Chapter 5: Advanced Python Features

Exception handling

Exception handling is a mechanism that allows you to handle errors and exceptions that may occur in your code. It allows you to write code that can gracefully handle unexpected input, errors, and exceptions, and it can help you to write more robust and reliable programs.

Here is an example of how to use the **try** and **except** statements to handle an exception:

```
try:
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))
    result = num1 / num2
    print(result)
except ZeroDivisionError:
    print('Cannot divide by zero')
```

In this example, we have used the **try** statement to enclose a block of code that may raise an exception. If an exception is raised, the code in the **except** block will be executed. In this case, we are handling the **ZeroDivisionError** exception, which is raised when trying to divide by zero.

You can also use the **finally** statement to execute a block of code *regardless* of whether an exception is raised or not. For example:

```
try:
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))
    result = num1 / num2
    print(result)
except ZeroDivisionError:
    print('Cannot divide by zero')
finally:
    print('Exiting the program')
```

In this example, the code in the **finally** block will be executed regardless of whether an exception is raised or not.

Generators and iterators

A generator is a function that generates a sequence of values on-demand, rather than returning a whole sequence at once. Generators are implemented using the **yield** keyword, which allows the function to return a value and then resume execution at the same point the next time it is called.

Here is an example of a generator function that generates a sequence of numbers:

```
def my_range(n):
    i = 0
    while i < n:
        yield i
        i += 1

# Use the generator
for i in my_range(5):
    print(i)
# Output: 0 1 2 3 4
```

In this example, the **`my_range`** function is a generator that generates a sequence of numbers from 0 to **`n-1`**. We can use a **`for`** loop to iterate over the sequence generated by the generator.

An iterator is an object that allows you to iterate over a sequence of values. All objects that support iteration are also iterators.

Here is an example of how to use the **`iter()`** function to create an iterator from a list:

```
# Create a list
my_list = [1, 2, 3, 4, 5]

# Create an iterator from the list
it = iter(my_list)

# Iterate over the iterator
print(next(it)) # Output: 1
print(next(it)) # Output: 2
print(next(it)) # Output: 3
print(next(it)) # Output: 4
print(next(it)) # Output: 5

# The iterator is exhausted
print(next(it)) # Raises StopIteration
```

In this example, we have used the **`iter()`** function to create an iterator from the **`my_list`** object. We can then use the **`next()`** function to retrieve the next value from the iterator. When the iterator is exhausted, the **`next()`** function raises a **`StopIteration`** exception.

Decorators

A decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying its code. Decorators are a powerful and convenient way to modify or enhance the functionality of a function, and they are often used to add additional behavior to functions that are called before or after the original function is executed.

Here is an example of how to use a decorator to add additional behavior to a function:

```
def my_decorator(func):
    def wrapper():
        print('Before calling the function')
        func()
        print('After calling the function')
    return wrapper

@my_decorator
def my_function():
    print('Inside the function')

# Call the decorated function
my_function()
# Output: Before calling the function
#         Inside the function
#         After calling the function
```

In this example, we have defined a decorator function called **`my_decorator`** that takes a function and returns a wrapper function. The wrapper function adds additional behavior before and after calling the original function.

We have then used the **`@`** symbol to decorate the **`my_function`** function with the **`my_decorator`** decorator. When we call the decorated **`my_function`**, the additional behavior added by the decorator is executed before and after the function is called.

Decorators are often used to add logging, authentication, or other types of behavior to functions. They are a powerful and convenient way to modify the behavior of a function without modifying its code.

Working with modules and packages

Modules

Modules are files that contain a collection of functions, variables, and other code that can be used in other Python programs. Modules are a way to organize and reuse code, and they help make Python programs more modular and maintainable.

To use a module in a Python program, you can use the **`import`** statement. For example, to import the **`math`** module, which contains a collection of mathematical functions, you can use the following code:

```python
import math

result = math.sqrt(16)
print(result)
# Output: 4.0
```

You can also import specific functions or variables from a module using the **`from`** keyword. For example:

```python
from math import sqrt

result = sqrt(16)
print(result)
# Output: 4.0
```

You can also use the **`as`** keyword to give a function or variable a different name when importing it. For example:

```python
from math import sqrt as square_root

result = square_root(16)
print(result)
# Output: 4.0
```

Packages

Packages are collections of modules that are organized into a directory structure. Packages are a way to organize larger Python programs, and they allow you to divide your code into smaller, more reusable pieces.

To use a package in a Python program, you can use the **`import`** statement with the name of the package and the name of the module you want to use, separated by a dot. For example:

```python
import mypackage.mymodule

result = mypackage.mymodule.some_function()
print(result)
```

You can also import specific functions or variables from a package using the **`from`** keyword. For example:

```python
from mypackage import mymodule

result = mymodule.some_function()
print(result)
```

You can also use the **`as`** keyword to give a function or variable a different name when importing it. For example:

```python
from mypackage import mymodule as mm

result = mm.some_function()
print(result)
```

In addition to using the **import** statement, you can also use the **__init__.py** file to define what gets imported when you use the **import** statement with the name of a package. For example, you can use the **__init__.py** file to import specific modules or functions from the package and make them available when the package is imported.

Working with dates and times

You can use the **datetime** module to work with dates and times. The **datetime** module provides classes for representing dates, times, and timestamps, and it also provides functions for working with these objects.

Here is an example of how to use the **datetime** module to get the current date and time:

```python
from datetime import datetime

# Get the current date and time
now = datetime.now()
print(now)
# Output: 2022-12-26 14:47:59.438123
```

In this example, we have imported the **datetime** module and used the **datetime.now()** function to get the current date and time. The **now** variable is set to a **datetime** object that represents the current date and time.

You can also use the **datetime** module to create **datetime** objects for specific dates and times, and to perform operations on these objects, such as formatting, arithmetic, and comparison.

Here is an example of how to create a **datetime** object for a specific date and time, and how to format it:

```python
from datetime import datetime

# Create a date and time
dt = datetime(2022, 12, 26, 14, 50, 0)

# Format the date and time
formatted = dt.strftime('%B %d, %Y %I:%M %p')
print(formatted)
# Output: December 26, 2022 02:50 PM
```

In this example, we have used the **datetime** constructor to create a **datetime** object for the date and time December 26, 2022, 2:50 PM. We have then used the **strftime()** method to format the date and time as a string.

Working with databases

You can use various libraries and frameworks to interact with databases. Some popular options include:

- **sqlite3**: A built-in Python library for working with SQLite databases
- **psycopg2**: A library for working with PostgreSQL databases
- **MySQLdb**: A library for working with MySQL databases
- **SQLAlchemy**: A powerful and flexible library for working with a variety of databases, including MySQL, PostgreSQL, and SQLite

Here is an example of how to use the **sqlite3** library to create a database and table, insert data into the table, and query the data:

```python
import sqlite3

# Connect to the database
conn = sqlite3.connect('mydatabase.db')

# Create a cursor
cursor = conn.cursor()

# Create a table
cursor.execute('''CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, email TEXT)''')

# Insert data into the table
cursor.execute('''INSERT INTO users (name, email) VALUES (?, ?)''', ('John', 'john@example.c
cursor.execute('''INSERT INTO users (name, email) VALUES (?, ?)''', ('Jane', 'jane@example.c

# Commit the changes
conn.commit()

# Query the data
cursor.execute('''SELECT * FROM users''')

# Fetch the results
results = cursor.fetchall()

for result in results:
    print(result)

# Close the connection
conn.close()
```

In this example, we have imported the **sqlite3** library and used it to connect to a database, create a table, insert data into the table, and query the data. We have also used a cursor to execute SQL statements and fetch the results.

Working with regular expressions

Regular expressions are a powerful tool for matching and manipulating strings. They are used in many programming languages, including Python, to search for patterns in strings, extract information from strings, and replace or manipulate parts of strings.

You can use the **re** module to work with regular expressions. Here is an example of how to use the **re** module to search for a pattern in a string:

```
import re

string = 'The quick brown fox jumps over the lazy dog.'
pattern = r'quick'

match = re.search(pattern, string)
if match:
    print('Match found:', match.group())
else:
    print('Match not found')
# Output: Match found: quick
```

In this example, we have imported the **re** module and defined a string and a pattern to search for. We have then used the **re.search()** function to search for the pattern in the string. If a match is found, the **match** variable is set to a **Match** object that contains information about the match, and we can use the **group()** method to get the matched string.

Here is an example of how to use the **re** module to extract information from a string:

```
import re

string = 'The quick brown fox jumps over the lazy dog.'
pattern = r'(quick) (brown) (fox)'

match = re.search(pattern, string)
if match:
    print('Match found:', match.groups())
else:
    print('Match not found')
# Output: Match found: ('quick', 'brown', 'fox')
```

In this example, we have defined a pattern that contains three groups, which are enclosed in parentheses. When we search for the pattern in the string and find a match, the **groups()** method returns a tuple of the matched strings for each group.