Python Flow-Control Statements

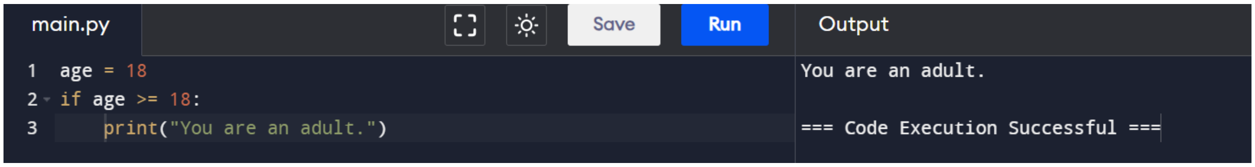
If-Else

Conditional statements allow you to make decisions and execute specific blocks of code based on certain conditions. There are three types of conditional statements: **if**, **if-else**, **if-elif-else**, and nested **if**.

If statement:

The simplest form of a conditional statement is the if statement. It allows you to execute a block of code only if a particular condition is true.

```
if condition:
        # code to be executed if the condition is true
```



```
age = 18
if age >= 18:
    print("You are an adult.")
```

```
You are an adult.

=== Code Execution Successful ===
```

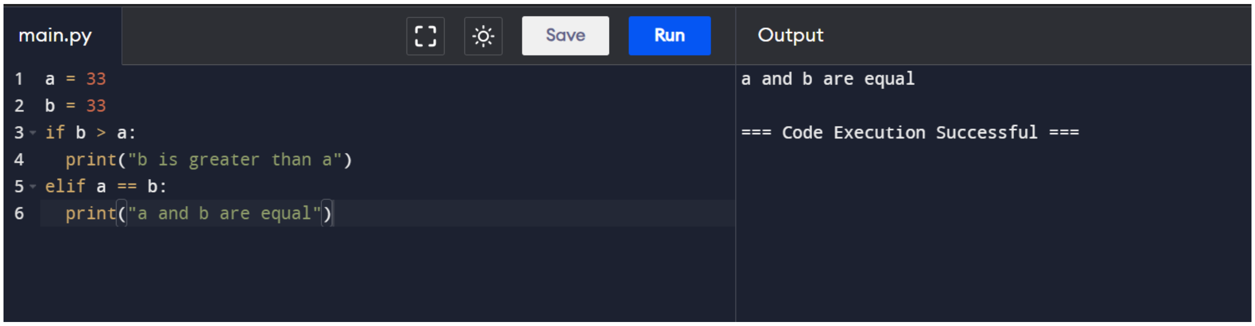
If-else statement:

An if-else statement allows you to execute one block of code if the condition is true and another block of code if the condition is false.

```
if condition:
        # code to be executed if the condition is true

 else:
        # code to be executed if the condition is false
```

```
a = 33

b = 33

if b > a:

  print("b is greater than a")

elif a == b:

  print("a and b are equal")
```

Python Flow-Control Statements



```python
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

```
a and b are equal

=== Code Execution Successful ===
```

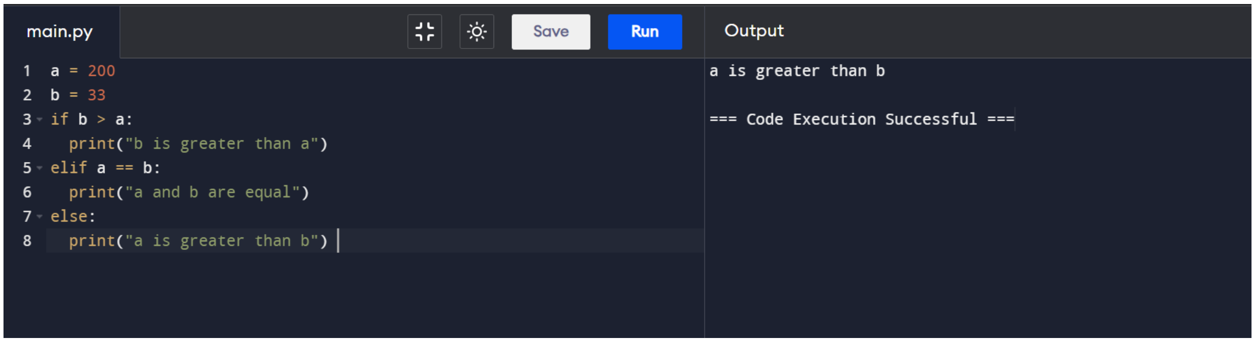
If-elif-else statement:

An if-elif-else statement is used when you have multiple conditions to check. It allows you to check each condition one by one and execute the code block associated with the first condition that is true. If none of the conditions are true, the code block under the e lse statement is executed.

```
if condition1:
        # code to be executed if condition1 is true

elif condition2:
        # code to be executed if condition2 is true

else:
        # code to be executed if both conditions are false
```

```python
a = 200

b = 33

if b > a:

  print("b is greater than a")

elif a == b:

  print("a and b are equal")

else:

  print("a is greater than b")
```

Python Flow-Control Statements



```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

```
a is greater than b

=== Code Execution Successful ===
```

Nested if statement:

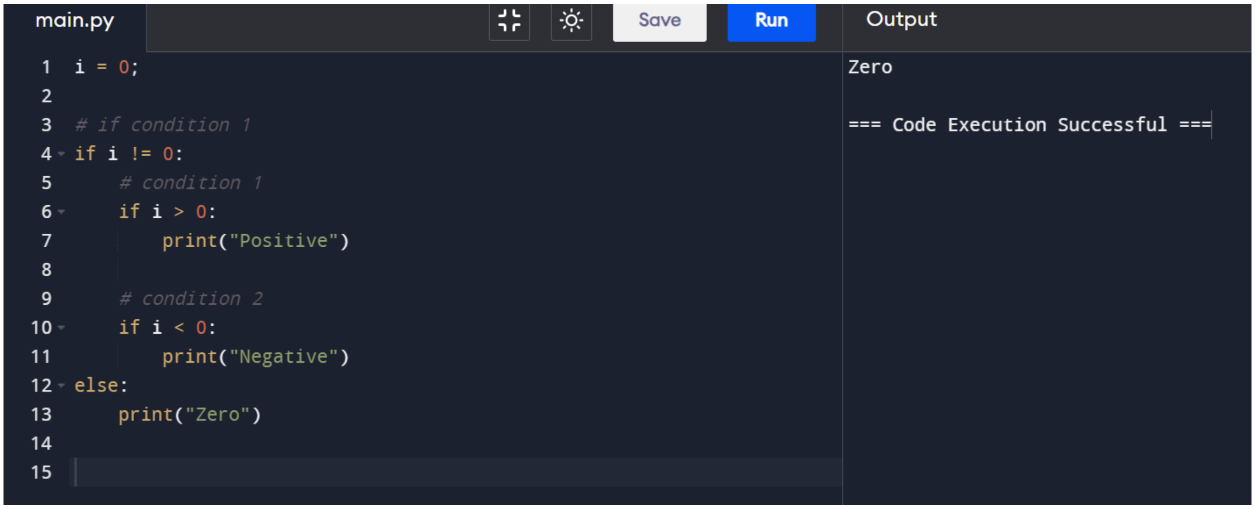
A nested if statement is one where an if statement is placed inside another if statement. It allows you to make more complex decisions based on multiple conditions.

```
 if condition1:
        # code to be executed if condition1 is true

        if condition2:
              # code to be executed if both condition1 and condition2 are true

        else:
        # code to be executed if condition1 is true but condition2 is false

  else:
        # code to be executed if condition1 is false
```

Try Yourself:

```
i = 0;


# if condition 1

if i != 0:

   # condition 1

   if i > 0:

      print("Positive")
```

```
    # condition 2
    if i < 0:
        print("Negative")
else:
    print("Zero")
```



```
i = 0;

# if condition 1
if i != 0:
    # condition 1
    if i > 0:
        print("Positive")

    # condition 2
    if i < 0:
        print("Negative")
else:
    print("Zero")

```

```
Zero

=== Code Execution Successful ===
```

While

A **while** loop is used to repeatedly execute a block of code as long as a specified condition is true. This provides a way to perform iterative tasks, such as iterating over elements in a list or processing user input until a certain condition is met.

The syntax of a **while** loop is as follows:

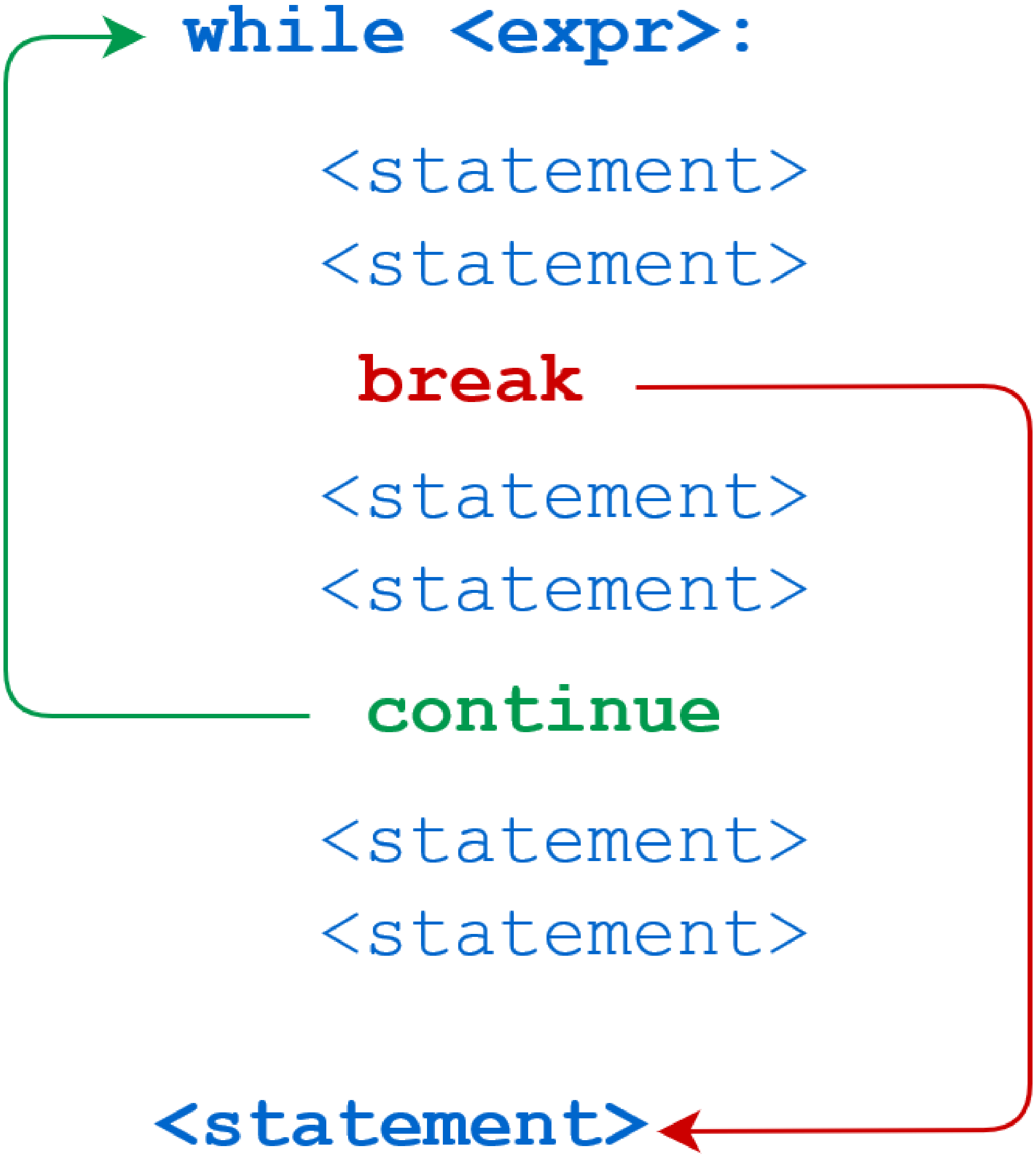
```
while <expr>:
    <statement(s)>
```

Python provides two keywords that terminate a loop iteration prematurely:

- The **break** statement immediately terminates a loop entirely. Program execution proceeds to the first statement following the loop body.
- The **continue** statement immediately terminates the current loop iteration. Execution jumps to the top of the loop, and the controlling expression is re-evaluated to determine whether the loop will execute again or terminate.

The distinction between **break** and **continue** is demonstrated in the following diagram:

Python Flow-Control Statements

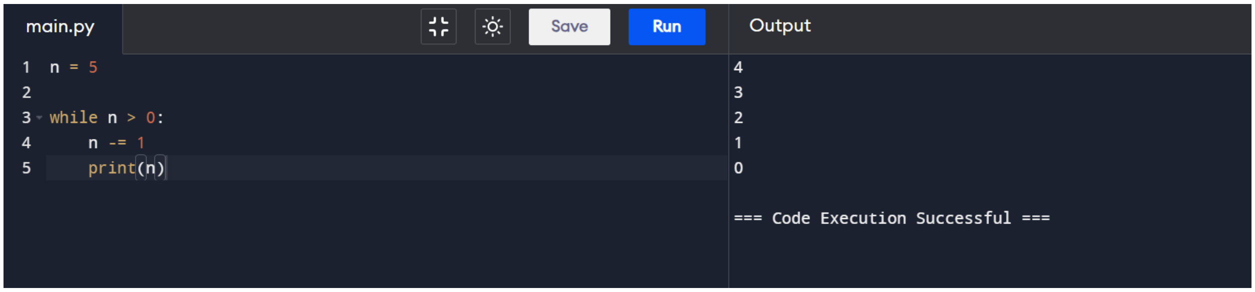


```
while <expr>:
        <statement>
        <statement>
         break
        <statement>
        <statement>
         continue
        <statement>
        <statement>

<statement>
```

Try yourself:

```
n = 5

while n > 0:
    n -= 1
    print(n)
```

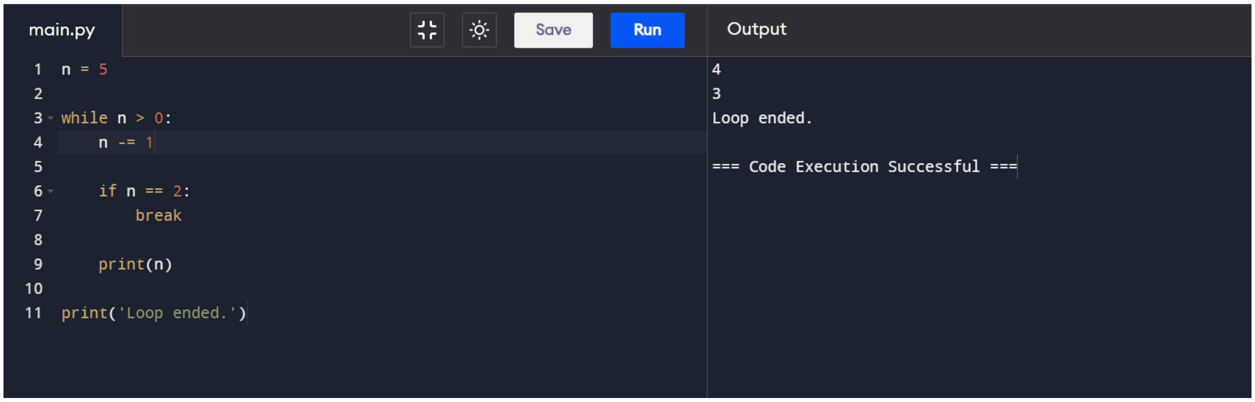
Python Flow-Control Statements



```
n = 5

while n > 0:
    n -= 1
    print(n)
```

```
4
3
2
1
0

=== Code Execution Successful ===
```

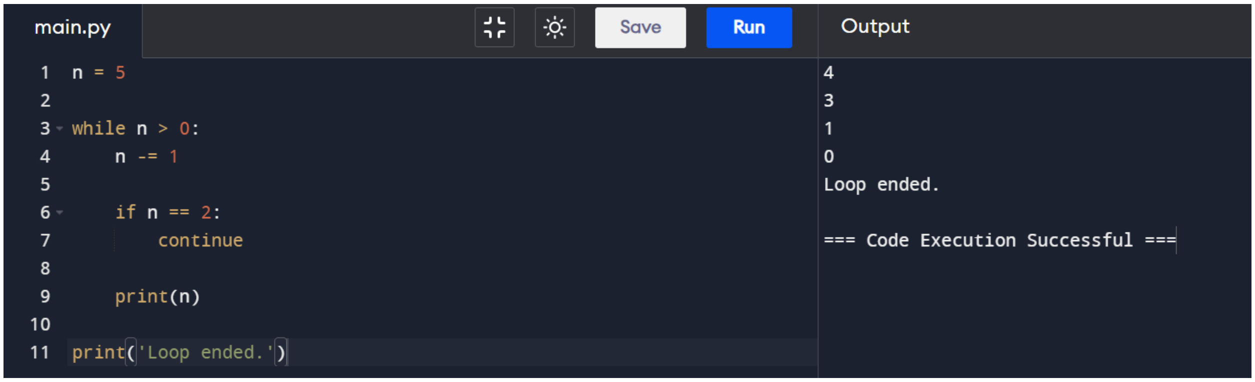
```
n = 5


while n > 0:

   n -= 1



   if n == 2:

      break



   print(n)



print('Loop ended.')
```



```
n = 5

while n > 0:
    n -= 1

    if n == 2:
        break

    print(n)

print('Loop ended.')
```

```
4
3
Loop ended.

=== Code Execution Successful ===
```

Python Flow-Control Statements



```
n = 5

while n > 0:
    n -= 1

    if n == 2:
        continue

    print(n)

print('Loop ended.')
```

```
4
3
1
0
Loop ended.

=== Code Execution Successful ===
```

For

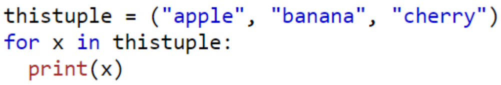
The for loop in Python is an iterating function. If you have a sequence object like a list, you can use the for loop to iterate over the items contained within the list.

Exercise :

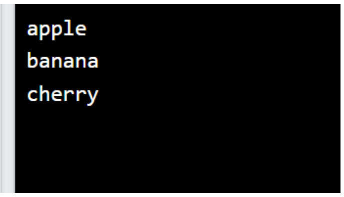
thistuple = ("apple", "banana", "cherry")

for x in thistuple:

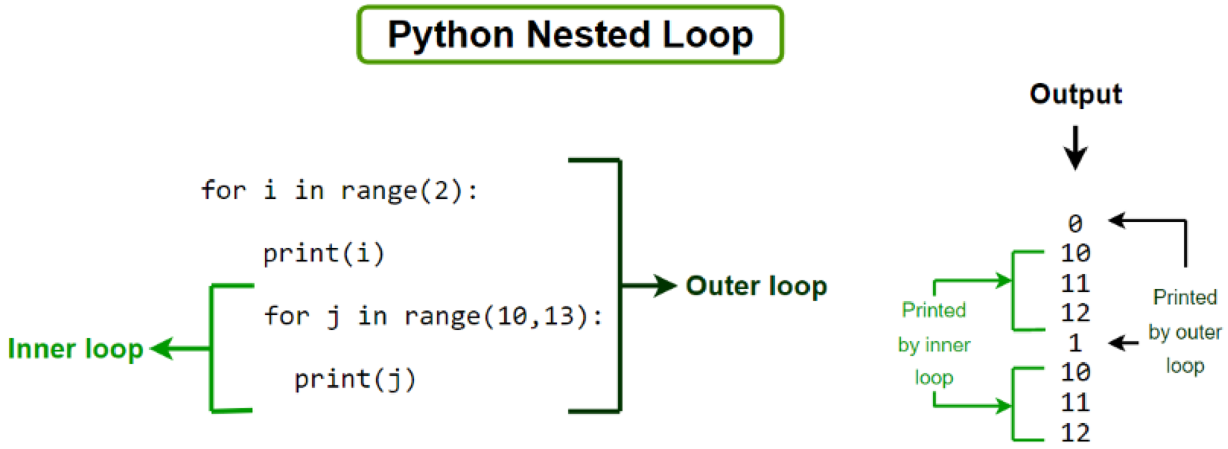
  print(x)



```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```



```
apple
banana
cherry
```



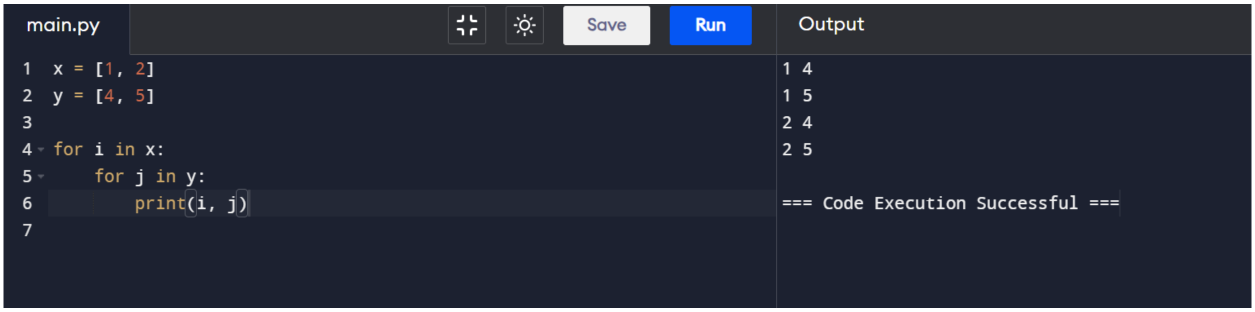
```
for i in range(2):
    print(i)
    for j in range(10,13):
        print(j)
```

```
0
10
11
12
1
10
11
12
```

x = [1, 2]

Python Flow-Control Statements

```
y = [4, 5]


for i in x:
    for j in y:
        print(i, j)
```

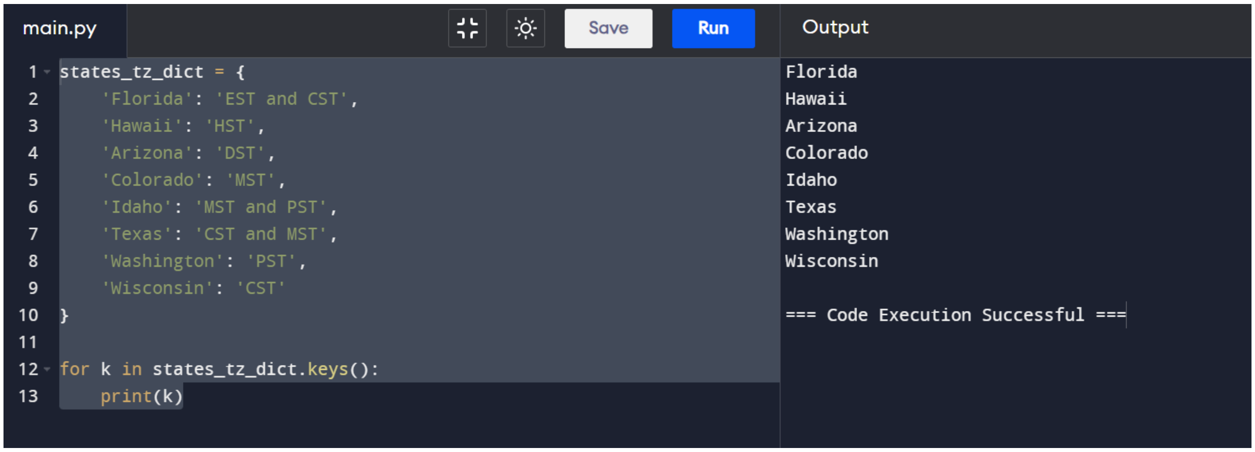


```
x = [1, 2]
y = [4, 5]

for i in x:
    for j in y:
        print(i, j)
```

```
1 4
1 5
2 4
2 5

=== Code Execution Successful ===
```

```
states_tz_dict = {
    'Florida': 'EST and CST',
    'Hawaii': 'HST',
    'Arizona': 'DST',
    'Colorado': 'MST',
    'Idaho': 'MST and PST',
    'Texas': 'CST and MST',
    'Washington': 'PST',
    'Wisconsin': 'CST'
}


for k in states_tz_dict.keys():
    print(k)
```

Python Flow-Control Statements



```python
states_tz_dict = {
    'Florida': 'EST and CST',
    'Hawaii': 'HST',
    'Arizona': 'DST',
    'Colorado': 'MST',
    'Idaho': 'MST and PST',
    'Texas': 'CST and MST',
    'Washington': 'PST',
    'Wisconsin': 'CST'
}

for k in states_tz_dict.keys():
    print(k)
```

```
Florida
Hawaii
Arizona
Colorado
Idaho
Texas
Washington
Wisconsin

=== Code Execution Successful ===
```

Function

Functions in are reusable blocks of code that perform specific tasks when called. They help in organizing code, improving readability, and promoting code reuse.

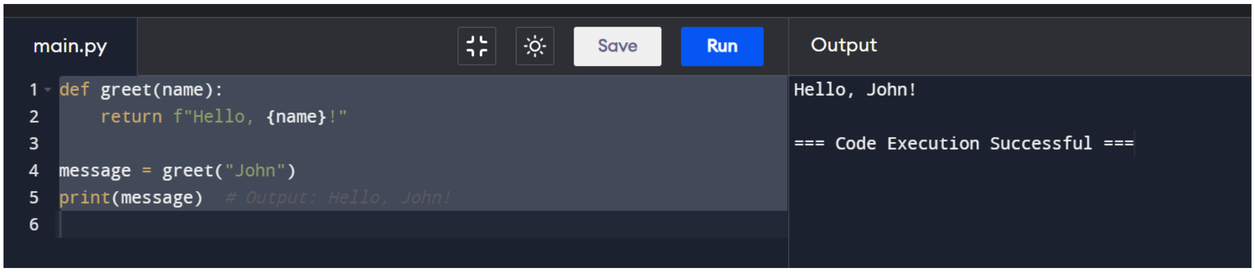
Basic Concepts about Functions:

- Functions are defined using the def keyword followed by the function name and parentheses containing optional parameters.
- Parameters are variables passed to the function for it to work on. You can also think of them as placeholders for values that will be used in the method later when the method is called.
- Functions can return values using the return statement.
- Functions can have default parameter values, making them flexible.
- The scope of variables inside a function is local unless explicitly defined as global

Try Yourself:

```
def greet(name):
    return f"Hello, {name}!"

message = greet("John")
print(message)  # Output: Hello, John!
```

Python Flow-Control Statements

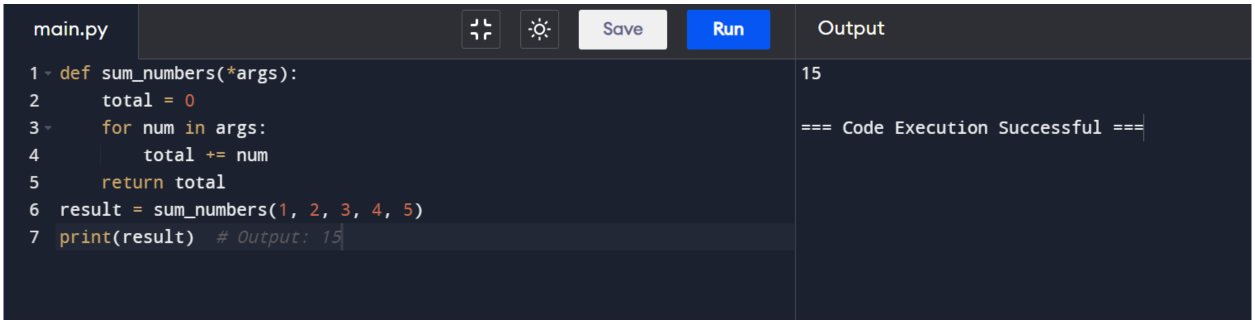


```
def greet(name):
    return f"Hello, {name}!"

message = greet("John")
print(message)  # Output: Hello, John!
```

```
Hello, John!

=== Code Execution Successful ===
```

Function with Variable Number of Arguments (*args):

```
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total
result = sum_numbers(1, 2, 3, 4, 5)
print(result)  # Output: 15
```



```
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total
result = sum_numbers(1, 2, 3, 4, 5)
print(result)  # Output: 15
```

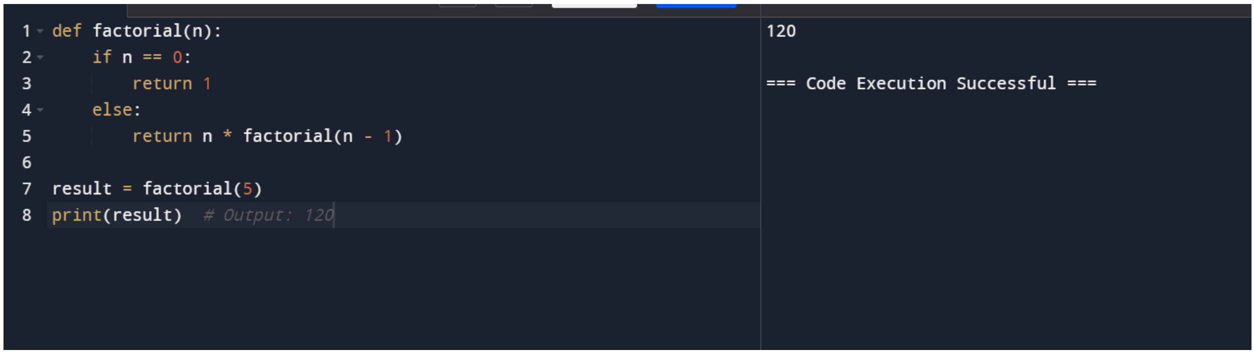
```
15

=== Code Execution Successful ===
```

Try yourself:

Function with Keyword Arguments (**kwargs):

```
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
display_info(name="John", age=30, city="New York")
```

Python Flow-Control Statements



```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print(result)  # Output: 120
```

```
120

=== Code Execution Successful ===
```

Exercise:

Higher-Order Function (Function as Parameter):

```
def apply_operation(operation, x, y):
    return operation(x, y)
def add(a, b):
    return a + b
def multiply(a, b):
    return a * b
result1 = apply_operation(add, 3, 5)
result2 = apply_operation(multiply, 3, 5)
```

Lambda

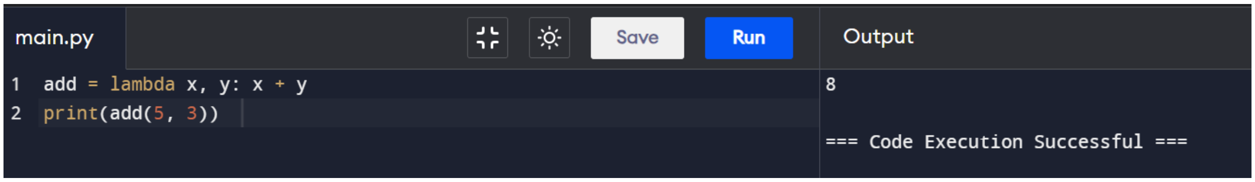
Lambda functions, also known as anonymous functions or lambda expressions, are small, single-line functions that can have any number of arguments but only one expression.

They are defined using the lambda keyword and are commonly used when a small function is required for a short period.

Basic Syntax:

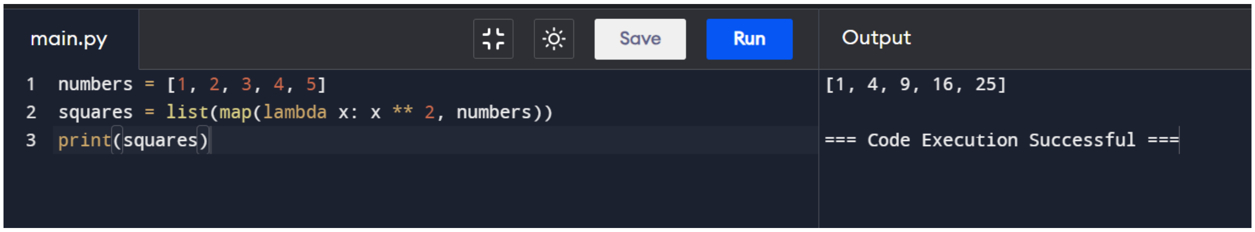
```
lambda arguments: expression


add = lambda x, y: x + y

print(add(5, 3))
```

Python Flow-Control Statements



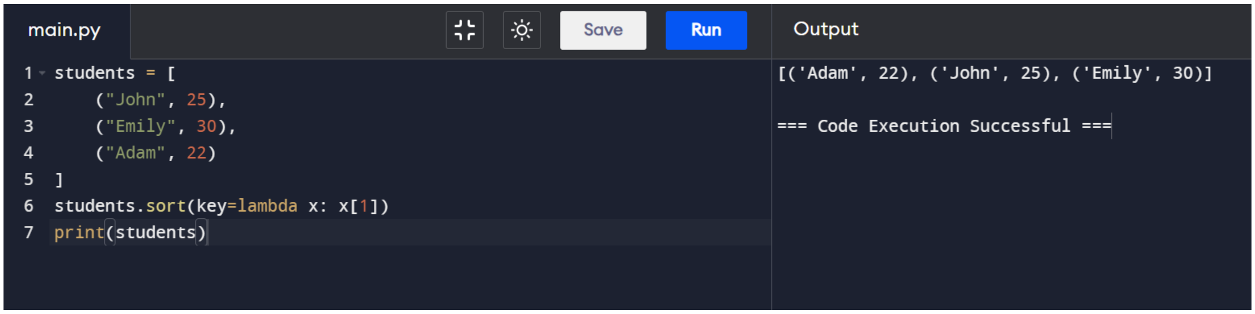
```python
add = lambda x, y: x + y
print(add(5, 3))
```

```
8

=== Code Execution Successful ===
```



```python
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x ** 2, numbers))
print(squares)
```

```
[1, 4, 9, 16, 25]

=== Code Execution Successful ===
```



```python
students = [
    ("John", 25),
    ("Emily", 30),
    ("Adam", 22)
]
students.sort(key=lambda x: x[1])
print(students)
```

```
[('Adam', 22), ('John', 25), ('Emily', 30)]

=== Code Execution Successful ===
```

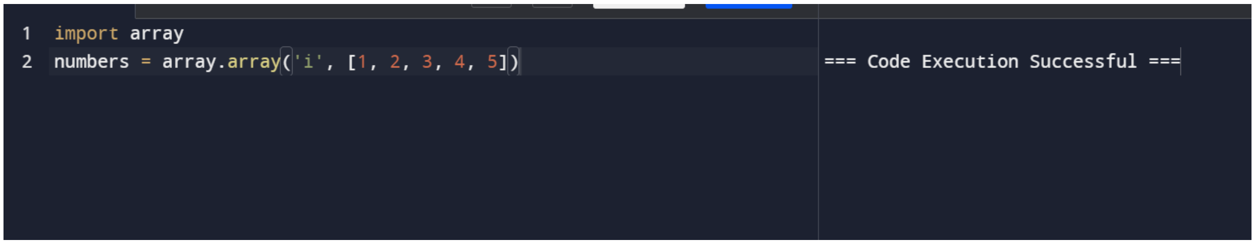
Arrays

- An array is a block of memory where elements of the type are stored sequentially.
- Each element in an array is accessed using an index starting from 0 for the element.
- Arrays allow access to elements based on their positions facilitating retrieval and modification operations.
- Pythons array module provides an approach to creating and working with arrays compared to lists.

Creating an array:

import array

numbers = array.array('i', [1, 2, 3, 4, 5])

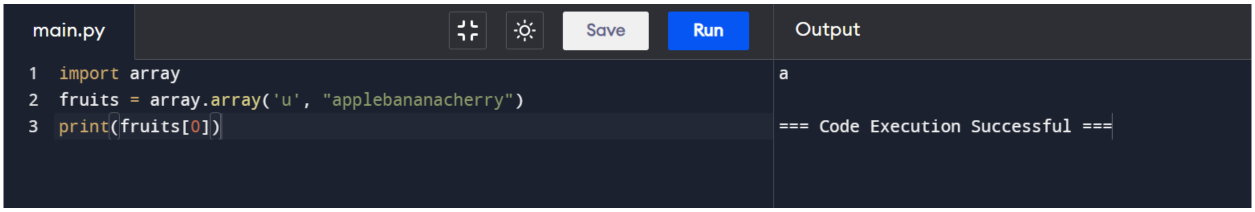


```python
import array
numbers = array.array('i', [1, 2, 3, 4, 5])
```

```
=== Code Execution Successful ===
```

Python Flow-Control Statements

```
import array

fruits = array.array('u', "applebananacherry")

print(fruits[0])
```



```
import array
fruits = array.array('u', "applebananacherry")
print(fruits[0])
```

```
a

=== Code Execution Successful ===
```

u represents a Unicode character which acts as the typecode for the array fruits.

Try yourself:

Exercise 1:

```
import array
fruits = array.array('u', "applebananacherry")
for fruit in fruits:
    print(fruit)
```

Exercise 2:

```
import array
numbers = array.array('i', [1, 2, 3, 4, 5])
length = len(numbers)
print(length)
```

Exercise 3:

```
import array
numbers = array.array('i', [3, 1, 4, 1, 5, 9])
numbers_sorted = sorted(numbers)
print(numbers_sorted)  # Output: array('i', [1, 1, 3, 4, 5, 9])
```