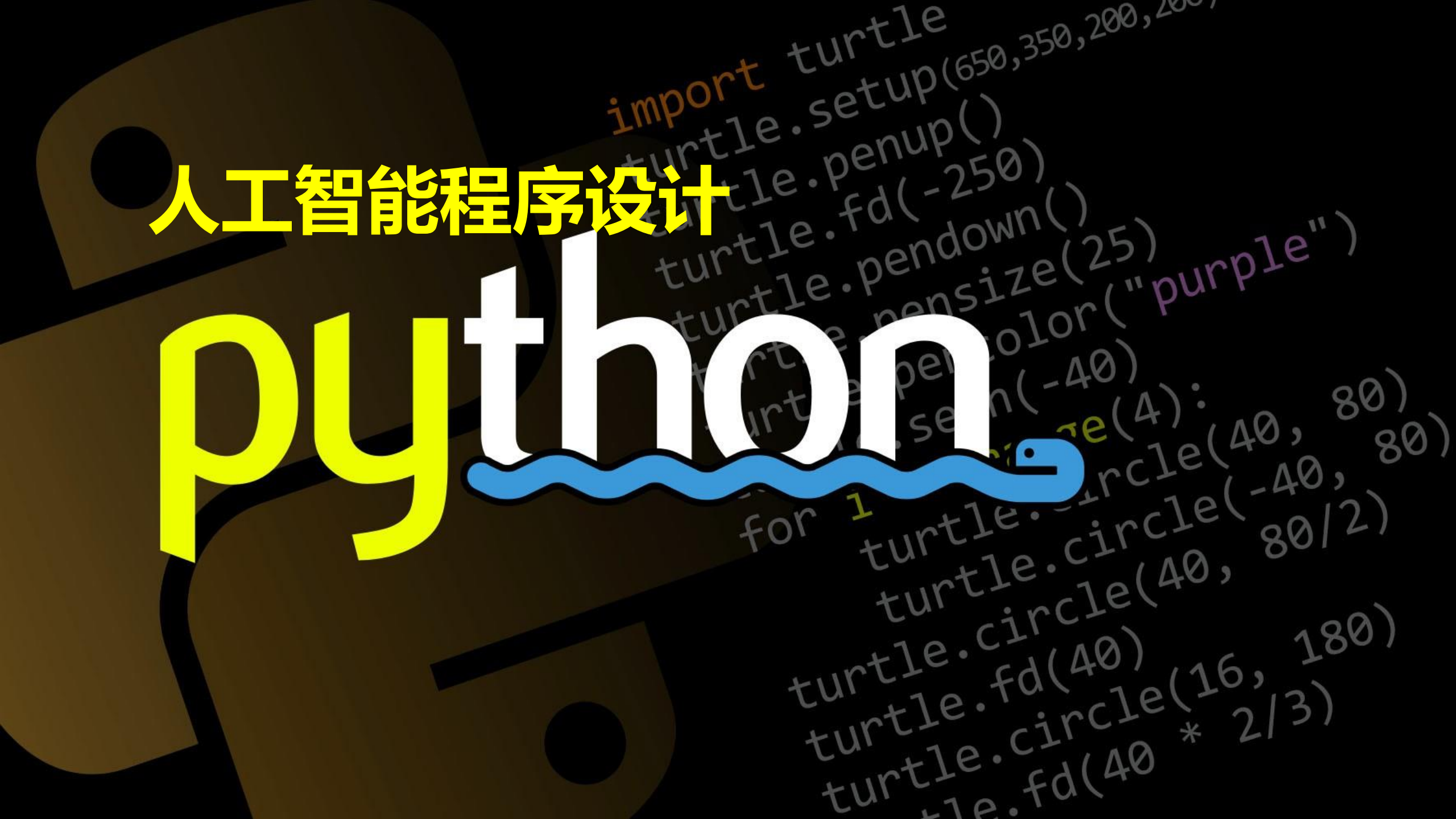
人工智能程序设计


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

17.2 单智能体开发：LANGGRAPH工作流

北京石油化工学院 人工智能研究院

刘 强

学习内容

- LangGraph基础与状态图设计
- 工具调用与任务编排
- 记忆管理与状态持久化

17.2.1 LangGraph基础与状态图设计

学习内容：

- 状态图核心概念
- 条件分支设计
- 工作流编排

LangGraph简介

LangGraph是用于构建状态化智能体应用的框架。

核心特点：

- 通过有向图表示工作流程
- 将复杂推理分解为多个节点
- 通过状态管理实现持续运行

示例 17.2.1：智能任务处理工作流

状态图由节点、边和状态组成，定义了智能体的工作流程和状态转换逻辑。

```
from langgraph.graph import StateGraph
from typing import TypedDict

class AgentState(TypedDict):
    messages: list
    current_step: str
    result: str

def analyze_task(state):
    return {"current_step": "analyzing", "result": "任务分析完成"}

def execute_task(state):
    return {"current_step": "executing", "result": "任务执行完成"}
```

创建工作流

将节点和边组合成完整的工作流：

StateGraph定义了智能体的状态结构和工作流程。

```
# 创建工作流
workflow = StateGraph(AgentState)
workflow.add_node("analyze", analyze_task)
workflow.add_node("execute", execute_task)
workflow.add_edge("analyze", "execute")
workflow.set_entry_point("analyze")

app = workflow.compile()
```

条件分支设计

智能体需要根据不同情况选择不同的执行路径。

```
def should_continue(state):
    if state.get("result") == "成功":
        return "finish"
    else:
        return "retry"

def retry_task(state):
    return {"current_step": "retrying", "result": "重试完成"}

def finish_task(state):
    return {"current_step": "finished", "result": "任务完成"}
```

添加条件边

条件分支允许智能体根据执行结果动态选择下一步行动。

```
workflow.add_node("retry", retry_task)
workflow.add_node("finish", finish_task)
workflow.add_conditional_edges("execute", should_continue, {
    "retry": "retry",
    "finish": "finish"
})
```

17.2.2 工具调用与任务编排

学习内容：

- 工具定义与集成
- 任务编排策略
- 工具协调管理

工具定义与集成

工具是智能体与外部系统交互的接口。

```python
def search_tool(query: str) -> str:
    # 模拟搜索功能
    return "搜索结果: {}相关信息".format(query)

def calculator_tool(expression: str) -> str:
    # 模拟计算功能
    try:
        result = eval(expression)
        return "计算结果: {}".format(result)
    except:
        return "计算错误"
```

工具选择逻辑

根据任务需求选择合适的工具：

```
def use_tools(state):
    query = state.get("query", "")
    if "搜索" in query:
        result = search_tool(query)
    elif "计算" in query:
        result = calculator_tool(query)
    else:
        result = "无法处理该请求"

    return {"result": result, "current_step": "tool_used"}
```

任务编排策略

复杂任务需要多个工具的协同工作。

```
def orchestrate_tools(state):
    task = state.get("task", "")
    steps = []

    if "研究" in task:
        steps = ["search", "analyze", "summarize"]
    elif "计算" in task:
        steps = ["parse", "calculate", "format"]

    results = []
    for step in steps:
        result = execute_step(step, state)
        results.append(result)
        state["intermediate_results"] = results

    return {"results": results, "current_step": "orchestrated"}
```

17.2.3 记忆管理与状态持久化

学习内容：

- 短期记忆管理
- 长期记忆存储
- 状态持久化机制

短期记忆管理

短期记忆存储当前对话或任务执行过程中的临时信息。

```python
class ShortTermMemory:
    def __init__(self, max_size=10):
        self.memory = []
        self.max_size = max_size

    def add(self, item):
        self.memory.append(item)
        if len(self.memory) > self.max_size:
            self.memory.pop(0)

    def get_recent(self, count=5):
        return self.memory[-count:]
```

记忆管理节点

确保智能体能够记住最近的交互历史。

```
def manage_memory(state):
    memory = state.get("memory", ShortTermMemory())
    current_interaction = {
        "timestamp": time.time(),
        "input": state.get("input", ""),
        "output": state.get("result", "")
    }
    memory.add(current_interaction)

    return {"memory": memory, "context": memory.get_recent()}
```

状态持久化机制

对于长期运行的智能体，状态持久化确保重启后能够恢复工作状态。

```
import json
import os

def save_state(state, filename="agent_state.json"):
    serializable_state = {
        "messages": [str(msg) for msg in state.get("messages", [])],
        "current_step": state.get("current_step", ""),
        "memory": state.get("memory", []),
        "timestamp": time.time()
    }

    with open(filename, 'w', encoding='utf-8') as f:
        json.dump(serializable_state, f, ensure_ascii=False, indent=2)
```

状态加载

支持智能体的长期运行和状态恢复。

```
def load_state(filename="agent_state.json"):
    if os.path.exists(filename):
        with open(filename, 'r', encoding='utf-8') as f:
            return json.load(f)
    return {}
```

17.2.4 Ask AI：LangGraph进阶应用

想要学习更多LangGraph的高级功能，可以向AI助手询问：

- "如何设计复杂的多分支工作流？"
- "LangGraph中如何实现循环和递归逻辑？"

实践练习

练习 17.2.1：工作流设计

使用LangGraph设计一个包含分析、规划、执行三个阶段的智能体工作流，实现任务的自动化处理。

实践练习

练习 17.2.2：工具集成

集成搜索、计算、文件操作等多种工具，构建一个能够处理不同类型任务的智能体系统。

实践练习

练习 17.2.3：记忆系统

实现智能体的短期和长期记忆管理，使智能体能够在多次交互中保持上下文连续性。