Python Continue Statement

The `continue` statement skips the current iteration of a loop and continues with the next iteration.

Continue in for and while Loop

Here’s how you can implement continue statement in a for and while loop.

```
# skip 'blue' while iterating a list
colors = ['red', 'green', 'blue', 'yellow']
for x in colors:
    if x == 'blue':
        continue
    print(x)
# Prints red green yellow

# Print values from 6 through 0 while skipping odd numbers
x = 6
while x:
        x -= 1
        if x % 2 != 0:
                continue
        print(x)
# Prints 4 2 0
```

Continue Inside try-finally Block

If you have try-finally block inside a `for` or `while` statement; after execution of a continue statement, the finally clause is executed before starting the next iteration.

```
# in a for Statement
```

```
for x in range(2):
  try:
      print('trying...')
      continue
      print('still trying...')
  except:
      print('Something went wrong.')
  finally:
      print('Done!')
print('Loop ended.')
# Prints trying...
# Prints Done!
# Prints trying...
# Prints Done!
# Prints Loop ended.

# in a while statement
x = 2
while x:
  try:
      print('trying...')
      x -= 1
      continue
      print('still trying...')
  except:
      print('Something went wrong.')
  finally:
      print('Done!')
print('Loop ended.')
# Prints trying...
# Prints Done!
# Prints trying...
# Prints Done!
# Prints Loop ended.
```

Python break Statement

Python `break` statement is used to exit the loop immediately. It simply jumps out of the loop altogether, and the program continues after the loop.

Break in for and while Loop

Here's how you can implement break in a for and while loop.

```
# Break the for loop at 'blue'
colors = ['red', 'green', 'blue', 'yellow']
for x in colors:
    if x == 'blue':
        break
    print(x)
# Prints red green
# Break the while loop when x becomes 3
x = 6
while x:
    print(x)
    x -= 1
    if x == 3:
        break
# Prints 6 5 4
```

The Else Clause

If the loop terminates prematurely with `break`, the else clause won't be executed.

```
# Break the for loop at 'blue'
colors = ['red', 'green', 'blue', 'yellow']
for x in colors:
    if x == 'blue':
        break
    print(x)
else:
    print('Done!')
# Prints red green
# Break the while loop when x becomes 3
x = 6
while x:
    print(x)
    x -= 1
    if x == 3:
        break
else:
    print('Done!')
# Prints 6 5 4
```

Break Inside try...finally Block

If you have try-finally block inside a `for` **or** `while` **statement; after execution of break statement, the finally clause is executed before leaving the loop.**

```
# in a for statement
for x in range(5):
  try:
      print('trying...')
      break
      print('still trying...')
  except:
    print('Something went wrong.')
  finally:
    print('Done!')
# Prints trying...
# Prints Done!

# in a while statement
while 1:
  try:
    print('trying...')
    break
    print('still trying...')
  except:
    print('Something went wrong.')
  finally:
    print('Done!')
# Prints trying...
# Prints Done!
```

Python global Keyword

Creates or Updates a global variable from a nonglobal scope

Usage

The `global` keyword is used to create or update a global variable from a nonglobal scope (such as inside a function or a class).

Syntax

```
global var1,var2,…
```

Parameter	Condition	Description
var1,var2,…	**Required**	**List of identifiers you want to declare global**

Modifying Globals Inside a Function

A variable declared outside all functions has a GLOBAL SCOPE. It is accessible throughout the file, and also inside any file which imports that file.

```
x = 42        # global scope x

def myfunc():
    print(x)    # x is 42 inside def

myfunc()
print(x)        # x is 42 outside def
```

Although you can access global variables inside or outside of a function, you cannot modify it inside a function.

Here's an example that tries to reassign a global variable inside a function.

```
x = 42        # global scope x
def myfunc():
    x = 0
    print(x)    # local x is now 0
```

```
myfunc()
print(x)       # global x is still 42
```

Here, the value of global variable x didn't change. Because Python created a new local variable named x; which disappears when the function ends, and has no effect on the global variable.

To access the global variable rather than the local one, you need to explicitly declare x global, using the global keyword.

```
x = 42         # global scope x

def myfunc():

    global x    # declare x global

    x = 0

    print(x)    # global x is now 0


myfunc()
print(x)       # global x is 0
```

The x inside the function now refers to the x outside the function, so changing x inside the function changes the x outside it.

Here's another example that tries to update a global variable inside a function.

```
x = 42         # global scope x


def myfunc():

    x = x + 1   # raises UnboundLocalError

    print(x)


myfunc()
```

Here, Python assumes that x is a local variable, which means that you are reading it before defining it.

The solution, again, is to declare x global.

```
x = 42         # global scope x
```

```
def myfunc():
    global x
    x = x + 1   # global x is now 43
    print(x)

myfunc()
print(x)       # global x is 43
```

There’s another way to update a global variable from a no-global scope – use globals() function.

Create Globals Inside a Function

When you declare a variable global, it is added to global scope, if not already present. For example, you can declare x global inside a function and access it outside the function.

```
def myfunc():
    global x    # x should now be global
    x = 42

myfunc()
print(x)       # x is 42
```

Python nonlocal Keyword

Usage

If a variable is declared in an enclosing (outer) function, it is nonlocal to nested (inner) function.

The nonlocal keyword is used to update these variables inside a nested function. The usage of nonlocal is very similar to that of global, except that the former is primarily used in nested functions.

Syntax

$$\text{nonlocal var1,var2,…}$$

Parameter	Condition	Description
var1,var2,…	**Required**	**List of identifiers you want to declare nonlocal**

Basic Example

Here's a basic example that tries to reassign enclosing function's local variable inside a nested function.

```
# enclosing function
def f1():
    x = 42

    # nested function
    def f2():
        x = 0
        print(x)    # x is 0
    f2()
    print(x)       # x is still 42

f1()
```

Here, the value of existing variable `x` didn’t change. Because, Python created a new local variable named `x` that shadows the variable in the outer scope.

Preventing that behavior is where the nonlocal keyword comes in.

```
# enclosing function
def f1():
    x = 42

    # nested function
    def f2():
        nonlocal x
        x = 0
        print(x)    # x is now 0
    f2()
    print(x)       # x remains 0

f1()
```

The `x` inside the nested function now refers to the `x` outside the function, so changing `x` inside the function changes the `x` outside it.