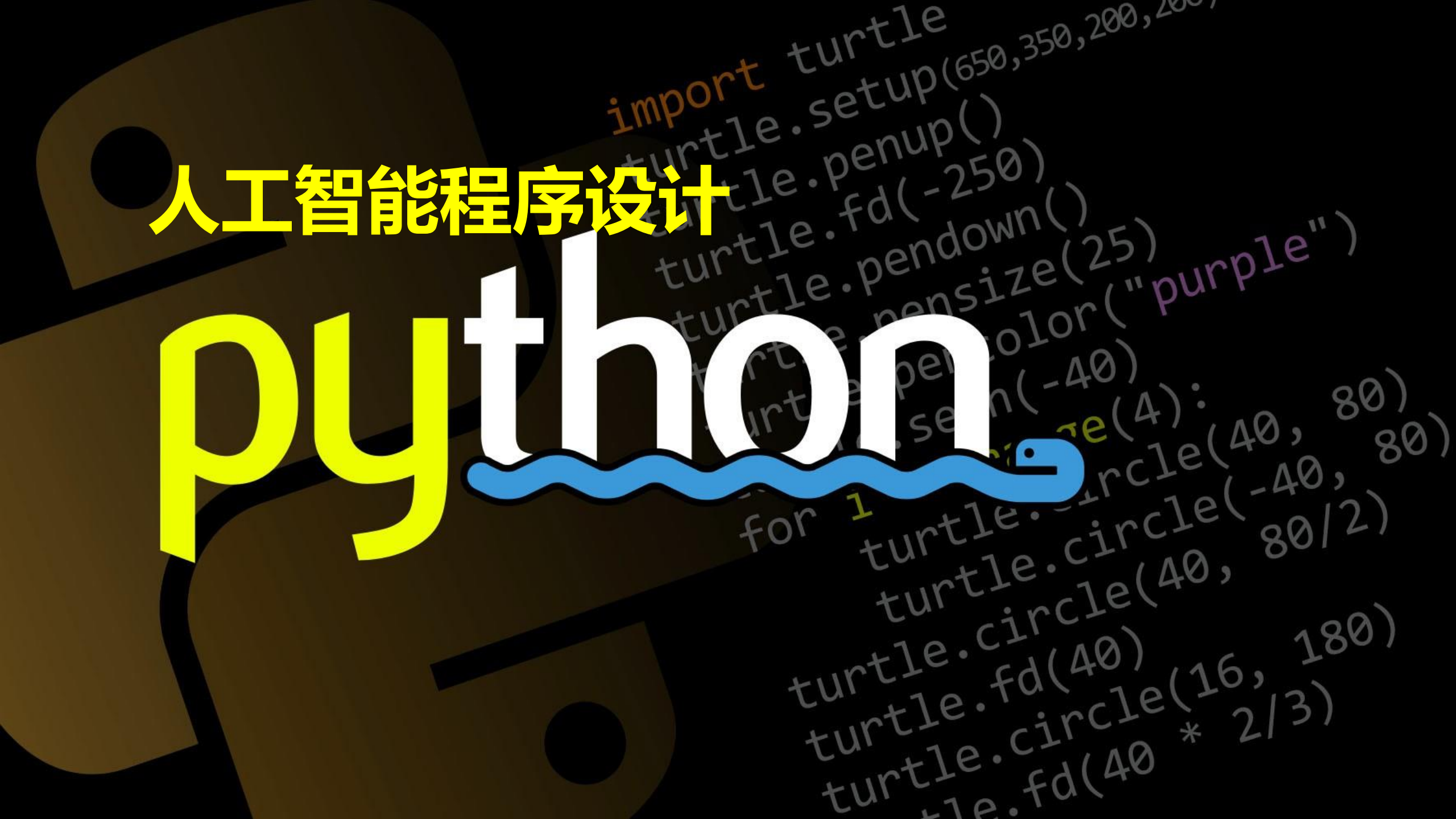
人工智能程序设计


python
import turtle
turtle.setup(650,350,200,200)
turtle.penup()
turtle.fd(-250)
turtle.pendown()
turtle.pensize(25)
turtle.pencolor("purple")
turtle.seth(-40)
for 1 in range(4):
turtle.circle(40, 80)
turtle.circle(-40, 80)
turtle.fd(40)
turtle.circle(40, 80/2)
turtle.fd(40)
turtle.circle(16, 180)
turtle.fd(40 * 2/3)

人工智能程序设计

17.1 从大模型应用到智能体

北京石油化工学院 人工智能研究院

刘 强

学习内容

- 智能体基本概念与架构
- 智能体与传统应用的区别
- 智能体开发框架选择

17.1.1 什么是智能体

学习内容：

- 智能体的定义
- 智能体基本架构
- 与传统应用的区别

智能体定义

智能体（Agent）是人工智能领域的核心概念，代表着能够感知环境、做出决策并采取行动以实现特定目标的自主系统。

与传统的大模型应用相比，智能体具有：

- 更强的**主动性**
- 更高的**自主性**
- 更好的**适应性**

智能体基本架构

智能体通常包含四个核心组件：

组件	功能
感知	获取环境信息
思考	基于感知制定计划
行动	执行具体操作
学习	优化行为策略

智能体基础代码

Agent 类封装了智能体的基本功能，包括环境感知、决策思考和行动执行，为构建复杂智能体系统提供基础框架。

```python
class Agent:
    def __init__(self, name, goals):
        self.name = name
        self.goals = goals
        self.memory = []
        self.tools = {}

    def perceive(self, environment):
        # 感知环境信息
        return environment.get("user_input", "")

    def think(self, perception):
        # 基于感知制定计划
        return {"action": "respond", "content": perception}

    def act(self, decision):
        # 执行具体行动
        return "执行行动: {}".format(decision['content'])
```

智能体与传统应用的区别

特性	传统应用	智能体
主动性	被动响应	主动感知
持续性	单次交互	持续运行
自主性	固定流程	动态决策
适应性	预设规则	自主学习

传统应用 vs 智能体应用

智能体通过感知-思考-行动-学习的完整循环处理任务。

```
# 传统大模型应用
def traditional_app(user_input):
    response = llm.generate(user_input)
    return response

# 智能体应用
def agent_app(user_input, context):
    perception = agent.perceive({"user_input": user_input, "context": context})
    decision = agent.think(perception)
    result = agent.act(decision)
    agent.learn({"input": user_input, "result": result})
    return result
```

17.1.2 智能体架构原理

学习内容：

- 反应式架构
- 深思熟虑式架构
- 混合式架构

反应式架构

反应式架构直接将感知映射到行动，适合需要快速响应的简单任务。

优势： 响应速度快

局限： 缺乏复杂推理能力

```
class ReactiveAgent:
    def __init__(self):
        self.rules = {
            "greeting": "你好！我是智能助手",
            "question": "让我来帮您解答这个问题",
            "task": "我会立即处理这个任务"
        }

    def respond(self, input_type):
        return self.rules.get(input_type, "我不确定如何回应")
```

深思熟虑式架构

深思熟虑式架构在行动前进行复杂的推理和规划，适合多步骤解决的复杂任务。

```
class DeliberativeAgent:
    def __init__(self, llm):
        self.llm = llm
        self.planning_prompt = """
        任务: {task}
        目标: {goal}
        请制定详细的执行计划:
        """

    def plan_and_execute(self, task, goal):
        plan = self.llm.generate(
            self.planning_prompt.format(task=task, goal=goal))
        steps = plan.split('\n')
        results = []
        for step in steps:
            if step.strip():
                result = self.execute_step(step)
                results.append(result)
        return results
```

17.1.3 智能体开发框架选择

学习内容：

- LangGraph框架特点
- CrewAI框架特点
- 框架选择建议

主流智能体开发框架

框架	特点	适用场景
LangGraph	基于状态图，工作流管理	单智能体复杂工作流
CrewAI	多智能体协作	团队化问题解决
AutoGPT	自主任务分解	开放式任务探索

LangGraph框架

LangGraph是基于状态图的智能体开发框架，适合构建复杂的工作流程。

通过有向图组织工作流程，使复杂推理过程直观可控。

```
from langgraph import StateGraph

# 创建状态图
workflow = StateGraph()
workflow.add_node("analyze", analyze_task)
workflow.add_node("plan", create_plan)
workflow.add_node("execute", execute_plan)
workflow.add_edge("analyze", "plan")
workflow.add_edge("plan", "execute")

app = workflow.compile()
result = app.invoke({"task": "分析市场数据"})
```

CrewAI框架

CrewAI专注于多智能体协作，通过角色分工实现团队化问题解决。

```python
from crewai import Agent, Task, Crew

# 创建专业化智能体
researcher = Agent(role="研究员", goal="收集和分析信息")
writer = Agent(role="写作者", goal="创作高质量内容")

# 定义协作任务
research_task = Task(description="研究AI发展趋势", agent=researcher)
writing_task = Task(description="撰写技术报告", agent=writer)

# 组建智能体团队
crew = Crew(agents=[researcher, writer],
            tasks=[research_task, writing_task])
result = crew.kickoff()
```

框架选择建议

应用场景	推荐框架
单智能体复杂工作流	LangGraph
多智能体团队协作	CrewAI
需要精细状态控制	LangGraph
需要角色专业分工	CrewAI

实践练习

练习 17.1.1：智能体概念理解

设计一个简单的智能体类，实现基本的感知、思考、行动功能，体验智能体与传统程序的区别。

实践练习

练习 17.1.2：架构模式对比

分别实现反应式和深思熟虑式智能体，比较它们在不同任务场景下的表现差异。

实践练习

练习 17.1.3：框架选择分析

分析LangGraph和CrewAI的特点，根据具体应用场景选择合适的开发框架。