Python程序设计实践

主讲：张琦

- 2.1 函数的定义与调用
- 2.2 模块与包
- 2.3 文件操作
- 2.4 异常处理

2.1 函数定义与调用

- 函数的定义
- 函数的调用
- 参数传递
- 返回值
- 实践指导

函数的定义、调用

- **函数的定义**
  - def关键字
  - 函数名
  - 函数体
  - 返回值
- **函数的调用**

```
# 定义函数
def function_name(parameter1, parameter2):
    # 函数体
    result = parameter1 + parameter2
    return result

# 调用函数
result = function_name(5, 3)
print(result)
```

参数传递

- **位置参数**
  - 按照参数在函数定义中声明的顺序传递值。

```
def add(a, b):
    return a + b

result = add(2, 3)  # a=2, b=3
```

参数传递

- **关键字参数**
  - 使用参数的名称直接对它们进行赋值，这样参数的顺序就不重要了。

```
def add(a, b):
    return a + b

result = add(b=3, a=2)  # a=2, b=3
```

参数传递

- **默认参数**
  - 在函数定义时为参数提供默认值。调用函数时，如果没有传递该参数，则使用默认值。

```
def add(a, b=2):
    return a + b

result = add(3)  # a=3, b=2（默认值）
```

参数传递

- **可变参数**
  - 使用*args接收任意数量的位置参数，使用**kwargs接收任意数量的关键字参数。

```
def func(*args, **kwargs):
    print("位置参数:", args)
    print("关键字参数:", kwargs)

func(1, 2, 3, a=4, b=5)
```

参数传递

- **强制关键字参数**
  - 如果在可变位置参数（*args）之后定义参数，那么调用函数时，这些参数只能作为关键字参数传递。

```
def func(*args, a, b):
    print("位置参数:", args)
    print("关键字参数 a:", a, "b:", b)

func(1, 2, a=3, b=4)
```

实践指导

- **编写函数，练习参数传递和返回值**
  - 示例1: 计算阶乘

```
def factorial(n):
    # 如果输入为0或1，阶乘结果为1
    if n == 0 or n == 1:
        return 1
    else:
        # 递归调用计算n的阶乘
        return n * factorial(n-1)
```

实践指导

- **编写函数，练习参数传递和返回值**
  - 示例2: 判断素数

```
def is_prime(n):
    # 小于2的数不是素数
    if n < 2:
        return False
    # 从2到sqrt(n)检查是否有因子
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

2.2 模块与包

- 模块的导入与使用
- 创建和使用自定义模块
- 包的创建与使用
- 实践指导

模块的导入与使用

- **导入整个模块**
  - 使用import语句导入整个模块，通过模块名访问其内容。

```
import math
result = math.sqrt(16)  # 使用math模块的sqrt函数
```

- **导入特定的项**
  - 使用from ... import …语句从模块中导入特定的函数、类等。

```
from math import sqrt
result = sqrt(16)  # 直接使用sqrt函数，无需通过模块名
```

模块的导入与使用

- **导入模块并为其设置别名**
  - 使用`import ... as …`语句导入模块，并为其设置别名，以简化代码。

```
import math as m
result = m.sqrt(16)  # 使用别名m代替模块名math
```

- **导入模块中的所有项**
  - 使用`from ... import *`语句导入模块中的所有公开项，不推荐此法。

```
from math import sqrt
result = sqrt(16)  # 直接使用sqrt函数，无需通过模块名
```

模块的导入与使用

- **模块的使用**

  - 导入模块后，就可以使用模块中定义的函数、类和变量等。模块可以帮助你组织代码，提高代码的可重用性。Python自身带有许多标准模块，如`math`、`datetime`等，你也可以创建自己的模块，或者安装和使用第三方模块。

  - 记得在使用模块之前，确保模块已经安装在你的环境中（对于第三方模块）。标准库中的模块无需安装即可直接使用。

创建和使用自定义模块

- **示例：计算器模块**
  - 创建一个简单的计算器模块，我们将其命名为 `calculator.py`。这个模块将包含一些基本的数学运算函数：加法、减法、乘法和除法。

```python
# calculator.py

def add(x, y):
    """加法"""
    return x + y

def subtract(x, y):
    """减法"""
    return x - y

def multiply(x, y):
    """乘法"""
    return x * y

def divide(x, y):
    """除法"""
    if y == 0:
        return "错误：除数不能为0"
    return x / y
```

创建和使用自定义模块

- **示例：计算器模块**
  - 在同一目录下创建另一个Python文件，比如`use_calculator.py`，然后导入并使用`calculator`模块。
  - 运行`use_calculator.py`

```python
# use_calculator.py
import calculator

# 使用模块中的函数
result1 = calculator.add(10, 5)
print(f"10 + 5 = {result1}")

result2 = calculator.subtract(10, 5)
print(f"10 - 5 = {result2}")

result3 = calculator.multiply(10, 5)
print(f"10 * 5 = {result3}")

result4 = calculator.divide(10, 5)
print(f"10 / 5 = {result4}")

result5 = calculator.divide(10, 0)
print(f"10 / 0 = {result5}")
```

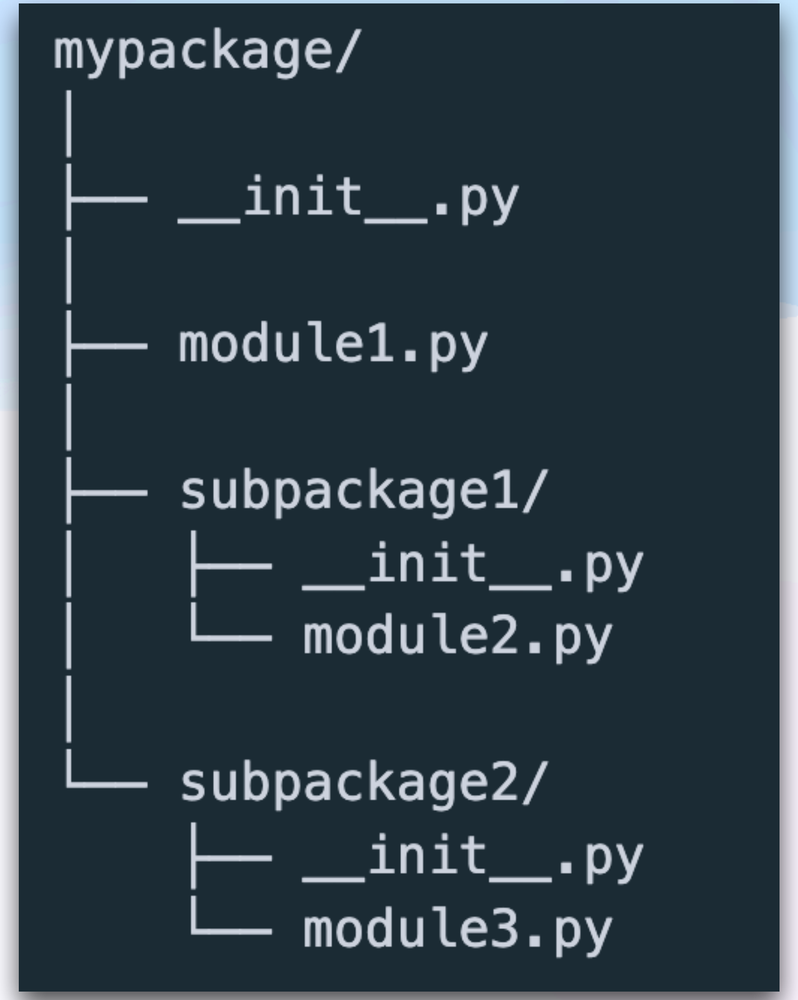
包的创建与使用

创建和使用Python包涉及到组织模块（Python文件）到目录结构中，并通过`__init__.py`件将这些目录标记为Python的包。这样可以帮助你组织大型项目。以下是创建和使用包的步骤：

- **创建包**

1. **创建一个目录结构**：假设你想创建一个名为`mypackage`的包，它包含两个子包`subpackage1`和`subpackage2`，每个子包中都有模块。

- 每个目录下的`__init__.py`文件都是必需的，它可以为空，但它告诉Python这个目录应该被视为一个Python包。



包的创建与使用

2. **添加代码到模块**：在module1.py、module2.py和module3.py中添加一些函数或类。

```
# module1.py
def func1():
    print("Function 1 from module 1")

# subpackage1/module2.py
def func2():
    print("Function 2 from module 2")

# subpackage2/module3.py
def func3():
    print("Function 3 from module 3")
```

包的创建与使用

- **使用包**

在同一项目中或者在mypackage所在目录的外部，你可以导入并使用这个包中的模块和函数。

1. **导入整个模块：**

```
import mypackage.module1
mypackage.module1.func1()
```

2. **从模块中导入特定函数：**

```
import mypackage.subpackage2.module3
mypackage.subpackage2.module3.func3()
```

包的创建与使用

- **使用包**

  3. **导入子包中的模块：**

```
import mypackage.subpackage2.module3
mypackage.subpackage2.module3.func3()
```

- **注意事项**

  - 确保你的Python环境已经设置了正确的路径，以便它可以找到你的包。
  - 如果你想要将你的包分发给其他人使用，你可能需要创建一个`setup.py`文件，并使用`setuptools`来打包你的项目。

通过这种方式，你可以创建复杂的包结构，以逻辑和有组织的方式管理你的Python代码。

实践指导

- **创建和使用自定义模块与包**
  - 示例: 字符串处理模块

2.3 文件操作

- 文件的打开与关闭
- 文件的读写操作
- 实践指导

文件的打开与关闭

- **打开文件**

使用`open()`函数打开文件。这个函数需要至少一个参数：文件的路径和名称。它还可以接受一个模式参数，如`'r'`（读取，默认）、`'w'`（写入，存在则覆盖）、`'a'`（追加）等。

```
# 打开文件进行读取
file = open('example.txt', 'r')
```

- **读取文件内容**

打开文件后，可以使用不同的方法读取内容，例如`read()`, `readline()`, 或`readlines()`。

```
content = file.read()  # 读取整个文件内容
```

文件的打开与关闭

- **关闭文件**

操作完成后，使用`close()`方法关闭文件是一个好习惯。

```
file.close()
```

- **使用`with`语句**

为了更好地管理文件的打开和关闭（确保即使发生错误也能正确关闭文件），推荐使用`with`语句。这种方式会在代码块执行完毕时自动关闭文件。

```
with open('example.txt', 'r') as file:
    content = file.read()
    # 文件在这个代码块结束时自动关闭
```

文件的读写操作

- **写入文件**

覆盖写入：如果文件已存在，此操作会覆盖原有内容。

```
with open('example.txt', 'w') as file:
    file.write("Hello, world!\n")
    file.write("This is another line.")
```

追加内容：如果想在文件末尾添加内容，而不是覆盖原有内容。

```
with open('example.txt', 'a') as file:
    file.write("\nAppending a new line.")
```

文件的读写操作

- **读取文件**

读取全部内容：使用read方法读取整个文件内容。

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

逐行读取：使用readline()或readlines()方法。

```
# 使用readline()
with open('example.txt', 'r') as file:
    line = file.readline()
    while line:
        print(line, end='')  # 使用end=''避免打印额外的换行符
        line = file.readline()

# 使用readlines()
with open('example.txt', 'r') as file:
    lines = file.readlines()
    for line in lines:
        print(line, end='')
```

文件的读写操作

- **读写模式**

'r'：只读模式。文件必须存在，否则会抛出异常。

'w'：写入模式。如果文件存在，会被覆盖。如果文件不存在，会被创建。

'a'：追加模式。如果文件存在，写入的数据会被追加到文件末尾。如果文件不存在，会被创建。

'r+'：读写模式。文件必须存在，允许读取和写入。

'w+'：读写模式。如果文件存在，会被覆盖。如果文件不存在，会被创建。允许读取和写入。

'a+'：读写模式。如果文件存在，写入的数据会被追加到文件末尾。如果文件不存在，会被创建。允许读取和写入。

- 选择合适的模式对于完成你的文件操作任务至关重要

实践指导

- **完成文件操作的练习**
  - 示例: 读取配置文件、写入日志文件

2.4 异常处理

- 异常的捕获与处理
- 自定义异常
- 实践指导

异常的捕获与处理

- **异常处理的基本结构**

在Python中，异常的捕获和处理使用try、except、else和finally语句块。

```
#include<iostream>
using namespace std;

// 递归函数来计算阶乘
int factorial(int n) {
    // 基本情况
    if(n == 0) {
        return 1;
    }
    // 递归步骤
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number;
    cout << "Enter a positive integer: ";
    cin >> number;
    cout << "Factorial of " << number << " = " << factorial(number);
    return 0;
}
```

异常的捕获与处理

- **多个异常的捕获**

你可以通过指定多个except语句来捕获不同类型的异常。

```
try:
    # 可能抛出多种异常的代码
    file = open('不存在的文件.txt', 'r')
except FileNotFoundError:
    print("文件未找到! ")
except Exception as e:
    # 捕获其他所有异常
    print(f"发生了一个错误: {e}")
```

异常的捕获与处理

- **使用异常的信息**

在except块中，你可以通过异常对象获取异常的详细信息。

```
try:
    # 尝试执行的代码
    result = 10 / 0
except ZeroDivisionError as e:
    # 使用异常对象e
    print(f"发生错误: {e}")
```

- **异常的传递**

如果在当前作用域内没有捕获到异常，它会被传递到上一级的作用域中，直到被捕获或导致程序终止。

自定义异常

- **自定义异常**

你还可以定义自己的异常类型，以满足特定的业务需求。

```
class MyError(Exception):
    """自定义异常类"""
    def __init__(self, message):
        self.message = message

try:
    # 触发自定义异常
    raise MyError("出错了！")
except MyError as e:
    print(e.message)
```

通过这种方式，你可以更精细地控制异常处理的逻辑，使代码更加清晰和健壮。

实践指导

- **完成异常处理的练习**
  - 示例: 处理文件不存在、输入数据格式错误

课程总结

- 掌握Python编程的基本语法和常用数据结构
- 能够编写简单的Python程序并进行基本的数据处理和文件操作
- 理论与实践相结合，提升编程能力和解决问题的能力