# 19. Try-Except Blocks

Try-except blocks, also known as exception handling, allow you to handle errors and exceptions gracefully in Python code. They provide a way to anticipate and manage potential errors that may occur during program execution. Here's how try-except blocks work and best practices for using them:

**Basic Syntax:**

The try block contains the code that may raise an exception.

The except block catches and handles exceptions raised in the try block.

Optionally, you can include an else block that executes if no exceptions are raised, and a finally block that executes regardless of whether an exception occurs.

**Example:**

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Handle the exception
else:
    # Execute if no exceptions are raised
finally:
    # Execute regardless of exceptions
```

**Handling Specific Exceptions:**

You can specify the type of exception to catch in the except block to handle specific types of errors.

Handling specific exceptions allows you to provide tailored error messages or take appropriate actions based on the type of error encountered.

**Example:**

```
try:
    # Code that may raise an exception
except FileNotFoundError:
    # Handle file not found error
except ValueError:
    # Handle value error
```

**Handling Multiple Exceptions:**

You can handle multiple exceptions in a single except block by specifying multiple exception types separated by commas.

This approach reduces code duplication and improves readability.

**Example:**

```
try:
    # Code that may raise an exception
except (ValueError, TypeError):
    # Handle value error or type error
```

**Generic Exception Handling:**

It's generally recommended to catch specific exceptions whenever possible to handle errors more precisely.

However, you can also catch a generic Exception to handle any type of exception.

**Example:**

```
try:
    # Code that may raise an exception
except Exception as e:
    # Handle any exception
    print(f"An error occurred: {e}")
```

**Raising Exceptions:**

Inside the except block, you can raise another exception to propagate the error or raise a custom exception to provide additional context.

**Example:**

```
try:
    # Code that may raise an exception
except ValueError:
    # Handle value error
    raise RuntimeError("Encountered a runtime error")
```

**Finally Block:**

The finally block is used to execute cleanup code that should always run, regardless of whether an exception occurs or not.

Common use cases include closing files, releasing resources, or finalizing operations.

**Example:**

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Handle the exception
finally:
    # Cleanup code (e.g., closing files)
```

By using try-except blocks effectively, you can write more robust and fault-tolerant Python code, ensuring that your programs handle errors gracefully and continue to function correctly even in the presence of unexpected conditions.

# 20. Handling Multiple Exceptions

Handling multiple exceptions in Python allows you to catch and handle different types of errors that may occur within the same block of code. This approach improves code readability and reduces redundancy.

Here's how to handle multiple exceptions using try-except blocks:

```
try:
    # Code that may raise exceptions
except ExceptionType1:
    # Handle ExceptionType1
except ExceptionType2:
    # Handle ExceptionType2
except (ExceptionType3, ExceptionType4):
    # Handle ExceptionType3 or ExceptionType4
except:
    # Handle any other exception
```

**In the above code:**

The try block contains the code that may raise exceptions.

Each except block catches and handles a specific type of exception.

You can specify multiple exceptions in a single except block by enclosing them in parentheses and separating them with commas.

A generic except block without specifying any exception type can be used to catch any other exception that is not handled by the preceding except blocks. However, it's generally recommended to catch specific exceptions whenever possible.

**Here's a more concrete example:**

```
 try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result:", result)
except ValueError:
    print("Please enter a valid integer.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
except KeyboardInterrupt:
    print("Operation interrupted.")
except:
    print("An unexpected error occurred.")
```

**In this example:**

If the user enters a non-integer value, a ValueError will be raised and caught by the first except block.

If the user enters zero as the input, a ZeroDivisionError will be raised and caught by the second except block.

If the user interrupts the operation (e.g., by pressing Ctrl+C), a KeyboardInterrupt exception will be raised and caught by the third except block.

Any other unhandled exception will be caught by the last except block, providing a generic error message.

Handling multiple exceptions in this way allows you to tailor error handling for different scenarios and improve the robustness of your code.

# 21. Custom Exceptions

Custom exceptions in Python allow you to define your own exception classes tailored to specific error conditions in your application. This enables you to create more meaningful and descriptive error messages, making it easier to debug and maintain your code.

Here's how to create and use custom exceptions:

**Defining Custom Exception Classes:**

Custom exceptions are created by subclassing the built-in Exception class or one of its subclasses.

You can define additional attributes or methods in your custom exception class as needed.

**Example:**

```
class CustomError(Exception):
    def __init__(self, message="An error occurred"):
        self.message = message
        super().__init__(self.message)
```

**Raising Custom Exceptions:**

To raise a custom exception, simply create an instance of your custom exception class and raise it using the raise statement.

You can optionally pass a custom error message to the exception constructor.

**Example:**

```
try:
    validate_input(-5)
```

```
except CustomError as e:
    print("Custom error:", e.message)
```

**Inheriting from Built-in Exceptions:**

You can subclass built-in exception classes like ValueError, TypeError, or RuntimeError to create more specific custom exceptions.

This allows you to leverage existing exception behavior and error handling mechanisms.

**Example:**

```
class CustomValueError(ValueError):
    def __init__(self, value):
        self.value = value
        self.message = f"Invalid value: {value}"
        super().__init__(self.message)
```

**Using Custom Exceptions in Modules:**

Custom exceptions can be defined in modules and imported into other modules where they are needed.

This promotes code reuse and organization, allowing you to centralize error handling logic.

**Example:**

```
# custom_exceptions.py
class CustomError(Exception):
    pass

# main.py
from custom_exceptions import CustomError
```

```
try:
    raise CustomError("An error occurred")
except CustomError as e:
    print("Custom error:", e)
```

Custom exceptions provide a flexible and powerful mechanism for error handling in Python applications.

# 22. Lambda Functions

Lambda functions, also known as anonymous functions or lambda expressions, are a concise way to create small, one-line functions in Python. They are often used when you need a simple function for a short period of time or when you want to pass a function as an argument to another function.

Here's an overview of lambda functions:

**Basic Syntax:**

Lambda functions are defined using the lambda keyword, followed by a list of parameters, a colon (:), and the expression to evaluate.

The syntax is: lambda parameters: expression

**Example:**

```
add = lambda x, y: x + y
```

**Usage:**

Lambda functions can be assigned to variables and used like regular functions.

They can also be passed as arguments to other functions or used within list comprehensions, map(), filter(), and reduce() functions.

**Example:**

```
# Using a lambda function to define a custom sorting key
points = [(1, 2), (3, 1), (5, 3)]
sorted_points = sorted(points, key=lambda point: point[1])
```

**Single Expression:**

Lambda functions can only contain a single expression.

The expression is evaluated and returned as the result of the function.

**Example:**

```
# Lambda function to check if a number is even
is_even = lambda x: x % 2 == 0
```

**No Statements:**

Lambda functions cannot contain statements like return, pass, assert, or raise.

They are limited to a single expression for evaluation.

**Example (incorrect):**

```
# Incorrect lambda function with a return statement
square = lambda x: return x ** 2  # Raises SyntaxError
```

**Implicit Return:**

Lambda functions automatically return the result of the expression.

You don't need to use the return keyword explicitly.

**Example:**

```
# Lambda function to square a number
square = lambda x: x ** 2
```

Lambda functions are useful for writing quick, throwaway functions where defining a named function would be overkill.

# 23. Map, Filter, and Reduce

map(), filter(), and reduce() are three built-in functions in Python that are commonly used for processing iterables (such as lists, tuples, or sets) in a concise and functional programming style. They allow you to perform operations on elements of an iterable in a more compact and expressive way than using loops.

Here's an overview of each function:

**map() Function:**

The map() function applies a given function to each item of an iterable (e.g., a list) and returns a new iterator containing the results.

**Syntax:** map(function, iterable)

**Example:**

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
# squared is now an iterator containing [1, 4, 9, 16, 25]
```

**filter() Function:**

The filter() function constructs a new iterator from elements of an iterable for which a function returns true (i.e., the function filters the elements).

**Syntax:** filter(function, iterable)

**Example:**

```
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
# even_numbers is now an iterator containing [2, 4]
```

**reduce() Function:**

The reduce() function, which was part of the functools module in Python 3, applies a rolling computation to pairs of elements from an iterable, producing a single result.

**Syntax:** reduce(function, iterable, initializer)

**Example:**

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
# product is now 120 (1 * 2 * 3 * 4 * 5)
```

**Using with Named Functions:**

Instead of using lambda functions, you can use named functions with map() and filter() for better readability and reusability.

**Example:**

```
def square(x):
    return x ** 2

numbers = [1, 2, 3, 4, 5]
squared = map(square, numbers)
```

**List Conversion:**

The result of map() and filter() functions can be converted to lists using the list() function.

**Example:**

```
numbers = [1, 2, 3, 4, 5]
```

```
squared = list(map(lambda x: x**2, numbers))
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
```

These functions are powerful tools for functional programming in Python, allowing you to write more expressive and concise code when dealing with collections of data.

# 24. List Comprehensions and Generator Expressions

List comprehensions and generator expressions are concise and efficient ways to create lists and generators, respectively, in Python. They provide a more compact syntax for generating sequences of values compared to traditional for loops.

Here's an overview of list comprehensions and generator expressions:

**List Comprehensions:**

List comprehensions provide a concise way to create lists based on existing lists or other iterable objects.

**Syntax:** [expression for item in iterable if condition]

**Example:**

```
# Create a list of squares of numbers from 0 to 9
squares = [x**2 for x in range(10)]
```

List comprehensions can also include an optional conditional expression to filter elements.

**Example:**

```
# Create a list of even numbers from 0 to 9
even_numbers = [x for x in range(10) if x % 2 == 0]
```

**Generator Expressions:**

Generator expressions are similar to list comprehensions but return a generator object instead of a list.

They are evaluated lazily, meaning they generate values on-the-fly as they are needed, which can save memory for large sequences.

**Syntax:** (expression for item in iterable if condition)

**Example:**

```
# Create a generator of squares of numbers from 0 to 9
squares_generator = (x**2 for x in range(10))
```

Generator expressions can also include an optional conditional expression to filter elements.

**Example:**

```
# Create a generator of even numbers from 0 to 9
even_numbers_generator = (x for x in range(10) if x % 2 ==
0)
```

**Usage:**

List comprehensions and generator expressions are commonly used for creating lists or generators based on simple transformations or filters of existing data.

They provide a more concise and readable alternative to using traditional for loops.

**Example:**

```
# Traditional approach using a for loop
squares = []
for x in range(10):
    squares.append(x**2)

# Using a list comprehension
squares = [x**2 for x in range(10)]
```

```
# Using a generator expression
squares_generator = (x**2 for x in range(10))
```

**Memory Efficiency:**

Generator expressions are more memory-efficient than list comprehensions because they generate values on-demand rather than storing them all in memory at once.

This makes generator expressions suitable for processing large datasets or infinite sequences.

**Example:**

```
# List comprehension (stores all squares in memory)
squares = [x**2 for x in range(10**6)]

# Generator expression (generates squares on-demand)
squares_generator = (x**2 for x in range(10**6))
```

Both list comprehensions and generator expressions are powerful tools for creating sequences of values in Python.

# 25. Decorators

Decorators are a powerful feature in Python that allow you to modify or extend the behavior of functions or methods without changing their source code. They provide a way to add functionality to existing functions dynamically.

Here's an overview of decorators:

**Basic Syntax:**

Decorators are implemented as functions that take another function as an argument and return a new function.

They are typically used with the @decorator_name syntax, placed above the function definition.

**Example:**

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function
is called.")
        func()
        print("Something is happening after the function is
called.")
    return wrapper

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

say_hello()
```

**Decorator Execution Flow:**

When a decorated function is called, the original function is replaced by the inner wrapper function defined in the decorator.
The wrapper function can perform actions before and after calling the original function, such as logging, timing, or input validation.

**Example:**

```
# Output:
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.
```

**Decorator with Arguments:**

Decorators can accept arguments by defining a decorator factory function that returns a decorator function.

**Example:**

```
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator_repeat

@repeat(num_times=3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

**Common Use Cases:**

**Logging:** Add logging statements before and after function calls.

**Authentication:** Check user authentication before executing a function.

**Caching:** Cache expensive function calls to improve performance.

**Rate Limiting:** Limit the rate at which a function can be called.

**Timing**: Measure the execution time of a function.

**Validation:** Validate input arguments before calling a function.

**Preserving Function Metadata:**

Decorators should use functools.wraps to preserve the metadata of the original function, such as its name, docstring, and annotations.

This ensures that the decorated function retains its identity and remains introspectable.

**Example:**

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Function body
        pass
    return wrapper
```

Decorators are a versatile tool in Python for extending and customizing the behavior of functions.

# 26. Regular Expressions

Regular expressions (regex) in Python provide a powerful and flexible way to search, match, and manipulate text strings based on patterns. They are implemented through the re module in Python.

Here's an overview of regular expressions:

**Basic Syntax:**

Regular expressions are patterns used to match character combinations in strings.

Common regex patterns include literal characters, character classes, quantifiers, anchors, and groups.

**Example:**

```
import re

pattern = r'apple'
text = 'I have an apple'

match = re.search(pattern, text)
if match:
    print('Found')
```

**Pattern Matching Functions:**

The re module provides various functions for pattern matching, including search(), match(), findall(), finditer(), and sub().

- **search():** Searches the string for a match and returns a match object if found.
- **match():** Matches the pattern only at the beginning of the string.

- **findall():** Finds all occurrences of the pattern in the string and returns them as a list.

- **finditer():** Finds all occurrences of the pattern in the string and returns them as an iterator of match objects.

- **sub():** Substitutes occurrences of the pattern in the string with a replacement string.

**Example:**

```
import re

pattern = r'apple'
text = 'I have an apple and another apple'

matches = re.findall(pattern, text)
print(matches)  # Output: ['apple', 'apple']
```

**Metacharacters:**

Regular expressions use metacharacters like . (any character), * (zero or more occurrences), + (one or more occurrences), ? (zero or one occurrence), \d (digit), \w (word character), \s (whitespace), etc.

**Example:**

```
import re

pattern = r'\d+'
text = 'I have 2 apples and 3 oranges'

matches = re.findall(pattern, text)
print(matches)  # Output: ['2', '3']
```

**Grouping and Capturing:**

Parentheses () are used for grouping characters or patterns and capturing matched substrings.

Captured groups can be accessed using the group() method of the match object or through backreferences in replacement strings.

**Example:**

```
import re

pattern = r'(\d{3})-(\d{3})-(\d{4})'
text = 'Phone numbers: 123-456-7890, 987-654-3210'

matches = re.findall(pattern, text)
for match in matches:
    print(match)
```

**Flags:**

Flags modify the behavior of regex patterns. Common flags include re.IGNORECASE, re.MULTILINE, re.DOTALL, etc.

Flags are passed as an argument to regex functions or embedded in the regex pattern using (?i), (?m), (?s), etc.

**Example:**

```
import re

pattern = r'apple'
text = 'I have an APPLE'

match = re.search(pattern, text, re.IGNORECASE)
if match:
```

```
    print('Found')
```

Regular expressions are a powerful tool for text processing and manipulation in Python. They provide a concise and flexible way to define complex patterns for searching and extracting information from strings.

However, regex patterns can be cryptic and challenging to read, so they should be used judiciously and with appropriate documentation.

# 27. Syntax and Patterns

In Python, regular expressions (regex) are strings of characters that define search patterns. These patterns are then used by regex functions from the re module to search for matches within text strings.

Here's an overview of regex syntax and common patterns:

**Literal Characters:**

Literal characters in a regex pattern match themselves in the text string.
Example: The regex pattern apple matches the substring "apple" in a text string.

**Character Classes:**

Character classes match any one of a set of characters enclosed in square brackets [ ].
You can use a hyphen - to specify a range of characters.

**Example:** The regex pattern [abc] matches either "a", "b", or "c".

**Quantifiers:**

Quantifiers specify the number of occurrences of the preceding element in the pattern.

Common quantifiers include * (zero or more occurrences), + (one or more occurrences), ? (zero or one occurrence), {n} (exactly n occurrences), {m,n} (between m and n occurrences).

**Example:** The regex pattern a+ matches one or more occurrences of the character "a".
**Anchors:**

Anchors specify positions in the text string, such as the beginning (^) or end ($) of a line, or word boundaries (\b).

Example: The regex pattern ^start matches "start" only if it occurs at the beginning of a line.

**Alternation:**

Alternation is represented by the pipe symbol | and allows matching of multiple alternatives.

**Example:** The regex pattern cat|dog matches either "cat" or "dog" in the text string.

**Grouping:**

Parentheses ( ) are used for grouping characters or subpatterns together.

Groups can be quantified as a single unit and can be captured for later use.
Example: The regex pattern (ab)+ matches one or more occurrences of the sequence "ab".

**Character Escapes:**

Certain characters have special meanings in regex patterns (metacharacters). To match them literally, you need to escape them with a backslash \.

**Example:** The regex pattern \$ matches the character "$" in the text string.

**Flags:**

Flags modify the behavior of the regex engine. Common flags include re.IGNORECASE, re.MULTILINE, and re.DOTALL.

Flags can be passed as arguments to regex functions or embedded in the pattern itself using (?i), (?m), (?s), etc.

These are just a few examples of regex syntax and patterns. Regular expressions provide a powerful and flexible way to search for and manipulate text data in Python.

# 28. Matching and Searching

The **re** module provides functions for matching and searching text using regular expressions. These functions allow you to find occurrences of patterns within strings and extract or manipulate the matched substrings.

Here's an overview of matching and searching operations using regular expressions:

**re.match() Function:**

The **re.match()** function attempts to match the regex pattern at the beginning of the string.

It returns a match object if the pattern matches at the beginning of the string, or None otherwise.

**Example:**

```
import re

pattern = r'apple'
text = 'apple pie'

match = re.match(pattern, text)
if match:
    print('Match found:', match.group())
else:
    print('No match')
```

**re.search() Function:**

The re.search() function searches for the first occurrence of the regex pattern anywhere in the string.

It returns a match object if the pattern is found, or None otherwise.

**Example:**

```
import re

pattern = r'apple'
text = 'I have an apple and a banana'

match = re.search(pattern, text)
if match:
    print('Match found:', match.group())
else:
    print('No match')
```

**re.findall() Function:**

The re.findall() function finds all occurrences of the regex pattern in the string and returns them as a list of strings.

**Example:**

```
import re

pattern = r'\d+'
text = 'I have 2 apples and 3 oranges'

matches = re.findall(pattern, text)
print('Matches found:', matches)
```

**re.finditer() Function:**

The re.finditer() function finds all occurrences of the regex pattern in the string and returns them as an iterator of match objects.

**Example:**

```
import re
```

```
pattern = r'\d+'
text = 'I have 2 apples and 3 oranges'

matches = re.finditer(pattern, text)
for match in matches:
    print('Match found:', match.group())
```

**Match Object:**

Match objects contain information about the matched substring, such as its start and end positions, the matched text, and any captured groups.

You can access this information using methods like group(), start(), end(), span(), etc.

**Example:**

```
import re

pattern = r'\d+'
text = 'I have 2 apples and 3 oranges'

match = re.search(pattern, text)
if match:
    print('Match found:', match.group())
    print('Start position:', match.start())
    print('End position:', match.end())
    print('Start and end positions:', match.span())
```

Regular expressions provide a powerful way to search for patterns within text strings in Python. By using the re module functions, you can efficiently find and extract relevant information from text data, enabling various text processing and analysis tasks.

# 29. Substitution and Grouping

the re module provides functions for substitution and grouping using regular expressions. These functions allow you to replace matched substrings with other strings and organize complex patterns into groups for more advanced matching and manipulation.

Here's an overview of substitution and grouping operations using regular expressions:

**Substitution with re.sub():**

The re.sub() function replaces occurrences of a regex pattern in a string with a specified replacement string.

**Syntax:** re.sub(pattern, replacement, string, count=0, flags=0)

**Example:**

```
import re

pattern = r'\d+'
text = 'I have 2 apples and 3 oranges'

result = re.sub(pattern, 'X', text)
print('Result:', result)
```

**Substitution with Function:**

Instead of a replacement string, you can specify a function to dynamically generate the replacement based on the match.

The function takes a single argument (the match object) and returns the replacement string.

**Example:**

```
import re

def square(match):
    num = int(match.group())
    return str(num ** 2)

pattern = r'\d+'
text = 'I have 2 apples and 3 oranges'

result = re.sub(pattern, square, text)
print('Result:', result)
```

**Grouping with Parentheses:**

Parentheses ( ) are used to create groups within a regex pattern.

Groups can be quantified as a single unit and captured for later use.

**Example:**

```
import re

pattern = r'(\w+), (\w+)'
text = 'Lastname, Firstname'

match = re.match(pattern, text)
if match:
    print('Last name:', match.group(1))
    print('First name:', match.group(2))
```

**Named Groups:**

You can assign names to groups using the (?P<name>...) syntax.