**OBJECTS\_&\_CLASSES-Week04\_13-17-Nov2023­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­**

**WEEK04: OBJECTS AND CLASSES­**

**DAY 1: INHERITANCE AND PRIVATE VARIABLES**

**INHERITANCE**

* **Inheritance=**It is a mechanism that **allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class.**
* Inheritance is the **capability of one class to derive or inherit the properties from another class.**
* Inheritance **allows us to define a class that inherits all the methods and properties from another class.**
* Inheritance ***allows you to inherit the properties of a class***, i.e., base class to another, i.e., derived class.
* **Parent class** is the ***class being inherited from***, also called base class.
* **Child class** is the **class that inherits from another class**, also called derived class.

**The benefits of Inheritance in Python are as follows:**

* It represents real-world relationships well.
* It provides the **reusability of a code**. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
* It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.
* Inheritance offers a simple, understandable model structure.
* Less development and maintenance expenses result from an inheritance.

**Python Inheritance Syntax**

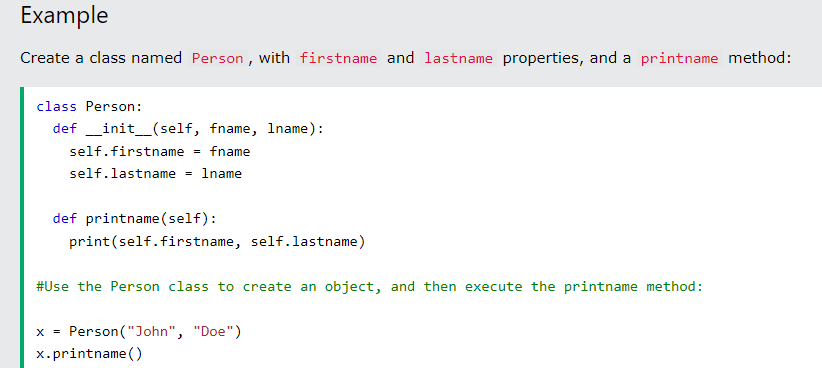
**Class BaseClass:**

**{Body}**

**Class DerivedClass(BaseClass):**

**{Body}**

**CREATA A PARENT CLASS[BASE CLASS]**

* Example of a parent class below showing a Person class with a printname method
* 

**Create a Child Class**

* To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:
* Example below of creating a child class
* **class Student(Person):**
* **pass**
* Use the pass keyword when you do not want to add any other properties or methods to the class.
* Now the Student class has the same properties and methods as the Person class.
* More detailed example of creating a Child Class
* A computer screen shot of a video game

  Description automatically generated
* **MULTIPLE INHERITANCE**=When a class is derived from more than one base class it is called multiple Inheritance. The derived class inherits all the features of the base case.

**PRIVATE VARIABLE**

* In actual terms (practically), python doesn’t have anything called private member variable in Python. However, adding two underlines(\_\_) at the beginning makes a variable or a method private is the convention used by most python code.

**NAME MANGLING**

* In Python, **name mangling is a way to handle class-private members** (variables or methods) **to avoid conflicts with names in subclasses**. When an identifier has at least two leading underscores or at most one trailing underscore (e.g., \_\_geek), it gets replaced with \_classname\_\_geek, where classname is the current class name with leading underscores stripped.
* For example, if you have a class with a private variable \_\_geek, Python internally changes it to \_classname\_\_geek. This helps prevent accidental name clashes, especially when subclasses want to override methods without causing issues with intraclass method calls.
* So, name mangling is a mechanism that occurs within the class definition, ensuring that these "mangled" names are unique to each class, which is helpful for maintaining a clear and organized structure in your code.

**Day 2 – Iterators, Generators, and more**

**ITERATORS AND GENERATOR**

* **A process that is repeated more than one time by applying the same logic is called an Iteration**
* In programming languages like python, a loop is created with few conditions to perform iteration till it exceeds the limit
* **Iterator definition: An iterator is like a tool that helps you go through each item in a collection (like a list, tuple, or set) one at a time.**
* **Implemented using a class:** Think of it as a special kind of object that's created using a blueprint called a class. This object knows how to move through the items in a collection.
* **No need for a separate counter variable:** Unlike a regular loop where you might use a variable to keep track of your position in the collection, an iterator handles this internally. You just ask it for the next item.
* **Lazy evaluation:** This means the iterator doesn't do all the work at once. It only does the work when you ask for the next item. It's like bookmarking where you are, and only turning the page when you want to see what's next. This can be more efficient because it doesn't calculate everything upfront.
* **Avoids repeated evaluation:** If there's a complex calculation involved in getting the next item, the iterator only does it when you need that item. It doesn't calculate everything in advance, saving memory and processing power.
* So, **an iterator is a smart way to move through a bunch of things, doing work only when needed, and without needing extra variables to keep track of where you are.**

**Using an iterator-**

* **iter() keyword=** is used to create an iterator containing an iterable object.
* **next() keyword=** is used to call the next element in the iterable object.
* After the iterable object is completed, to use them again reassign them to the same object**.**
* **Iterator vs Iterable**
* **Lists, tuples, dictionaries, and sets are all iterable objects**. They are iterable containers which you can get an iterator from.
* **All these objects have a iter() method which is used to get an iterator**:
* Eg below returning an iterator from a tuple and printing each value
* A black rectangular object with white text

  Description automatically generated
* **Looping Through an Iterator**
* We can also use a for loop to iterate through an iterable object:
* Eg below:iterate the vales of a tuple
* 
* Can also iterate the characters of a string
* A black rectangular object with white text

  Description automatically generated with medium confidence
* **Create an Iterator**
* To create an object/class as an iterator you have to implement the **methods \_\_iter\_\_() and \_\_next\_\_() to your object.**
* As you have learned in the Python Classes/Objects chapter, all classes have a function called \_\_init\_\_(), which allows you to do some initializing when the object is being created.
* **The \_\_iter\_\_() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.**
* **The \_\_next\_\_() method also allows you to do operations, and must return the next item in the sequence.**
* Eg Create an iterator that returns numbers, starting with 10, and each sequence will increase by one (returning 1,2,3,4,5
* **StopIteration**
* The example above would continue forever if you had enough next() statements, or if it was used in a for loop.
* **To prevent the iteration from going on forever, we can use the StopIteration statement**.
* In the \_\_next\_\_() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:
* Eg showing how to use the StopIteration method
* A blue background with white text

  Description automatically generated

**Generators**

* **Generator definition:** **A generator is a special type of function that helps you create iterators in a simpler way. Instead of using "return" to give you a value, it uses the keyword "yield."**
* **Implemented using a function:** Generators are made using a function, just like regular functions you might write. But they **have the special "yield" keyword**.
* **Lazy evaluation, just like iterators:** Similar to iterators, generators also follow lazy evaluation. They don't do all the work at once**; they generate (or yield) values one at a time when you ask for them.**
* **Yield function:** Instead of returning a value and exiting the function like a regular function, a generator uses "yield" to temporarily give you a value without leaving the function. It's like a pause button; the function can resume where it left off.
* **Iterable sequence:** When you use a generator, it gives you a sequence of data that you can go through one step at a time. You don't have to store the entire sequence in memory; it's generated as you go along.
* So, generators are a handy way to create iterators. They use "yield" to provide values one at a time, and they're efficient because they don't generate the entire sequence upfront, saving memory.

**DIFFERENCE BETWEEN ITERATOR AND GENERATOR**

|  |  |
| --- | --- |
| * **ITERATOR** | * **GENERATOR** |
| Class is used to implement an iterator | Function is used to implement a generator. |
| Local Variables aren’t used here. | All the local variables before the yield function are stored. |
| Iterators are used mostly to iterate or convert other objects to an iterator using iter() function. | Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop |
| Iterator uses iter() and next() functions | Generator uses yield keyword |
| Every iterator is not a generator | Every generator is an iterator |

**Generator Expressions**

* In Python, **to create iterators, we can use both regular functions and generators.**
* Generators are written just like a normal function but **we use yield() instead of return() for returning a result.**
* It is more powerful as a tool to implement iterators. It is easy and more convenient to implement because it offers the evaluation of elements on demand.
* Unlike regular functions which on encountering a return statement terminates entirely, **generators use a yield statement in which the state of the function is saved from the last call and can be picked up or resumed the next time we call a generator function.**
* Another great advantage of the generator over a list is that it takes much less memory.
* **Generator Expression:** In Python, a generator expression is a quick and short way to make a special kind of object called a generator.
* **Similar to List Comprehension:** You might have heard of list comprehensions, which are a way to make lists in a single line. A generator expression is similar, but it doesn't create a list; it makes a generator object.
* **Creates a Generator Object:** Instead of immediately making a list with all the values, a generator expression makes an object that can give you values one at a time. It's like a list, but it doesn't store everything in memory at once
* **Iterate Over for Values:** You can use this generator object in a loop to get each value one by one. It's a handy way to work with a sequence of values without creating a big list in memory.
* So, a generator expression is a cool and concise way in Python to create a generator, which is a special kind of object that lets you work with values one at a time without using a lot of memory.

**Error Output Redirection and Program Termination**

* **Sys Module:**
* The sys module in Python has attributes for standard input (stdin), standard output (stdout), and standard error (stderr).
* **sys.stderr is particularly useful for showing warnings and error messages, even if the standard output (stdout) has been redirected.**
* Example:
* A computer screen with green text

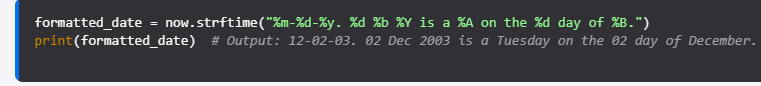
  Description automatically generated
* **Exiting a Script:**
* The sys.exit() function is the most direct way to terminate a Python script. **It's used to stop the execution of the program.**
* Example:
* A black rectangular object with a white border

  Description automatically generated
* **String Pattern Matching with re Module:**
* The re module in Python **provides tools for working with regular expressions**, which are advanced techniques for processing strings.
* **String Methods for Simple Tasks:**
* For simple string manipulations, using string methods is preferred over regular expressions because they are easier to read and debug.
* Example:
* **Internet Access:**
* There are modules in Python for internet access and processing internet protocols. Examples include **urllib.request** for retrieving data from URLs and **smtplib** for sending emails.
* Example:
* A screen shot of a computer

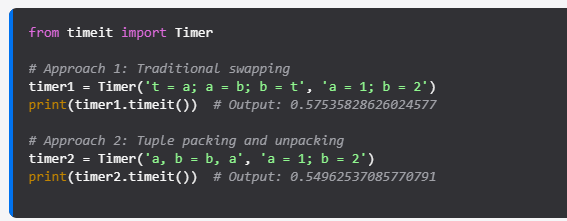
  Description automatically generated

**Day 3 – Dates and Times, Data Compression, Output Formatting, and more!**

* The datetime module in Python provides classes for working with dates and times. It allows you to manipulate dates and times in various ways, perform arithmetic operations, and format them for output. The module also supports working with timezones.
* **Working with Dates**
* To work with dates, you can import the date class from the datetime module.
* Example below using datetime module
* A black background with white text

  Description automatically generated
* You can also format dates using the **strftime method,** which stands for "string format time". It takes a format string as an argument and returns a formatted string representation of the date.
* 
* **Working with Date Arithmetic**
* The datetime module also supports arithmetic operations on dates. You can perform calculations like finding the difference between two dates.eg Below
* A screenshot of a computer

  Description automatically generated
* **Data Compression**
* Python provides modules for working with common data archiving and compression formats such as **zlib, gzip, bz2, lzma, zipfile, and tarfile**. These modules allow you to compress and decompress data, calculate checksums, and work with compressed files.
* Here's an example using the zlib module to compress and decompress data:
* A screenshot of a computer program

  Description automatically generated
* In this example, we compress the string s using zlib, calculate the length of the compressed data, decompress it back to the original string, and calculate the CRC32 checksum of the original string.
* **Performance Measurement**
* Python provides tools for measuring the performance of your code. The timeit module allows you to time the execution of small code snippets. It can help you compare the performance of different approaches to the same problem.
* Here's an example using the timeit module to compare the performance of two different approaches to swapping variable values:
* 
* In this example, we use the Timer class from the timeit module to measure the execution time of two different swapping approaches. The timeit() method returns the execution time in seconds.
* **Quality Control**
* Python provides modules for writing and running tests to ensure the quality of your code. The doctest module allows you to write tests directly in your code's docstrings and validate them automatically.
* Here's an example using the doctest module to test a function:
* A screen shot of a computer program

  Description automatically generated
* In this example, we define a test case class TestStatisticalFunctions that inherits from unittest.TestCase. We define test methods within the class, each containing assertions to validate the expected behavior of the code under test. We then use unittest.main() to run all the tests in the test case.
* By writing tests and running them frequently, you can ensure that your code behaves as expected and catch any regressions or bugs early in the development process.

**Output Formatting**

* **reprlib module:**
* Purpose: Provides a customized version of the repr() function for displaying large or deeply nested containers in a more concise way.
* **pprint module:**
* Purpose: Offers more sophisticated control over printing objects, especially those that are nested or complex. It adds line breaks and indentation for better readability.
* **textwrap module:**
* Purpose: Formats paragraphs of text to fit a given screen width by adding line breaks.
* **locale module:**
* Purpose: Accesses a database of culture-specific data formats. The locale.format() function is used to format numbers with group separators.
* **string module (Template class):**
* Purpose: Provides a versatile Template class for string templating with a simplified syntax. It uses placeholders and allows customization without altering the application.
* **Template subclass with custom delimiter:**
* Purpose: Demonstrates a custom Template subclass with a different placeholder delimiter. In this example, a batch renaming utility uses percent signs as delimiters

**DAY 4:Logging, Managing packages with pip, and Floating point Arithmetic.**

**LOGGING**

* The logging module in Python provides a comprehensive and flexible logging system. **It allows you to send log messages to various destinations**, such as a file or the standard error output
* **Logging Levels**
* The logging module provides several levels of severity for log messages, including **DEBUG, INFO, WARNING, ERROR, and CRITICAL. Each level represents a different level of importance or severity. In** the code snippet, we have used the following levels:
* logging.debug: Logs a debugging message.
* logging.info: Logs an informational message.
* logging.warning: Logs a warning message.
* logging.error: Logs an error message.
* logging.critical: Logs a critical error message.
* **Log Output**
* By default, log messages with severity levels of INFO and DEBUG are suppressed, and the output is sent to the standard error. In the code snippet, the log messages are prefixed with the severity level and the logger name (root in this case).
* **Customization and Configuration**
* The logging system in Python is highly customizable. You can configure it to send log messages to different destinations, such as email, datagrams, sockets, or an HTTP server. You can also customize the logging format and filter log messages based on their severity or other criteria.
* The logging system can be configured directly from Python code or loaded from a user-editable configuration file. This allows for customized logging without altering the application code.

**Virtual Environments**

* When developing Python applications, you may need to use packages and modules that are not part of the standard library. Sometimes, different applications require different versions of the same library, which can create conflicts. For example, if application A needs version 1.0 of a module and application B needs version 2.0, installing either version would make one application unable to run.
* To solve this problem, you can create a virtual environment. A **virtual environment is a self-contained directory that includes a specific version of Python and additional packages. Each application can have its own virtual environment, allowing them to have different versions of libraries without conflicts.**
* To **create a virtual environment, you can use the venv module** in Python. This module will install the most recent version of Python available on your system. If you have multiple Python versions, you can specify a particular version**.**
* Once you've created a virtual environment, you can activate it on Windows, IOS using different commands
* Activating the virtual environment will change your shell's prompt to indicate which virtual environment you're using. It will also modify the environment so that running python will use the Python version and installation within the virtual environment.
* By using virtual environments, you can ensure that each application has its own isolated environment with the required versions of libraries. This way, upgrading or changing libraries for one application won't affect others.
* If you want to learn more about virtual environments and their usage, you can refer to the Python documentation or search for tutorials and guides online.

**Managing packages with pip**

* The **pip program is used to install, upgrade, and remove Python packages. By** default, pip installs packages from the Python Package Index (PyPI), which is a repository of Python packages available for use.
* You can use pip to search for packages on PyPI. For example, if you want to find packages related to astronomy, you can use the command pip search astronomy. This will display a list of packages related to astronomy, along with their descriptions.
* To **install a package, you can use the pip install command followed by the package name. For example, pip install novas will install the package named "novas".** You can **also specify a specific version of a package** by using the == operator. For example, pip install requests==2.6.0 will install version 2.6.0 of the "requests" package.
* If you want **to upgrade a package** to the latest version, you can use the --upgrade option. For **example, pip install --upgrade requests** will upgrade the "requests" package to the latest version available.
* **To uninstall a package, you can use the pip uninstall command** followed by the package name. For example, pip uninstall requests will remove the "requests" package from your environment.
* You can use the **pip show command to display information about a specific package**, such as its version, author, and license.
* **The pip list command will display a list** of all the packages installed in your environment.
* To generate a list of installed packages in a format that can be easily shared or used to recreate the environment, you can use the pip freeze command. This command will output a list of installed packages along with their versions, which can be saved to a file. For example, pip freeze > requirements.txt will save the list of packages to a file named "requirements.txt".
* To install all the packages listed in a requirements file, you can use the pip install -r command followed by the file name. For example, pip install -r requirements.txt will install all the packages listed in the "requirements.txt" file.
* These are some of the basic commands and features of pip. It is a powerful tool for managing Python packages and dependencies, allowing you to easily install, upgrade, and remove packages as needed.

**Floating point Arithmetic**

* Floating point numbers are represented in binary format inside computers rather than decimal. This causes some decimal numbers like 0.1 to be endlessly repeating numbers in binary format.
* Some key points:
* **Representation as fractions**
* Floating point numbers store values as fractions of the form J/2^N where J is an integer and N is the exponent. For example, the binary representation of 0.1 is an infinitely repeating fraction.
* **Approximation errors**
* Since most decimals can't be represented exactly, floating point values are approximations. This leads to small rounding errors in calculations.
* **Display format vs internal value**
* Languages often display a rounded string rather than the full internal binary value, even though the stored value is an approximation.
* **Solutions for errors**
* Use rounding/formatting when displaying results
* Check for equality within a small tolerance range rather than exact equality
* Use decimal or fractions modules for applications requiring exact precision