All Python Keywords

1. `False`:

> ⓘ Description
>
> Represents the boolean value false.

Example:

```
variable = False
```

2. `None`:

> ⓘ Info
>
> Represents a null or empty object.

Example:

```
variable = None
```

3. `True`:

> ⓘ Info
>
> Represents the boolean value true.

Example:

```
variable = True
```

4. `and`:

> ⓘ Info

A logical operator that returns `True` if both expressions are true.

Example:

```
if condition1 and condition2:
   # do something
```

5. `as`:

> **Info**
>
> Used in an `import` statement to rename a module or object by giving it an alias.

Example:

```
import math as m
```

The reserved word `as` in Python is not only used in the import context. Although it is commonly used in that context, it also has other applications in the language.

Different contexts:

1. Import with aliases:

```
import math as m
print(m.sqrt(25)) # Using the alias "m" instead of the full name "math".
```

In this example, the reserved word `as` is used to assign an alias to the module `math`. In this way, we can access the module functions using the alias `m` instead of the full name `math`.

2. rename variables or elements:

```
original_name = "Jhon"
new_name = original_name as copy_name
print(copy_name) # print "Jhon"
```

Here, the reserved word `as` is used to assign a new name to a variable or element. In this case, the variable `original_name` is renamed to `copy_name` using `as`.

3. Administration context with aliases:

```
with open("file.txt") as file:
    content = file.read()
```

In this example, the reserved word `as` is used to assign an alias to the object created by the management context (`open()` in this case). The file is opened within the `with` block and assigned to the `file` alias, allowing operations to be performed on the file within the block. At the end of the block, the file is automatically closed.

These are just a few additional examples of the use of `as` in different contexts in Python. Although importing with aliases is the most common usage, the `as` reserved word has flexibility in its application and can be used to assign aliases, rename elements, and more, depending on the context in which it is used.

6. `assert`:

> **Info**
>
> Used to verify that a condition is true, otherwise it throws an `AssertionError` exception.

Example:

```
assert x > 0, "x must be greater than zero."
```

7. `async`:

> **Info**
>
> Used to define an asynchronous function.

Example:

```
async def my_function():
   # do something
```

In Python, when a function is declared as an asynchronous function using the `async` reserved word, it returns a special object called a `coroutine`.

> **Note**

A corroutine is not directly equivalent to a promise in JavaScript, but it shares some similarities in terms of asynchrony.

When you invoke an asynchronous function, you don't directly get the expected result right away. Instead, you get a corrutine object that represents the asynchronous execution of the function. To get the result of the asynchronous function, you need to "wait" for the completion of the corrutine using the `await` reserved word.

Here is an example to illustrate this concept:

```
import asyncio

async def my_function():
    await asyncio.sleep(2).
    return "Asynchronous function completed!"

async def main():
    print("Starting asynchronous function")
    result = await my_function()
    print(result)

asyncio.run(main())
```

In this example, the `my_function()` function is an asynchronous function that uses `asyncio.sleep(2)` to simulate an asynchronous operation that takes 2 seconds. Within the `main()` function, we wait for the completion of `my_function()` using `await`. Only when the corrutine completes, the program continues and the result is printed.

The similarity with promises in JavaScript lies in the fact that both allow asynchronous execution and the ability to wait for the completion of a task before continuing with the code. However, corroutines in Python have their own language-specific implementation and syntax.

> **Note**
>
> It is important to note that, unlike JavaScript, where promises are resolved or rejected, corroutines in Python always return a value using the `return` statement. In addition, corroutines can be executed in an event loop using Python's `asyncio` module for more advanced asynchronous execution.

`async for`:

The `async for` statement is used to asynchronously iterate over an iterable and perform asynchronous operations within the loop. It allows waiting for the completion of asynchronous tasks on each iteration.

```
import asyncio

async def my_function(item):
        print(item)
    # Function code

async def main():
    iterable = [1, 2, 3, 3, 4, 5]

    async for item in iterable:
        await my_function(item)
        # asynchronous operations inside the loop

asyncio.run(main())
```

`async with`:

The `async with` statement is used to manage the context of an asynchronous object and ensure that resources are properly released upon completion. It is similar to the conventional `with`, but in this case, asynchronous objects are used.

```
import asyncio

async def chorus(name, lock):
        print('chorus {}: waiting for lock'.format(name)).
        async with lock:
                print('chorus {}: holding the lock'.format(name))
                await asyncio.sleep(1)
                print('chorus {}: releasing the lock'.format(name))

loop = asyncio.get_event_loop()
lock = asyncio.Lock()
chorus = asyncio.gather(chorus(1, lock), chorus(2, lock))
try:
        loop.run_until_complete(chorus).
finally:
        loop.close()
```

Note that both `async for` and `async with` can only be used inside a coroutine function declared with `async def`. Both were introduced in version 3.5.

In short, `async for` is used to asynchronously iterate over an iterable and perform asynchronous operations on each iteration, while `async with` is used to manage the context of an asynchronous object and ensure that resources are properly released. Both constructs are part of the asynchronous programming features in Python and allow you to work with asynchronous tasks efficiently.

8. `await`:

> ⓘ Info
>
> Used within an asynchronous function to wait for an asynchronous operation to complete.

Example:

```
async def my_function():
   await some_asynchronous_operation().
```

9. `break`:

> ⓘ Info
>
> Used to exit a loop (`for` or `while`) before its normal execution completes.

Example:

```
for i in range(10):
   if i == 5:
            break
   print(i)
```

10. `class`:

> ⓘ Info
>
> Used to define a class.

Example:

```
class MyClass:
        # Methods and attributes of the class.
```

11. `continue`:

> ⓘ Info
>
> Used to jump to the next iteration of a loop (`for` or `while`) without executing the rest of the code inside the loop for that particular iteration.

Example:

```
for i in range(10):
        if i == 5:
                continue
        print(i)
```

12. `def`:

> ⓘ Info
>
> Used to define a function.

Example:

```
def my_function():
        # do something
```

13. `del`:

> ⓘ Info
>
> Used to remove a reference to an object or remove a variable.

Example:

```
del variable
```

14. `elif`:

> ⓘ Info
>
> Used in an `if` structure to check for an additional condition if the previous conditions are false.

Example:

```
if condition1:
        # do something
elif condition2:
        # Do something different
else:
        # Do something else
```

15. `else`:

> ⓘ Info
>
> Used in an `if`, `for` or `while` structure to specify a block of code to be executed if all of the above conditions are false.

Example:

```
if condition:
        # do something.
else:
        # Do another
```

The `else` reserved word in Python can be used in conjunction with the `for` and `while` control structures to add a block of code that will be executed if the loop ends normally, that is, without interrupts.

use of `else` with a `for` loop:

```
for element in sequence:
  # code inside the loop.
  if condition:
         break
```

```
else:
  # Code to execute if loop ends normally (without interrupts).
  print("The for loop has finished without interrupts")
```

In this case, the `else` block will be executed if the `for` loop completes without a `break` statement being triggered. That is, if the loop iterates over all elements of the sequence without the condition to stop it prematurely being met.

Using `else` with a `while` loop:

```
while condition:
  # Code inside the loop.
  if condition:
        break
else:
  # Code to execute if the loop ends normally (without interrupts).
  print("The while loop has finished without interrupts")
```

In this case, the `else` block will be executed if the `while` loop completes without a `break` statement being triggered and the loop condition becomes false.

The use of `else` in `for` and `while` loops can be useful when you want to perform some specific action when the loop runs to completion without breaks. This can be useful to check if specific elements have been found or to execute some logic after the loop has finished iterating. Remember that the `else` block in these cases is not executed if the loop is interrupted with a `break`.

16. `except`:

> ⓘ **Info**
>
> Used in a `try` structure to catch and handle a specific exception.

Example:

```
try:
        # Code that can throw an exception.
except ValueError:
        # Handle exception ValueError.
```

17. `finally`:

Info

Used in a `try` structure to specify a block of code that will always be executed, regardless of whether an exception occurs or not.

Example:

```
try:
        # Code that can throw an exception.
finally:
        # Code that is always executed.
```

18. `for`:

Info

Used to create a loop that loops through a sequence of elements.

Example:

```
for element in sequence:
        # do something with each element.
```

19. `from`:

Info

Used in an `import` statement to import only certain elements of a specific module.

Example:

```
from math import sqrt
```

20. `global`:

Info

Used within a function to indicate that a variable refers to the global variable and not a local variable within the function.

Example:

```
def my_function():
        global variable
        # Do something with the global variable
```

21. `if`:

> ⓘ Info
>
> Used to evaluate a condition and execute a block of code if the condition is true.

Example:

```
if condition:
        # do something.
```

22. `import`:

> ⓘ Info
>
> Used to import a module for use in code.

Example:

```
import math
```

23. `in`:

> ⓘ Info
>
> It is used to check if a value is present in a sequence.

Example:

```
if element in list:
        # do something
```

The use of the reserved word `in` can have different meanings depending on its context, although the underlying concept is the same: check if a value is present in a sequence or in a set of elements.

Contexts in which `in` is used for different purposes:

1. Checking membership in a sequence: The most common use of `in` is to check whether a value is present in a sequence, such as a `list`, a `tuple` or a string.

```
list = [1, 2, 3, 3, 4, 5]
if 3 in list:
   print("Value 3 is in list.")
```

In this case, it checks if the value 3 is present in the list. If so, the message "Value 3 is in the list" is printed.

2. Iteration in a sequence: The reserved word `in` is also used in `for` loops to iterate over the elements of a sequence one by one.

```
list = [1, 2, 3, 3, 4, 5]
for element in list:
   print(element)
```

Here, the `for` loop iterates over each element in the list and prints it.

3. Checking keys in a dictionary: The reserved word `in` is used to check if a key is present in a `dictionary`.

```
dictionary = {"name": "John", "age": 30}
if "name" in dictionary:
   print("The key `name` is in the dictionary.")
```

In this example, it is checked if the key "name" is present in the dictionary. If so, the corresponding message is printed.

4. Use in list comprehensions: The reserved word `in` is used in list comprehensions to filter elements of a sequence according to a condition.

```
numbers = [1, 2, 3, 3, 4, 5]
pairs = [num for num in numbers if num % 2 == 0]
```

In this example, `in` is used in the list comprehension to iterate over the numbers in the list and filter out only the even numbers.

5. Check for the existence of a substring in a string: You can use `in` to check if a substring is present in a larger string.

```
string = "Hello, how are you?"
if ``how`` in string:
  print("The substring `how` is present in string.")
```

In this case, it checks if the substring "how" is present in the given string.

6. Use in boolean expressions: The reserved word `in` can be used in boolean expressions to combine multiple conditions and check if a variable or value meets any of those conditions.

```
age = 25
if age in range(18, 25):
  print("The person is between 18 and 24 years old.")
```

Here, `in` is used to check if the variable `age` is within the specified age range.

7. Check if an element is present in a set:

```
set = {1, 2, 3, 4, 5}
if 3 in set:
        print("The number 3 is in the set.")
```

In this example, we check if the number 3 is present in the set using the `in` operator.

8. Use in control structures such as `if`, `elif` and `while`:

```
value = 5
if value in [1, 3, 5, 7, 9]:
    print("Value is odd")

while element in list:
    # perform some operation while element is present in list.
```

In the first example, we check if the value is an odd number using the `in` operator in an `if` structure.

In the second example, `in` is used in a `while` loop to perform some operation while an element is present in the list.

9. Checking for the existence of an element in a custom data structure:

```
class MyStruct:
        def __init__(self, elements):
           self.elements = elements

        def __contains__(self, element):
           return element in self.elements

structure = MyStructure([1, 2, 3, 4, 5])
if 4 in structure:
        print("Element 4 is in structure.")
```

In this example, a custom data structure `MyStructure` is created that has a list of elements. The special method `__contains__` is implemented to allow membership checking using the `in` operator. Then, we check if element 4 is present in the structure.

10. Check for the presence of a value in a generator or an iterator:

```
generator = (x for x in range(10))
if 5 in generator:
    print("The number 5 is in generator.")
```

In this example, we create a generator that generates numbers from 0 to 9. Next, we check if the number 5 is present in the generator using the `in` operator.

11. Used with the `any()` or `all()` function to check if any or all of the elements of a sequence meet a given condition:

```
num = [1, 2, 3, 3, 4, 5]
if any(num > 3 for num in numbers):
    print("At least one number is greater than 3").

if all(num > 0 for num in numbers):
    print("All numbers are positive").
```

In the first example, `any()` is used to check if at least one number in the list is greater than 3.

In the second example, `all()` is used to check if all numbers in the list are positive.

12. Use in unit tests to check if an element is found in a list of expected results:

```
import unittest

class MyTest(unittest.TestCase):
    def test_result(self):
           expected_results = [1, 2, 3, 3, 4, 5]
```

```
            result_obtained = get_result().
            self.assertIn(get_result, expected_results)

if __name__ == '__main__':
    unittest
```

In this example, `self.assertIn()` is used within a unit test to check if the result obtained is present in a list of expected results.

The use of the reserved word `in` is related to membership checking and may vary as to the type of sequence or data structure in which it is used. However, the general concept of checking whether a value is present remains the same.

24. `is`:

> **Info**
>
> It is specifically used to compare whether two objects are the same object in memory and its usage does not change according to its context.

Example:

```
if object1 is object2:
        # do something
```

When you use the `is` operator, you check whether two objects refer to the same memory location, that is, whether they are the same object in terms of identity. This is different from the equality comparison, which is performed with the operator `==`, ==where you check whether two objects have the same value.

Example illustrating the use of `is`:

```
x = [1, 2, 3]
y = [1, 2, 3]
z = x

print(x is y) # False, x and y are different objects
print(x is z) # True, x and z are the same object
```

In this example, two lists are created, `x` and `y`, which contain the same elements. Although the contents of `x` and `y` are the same, the objects themselves are different, so the expression `x is`

`y` returns `False`. However, the variable `x` is assigned to `z`, which means that `z` and `x` refer to the same object in memory. Therefore, the expression `x is z` returns `True`.

> **Note**
>
> It is important to note that the use of `is` is restricted to object identity comparison and should not be used to compare the value of objects. To compare the value of objects, the `==` operator must be used.

25. `lambda`:

> **Info**
>
> Used to create an anonymous (unnamed) function on a single line.

Example:

```
my_function = lambda x: x * 2
```

26. `nonlocal`:

> **Info**
>
> Used within a nested function to indicate that a variable is not local to that function or the outer function, but to an even more external function.

Example:

```
def outer_function():
        variable = 10

        def nested_function():
                nonlocal variable
                variable += 5
```

27. `not`:

> **Info**

A logical operator that inverts the Boolean value of an expression.

Example:

```
if not condition:
        # do something
```

28. `or`:

> **Info**
>
> A logical operator that returns `True` if at least one of the expressions is true.

Example:

```
if condition1 or condition2:
        # do something
```

29. `pass`:

> **Info**
>
> Used as a placeholder when no action is required in a code block.

Example:

```
if condition:
        pass # Do nothing for now.
```

30. `raise`:

> **Info**
>
> Used to generate an exception manually.

Example:

```
if condition:
        raise ValueError("An error occurred").
```

31. `return`:

> ⓘ Info
>
> Used to return a value from a function.

Example:

```
def my_function():
        return result
```

32. `try`:

> ⓘ Info
>
> Used to define a block of code in which exceptions may occur.

Example:

```
try:
        # Code that can throw an exception.
except Exception:
        # Handle exception.
```

33. `while`:

> ⓘ Info
>
> Used to create a loop that executes as long as a condition is met.

Example:

```
while condition:
        # Do something as long as the condition is true.
```

34. `with`:

> **Info**
>
> It is used to define an execution context in which some action is performed before and after the code block.

Example:

```
with open("file.txt", "r") as file:
        # do something with file.
```

The `with` statement in Python is used to work with external resources, such as files or database connections, safely and efficiently. It provides a clear and readable syntax to ensure that resources are handled properly, even in case of exceptions.

The basic structure of a `with` statement is as follows:

```
with resource as variable:
# Code to work with resource.
```

How the `with` statement is used:

1. opening the resource: The `with` statement is used to open an external resource, such as a file, using a function or method that can handle the proper opening and closing of the resource. This is done in order to ensure that the resource is closed properly, even if exceptions occur.
2. Resource context: The open resource is associated with a variable in the `with` statement. This variable is used to access and work with the resource within the code block inside the `with`.
3. Working with the resource: Within the `with` code block, you can perform any necessary operation or manipulation using the open resource. This may include reading, writing, or any other resource-specific operation.
4. Closing the resource: Once the `with` code block is exited, the resource is guaranteed to close automatically, even if exceptions occur while working with the resource. This avoids problems with unreleased resources and ensures proper management of the resource.

Example using `with` to work with a file:

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

In this example, `with` is used together with the `open()` function to open the file "file.txt" in read mode (`"r"`). The file is associated with the variable `file` inside the `with` block. Within the block, the contents of the file are read and printed to the console. Once the `with` block is exited, the file is automatically closed, regardless of whether exceptions occurred while reading the file.

The `with` statement is especially useful when working with resources that must be explicitly closed, such as files or database connections. It provides a cleaner and safer way to work with these resources by avoiding common mistakes of forgetting to close them properly.

35. `yield`:

> **Info**
>
> Used in a generator function to return a value without terminating the function, and can be resumed from where it left off on the next call.

Example:

```
def generator():
        yield value
```