## Inheritance and private variables

Inheritance Syntax

To define a derived class inheriting from a base class.

Inheritance supports recursive attribute search, looking up the chain of base classes.

**class DerivedClassName(BaseClassName):**

Instantiation and method calling

**DerivedClassName()** creates a new instance; method references are resolved by searching attributes in class chains.

* Methods of base classes can be overridden in derived classes, allowing extension or replacement of base class methods.

BaseClassName.methodName(self, arguments)

Multiple Inheritance

Define a class with multiple base classes using **class DerivedClassName(Base1, Base2, Base3):**.

Attribute search in multiple inheritance generally follows a depth-first, left-to-right approach.

**Dynamic Ordering and Method Resolution Order:**

* Python employs a dynamic method resolution order supporting cooperative calls to **super()**.
* The method resolution order dynamically changes to prevent multiple accesses to base classes in diamond relationship cases.
* Linearizing the search order ensures monotonicity and precedence of parents, allowing reliable and extensible classes with multiple inheritance.

**Built-in Functions for Inheritance**

* **isinstance(obj, int)** to check an instance’s type
* **issubclass(bool, int)** to check class inheritance.

## Private Variables

Private Instance Variables in Python

Python does not have strict private instance variables that cannot be accessed except from inside an object.

Instead, it follows a convention: any name starting with a single underscore (e.g., **\_spam**) is considered non-public, intended for internal use, and subject to change without notice. It's a convention, not an enforced rule.

Limited Support for Class-Private Members - Name Mangling

* To address the need for class-private members and avoid potential clashes with subclass names, Python offers a mechanism called name mangling.
* Any identifier in the form **\_\_spam** (at least two leading underscores, at most one trailing underscore) is textually replaced with **\_classname\_\_spam**, where **classname** is the current class name stripped of leading underscores. This occurs within the class definition.

Usefulness of Name Mangling

Name mangling helps subclasses to override methods without breaking intraclass method calls.

Even if a subclass introduces a **\_\_update** identifier, it gets replaced by **\_Mapping\_\_update** or **\_MappingSubclass\_\_update** respectively, ensuring its uniqueness within its class hierarchy.

Access to Private Variables

Although name mangling aims to prevent accidental access to private variables, it's still possible to access or modify these "private" variables.

This flexibility aligns with Python's philosophy of being pragmatic and treating programmers as consenting adults.

Limitations and Restrictions

Certain functions (**exec()**, **eval()**, **getattr()**, **setattr()**, **delattr()**, referencing **\_\_dict\_\_**) do not strictly follow the name-mangling rules, considering the current class name, akin to how the global statement works in Python. This behavior is restricted to code that is byte-compiled together.

class MyClass:

    def \_\_init\_\_(self):

        self.\_\_private\_var = 10  # Private variable using name mangling

    def get\_private\_var(self):

        return self.\_\_private\_var

    def set\_private\_var(self, value):

        self.\_\_private\_var = value

# Creating an instance of MyClass

obj = MyClass()

# Trying to access the private variable directly

# This will result in an AttributeError as the name is mangled

try:

    print(obj.\_\_private\_var)

except AttributeError as e:

    print("AttributeError:", e)

# Accessing the private variable using a getter method

print(obj.get\_private\_var())  # Output: 10

# Trying to modify the private variable directly

# This will not affect the actual variable as it's name-mangled

obj.\_\_private\_var = 20

# Accessing the private variable again using the getter method

# It remains unchanged despite the attempted modification

print(obj.get\_private\_var())  # Output: 10

# Modifying the private variable using a setter method

obj.set\_private\_var(30)

# Accessing the private variable again using the getter method

# It reflects the change made using the setter method

print(obj.get\_private\_var())  # Output: 30

## Iterators and Generators

Iterators

**Looping Convenience:** Various container objects can be looped over using a **for** statement.

**Iterator Access:** Behind the scenes, the **for** statement calls **iter()** on the container object, returning an iterator.

**Iterator Protocol:** Iterator objects define **\_\_next\_\_()** to access elements one at a time, raising **StopIteration** when done.

**s = 'abc'**

**it = iter(s)**

**next(it)** # 'a'

**next(it)** # 'b'

**next(it)** # 'c'

**next(it)** # Raises StopIteration

Adding Iterator Behavior:

Define **\_\_iter\_\_()** returning an object with **\_\_next\_\_()** or define **\_\_next\_\_()** directly.

**class Reverse:**

**def \_\_init\_\_(self, data):**

**self.data = data**

**self.index = len(data)**

**def \_\_iter\_\_(self):**

**return self**

**def \_\_next\_\_(self):**

**if self.index == 0:**

**raise StopIteration**

**self.index -= 1**

**return self.data[self.index]**

Generators

Iterator Creation: Written like regular functions but use yield statement to return data.

Resumable Execution: Each next() call on a generator resumes where it left off.

**def reverse(data):**

**for index in range(len(data)-1, -1, -1):**

**yield data[index]**

Generator Advantages:

Automatic Method Creation: \_\_iter\_\_() and \_\_next\_\_() methods are created automatically.

State Saving: Local variables and execution state are saved between calls, improving clarity and ease of writing.

Automatic StopIteration: Terminating generators automatically raise StopIteration.

## Generator Expressions

**Compact Syntax:** Similar to list comprehensions but using parentheses.

**Usage:** Immediate use within an enclosing function.

**Memory-Friendly:** More memory-efficient compared to list comprehensions.

**sum(i\*i for i in range(10))** # Sum of squares

**sum(x\*y for x,y in zip(xvec, yvec))** # Dot product of vectors

**sine\_table = {x: sin(x\*pi/180) for x in range(0, 91)}** # Generating sine values

**unique\_words = set(word for line in page for word in line.split())** # Unique words from lines

**valedictorian = max((student.gpa, student.name) for student in graduates)** # Top student

**list(data[i] for i in range(len(data)-1, -1, -1))** # Reversing a string

**OS Module (Operating System Interaction):**

* **Functionality:** Offers multiple functions for OS interaction.

**os.getcwd()** # Get current working directory

**os.chdir('/server/accesslogs')** # Change current directory

**os.system('mkdir today')** # Run system shell command

**Important Note:** Prefer **import os** over **from os import \*** to prevent clashes with built-in functions.

**shutil Module:** Provides a simpler interface for file and directory management.

**shutil.copyfile('data.db', 'archive.db')** # Copy file

**shutil.move('/build/executables', 'installdir')** # Move directory

Command Line Arguments:

**sys.argv:** Stores command line arguments as a list in the **sys** module.

**import sys**

**print(sys.argv)** # Displays command line arguments passed.

**argparse Module:** Preferred over manual **sys.argv** processing for better command line argument handling.

**import argparse**

**parser = argparse.ArgumentParser(description='An argparse example.')**

# Define arguments and parse them

**args = parser.parse\_args()**