1.1

Q. What is difference between static and dynamic variable in python?

Answer -

**- Static Variables:**

- Shared among all instances of a class.

- Defined directly within the class outside of any method.

- Accessed using the class name or instances of the class.

**- Dynamic (Instance) Variables:**

- Specific to each instance of a class.

- Defined inside the `\_\_init\_\_` method using `self`.

- Accessed using instances of the class (`self`).

Q. Explain the purpose of ‘pop’, ‘popitem’,’clear()’ in a dictionary with suitable example.

Answer-

**pop(key[, default])**:

* Removes the key-value pair from the dictionary and returns the corresponding value.
* If the key is not found and a default value is provided, that default value is returned.
* If the key is not found and no default value is provided, a KeyError is raised.

Example:

# Example of pop() method

my\_dict = {'a': 1, 'b': 2, 'c': 3}

# Remove 'b' from the dictionary

removed\_value = my\_dict.pop('b')

print(removed\_value) # Output: 2

print(my\_dict) # Output: {'a': 1, 'c': 3}

# Using pop() with default value

removed\_value = my\_dict.pop('x', 'No such key')

print(removed\_value) # Output: 'No such key'

2 **popitem()**:

* Removes and returns an arbitrary (key, value) pair from the dictionary.
* This method is useful when you need to remove and process items in an unspecified order.

Example:

# Example of popitem() method

my\_dict = {'a': 1, 'b': 2, 'c': 3}

# Remove and get an arbitrary item

removed\_item = my\_dict.popitem()

print(removed\_item) # Output: ('c', 3)

print(my\_dict) # Output: {'a': 1, 'b': 2}

**3. clear()**:

* Removes all items from the dictionary, making it empty.
* This method modifies the dictionary in place and returns None.

Example:

# Example of clear() method

my\_dict = {'a': 1, 'b': 2, 'c': 3}

# Clear all items from the dictionary

my\_dict.clear()

print(my\_dict) # Output: {}

**Q. what do you mean by FrozenSet? Explain it with suitable examples?**

Answers-

In Python, a `frozenset` is an immutable and hashable collection of unique elements. This means once a `frozenset` is created, its elements cannot be changed or updated. This is similar to the built-in `set` type in Python, but whereas `set` is mutable (you can add or remove elements), `frozenset` is immutable (you cannot change its elements after creation).

### Properties of `frozenset`:

1. \*\*Immutable\*\*: Once a `frozenset` is created, you cannot add, remove, or modify its elements.

2. \*\*Hashable\*\*: `frozenset` instances can be used as dictionary keys because they are hashable (assuming their elements are also hashable).

3. \*\*Unique Elements\*\*: Like a regular `set`, a `frozenset` contains only unique elements. If you try to create a `frozenset` with duplicate elements, only unique elements will be retained.

### Example Usage:

Here are some examples to illustrate the usage of `frozenset`:

```python

# Creating a frozenset

f\_set = frozenset([1, 2, 3, 4, 5])

print(f\_set) # Output: frozenset({1, 2, 3, 4, 5})

# Attempting to add an element (which is not possible)

# f\_set.add(6) # This would raise an AttributeError: 'frozenset' object has no attribute 'add'

# Attempting to remove an element (which is not possible)

# f\_set.remove(3) # This would raise an AttributeError: 'frozenset' object has no attribute 'remove'

# Operations like union, intersection, difference, etc., are supported

f\_set2 = frozenset([4, 5, 6, 7, 8])

print(f\_set.union(f\_set2)) # Output: frozenset({1, 2, 3, 4, 5, 6, 7, 8})

# Creating a frozenset from a regular set

regular\_set = {1, 2, 3}

frozen\_from\_set = frozenset(regular\_set)

print(frozen\_from\_set) # Output: frozenset({1, 2, 3})

# Using frozenset as keys in a dictionary

employee1 = frozenset({'name', 'age', 'department'})

employee2 = frozenset({'name', 'age', 'department'})

employees = {employee1: 'John', employee2: 'Jane'}

print(employees) # Output: {frozenset({'department', 'age', 'name'}): 'John'}

# Iterating over a frozenset

for element in f\_set:

print(element)

# Output:

# 1

# 2

# 3

# 4

# 5

```

### When to Use `frozenset`:

- \*\*As Dictionary Keys\*\*: Because `frozenset` is hashable, it can be used as a key in dictionaries if you need a set of elements as a key.

- \*\*Ensuring Immutability\*\*: Use `frozenset` when you need to guarantee that a set of elements cannot be changed after creation. This can be useful in contexts where immutability is desired for safety or clarity.

- \*\*Compatibility\*\*: If you need a set-like object that can be safely used in scenarios requiring immutability (like being stored in another set or passed as an argument to a function that requires immutable objects).

In summary, `frozenset` provides an immutable and hashable alternative to the mutable `set` in Python, suitable for situations where you need a fixed set of unique elements that should not change after creation.

**Q. what is difference between mutable and immutable data types in Python and give example of mutable and immutable data types.**

Mutable vs. Immutable Data Types in Python

* Mutable: Can be changed after creation.
  + Examples: list, dict, set.

Immutable: Cannot be changed after creation.

* Examples: str, tuple, int, float.

Key Difference: Mutable objects can be modified in place, while immutable objects require creating a new object for any modification

**Q.what is \_\_init\_\_ ? explain it with an example.**

\_\_init\_\_ is a special method in Python, also known as a constructor. It is automatically called when an object of a class is created. The purpose of \_\_init\_\_ is to initialize the object's attributes and set up any necessary initial state.

Key Points about \_\_init\_\_:

* Initialization: \_\_init\_\_ sets up the initial state of an object by assigning values to the object's properties.
* Automatic Invocation: It is called automatically when a new instance of the class is created.
* Not a Constructor: Unlike constructors in other programming languages, \_\_init\_\_ is not responsible for creating the object; it just initializes it.

Example:

1. Class Definition: A class named Person is defined with an \_\_init\_\_ method.
2. \_\_init\_\_ Method: It takes self, name, and age as parameters.
   * self.name = name assigns the value of name to the instance attribute name.
   * self.age = age assigns the value of age to the instance attribute age.
3. Creating an Object: person1 = Person("Alice", 30) creates a new instance of Person. The \_\_init\_\_ method is called with "Alice" and 30 as arguments, initializing person1 with those values.
4. Using the Object: The object person1 can access its attributes (name and age) and methods (introduce()).

In summary, \_\_init\_\_ is essential for setting up the properties of a new object in Python, allowing you to define how objects of a class are initialized.

**Q, what is docstring in python? Explain with an example.**

A docstring (documentation string) in Python is a special type of comment that is used to describe what a function, method, class, or module does. It is a string literal that occurs as the first statement in a function, method, class, or module definition. Docstrings are used to provide a convenient way of associating documentation with Python code. This documentation can then be accessed via the help system using the help() function or \_\_doc\_\_ attribute.

Key Points about Docstrings:

1. Syntax: A docstring is written using triple quotes (""" or '''). It can span multiple lines.
2. Location: It should be the first statement within a module, class, method, or function.
3. Purpose: Provides a clear explanation of the purpose of the code, its parameters, return values, and any other relevant information.

**Example:**

1. **Defining the Function**: The function add\_numbers takes two parameters a and b.
2. **Docstring**: The triple-quoted string right after the function definition is the docstring. It describes:
   * **What the function does**: "Adds two numbers and returns the result."
   * **Parameters**: Describes each parameter's type and role.
   * **Returns**: Describes what the function returns.
3. **Accessing the Docstring**:
   * Using help(add\_numbers), you can see the docstring, which helps understand what the function does without looking at the code.
   * The \_\_doc\_\_ attribute can also be used to access the docstring directly: add\_numbers.\_\_doc\_\_.

**Benefits of Using Docstrings:**

* **Improved Readability**: Helps others (and yourself) understand what the code does.
* **Easy Access**: Documentation is integrated within the code and can be accessed without external documentation tools.
* **Standardization**: Many IDEs and tools can automatically extract and display docstrings, providing a consistent way to document code.

Docstrings are an important part of writing clean, maintainable, and professional Python code, as they help provide context and clarity for how a piece of code is supposed to work.

**Q. What are unit tests in python?**

**Unit tests** in Python are a way to automatically verify that individual pieces of code, typically functions or methods, work as expected. The purpose of unit tests is to isolate each part of the program and show that the individual parts are correct. Unit tests help identify bugs early in the development process and ensure that code changes do not introduce new issues.

**Key Points about Unit Tests:**

1. **Purpose**: To test the functionality of specific sections of code, usually functions or methods, to ensure they produce the correct output for a given input.
2. **Isolation**: Unit tests should test only a small piece of functionality (a "unit") independently of other parts of the program.
3. **Automation**: Once written, unit tests can be run automatically whenever the code changes, making it easier to detect bugs early.
4. **Regression Testing**: They help catch bugs that are introduced by new changes, ensuring that previously working code still functions as expected.

**Writing Unit Tests in Python**

Python has a built-in module called unittest which provides a framework for writing and running tests

**Example:**

1. **Importing unittest**: The unittest module provides tools for constructing and running tests.
2. **Test Class**: TestMathOperations is a subclass of unittest.TestCase. This class contains test methods that test different aspects of the code.
3. **Test Method**: test\_add is a test method. It uses self.assertEqual to check if the output of the add function matches the expected result.
   * **self.assertEqual(add(2, 3), 5)**: Checks if calling add(2, 3) returns 5.
   * **self.assertEqual(add(-1, 1), 0)**: Checks if calling add(-1, 1) returns 0.
   * **self.assertEqual(add(0, 0), 0)**: Checks if calling add(0, 0) returns 0.
4. **Running the Test**: The unittest.main() function is called to run the test when the script is executed.

**Benefits of Unit Tests:**

* **Error Detection**: Quickly identify bugs and errors in the code.
* **Code Refactoring**: Ensure that changes or optimizations in the code do not break existing functionality.
* **Documentation**: Serve as a form of documentation by demonstrating how the code is expected to work.
* **Continuous Integration**: Integration with CI/CD pipelines to automate testing and ensure code quality.

Unit tests are a fundamental part of modern software development practices, promoting code reliability and maintainability by ensuring each part of the codebase works as intended.

**Q. what is break, continue and pass in python?**

In Python, break, continue, and pass are control flow statements that are used to modify the behavior of loops and conditionals. They help manage the flow of the program in different ways. Here's an explanation of each, along with examples:

**1. break Statement**

The break statement is used to exit a loop prematurely. When break is encountered, the loop is immediately terminated, and the program control moves to the next statement following the loop.

**2. continue Statement**

The continue statement is used to skip the rest of the code inside the loop for the current iteration only. The loop then proceeds to the next iteration.

**3. pass Statement**

The pass statement is a null operation; it doesn't do anything when executed. It's often used as a placeholder for future code. It can be used where a statement is syntactically required, but no action is needed.

**Q. what is the use of self in python?**

In Python, self is a reference to the current instance of the class. It is used to access variables that belong to the class. In object-oriented programming (OOP), self represents the instance of the object itself, and it is used to differentiate between instance attributes and local variables. It also allows us to call other methods from within the same class.

Key Points about self:

1. Refers to the Instance: self refers to the specific instance of the class on which a method is called. Each time an instance method is called, self is passed automatically by Python as the first argument.
2. Access Instance Variables: It is used to access variables and methods associated with the object. Without self, Python would not be able to differentiate between instance variables and local variables.
3. Explicit First Argument: In Python, self must be explicitly declared as the first parameter of instance methods. It is not a keyword in Python, but it is a strong convention to use self as the name for the first parameter of an instance method.

Example:

1. Class Definition: A class named Dog is defined with an \_\_init\_\_ method and a bark method.
2. Using self in \_\_init\_\_:
   * self.name = name assigns the name parameter to the name attribute of the instance.
   * self.breed = breed assigns the breed parameter to the breed attribute of the instance.
3. Method bark: The method uses self.name to access the name attribute of the specific instance that calls this method.
4. Creating an Instance: dog1 = Dog("Buddy", "Golden Retriever") creates an instance of Dog. Here, "Buddy" and "Golden Retriever" are passed to the \_\_init\_\_ method, initializing self.name and self.breed.
5. Accessing Attributes and Methods:
   * dog1.name accesses the name attribute of the dog1 instance.
   * dog1.bark() calls the bark method for the dog1 instance, which uses self to refer to dog1.

Why Use self?

* Instance Differentiation: Each instance of a class has its own separate set of attributes. self ensures that each instance method operates on its own data.
* Method Calls: When calling methods from within another method in the same class, self is used to reference the method (e.g., self.another\_method()).
* Clarity and Convention: Although self could technically be named anything, using self is a Python convention that enhances readability and understanding of the code.

In summary, self is essential for writing object-oriented Python code, as it provides access to the attributes and methods of the current instance, ensuring that operations are performed on the correct instance data.

**Q. what are global, protected and private attributes in python?**

* Global Attributes: Accessible from anywhere in the code. Not tied to classes or objects.
* Protected Attributes: Accessible within the class and its subclasses. Indicated by a single underscore prefix (e.g., \_attribute).
* Private Attributes: Intended to be accessed only within the defining class. Indicated by a double underscore prefix (e.g., \_\_attribute). Access outside the class requires name mangling, which is discouraged.

**Q. What are modules and packages in python?**

Modules and packages are fundamental concepts in Python that help organize and manage code. They provide a way to structure Python programs by splitting them into smaller, manageable, and reusable components. Here's an explanation of both:

**Modules**

A **module** is a single file (with a .py extension) that contains Python code. This code can define functions, classes, variables, and runnable code. Modules allow you to logically organize your Python code and reuse it across different programs by importing the module.

**Key Points About Modules:**

1. **File-Based**: A module is simply a Python file, typically containing related functions, classes, or variables. For example, a file named math\_utils.py is a module that might contain mathematical functions.
2. **Importing Modules**: Modules can be imported into other Python scripts or modules using the import statement. Once imported, you can use the functions, classes, or variables defined in that module.
3. **Namespace Management**: When you import a module, a new namespace is created. This helps avoid name conflicts and keeps the code organized.

**Packages**

A **package** is a collection of related modules grouped together within a directory. A package typically contains a special file named \_\_init\_\_.py, which indicates to Python that the directory should be treated as a package. Packages are used to organize modules into a directory hierarchy.

**Key Points About Packages:**

1. **Directory-Based**: A package is a directory that contains multiple modules and a special \_\_init\_\_.py file. The \_\_init\_\_.py file can be empty or contain initialization code for the package.
2. **Sub-Packages**: Packages can contain sub-packages, which are directories within the main package directory, allowing for a hierarchical organization of modules.
3. **Organizing Code**: Packages allow for more structured and scalable organization of code. This is especially useful for larger projects where grouping related functionality into packages makes the code easier to manage.

**Q. what are list and tuples? what are key differences between them?**

| **Feature** | **Lists** | **Tuples** |
| --- | --- | --- |
| Syntax | Defined using square brackets [] | Defined using parentheses () |
| Mutability | Mutable (elements can be modified) | Immutable (elements cannot be modified) |
| Methods | Many methods (e.g., append(), remove(), sort()) | Fewer methods (e.g., count(), index()) |
| Performance | Slightly slower due to mutability | Slightly faster due to immutability |
| Use Cases | Use when collection needs to be modified | Use for fixed collections or to ensure data integrity |
| Memory Usage | Generally uses more memory | Generally uses less memory |

**Q. What is interpreted language and dynamically typed language? Write 5 difference between them?**

**Interpreted Language** and **Dynamically Typed Language** are terms that describe different aspects of a programming language's execution and type system. Here's an explanation of each and the differences between them:

**Interpreted Language**

An **interpreted language** is a programming language where most of the instructions are executed directly by an interpreter, rather than being compiled into machine code. The interpreter reads and executes the code line-by-line or statement-by-statement.

**Characteristics**:

* **Execution**: The code is executed directly by an interpreter, which translates the code into machine instructions at runtime.
* **Error Detection**: Errors are often detected at runtime, which means errors in code are caught during execution.
* **Development Cycle**: Typically allows for rapid development and testing since changes can be tested immediately without needing a separate compilation step.
* **Performance**: Generally slower compared to compiled languages, as code execution involves an additional layer (the interpreter).
* **Portability**: Code is generally more portable since the interpreter can be run on different platforms without needing recompilation.

**Example**: Python, Ruby, and JavaScript are examples of interpreted languages.

**Dynamically Typed Language**

A **dynamically typed language** is a programming language in which variable types are checked at runtime rather than at compile time. This means that the type of a variable is determined during program execution, not in advance.

**Characteristics**:

* **Type Checking**: Types are checked dynamically at runtime. This means that type errors are detected only when the specific code is executed.
* **Flexibility**: Variables can be reassigned to different types at runtime. For example, a variable initially holding an integer can later hold a string.
* **Error Detection**: Type-related errors are only caught during execution, not during the coding phase.
* **Syntax**: Code may be less verbose because you don’t have to explicitly declare types.
* **Type Safety**: Offers less strict type safety, which can lead to runtime errors if types are not handled properly.

**Example**: Python, JavaScript, and PHP are examples of dynamically typed languages.

**Key Differences**

| **Aspect** | **Interpreted Language** | **Dynamically Typed Language** |
| --- | --- | --- |
| **Execution** | Code is executed by an interpreter at runtime. | Type information is determined at runtime. |
| **Error Detection** | Errors are caught at runtime. | Type errors are also caught at runtime. |
| **Compilation** | No separate compilation step; code is executed directly. | No compile-time type checking; types are checked at runtime. |
| **Performance** | Generally slower due to interpretation overhead. | Not inherently related to performance; depends on the language and its runtime. |
| **Type System** | Type checking is not related to being interpreted. | Variables' types are determined and checked at runtime. |

**Q. how is memory managed in python?**

Memory management in Python involves several components and mechanisms to ensure efficient allocation, usage, and cleanup of memory. Here’s a breakdown of how Python handles memory management:

**1. Automatic Memory Management**

Python uses automatic memory management, which means that developers do not need to manually allocate or deallocate memory. This is largely handled through two main mechanisms:

**a. Reference Counting**

* **Concept**: Every object in Python has a reference count that tracks how many references point to that object.
* **Mechanism**: When a new reference to an object is created, its reference count is incremented. When a reference is deleted, the count is decremented.
* **Deallocation**: When the reference count drops to zero, meaning no references to the object exist, Python automatically deallocates the memory used by the object.

**b. Garbage Collection**

* **Concept**: In addition to reference counting, Python employs garbage collection to handle cyclic references that reference counting alone cannot clean up.
* **Mechanism**: Python's garbage collector detects and collects groups of objects that reference each other, but are no longer accessible from the root of the program.
* **Implementation**: Python uses a cyclic garbage collector that operates in the background and periodically scans for unreachable objects.

**2. Memory Allocation**

* **Heap Memory**: Python objects are allocated on the heap. The heap is a region of memory used for dynamic allocation where objects are managed during the lifetime of the program.
* **Memory Pools**: Python uses a system of memory pools to manage small object allocations efficiently. This is done to reduce the overhead of frequent memory allocations and deallocations.

**Example**:

* Small integers and strings are managed in a pool to optimize memory usage and performance. For example, integers from -5 to 256 are preallocated and reused.

**3. Object Interning**

* **Concept**: Python optimizes memory usage by reusing objects for commonly used immutable objects.
* **Mechanism**: Small integers and certain strings are interned, meaning that identical immutable objects are stored in a shared pool and reused rather than creating new objects.

**4. Memory Management Tools**

* **sys.getsizeof()**: Returns the size of an object in bytes.
* **gc Module**: Provides functions to interact with the garbage collector, such as gc.collect() to manually trigger garbage collection.

Python's memory management system combines these techniques to provide a robust, efficient, and developer-friendly approach to handling memory.

**Q. What is polymorphism? Explain it with an example.**

**Polymorphism** is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It is the ability for different classes to be used interchangeably, even if they implement different functionalities. Polymorphism can be achieved through method overriding (inheritance) and method overloading (although Python primarily supports method overriding).

**Types of Polymorphism**

1. **Method Overriding**: When a subclass provides a specific implementation of a method that is already defined in its superclass. This allows a subclass to provide a behavior that is different from the superclass.
2. **Method Overloading**: Python does not support method overloading in the traditional sense (multiple methods with the same name but different parameters). Instead, method overloading is often simulated through default arguments or variable-length argument lists.

**Example of Polymorphism**

Consider a scenario where we have a base class called Animal and two subclasses Dog and Cat. Both subclasses will override the make\_sound method of the Animal class to provide their own specific implementations.

**Explanation:**

1. **Base Class**: Animal is the base class with a method make\_sound that is intended to be overridden by subclasses. The raise NotImplementedError ensures that if make\_sound is not overridden, an error will be raised, enforcing the requirement for subclass implementation.
2. **Subclasses**: Dog and Cat are subclasses of Animal. Each subclass overrides the make\_sound method to provide a specific implementation.
3. **Function**: The function print\_animal\_sound accepts an Animal object (or any subclass thereof) and calls the make\_sound method. Due to polymorphism, this method call will execute the version of make\_sound defined in the actual class of the object passed.
4. **Polymorphic Behavior**: When print\_animal\_sound(dog) is called, the make\_sound method of Dog is invoked. Similarly, when print\_animal\_sound(cat) is called, the make\_sound method of Cat is invoked. Despite print\_animal\_sound only knowing that it is dealing with an Animal, the correct make\_sound method is called based on the actual object type.

**Summary**

Polymorphism allows objects of different classes to be used interchangeably based on a common interface or method. In the example above, polymorphism lets us call the same method make\_sound on different types of animals (Dog and Cat), and each animal responds in its own way. This leads to more flexible and reusable code, where functions and methods can operate on objects of different classes without knowing their exact types.