Exceptions and Decorators

Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. An exception is an object that wraps an error event that occurred within a method and contains:

- Information about the error including its type
- The state of the program when the error occurred
- Optionally, other custom information about the error

Types of Exceptions

In Python, the exceptions are categorized into two types:

1. `Built-in Exceptions` - These exceptions are raised when Python interpreter

encounters an error. Examples: `ZeroDivisionError`, `TypeError`, `NameError`, etc.
2. `User-defined Exceptions` - Developers can create their own custom exceptions to handle specific error conditions.

Exception Hierarchy

All built-in exceptions are derived from the base class BaseException, which itself is derived from object. The BaseException class has three subclasses that correspond to the exception classes that are raised when Python interpreter encounters an error:

1. `SystemExit` - Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, it causes the interpreter to exit.
2. `KeyboardInterrupt` - Raised when the user hits the interrupt key (Ctrl+C or Delete). If not handled in the code, it causes the interpreter to exit.
3. `Exception` - This is the base class for all built-in exceptions. It is derived from BaseException. All other built-in exceptions are derived from this class.

Creating Custom Exceptions

In Python, users can define custom exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from the built-in Exception class. Most of the built-in exceptions are also derived from this class. The following example shows how to create a custom exception class:

```
class CustomException(Exception):
    pass
```

Exception Handling

The raise keyword

Developers can raise exceptions explicitly using the raise statement. This can be useful when a certain condition is met and the program needs to indicate an exceptional situation.

```python
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(f"Error: {e}")
```

In the above example, if the denominator b is zero, a ValueError is raised explicitly.

You can also throw custom exceptions using the raise statement. For example:

```python
class CustomException(Exception):
    pass

def divide(a, b):
    if b == 0:
        raise CustomException("Cannot divide by zero")
    return a / b
```

The try and except keywords

The try statement allows you to define a block of code to be tested for errors while it is being executed. The except statement allows you to define a block of code to be executed, if an error occurs in the try block. The try and except keywords come in pairs.

```python
try:
    # do something
except:
    # handle exception
```

If an exception occurs in the try block, the program execution will jump immediately to the except block. If no exception occurs, the except block will be skipped.

You can also specify the type of exception to be handled in the except block. For example:

```
try:
    # do something
except ValueError:
    # handle ValueError
```

In the above example, the except block will only handle ValueError exceptions. If any other exception occurs, it will not be handled by the except block. You can also specify multiple exception types to be handled in the except block. For example:

```
try:
    # do something
except (ValueError, TypeError):
    # handle ValueError and TypeError
```

If the handling logic for different exceptions is different, you can specify multiple except blocks to handle each exception type separately. For example:

```
try:
    # do something
except ValueError:
    # handle ValueError
except TypeError:
    # handle TypeError
```

The finally keyword

The finally block is used to define a block of code that will be executed, no matter if there is an exception or not. For example:

```
try:
    # do something
except ValueError:
    # handle ValueError
finally:
    # do something
```

The finally block will be executed even if there is no exception in the try block. The finally block is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

The golden Rules of Exception Handling

1. `Never swallow an exception` - Swallowing an exception means that you catch it and do nothing with it. This is a bad practice because it means that you are ignoring the fact that an error occurred, which could cause your program to behave in unexpected ways and hide bugs.

```
try:
    result = divide(10, 0)
except ValueError:
    pass
```

2. `Never catch a generic exception` - Catching a generic exception means that you are catching all exceptions. This is a bad practice because it means that you are not handling exceptions in a meaningful way. You should always catch specific exceptions and handle them appropriately.

```
try:
    result = divide(10, 0)
except Exception:
    # handle exception
```

3. `Never throw a generic exception` - Raise specific exceptions instead of generic ones. This is a good practice because it allows you to handle different exceptions in different ways.

```
if b == 0:
    raise Exception("Cannot divide by zero")
```

4. `Use context managers for resources that need to be released` - Context managers are a good way to ensure that resources are released when they are no longer needed. For example, when you open a file, you should use a context manager to ensure that the file is closed when you are done with it.

```python
try:
    with open("example.txt", "r") as file:
        # Code to read from the file
except FileNotFoundError:
    print("File not found.")
```

Decorators

Decorators are a powerful and flexible feature in Python that allows the modification of functions or methods at the time of their definition. They provide a way to wrap or modify the behavior of functions without changing their source code. Decorators are widely used for various purposes, such as logging, access control, memoization, and more.

Why Use Decorators?

1. **Code Reusability:** Decorators allow you to encapsulate reusable functionality and apply it to multiple functions or methods.
2. **Separation of Concerns:** Decorators help separate the core logic of a function from additional concerns or behaviors. This promotes a cleaner and more maintainable codebase.
3. **Readability:** Decorators can enhance the readability of code by keeping the main logic of a function uncluttered with additional functionalities.
4. **Code Organization:** Decorators enable the organization of cross-cutting concerns in a modular way, making it easier to manage and maintain the code.
5. **Meta-Programming:** Decorators enable meta-programming by modifying the behavior of functions dynamically.

Popular Decorators in Python

1. `@staticmethod` and `@classmethod`

These decorators are built-in and are used for defining static and class methods in a class.

```python
class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method.")

    @classmethod
    def class_method(cls):
        print(f"This is a class method of {cls}.")
```

2. `@property`

The `@property` decorator allows you to define a method that can be accessed like an attribute.

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @property
    def area(self):
        return 3.14 * self._radius ** 2
```

3. `@classmethod` and `@staticmethod`

These decorators are used to define class methods and static methods in a class, respectively.

```python
class MathOperations:
    @classmethod
    def add(cls, x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y
```

Creating Custom Decorators

Creating a decorator involves defining a function that takes another function as its argument, performs some actions before or after the original function is called, and returns a new function.

Here is an example of a simple decorator:

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the function is called.")
        return result
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

In this example, `my_decorator` is a decorator that prints messages before and after the `say_hello` function is called. The `@my_decorator` syntax is a shorthand for `say_hello = my_decorator(say_hello)`.