Classes (hour 13)

- `object`.
- Additional features.
- Examples.

`object` class

In Python, everything is an object. The main class of all objects is the `object` class.

When you create a new class in Python without specifying a parent class, Python implicitly assumes that your class inherits from the `object` class. This means that all objects created from user-defined classes, as well as many built-in Python types like lists, strings, and integers, ultimately inherit from the object class.

Example

```python
# User-defined class without specifying a parent class
class MyClass:
    pass

# Create an instance of MyClass
obj = MyClass()

# Check the type of obj and its inheritance hierarchy
print(type(obj))                    # <class '__main__.MyClass'>
print(isinstance(obj, object))  # True
```

Additional features (type)

In Python, you can use several methods to determine the type of a variable or object. These methods will help you identify whether a variable is an integer, string, list, dictionary, or any other data type. Here are some commonly used methods to check variable types:

- **type()** function: The type() function is a built-in Python function that returns the type of an object. It takes a single argument and returns the type of that object as a class or type object.

```
x = 10
y = "Hello"
z = [1, 2, 3]

print(type(x))  # <class 'int'>
print(type(y))  # <class 'str'>
print(type(z))  # <class 'list'>
```

Additional features (type)

```
class Parent:
    pass
class Child(Parent):
    pass

obj = Child()

print(type(obj))  # <class '__main__.Child'>
```

Additional features (`issubclass`)

- `issubclass()` function: The `issubclass()` function checks whether a class is a subclass of another class. It takes two arguments: a class and a potential parent class. It returns `True` if the first class is a subclass of the second class, otherwise `False`.

```
class Parent:
    pass
class Child(Parent):
    pass

print(issubclass(Child, Parent))  # True
```

Additional features (`isinstance`)

- `isinstance()` function: The `isinstance()` function is used to check if an object is an instance of a specific class or type. It returns `True` if the object is an instance of the specified class, and `False` otherwise.

```
x = 10
y = "Hello"
z = [1, 2, 3]

print(isinstance(x, int))  # True
print(isinstance(y, str))  # True
print(isinstance(z, list)) # True
```

Additional features (__bases__)

- __bases__ attribute: Each class in Python has an attribute called .__bases__, which is a tuple containing its base classes. You can use this attribute to directly see the immediate parent classes of a given class.

```
class Parent:
    pass
class Child(Parent):
    pass

print(Child.__bases__)  # (<class '__main__.Parent'>,)
```

Additional features (__dict__)

- __dict__ attribute: objects are instances of classes, and each object can have its own unique set of attributes. The __dict__ attribute of an object is a dictionary that contains all the attributes (variables) defined for that specific object, along with their corresponding values.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

print(person1.__dict__)  # {'name': 'Alice', 'age': 30}
print(person2.__dict__)  # {'name': 'Bob', 'age': 25}
```

Additional features (__dict__)

Classes are themselves objects, and they have attributes too. The `__dict__` attribute of a class is a dictionary that contains all the attributes defined in the class, including methods and class variables.

```
class MyClass:
    class_variable = 100
    def __init__(self, x):
        self.x = x
    def print_x(self):
        print(self.x)


print(MyClass.__dict__)
```

Additional features (__dict__)

Output

```
{
    '__module__': '__main__',
    'class_variable': 100,
    '__init__': <function MyClass.__init__ at 0x...>,
    'print_x': <function MyClass.print_x at 0x...>,
    '__dict__': <attribute '__dict__' of 'MyClass' objects>,
    '__weakref__': <attribute '__weakref__' of 'MyClass'
objects>,
    '__doc__': None
}
```

Additional features (`__class__.__name__`)

- `__class__.__name__`: You can use the `__class__.__name__` attribute to get the name of the class of an object. This can be useful when you want to get a more human-readable representation of the object's type.

```
x = 10
y = "Hello"
z = [1, 2, 3]


print(x.__class__.__name__)  # int
print(y.__class__.__name__)  # str
print(z.__class__.__name__)  # list
```

These methods allow you to identify the type of a variable or object in Python. The choice of method depends on your specific use case and whether you want to get the actual class/type object or simply want to check if an object is an instance of a specific class.

Examples

```
class myList(list):
    def __str__(self):
        return f"this is list with values: {', '.join(str(el) for el in self)}"

    def __add__(self, other):
        if isinstance(other, list) and len(other) == len(self):
            return [self[i] + other[i] if isinstance(other[i], int) else None for i in range(len(self))]
        else:
            raise ValueError("Both lists should have the same length.")


# Test the myList class
my_list = myList([1, 5, 1, 3])
other_list = myList([2, "hello", 4, 5])

print(my_list)  # Output: "this is list with values: 1, 5, 1, 3"
print(my_list + [2, 3, 4, 5])  # Output: [3, 8, 5, 8]
print(my_list + other_list)  # Output: ValueError: Both lists should have the same length.
```

Examples

```python
# Base class 1
class Color:
    def __init__(self, color):
        self.color = color
        print("Color's __init__ called")

    def display_color(self):
        print(f"The shape color is {self.color}")


# Base class 2
class Area:
    area = 0

    def __init__(self):
        print("Area's __init__ called")

    def calculate_area(self):
        pass

    def display_area(self):
        print(f"The area is {self.area}")


# Derived class inheriting from both Shape and Area
class Rectangle(Color, Area):
    def __init__(self, color, width, height):
        # super().__init__(color)
        Color.__init__(self, color)
        Area.__init__(self)
        self.width = width
        self.height = height
        self.calculate_area()
        print("Rectangle's __init__ called")

    def calculate_area(self):
        self.area = self.width * self.height


# Create an instance of Rectangle
rectangle = Rectangle("blue", 5, 3)

# Access methods from both base classes
rectangle.display_color()  # Output: The shape color is blue
rectangle.display_area()  # Output: The area is 15
```

Type hint

- **type hints**: Python 3.5 introduced type hints as part of PEP 484. Type hints allow you to specify the expected type of a variable using annotations. These annotations are for informational purposes and do not affect the actual behavior of the code.

```
def add_numbers(a: int, b: int) -> int:
    return a + b
```

Type hints help make the code more readable and provide information to tools like linters and type checkers.

Overloading of functions

```
def func1(a: int) -> str:
   return "this is an integer“

def func1(a: str) -> str:
   return "this is a string"
```

Overloading of functions

```
def func1(a):
    if isinstance(a, int):
        return "this is an integer“
    elif isinstance(a, str):
        return "this is a string“
    else:
        return "unknown type"print(func1(10))
       # Output: "this is an integer“
print(func1("Hello"))  # Output: "this is a string“
print(func1(3))      # Output: " this is an integer“
print(func1(3.14))     # Output: "unknown type"
```

Iterables (hour 14)

- Iterables.
- Examples of iterables.
- Iterables usage.

Iterables

An iterable is an object in Python that can be iterated over, meaning it can be used in a loop or used to generate an iterator. Iterables are collections of items that you can iterate through one by one, accessing each item sequentially.

In Python, many built-in data types are iterable, including strings, lists, tuples, dictionaries, sets, and more. Additionally, custom objects can be made iterable by implementing special methods such as `__iter__()` and `__next__()`.

When we use an iterable in a loop or conversion function, Python uses the iterable's `__iter__()` method to create an iterator, which allows you to access the elements one by one. If the iterable doesn't have an `__iter__()` method, Python will raise an error.

Check if an object is *iterable*

All iterables in Python implement the iterable protocol, which means they have a special method called `__iter__()`.

```
# Check if the object is iterable
print(hasattr(list, '__iter__'))  # Output: True

print(hasattr(int, '__iter__'))  # Output: False
```

Examples of iterables

- **Lists, Tuples, and Strings**:

```
my_list = [1, 2, 3, 4, 5]
my_tuple = (10, 20, 30)
my_string = "Hello“
```

- **Dictionaries**:

```
my_dict = {'name': 'John', 'age': 30, 'occupation': 'Engineer'}
```

Examples of iterables

- **Sets:**

```
my_set = {1, 2, 3, 4, 5}
```

- **Range:**

```
my_range = range(1, 6)  # Creates a range object from 1 to 5 (exclusive)
```

- **File Objects:**

```
my_file = open('example.txt', 'r')
```

Iterables usage

To iterate over an iterable, you can use a loop like `for` loop or `while` loop, or you can use functions like `list()`, `tuple()`, or `set()` to convert the iterable into a list, tuple, or set, respectively.

- Example using a `for` loop:

```
my_list = [1, 2, 3, 4, 5]
for item in my_list:
    print(item)
```

- Example using the `list()` function:

```
my_tuple = (10, 20, 30)
my_list = list(my_tuple)
print(my_list)  # Output: [10, 20, 30]
```

List comprehensions (*hour 12*)

- List comprehensions.
- Creating and manipulating lists using comprehensions.
- Nested comprehensions.

Basic List Comprehension Syntax

The general syntax of a list comprehension is as follows:

```
new_list = [expression for item in iterable if condition]
```

- **expression**: The expression to be applied to each item in the iterable.
- **item**: The variable representing each element in the iterable.
- **iterable**: The data source, typically a list, tuple, or any other iterable.
- **condition (optional)**: An optional filter to include only items that satisfy the given condition.

Example: Creating a List with *Squares of* Numbers

```
# Using a for loopnumbers = [1, 2, 3, 4, 5]
squares = []
for num in numbers:
    squares.append(num ** 2)
print(squares)  # Output: [1, 4, 9, 16, 25]

# Using list comprehension
numbers = [1, 2, 3, 4, 5]
squares = [num ** 2 for num in numbers]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

Example: Filtering Odd Numbers

```
# Using a for loop
numbers = [1, 2, 3, 4, 5]
odd_numbers = []
for num in numbers:
    if num % 2 != 0:
        odd_numbers.append(num)
print(odd_numbers)  # Output: [1, 3, 5]
# Using list comprehension with condition
numbers = [1, 2, 3, 4, 5]
odd_numbers = [num for num in numbers if num % 2 != 0]
print(odd_numbers)  # Output: [1, 3, 5]
```

Nested List Comprehensions

List comprehensions can also be nested to create more complex data structures like nested lists.

```
# Creating a 2D list using nested comprehensions
matrix = [[i + j for i in range(3)] for j in range(4)]
print(matrix)  # Output: [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5]]
```

Nested comprehensions can be used for any level of nesting, allowing you to create even more complex data structures if needed.

List Comprehension with Conditionals

List comprehensions can include conditional expressions, allowing you to apply different transformations based on specific conditions.

```
# Creating a list of squared even numbers and cubed odd numbers
numbers = [1, 2, 3, 4, 5]
result =
      [num ** 2 if num % 2 == 0 else num ** 3 for num in numbers]
print(result)  # Output: [1, 4, 27, 16, 125]
```

List comprehensions are a powerful tool for generating and manipulating lists in a concise and readable way. They are widely used in Python and considered a Pythonic approach to list processing.

Generators (hour 13)

- Generators.
- The key characteristics of generators.
- Generators examples

Generators

Generators in Python are a special type of iterable, allowing you to create iterators in a more concise and efficient manner. They are functions defined using the yield keyword, which makes them different from regular functions. Generators enable you to produce a sequence of values on-the-fly, one at a time, as they are needed, rather than generating and storing all values in memory at once.

To create a generator, you define a function that uses the `yield` keyword to `yield` values. When the generator function is called, it returns a generator object, which can be used to iterate through the values produced by the generator.

The key characteristics of generators

- **Lazy Evaluation**: Generators use lazy evaluation, meaning they produce values only when requested. Each time a generator's `__next__()` method (or the `next()` function) is called, the generator executes its code until it reaches a yield statement. It then yields the value and pauses its execution, saving its state. When `__next__()` is called again, the generator resumes execution from where it left off.
- **Memory Efficiency**: Since generators produce values on-the-fly, they are more memory-efficient compared to other data structures like lists or tuples. Instead of storing all values in memory, generators yield each value as it's needed, reducing memory consumption.
- **Iterators**: Generators are iterators. They can be used in for loops or with functions like `next()` to iterate through the sequence of values they generate.

yield

In Python, `yield` is a keyword used in the context of generators and generator functions. It allows you to create custom iterators in a more concise and memory-efficient way. A generator is a special type of iterator that generates values on-the-fly, one at a time, as they are needed, rather than generating and storing all values in memory at once.

```
def my_generator():
    # Some code
    yield value
    # Some code
    yield value
    # More code
```

yield

When a generator function is called, it doesn't execute the entire function at once. Instead, it returns a generator object, which can be used to iterate through the values generated by the function.

Each time the generator function encounters a yield statement, it "pauses" its execution and returns the value specified in the yield statement to the caller. The state of the function is saved, allowing it to resume execution from where it left off the next time the generator's `__next__()` method is called.

Generators examples

```
def squares_generator(n):
    for i in range(n):
        yield i ** 2

# Using the generator to iterate through squares
squares = squares_generator(5)
print(next(squares))  # Output: 0
print(next(squares))  # Output: 1
print(next(squares))  # Output: 4
print(next(squares))  # Output: 9
print(next(squares))  # Output: 16

# The generator will raise a StopIteration exception when there are no more values to yield
print(next(squares))  # Raises StopIteration
```

Generators examples

```
# Using the generator to iterate through squares
squares = squares_generator(5)
while True:
    try:
        print(next(squares))
    except StopIteration:
        break
# output
# 0
# 1
# 4
# 9
# 16
```

Generators examples

```
my_sqrt_gnrt = squares_generator(5)
for val in my_sqrt_gnrt:
    print(val)
```

Or

```
my_sqrt_gnrt = squares_generator(5)
val = next(my_sqrt_gnrt, None)
while val is not None:
    print(val)
    val = next(my_sqrt_gnrt, None)
```