**Python**

**1) Iterators:**

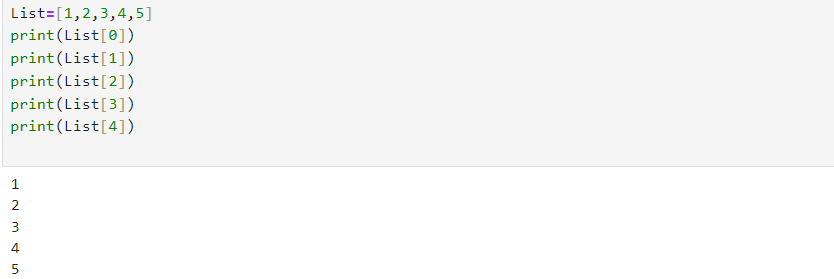
In Python, an iterator is an object that allows you to go through a collection of items one by one without needing to use loops or indexes directly.

**Methods for Accessing elements in List:**

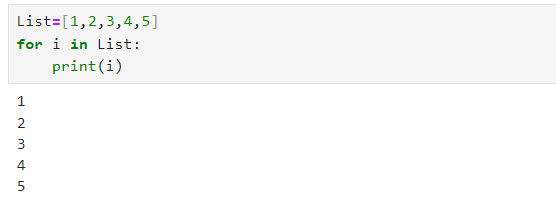
**Using print statements**:



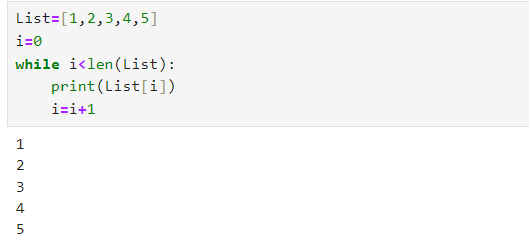
**Using indexing:**



**Using For Loop**

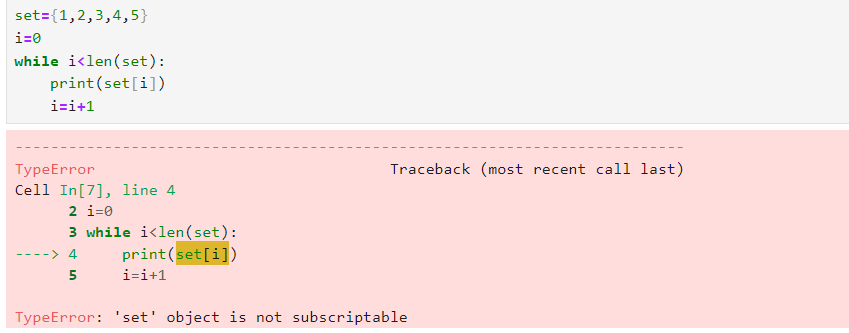


**Using While Loop**:



**Using while Loop for set**:

We



We can only use for loop to iterate through other datatypes like list, tuple, set etc. However while loops cannot access elements in sets. To address this limitation we can utilize ***iterators.***

**Why use iterator?**

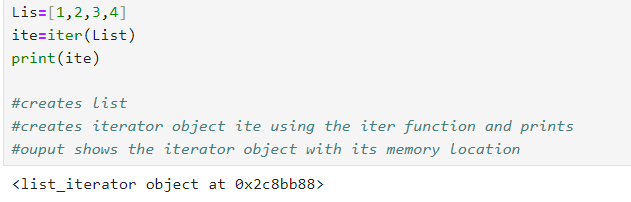
Works with any collection: Iterators work with lists, sets, tuples and more.

Save memory: Iterators only process one item at a time, which is more efficient.

**Creating and using an iterator:**

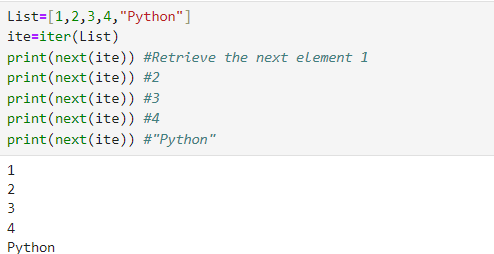
**Step 1: Create an Iterator**

Use the ‘iter()’ function which converts iterable objects into iterators, allowing us to access their elements one at a time

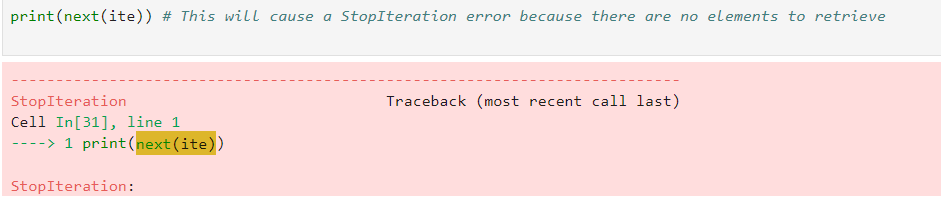


**Step 2: Get items with** ‘next()’

Using the ‘next()’ function to get items one at a time

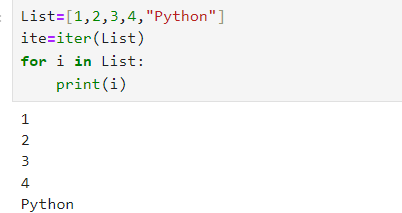


If you try to get more items than the list, you will get an error

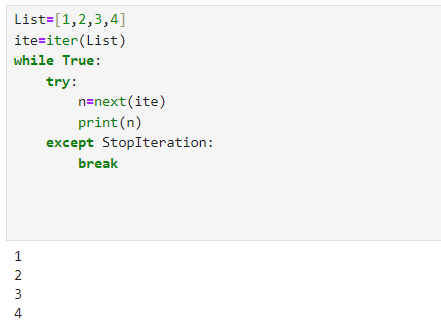


**Using iterator in For Loop:**

You can use iterator in for loop



**Using iterator in while loop**:



**Defining custom iterators**:

Now let’s see how we can define our own custom iterators in Python. To create a custom iterator, we need to define a class that implements the iterator protocol:

1. ‘ iter()’ method returns the next element in the iteration object itself and is called when the iterator is initialized.
2. ‘next()’ method returns the next element in the iteration sequence and is called each tie we request the next element

Create a custom iterator class named ‘Ten’ that generates number from 1 to 10



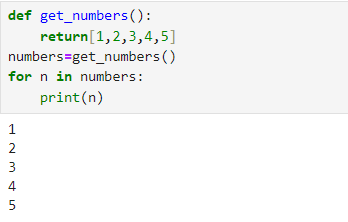
**Key points:**

* Iterator: Object providing access to items one at a time
* iter(): Converts a collection into an iterator
* next():Retrieves the next item from the iterator
* StopIteration: Exception raised when no more items available
* Customized iterator: Allows us to iterate over data in a customized manner tailored to our needs

**2) Generators:**

In Python, generators is a kind of function that returns an object called generator object which can return a series of values rather than a single value

**Normal Function:**

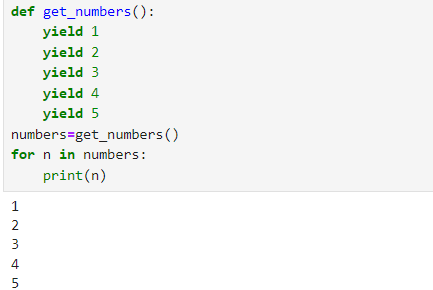


**Explanation:**

* The function ‘get\_numbers’ returns a list containing all numbers at once
* All numbers are stored in memory at the same time

**Generator function:**

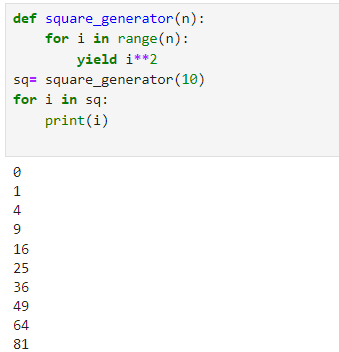
Creating a generator function is similar to creating a function. We use he keyword def and a yield statement to create a generator

****

**Explanation:**

* The function ‘get\_numbers’ yields each number one at a time.
* Only one number is in memory at a time, making it more memory efficient

**Example of generator function that yields square numbers:**

****

In this example, ‘square\_generator’ is a generator function that yields the square of each number from 0 to n-1. When we iterate over the generator object ‘sq’, it generates and yield one square number at a time.

**Generator Expressions:**

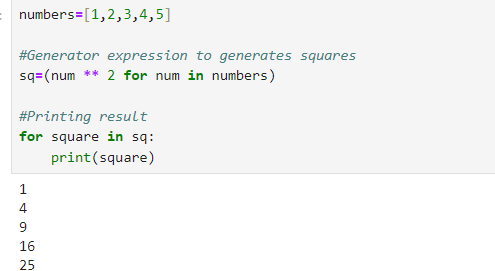
Generator expressions are a compact way of creating iterators. They are similar to list comprehension, but instead of creating a list they generate items one at a time.

**Simple Generator Expression**:

A simple generator expression goes through each item in an iterable and applies an expression to it

**Example**

We have a list of numbers ,and we want to generate a sequence of their squares:

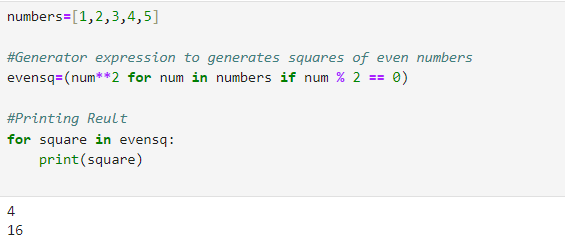


**Conditional Generator Expression**:

A conditional generator expression adds a condition to filter items before applying the expression. It generates based on whether they meet the condition

**Example**:

Generate a sequence of squares for only the even numbers in the list:

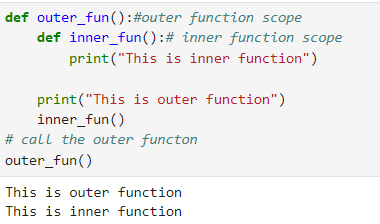


**3) Closure:**

To understand the concept of closures, we need to understand some basic concepts

**Nested Functions:**

A nested function is simply a function defined inside another function

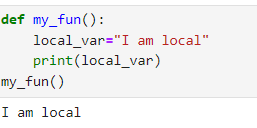


**Explanation:**

* Inner\_fun is defined inside the outer\_fun
* When outer\_fun is called,it prints *“This is outer function”* and then calls inner\_fun.
* Inner\_fun prints *“This is inner function”* when called

**Local Variables:**

A variable defined inside a function and only accessible within that function



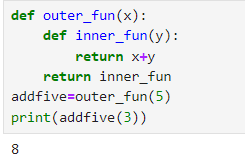
**Non-Local Variables:**

A variable that is defined outside of a function, but it is used inside that function



**Closure:**

A closure is a function that remembers the values from its enclosing lexical scope even when the scope has finished executing



**Explanation**:

* In the example, inner\_fun is defined inside outer\_fun.
* Whenever outer\_fun(5) is called, it returns inner\_fun with x set to 5, creating a closure.
* add\_five now holds the closure, allowing it to remember the value o x( which is 5) even after outer\_fun has finished executing.
* When add\_five(3) is called, it adds 5(captured from the closure) and 3, resulting in 8.

**Conditions for Closures in Python**:

1. **Nested Functions**:

Closure require nested functions, meaning a function defined inside another function

1. **Access to Outer variable**:

The nested function refer to variables defined in its outer scope, such as variables from the enclosing function.

1. **Return of Nested function**:

The outer function must return the nested function for the closure to be created and maintained.

**4) Decorators**:

**Definition**: A Python decorator is a function that accepts another function as an argument and returns a new function.

**Purpose**: Decorators allow you to enhance or modify the behaviour of the original function without altering its actual code.

**Functionality**: By "wrapping" the original function, decorators enable you to run additional code before or after the original function is executed.

**Non-Permanent Changes**: They adjust the function's behaviour temporarily, without making permanent changes to the function itself.

**Usage**: Decorators are applied using the @decorator\_name syntax placed above the function definition

**Why Do We Need Python Decorators?**

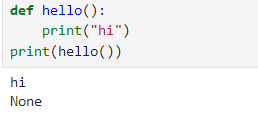
We use decorators when we want to add the same behaviour to multiple functions or change the inputs of existing functions. They help us apply these changes in one place, making the code easier to manage and read.

**To create Python decorators, follow these steps**:

1. Define an outer function that accepts another function as its argument.
2. Inside this outer function, create an inner function that includes the additional features you need.
3. Use the @ symbol followed by the name of the decorator function to apply it to the target function.

**Basic Function**

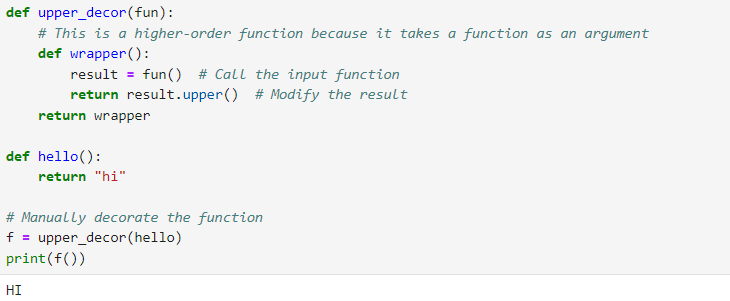
First, let's define a basic function:



This will print "hi" and then None because the hello function itself returns None.

**Adding Functionality with Decorators**

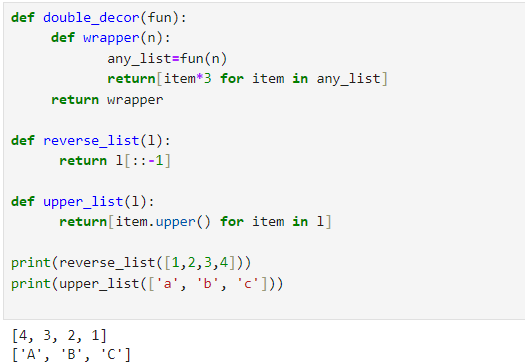
Now, we'll use a decorator to add additional functionality to this function



**Explanation:**

* upper\_decor(fun) defines a function wrapper that converts fun's output to uppercase and returns wrapper.
* hello() simply returns "hi".
* f = upper\_decor(hello) applies upper\_decor to hello, creating a new function f that will return the uppercase version of hello's output.
* print(f()) calls the decorated function f, which outputs "HI" by converting "hi" to uppercase

**Create a decorator function that takes another function as an argument**



**Explanation**:

* The double\_decor function defines a decorator that takes a function as input.
* Inside the decorator, there's a wrapper function (wrapper) that modifies the behaviour of the original function.
* The wrapper function triples each element of the list returned by the original function.
* reverse\_list: Reverses the input list.
* upper\_list: Converts input strings to uppercase

**5) Property:**

* The property() function in Python is utilized to establish class properties, primarily used within Python classes to define attributes with custom getter, setter, and deleter methods.
* It's a fundamental tool in Python's Object-oriented paradigm, accepting getter, setter, and deleter methods as arguments to define properties within classes.
* Python natively incorporates the property() function for declaring class properties.
* Alongside property(), Python offers the @property decorator, enhancing the ease of implementing getter and setter methods in Object-Oriented Programming

**Python getters and setters:**

In object-oriented programming, getters and setters are used to keep data safe and private within a class. Getters help to fetch the value of private attributes, while setters allow changing or updating these values. This way, we prevent other classes from accidentally changing our data.

**Python property**:

**Functionality:** In Python classes, the property() function defines properties, which are managed by the Properties class to handle class attributes.

**Usage**: It's a built-in function that creates and returns a property object, serving as an interface for instance attributes in Python

**Syntax:**

property(fget, fset, fdel, doc)

**Parameters:**

fget (optional): Function for retrieving an attribute value. Default is None.

fset (optional): Function for setting an attribute value. Default is None.

fdel (optional): Function for deleting an attribute value. Default is None.

doc (optional): String containing documentation (docstring) for the attribute. Default is None.

**Return Value**:

Returns property attributes based on the specified getter, setter, and deleter functions.

If no arguments are provided, property() returns a base property attribute with no getter, setter, or deleter.

If doc is not provided, property() uses the docstring from the getter function.

**How python property works?**

Consider a basic syntax example where we create a new class attribute called 'attr' and define its three parameters as properties:

attr = property(getter, setter, deleter)

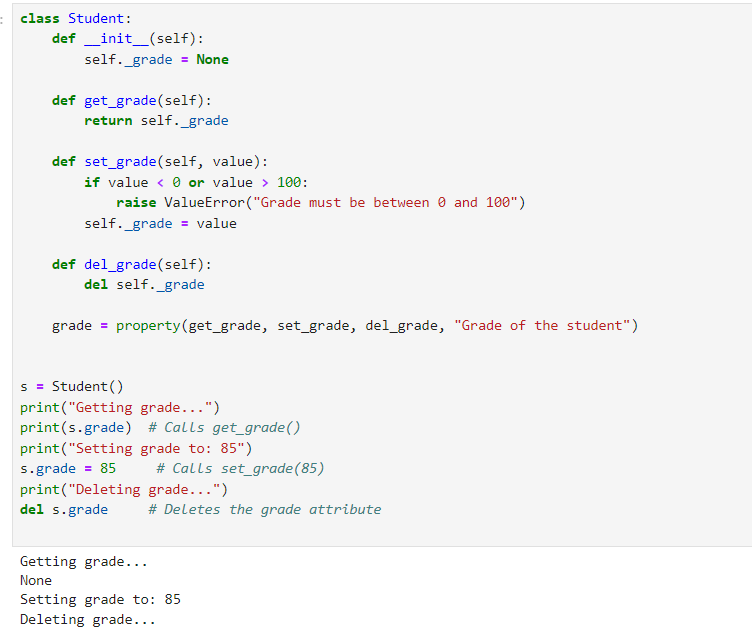
Now, if 'obj' represents an object instance of a class, accessing obj.attr in Python triggers the getter function.

When assigning a value to obj.attr like obj.attr = value, Python executes the setter function and passes the value as an argument.

Similarly, when del obj.attr is used, Python invokes the deleter method.

The argument provided as doc is utilized by Python as the attribute's docstring.

Example:



In this example:

* The Student class has a single property attribute called grade.
* It defines three methods: get\_grade, set\_grade, and del\_grade.
* get\_grade retrieves the grade of the student, set\_grade sets the grade within the valid range of 0 to 100, and del\_grade deletes the grade attribute.
* The grade property is created using the property() function.
* When accessing s.grade, it internally calls the getter method get\_grade().
* Assigning a new value to s.grade invokes the setter method set\_grade() to ensure the grade is within the valid range.
* Attempting to delete s.grade invokes the deleter method del\_grade().

**Regular Expressions:**

**Why regular expressions are used?**

Consider another scenario where regular expressions can be useful:

Suppose you have a large text document containing various email addresses, and you need to extract all the email addresses from it

*text = """*

*Lorem ipsum dolor sit amet, consectetur adipiscing elit.*

*Email addresses: john.doe@example.com, jane.doe@example.org,*

*johndoe1234@email.co.uk, test.email@email.domain*

*"""*

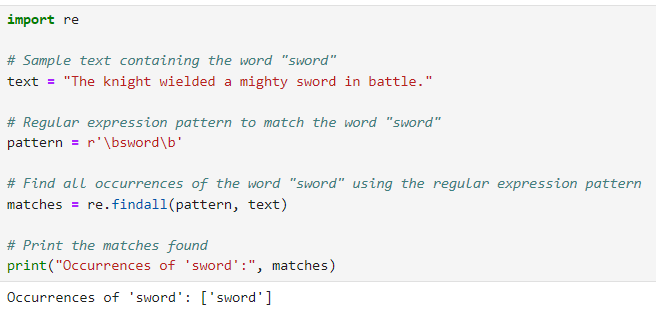
Regular Expressions can be used in this case to recognize the patterns and extract the required information easily.

**What are regular expressions?**

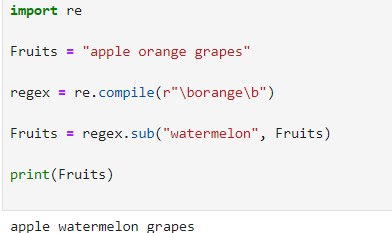
Regular Expressions are powerful tools used to identify specific patterns within text strings. They enable tasks such as validating data, searching for patterns, and performing operations like find-and-replace and formatting within text.

**Basic regular expressions:**

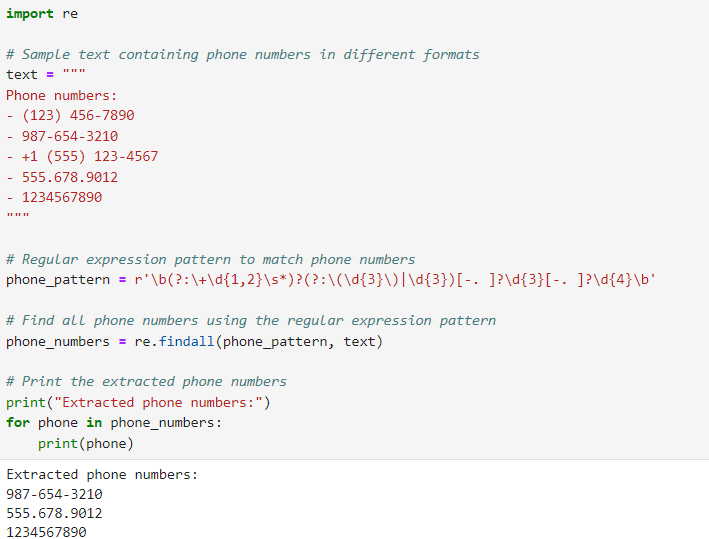
Finding a word in string



Replacing a string:



Consider an example where we have a text document containing various phone numbers in different formats, and we want to extract all the phone numbers from it using regular expressions.



**Explanation**:

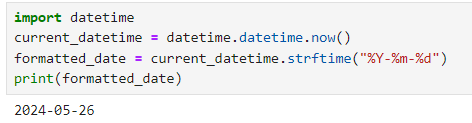
* We have a sample text document containing phone numbers in different formats.
* We define a regular expression pattern phone\_pattern to match phone numbers. This pattern accounts for various formats including those with area codes, country codes, and different separators like hyphens, dots, and spaces.
* We use the re.findall() function to find all occurrences of phone numbers in the text that match the specified pattern.
* Finally, we print out all the extracted phone numbers.

**Python Date and Time:**

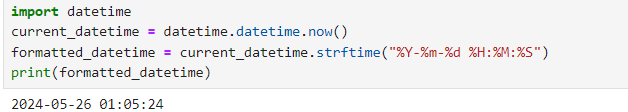
**Python datetime Module**

This module provides classes for manipulating dates and times in both simple and complex ways. It allows you to work with dates, times, and combined date-time representations. You can create datetime objects to represent specific dates and times, perform arithmetic operations on them, and format them as strings for display or storage.

Formatting a datetime object as a string:



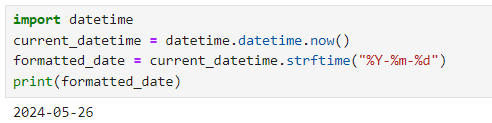
Formatting a datetime object with time:



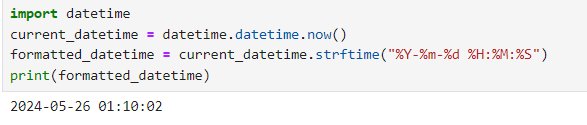
**Python datetime.strftime()**

The strftime() method is used to convert a datetime object into a string representation based on a specified format. You can specify the format using various directives, which are placeholders for different components of the datetime object, such as year, month, day, hour, minute, and second.

Formatting a datetime object as a string:



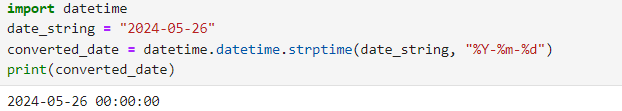
Formatting a datetime object with time:



**Python datetime.strptime()**

The strptime() method is the inverse of strftime(). It converts a string representation of a date and time into a datetime object. You need to specify the format of the input string using directives that match the components of the string representation.

Converting a string to a datetime object:



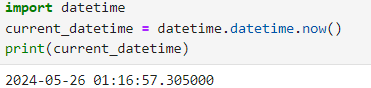
Converting a string with time to a datetime object:



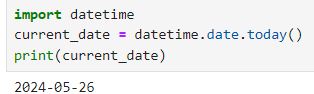
**Current date & time**

This refers to obtaining the current system date and time. You can use the datetime.now() function to get the current date and time as a datetime object. Alternatively, you can use date.today() to get just the current date without the time component.

Getting the current date and time:



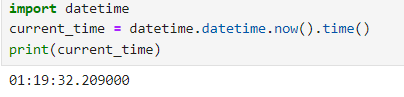
Getting the current date only:



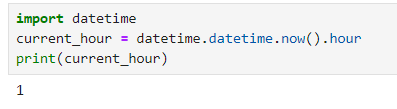
**Get current time**

This involves extracting the current time from the system clock. You can use datetime.now().time() to get the current time as a time object, which represents only the time portion without the date. Alternatively, you can access specific components like the hour using datetime.now().hour.

Getting the current time:



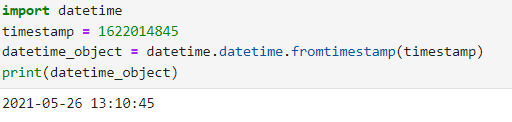
Getting the current hour:



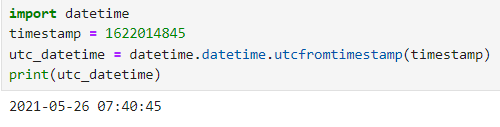
**Timestamp to datetime**

Timestamps are often used to represent a point in time as the number of seconds since a particular reference point (epoch). The fromtimestamp() method converts a Unix timestamp into a datetime object. You can also use utcfromtimestamp() to obtain a UTC datetime object from the timestamp.

Converting timestamp to datetime:



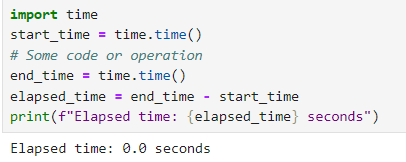
Converting timestamp to UTC datetime:



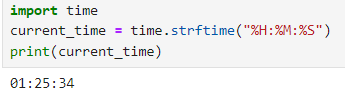
**Python time Module**

The time module provides various functions for working with time-related operations. It includes functions to obtain the current time, measure time intervals, and format time strings. You can use time.time() to get the current time in seconds since the epoch.

Getting the time elapsed using time module:



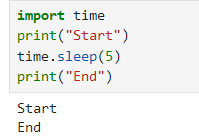
Formatting time using time module:



**Python time sleep()**

The sleep() function suspends the execution of the current thread for a specified number of seconds. It's useful for delaying the execution of code or creating time intervals between operations. You specify the duration of the pause in seconds as an argument to the function.

Pausing execution for 5 seconds:



Pausing execution for 1 minute:

