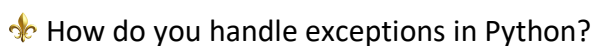




Here is an example of creating a tuple and a list:

Here is an example of trying to modify an element in a tuple and a list:

Keep in mind that tuples have slightly better performance than lists when working with large amounts of data.



In Python, exceptions can be handled using a try-except block. The try block contains the code that may raise an exception, and the except block contains the code that will be executed if an exception is raised.

Here is an example of handling a ZeroDivisionError exception:

```
try:
    x = 5 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

In this example, the code in the try block will raise a ZeroDivisionError exception when attempting to divide 5 by 0.

The code in the except block will be executed, printing "Cannot divide by zero" to the console.

"It's also possible to handle multiple exceptions in the same try-except block, using a tuple of exception types after the except keyword:"

```
try:
    x = int("abc")
    y = 5 / 0
except (ValueError, ZeroDivisionError):
    print("Invalid input or division by zero")
```

"It's also possible to handle the exception and assign it to a variable with the as keyword:"

```
try:
    x = int("abc")
except ValueError as e:
    print(f"Invalid input: {e}")
```

You can also use the finally block, which is executed after the try and except blocks, regardless of whether an exception was raised or not:

```
try:
    x = 5 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("This code will always be executed")
```

In the example above, the code in the finally block will be executed even if an exception was raised.

"In Python, decorators are a way to modify the behavior of a function or class, by wrapping the function or class with another function. Decorators are implemented using the @ symbol followed by the decorator function."

import time

```
@timer
def long_running_function():
    time.sleep(5)
    print('Function completed')
```

The wrapper function calculates the execution time of the original function, prints it and returns the result of the original function.

You can also pass arguments to decorator function:

```
import time
```

```

def timer(num):
    def actual_decorator(func):
        def wrapper(*args, **kwargs):
            start = time.time()
            result = func(*args, **kwargs)
            end = time.time()
            print(f'Executed in {end-start} seconds. {num} times')
            return result
        return wrapper
    return actual_decorator

@timer(num=3)
def long_running_function():
    time.sleep(5)
    print('Function completed')

```

In this case, the timer function takes an argument num, which is passed to the actual decorator function, actual\_decorator, that wraps the original function.

Decorators are a powerful tool in Python, they can be used to add functionality to functions, methods, and classes, such as logging, caching, authentication, etc, without modifying the original code, making the code more modular and easier to maintain.

\*\*\*

"Decorators in Python add some feature or functionality to an existing function without altering it."

Let's say we have the following simple function that takes two numbers as parameters and divides them.

```

def divide(first, second):
    print ("The result is:", first/second)

```

What will happen if we pass the number 4 first, and 16 after? The answer will be 0.25. But we don't want it to happen. We want a scenario where if we see that first < second, we swap the numbers and divide them. But we aren't allowed to change the function.

Let's create a decorator that will take the function as a parameter. This decorator will add the swipe functionality to our function.

```

def swipe_decorator(func):
    def swipe(first, second):
        if first < second:
            first, second = second, first

```

```
return swipe
```

We have passed the function as a parameter to the decorator. The decorator "swiped our values" and returned the function with swiped values. After that, we invoked the returned function to generate the output as expected.

```
def swiipe_decorator(func):
    def swiipe(first, second):
        if first < second:
            first, second = second, first
        return func(first, second)
```

```
return swipe
```

## 🌟 How do you implement inheritance in Python?

In Python, inheritance is implemented by defining a new class that inherits from an existing class. The new class is called the child class or derived class, and the existing class is called the parent class or base class.

Here is an example of implementing inheritance in Python:

```
class Parent:
    def __init__(self, name):
        self.name = name

    def print_name(self):
        print(self.name)
```

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

In this example, the Child class inherits from the Parent class. The Child class has access to all the attributes and methods defined in the Parent class, including the `__init__` and `print_name` methods.

The Child class can also define its own attributes and methods, and override the methods of the parent class:

```
class Parent:
    def __init__(self, name):
        self.name = name

    def print_name(self):
        print(self.name)

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def print_age(self):
        print(self.age)

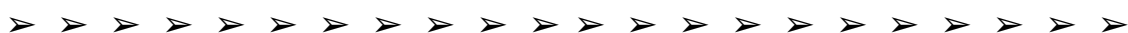
    def print_name(self):
        print(f'{self.name} is {self.age} years old')
```

In this example, the Child class has defined a new attribute `age` and new method `print_age`, it also overrides the `print_name` method of the parent class.

You can also use the `isinstance()` function to check if an object is an instance of a particular class or its subclasses.

```
c = Child("John", 25)
print(isinstance(c, Child)) # True
print(isinstance(c, Parent)) # True
```

Inheritance is a fundamental concept in object-oriented programming, it allows for code reusability and organization. It allows you to create a new class that inherits the properties and methods of an existing class, and then add or modify them as needed, making it easier to maintain and extend the codebase.





In Python, a shallow copy and a deep copy are two different ways to copy the contents of an object.

A shallow copy is a copy of an object that contains references to the original object's elements. In other words, the new object is a copy of the original object, but it shares the same memory address of the elements.

Here is an example of creating a shallow copy of a list:

```
original_list = [[1, 2], [3, 4]]
shallow_copy = original_list.copy()
```

In this example, `shallow_copy` is a copy of `original_list`, but the elements of the list `[1, 2]` and `[3, 4]` are still shared between the two lists.

On the other hand, a deep copy creates a new object that has its own memory address, and all the elements in the object are also copied.

Here is an example of creating a deep copy of a list:

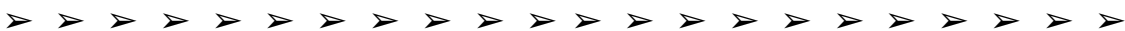
```
import copy
original_list = [[1, 2], [3, 4]]
deep_copy = copy.deepcopy(original_list)
```

In this example, `deep_copy` is a completely independent copy of `original_list`, it has its own memory address and the elements `[1, 2]` and `[3, 4]` are also independent copies.

It's important to note that when you copy an object, the default behavior is to create a shallow copy, unless you explicitly use the `deepcopy()` function or another way to make a deep copy.

Also, when you copy a basic data type (int, float, string, etc) the copy is always a deep copy, there is no such thing as a shallow copy for basic data types.

It's important to understand the difference between a deep copy and a shallow copy, because when working with complex data structures, a shallow copy may not be sufficient and can lead to unexpected behavior if the original object is modified.



In Python, a lambda function is a small, anonymous function that can have any number of arguments but can only have one expression. Lambda functions are also known as "anonymous functions" or "throwaway functions" because they are not bound to a name and are used for a short period of time.

```
add = lambda x, y: x + y
print(add(3, 4)) # prints 7
```

Lambda functions can also be used as arguments to other functions, such as `map()`, `filter()` and `reduce()`.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x**2, numbers)

print(list(squared_numbers)) # prints [1, 4, 9, 16, 25]
```

**▶ ▶**

In Python, a class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

```
class MyClass:
    pass
```

```
class MyClass:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def print_name(self):
        print(self.name)
```



```
my_object = MyClass("John", 25)
my_object.print_name() # prints "John"
```

In this example, the class `MyClass` is defined with an `__init__` method and a `print_name` method. The `__init__` method is a special method that is called when an object of the class is created, it is used to initialize the attributes of the class. The `print_name` method is a regular method that prints the value of the `name` attribute. The `self` parameter is a reference to the current instance of the class, and it must be included in the method definition, it's used to access the attributes of the class.

Once the class is defined, you can create objects (instances) of the class using the class name followed by parentheses. In this case, `my_object` is an instance of the class `MyClass`.

You can also define class variables and class methods, which are shared by all instances of the class.

```
class MyClass:
    x = [1,2,3]
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

🌟 Explain the use of generators in Python.

In Python, generators are a special type of function that allow you to iterate over a sequence of values without loading them into memory all at once. A generator function is defined like a normal function, but instead of using the return statement to return a value, it uses the yield statement.

Here is an example of a simple generator function that generates the squares of numbers up to a certain limit:

```
def squares(limit):
    for i in range(limit):
        yield i**2
```

```
for number in squares(5):
    print(number)
```

In this example, the squares generator function takes a single argument `limit`, and uses a `for` loop to iterate over the numbers from 0 to `limit-1`. The `yield` statement is used to generate the square of the current number in each iteration and return it to the caller.

Each time the generator's `next()` method is called, the generator resumes execution from where it was paused and continues until it encounters the next `yield` statement or until the function exits.

Here is an example of the usage of the generator in a more complex case, generating the Fibonacci numbers:

```
def fibonacci(limit):
    a, b = 0, 1
    for _ in range(limit):
        yield a
        a, b = b, a + b
```

```
for number in fibonacci(5):
    print(number)
```

Generators are useful when working with large data sets or when the data is generated on the fly. They allow you to iterate over the data without loading it into memory all at once, which can save a lot of memory and improve performance. They also provide a convenient way to implement iterators, which are objects that can be iterated upon, for example, in a for loop.



## 🌟 How do you perform file I/O operations in Python?

In Python, file I/O operations are performed using built-in functions such as `open()`, `write()`, `read()`, `readline()`, `readlines()`, and `close()`.

Here is an example of writing to a file:

```
# Open a file for writing
file = open("myfile.txt", "w")
```

```
# Write to the file
file.write("Hello World")
```

```
# Close the file
file.close()
```

In this example, the `open()` function is used to open a file named "myfile.txt" in write mode ("w"). The `write()` function is then used to write the string "Hello World" to the file. Finally, the `close()` function is used to close the file.

Here is an example of reading from a file:

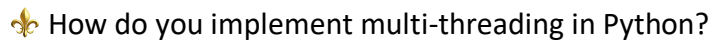


```
# mypackage/__init__.py

# mypackage/mymodule.py
def say_hello():
    print("Hello!")

# main.py
import mypackage.mymodule
mypackage.mymodule.say_hello()
```

Modules and packages allow you to organize and reuse your code, making it



Here is an example of creating two threads and running them simultaneously:

```
import threading

def first_function():
    print("First function started")
    for i in range(10):
        print("First function:", i)

def second_function():
    print("Second function started")
    for i in range(10):
        print("Second function:", i)

# Create the threads
first_thread = threading.Thread(target=first_function)
second_thread = threading.Thread(target=second_function)

# Start the threads
```

```
# Wait for the threads to finish
first_thread.join()
second_thread.join()
```

You can also use `ThreadPoolExecutor` class from `concurrent.futures` module to create a pool of worker threads to execute the function concurrently.

```
def some_function(arg):
    print(f"Argument: {arg}")

with ThreadPoolExecutor() as executor:
    executor.map(some_function, range(10))
```

In Python, the `with` statement is used to wrap the execution of a block of code with methods defined by a context manager. A context manager is an object that defines the methods `__enter__()` and `__exit__()`. The `__enter__()` method is run when the block of code is entered, and the `__exit__()` method is run when the block of code is exited, regardless of whether it finishes normally or with an exception.

```
with open("myfile.txt", "r") as file:
    contents = file.read()
    print(contents)
```

In this example, the `open()` function is used to open the file "myfile.txt" in read mode, and the file object is assigned to the variable `file`. The `with` statement is used to wrap the block of code that reads the contents of the file and prints them. When the block of code is exited, the `__exit__()` method of the file object is automatically called, which closes the file.

Here is another example of using the with statement to acquire and release a lock:

```
from threading import Lock
```

```
lock = Lock()
```

with lock:

```
print("Critical section 1")
```

```
print("Critical section 2")
```

In this example, the `Lock` class from the `threading` module is used to create a lock object. The `with` statement is used to wrap the block of code that represents the critical section of the program. When the block of code is entered, the `__enter__()` method of the lock object is automatically called, which acquires the lock. When the block of code is exited, the `__exit__()` method of the lock object is automatically called, which releases the lock.

The with statement is a convenient way to make sure that resources are properly managed and makes the code more readable and less prone to errors.

[illegible]

## 🌟 How do you implement polymorphism in Python?

In Python, polymorphism is the ability of objects of different types to be treated as objects of a common base type. This allows you to write code that can work with objects of different types, without having to know their specific types at runtime.

One way to implement polymorphism in Python is through the use of interfaces and abstract base classes (ABCs). An interface defines a set of methods that a class must implement, and an ABC is a class that defines an interface but cannot be instantiated.

Here is an example of using an ABC to define an interface for a shape:

```
from abc import ABC, abstractmethod
```

```
class Shape(ABC):
```

```
@abstractmethod
```

```
def area(self):
```

pass

```

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

```

```

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

```

```

shapes = [Rectangle(2, 3), Circle(4)]

```

```

for shape in shapes:
    print(shape.area())

```

In this example, the Shape class is defined as an ABC, and it defines a single method `area()` that must be implemented by its subclasses. The Rectangle and Circle classes inherit from the Shape class, and they implement the `area()` method in their own way. The shapes list is created to hold objects of different types that inherit from Shape class. The for loop iterates over the shapes list and calls the `area()` method on each object, without having to know the specific type of the object.

\*\*\*

In Python, polymorphism is the ability of an object to take on many forms. There are two main ways to implement polymorphism in Python: through inheritance and through function or method overloading.

Here is an example of polymorphism through inheritance:

```

class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

```

```

class Dog(Animal):
    def speak(self):
        return "Woof!"

```

```

class Cat(Animals):
    def speak(self):
        return "Meow!"

```

```
animals = [Dog("Fido"), Cat("Whiskers")]
```

```
for animal in animals:  
    print(animal.speak())
```

In this example, the Animal class is defined with a speak() method that does not have an implementation. The Dog and Cat classes inherit from the Animals class, and they both have their own implementation of the speak() method. This means that the speak() method can take on many forms depending on the specific object that is calling it.

Here is another example of polymorphism through function or method overloading:

```
def add(a, b):  
    return a + b
```

```
def add(a, b, c):  
    return a + b + c
```

```
print(add(1, 2)) # prints 3  
print(add(1, 2, 3)) # prints 6
```

In this example, the add() function is defined twice with different number of arguments, this is a form of polymorphism. Depending on the number of arguments passed, the add() function can take on many forms.

In Python, polymorphism allows you to write more general and flexible code that can work with multiple types of objects, and it is a key concept of Object-Oriented Programming (OOP).

What is the purpose of the "self" keyword in Python classes?

In Python, the self keyword is used to refer to the instance of a class. It is the first parameter of any method defined within a class, and it is used to access the attributes and methods of the class.

Here is an example of a class with a method that uses the self keyword:

```
class MyClass:  
    def __init__(self, name):  
        self.name = name  
  
    def print_name(self):  
        print(self.name)
```

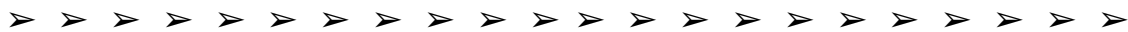
```
my_object = MyClass("John")  
my_object.print_name() # prints "John"
```



In this example, the MyClass class is defined with a constructor method `__init__` that initializes the name attribute of the class, and a method `print_name` that prints the value of the name attribute. The `self` keyword is used to refer to the instance of the class within the methods, it allows the methods to access the name attribute of the class.

When a method is called on an instance of the class, Python automatically passes the instance as the first argument to the method. In this case, when the `print_name()` method is called on `my_object`, Python automatically passes `my_object` as the first argument to the method, so `self` refers to `my_object`.

In Python, the `self` keyword is used to refer to the instance of the class, and it is a convention to use `self` as the name for the first parameter of methods in a class, but you can use any name you want, it's a common practice to use `self` but it's not mandatory.



## 🌟 How do you define and use global and local variables in Python?

In Python, a variable that is defined outside of a function or class is considered a global variable, and it can be accessed from anywhere in the code. A variable that is defined inside a function or class is considered a local variable, and it can only be accessed within the function or class.

Here is an example of using global and local variables:

```
# Global variable
```

$$x = 5$$

```
def my_function():
```

```
# Local variable
```

$$y = 10$$

```
print(x, y)
```

```
print(x) # prints 5
```

```
#print(y) # This will raise an error because y is a local variable
```

```
my function() # prints 5 10
```

In this example, the `x` variable is defined outside of any function or class and is considered a global variable. It can be accessed and used both inside and outside of the `my_function` function. The `y` variable is defined inside the `my_function` and is considered a local variable, it can only be accessed and used inside the `my_function`.

You can also use the `global` keyword to access a global variable from inside a function or class:

 $x = 5$

```
def my_function():
    global x
    x = 10
    print(x)
```

In this example, the global keyword is used inside the `my_function` to access the global variable `x`. Now the value of the global variable `x` is updated inside the function and it is accessible outside of the function.

🌟 Explain the use of the `map()`, `filter()`, and `reduce()` functions in Python.

## 🌟 How do you use the "assert" statement in Python?

In Python, the `assert` statement is used to check if a given condition is `True`, and if not, raise an `AssertionError` exception. It is often used for debugging and testing purposes to ensure that certain assumptions made in the code are valid.

Here is an example of using the assert statement to check if a variable has a certain value:

```
x = 5
assert x == 5, f"x is not 5, it is {x}"
```

In this example, the assert statement checks if the value of the x variable is equal to 5. If it is, the program continues to execute. If it is not, an AssertionError exception is raised with the message x is not 5, it is 5.

You can also use assert statement to check if a function returns a certain value

```
def my_function():
    return 5

assert my_function() == 5, "The function did not return 5"
```

In this example, the `assert` statement checks if the `my_function()` returns 5. If it does, the program continues to execute. If not, an `AssertionError` exception is raised with the message `The function did not return 5`.





ex.x = 5

```
print(ex.x) # output: 5
```

[illegible]

```
class Parent:
    def __init__(self, value):
        self.value = value

class Child(Parent):
    def __init__(self, value):
        super().__init__(value)
        self.new_value = value + 1
```

You can also use `super()` to call a method from a parent class if the method is overridden in the child class.

```
c = Child()
c.method()
# output: I am a child method
#         I am a parent method
```

It's important to note that the `super()` function should only be used when a class is subclassing another class, and it should be used inside a method of the subclass, not in the global scope.

while `a.extend([4, 5, 6])` would change the list to `[1, 2, 3, 4, 5, 6]`.

🌟 Explain the use of the "enumerate()" function in Python.

The "enumerate()" function in Python is a built-in function that takes an iterable object, such as a list, tuple, or string, and returns an iterator that produces tuples containing the index and the corresponding element of the iterable.

Here is an example of how to use the "enumerate()" function to iterate over a list:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

You can also specify a starting index for the enumeration by passing a second argument to the enumerate() function, like this:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits, 1):
    print(index, fruit)
```

The enumerate function is useful when you need to iterate over a list and also need the index of the current element. For example, you can use it to retrieve the index of a specific element in a list.

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    if fruit == "banana":
        print("The index of banana is", index)
```

This will output:

```
The index of banana is 1
```

You can also use the enumerate() function with a for loop and the zip() function to iterate over multiple lists in parallel.

```
fruits = ["apple", "banana", "cherry"]
prices = [1.2, 0.5, 2.3]
for index, (fruit, price) in enumerate(zip(fruits, prices)):
    print(index, fruit, price)
```

This will output:

```
0 apple 1.2
1 banana 0.5
2 cherry 2.3
```

Enumerate function is a useful and efficient tool for looping and indexing, as it allows you to keep track of the index of the current element, which can be very useful in some cases.

🌟 How do you use the "del" statement in Python?

🌟 Explain the difference between a "list comprehension" and a "generator expression" in Python.

In Python, both list comprehensions and generator expressions are used to create new iterable objects, such as lists or generators, from existing iterable objects, such as strings or lists. The main difference between the two is how they handle memory.

A list comprehension creates a new list in memory and stores all of its elements in that list, while a generator expression creates an iterator that generates the elements on the fly, without storing them all in memory at once. This means that a generator expression can be more memory-efficient when working with large data sets.

The syntax for a list comprehension is as follows: `[expression for item in iterable]`

The syntax for a generator expression is as follows: (expression for item in iterable)

In practical terms, when you use list comprehension, the list is created in memory, so you can use all of its feature like indexing and slicing, but for generator expression you can only iterate over it once.

➤ ➤

### 🌟 How do you use the "any()" and "all()" functions in Python?

🌟 Explain the difference between the "==" and "is" operators in Python.

In Python, the "==" operator is used to compare the values of two variables, while the "is" operator is used to compare the memory addresses of two variables.



```
x = [1, 2, 3]
y = [1, 2, 3]
print(x == y) # prints True
```

The "is" operator compares the memory addresses of two variables and returns True if they point to the same object in memory, and False otherwise. For example:

```
x = [1, 2, 3]
y = [1, 2, 3]
print(x is y) # prints False
```

```
x = [1, 2, 3]
y = [1, 2, 3]
print(x is y) # prints False
```

It is worth noting that for basic types (int, float, str, etc) the "is" operator behaves like the "==" operator, because they are immutable and have only one instance in memory. However, for complex types such as list or dict, multiple instances can exist even though they have the same values.

🌟 How do you use the "round()" function in Python?

[illegible]

🌟 How do you use the "in" operator in Python?

In Python, the "in" operator is used to check if an element or value is present in a given iterable object, such as a list, tuple, string, or dictionary.

For example, you can use the "in" operator to check if a specific element is in a list:

```
numbers = [1, 2, 3, 4, 5]
print(3 in numbers) # prints True
```

```
print(6 in numbers) # prints False
```

You can also use the "in" operator to check if a specific value is in a dictionary:

```
ages = {"Alice": 25, "Bob": 30, "Charlie": 35}
print("Bob" in ages) # prints True
print("David" in ages) # prints False
```

You can also use the "in" operator to check if a specific substring is in a string:

```
text = "Hello, World!"  
print("World" in text) # prints True  
print("Earth" in text) # prints False
```

You can also use the "in" operator to check if a specific element is in a tuple.

```
colors = ("red", "green", "blue")
print("green" in colors) # prints True
print("purple" in colors) # prints False
```

In summary, the "in" operator is used to check if an element or value is present in an iterable object and it returns a Boolean value (True or False) depending on whether the element is found or not.

A horizontal row of 20 small black arrows pointing to the right.

🌟 Explain the use of the "break" and "continue" statements in Python.

A decorative horizontal line composed of approximately 20 small, black, stylized arrowheads or chevrons pointing towards the right side of the page. The arrows are evenly spaced along a thin grey baseline.

## 🌟 How do you use the "ord()" and "chr()" functions in Python?

In Python, the `ord()` function is used to return the Unicode code point of a character, and the `chr()` function is used to return the character that corresponds to a given Unicode code point.

The `ord()` function takes a single character as an argument and returns an integer representing its Unicode code point. For example:

```
x = 'a'
print(ord(x)) # prints 97
```

In this example, the character 'a' has a Unicode code point of 97, so the `ord()` function returns 97.

The `chr()` function takes an integer as an argument, which is a Unicode code point, and returns a string representing the corresponding character. For example:

```
x = 97
print(chr(x)) # prints 'a'
```

In this example, the integer 97 corresponds to the character 'a', so the `chr()` function returns 'a'.

These functions are useful when working with Unicode characters and code points, and can be used to convert between the two representations.

For example, you can use `ord()` and `chr()` together to convert a string of characters to a list of integers, and vice versa:

```
string = 'Hello World!'
integer_list = [ord(c) for c in string]
print(integer_list)
# Output: [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33]
```

```
string_back = "".join(chr(i) for i in integer_list)
print(string_back)
# Output: "Hello World!"
```

In this example, the `ord()` function is used to convert each character in the string 'Hello World!' to its corresponding Unicode code point, and the `chr()` function is used to convert each code point back to its corresponding character.

In summary, `ord()` function returns the Unicode code point of a character and `chr()` returns the character corresponding to a given Unicode code point. These functions are useful when working with Unicode characters and code points and can be used to convert between the two representations.

🌟 Explain the use of the "id()" function in Python.

In Python, the `id()` function is used to return the identity of an object. The identity of an object is a unique integer that is assigned to the object when it is created and remains constant for the lifetime of the object. The `id()` function takes a single argument, which is the object for which the identity is to be returned.

For example:

```
x = [1, 2, 3]
y = [1, 2, 3]
print(id(x)) # prints the identity of the object x
print(id(y)) # prints the identity of the object y
```

In this example, even though the values of x and y are the same, they are not the same object in memory, so the identity returned by id() function will be different.

You can also use id() function to check if two variables refer to the same object or not

```
x = [1, 2, 3]
y = x
print(id(x) == id(y)) # prints True
```

In this example, x and y refer to the same object, so the comparison returns True.

It's worth noting that id() function return the memory address where the object is stored, so it will be different for different runs of your program, even for the same objects.

In short, id() function is a unique identifier for an object which can be useful for debugging, but it should not be used to compare the equality of objects, you should use == operator for that.

> >

🌟 How do you use the "format()" method in Python strings?

> >

🌟 Explain the use of the "try-finally" block in Python.

> >

🌟 How do you use the "exec()" function in Python?

In Python, the exec() function is used to execute a string of Python code. The code passed to exec() can be a single statement or multiple statements in the form of a string. The exec() function takes a single argument, which is a string containing the Python code to be executed.

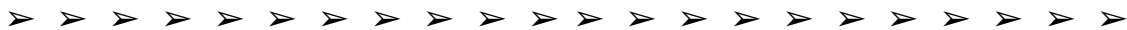
```
x = 5
exec("y = x + 2")
print(y) # prints 7
```

You can also use `exec()` to execute multiple statements by passing a string containing multiple lines of code.

This code will execute the for-loop which will print the numbers from 0 to 9.

Also, using `exec` can also change global namespace and if you are not careful, it can cause unexpected behavior.

It is recommended to use `eval()` or `exec()` only when you are sure about the input and the context in which it will be executed.



In Python, the `eval()` function is used to evaluate a string as a Python expression. The expression passed to `eval()` can be a single expression or multiple expressions in the form of a string. The `eval()` function takes a single argument, which is a string containing the Python expression(s) to be evaluated.

```
x = 5
y = eval("3 + x")
print(y) # prints 8
```

In this example, the `eval()` function is used to evaluate the string `"3 + x"` as a Python expression, which returns the value of `3 + x`, which is 8.

You can also use `eval()` to evaluate multiple expressions by passing a string containing multiple lines of code:

```
code = "a = 3; b = 4; c = a + b"
eval(code)
print(c) # prints 7
```

It is important to note that, although `eval()` can be useful for evaluating dynamic expressions, it can also be dangerous if used improperly. It is not recommended to use `eval()` with untrusted input, as it can open up a security vulnerability.

Also, using `eval()` can change the global namespace and if you are not careful, it can cause unexpected behavior. It is recommended to use `eval()` only when you are sure about the input and the context in which it will be executed.

In summary, `eval()` function is a powerful function that allows you to evaluate a string as a Python expression. However, it is important to use it with caution and only when you are sure about the input, as it can introduce security risks and unexpected behavior.

> >

✿ How do you use the `"slice()"` function in Python?

> >

✿ Explain the use of the `"set()"` data type in Python.

> >

✿ How do you use the `"frozenset()"` function in Python?

> >

✿ Explain the use of the `"complex()"` function in Python.







The array is implemented using the "array" module in Python and can be created using the array() function, which takes two arguments: the data type of the items in the array and an optional list of initial values for the array.

Example:

```
import array
```

```
# Create an array of integers
```

```
int_array = array.array("i", [1, 2, 3, 4, 5])
```

## # Create an array of floating-point numbers

```
float_array = array.array("f", [1.1, 2.2, 3.3, 4.4, 5.5])
```

A horizontal row of twenty small black arrows pointing to the right.

🦋 A multi-dimensional array can be created using nested lists, where each list represents a dimension of the array. The number of dimensions in the array is determined by the number of nested lists.

For example, "a 2-dimensional array can be create"d by creating a list of lists, where each inner list represents a row in the array:

## # Creating a 2-dimensional array

```
two_dimensional_array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

"Similarly, a 3-dimensional array can be created using a list of lists of lists:"

## # Creating a 3-dimensional array

```
three dimensional array = [[[1, 2, 3], [4, 5, 6], [7, 8, 9]], [[10, 11, 12], [13, 14, 15], [16, 17, 18]]]
```

It's important to note that the above examples are using lists and they can be used to create multidimensional arrays, but they are not as efficient as using numpy arrays or other specialized libraries.

You can also use Numpy library to create a multi-dimensional array, it provides a convenient way to create and manipulate arrays with a lot of useful methods and attributes. Here's an example of creating a 2D array using numpy:

```
import numpy as np
```

## # Creating a 2D array

```
two dimensional array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
import numpy as np
```

```
# Creating a 2D array
```

```
two_dimensional_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

