Python Data Types

Namespaces

What are Namespaces in Python?

All variables in Python exist in a namespace: the namespace the variable exists in determines what resources can and can't interact with that variable. There are four name spaces in Python

- Built-In
- Global
- Local
- Enclosing

Built-In Namespace

All variables, classes, functions, etc. that are built-in to Python exist in the "Built-In" scope (int, str, list len, print, input, etc.). These resources can be referenced anywhere in your code and should not have their values changed.

Global Namespace

Any reference you create directly in the module exists in the Global namespace: these are the resources that can be exported into other modules in your application. An easy way to check if a variable exists in the global scope is to check its indentation: if there is no tab or space before the reference then it exists in the Global namespace

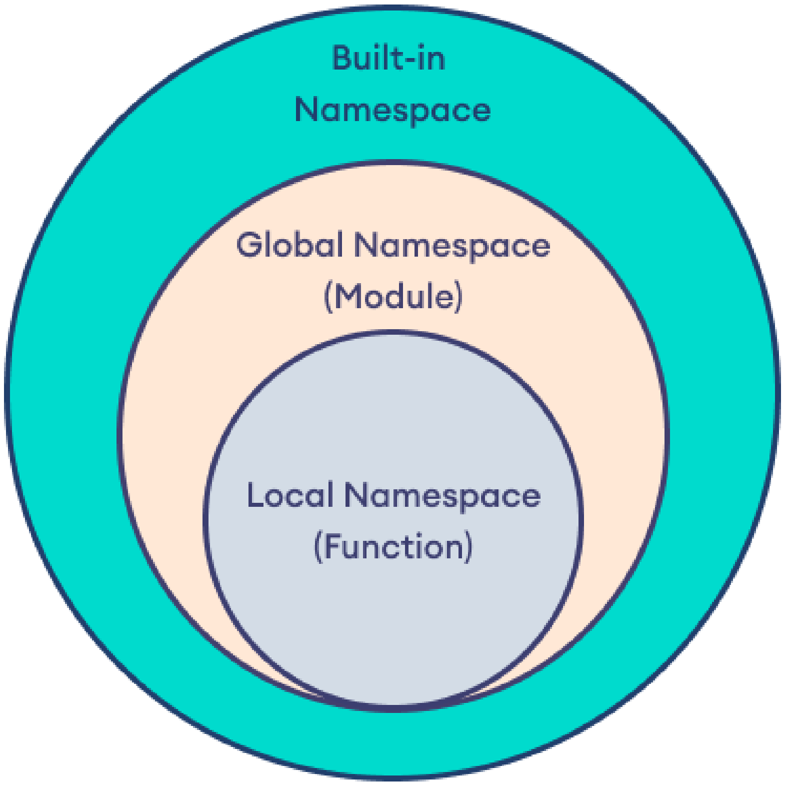
Local Namespace

The Local namespace is any block of nested code inside a class or function. Anything defined in this "local" space exists only within the space and can't be referenced outside that nested space directly.

Enclosing Namespace

Anytime you have a function defined within another function you create an Enclosing namespace: The Local namespace of the outer function is available to the inner function, but not vice versa.

Python Data Types



Built-In Namespace

```
# anything that comes pre-provided by Python exists in the Built-In namespace
print("Hello Built-In namespace!")
```

Global Namespace

Module One

```
# Because this variable is defined directly in the module it exists in the Global namespace
# This means we can export it into another module
person_name = "Billy
```

Module Two

# **this** will **import** the "person_name" variable into the module **and print** its value to the terminal

```
# we can do this because the variable exists in the Global namespace
```

Python Data Types

```
from module_one import person_name
print(person_name)
```

Local Namespace

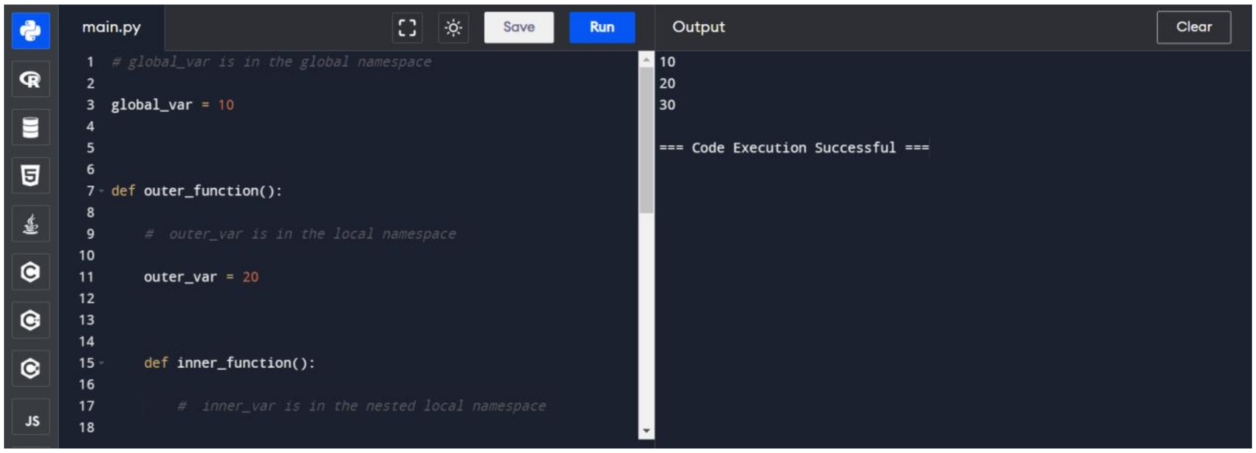
```
def double_number(num):
    # this variable exists only in the Local namespace of the function
    doubler = 2
    return num * doubler
print(doubler) # this will give you a NameError, since doubler is not defined outside the Local
               # namespace of the function
```

Enclosing Namespace

```
def doubler_function(num):
    # because we have an inner function this doubler function exists in the Enclosing namespace
    doubler = 2
    def multiplier_function():
        # the inner function has access to Enclosing space of the function it is being defined in
        result = num * doubler
        return result
    # because the doubler_function does not have access to the Local namespace of the
    # multiplier_function we have to call the multiplier_function to gain access to "result"
    return multiplier_function()
```

Try the code:

```
# global_var is in the global namespace

global_var = 10


def outer_function():

    #  outer_var is in the local namespace

    outer_var = 20
    def inner_function():
        #  inner_var is in the nested local namespace
        inner_var = 30

        print(inner_var)

    print(outer_var)

    inner_function()

# print the value of the global variable
print(global_var)

# call the outer function and print local and nested local variables
outer_function()
```



```
# global_var is in the global namespace

global_var = 10



def outer_function():

    #  outer_var is in the local namespace

    outer_var = 20



    def inner_function():

        #  inner_var is in the nested local namespace

```

```
10
20
30

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

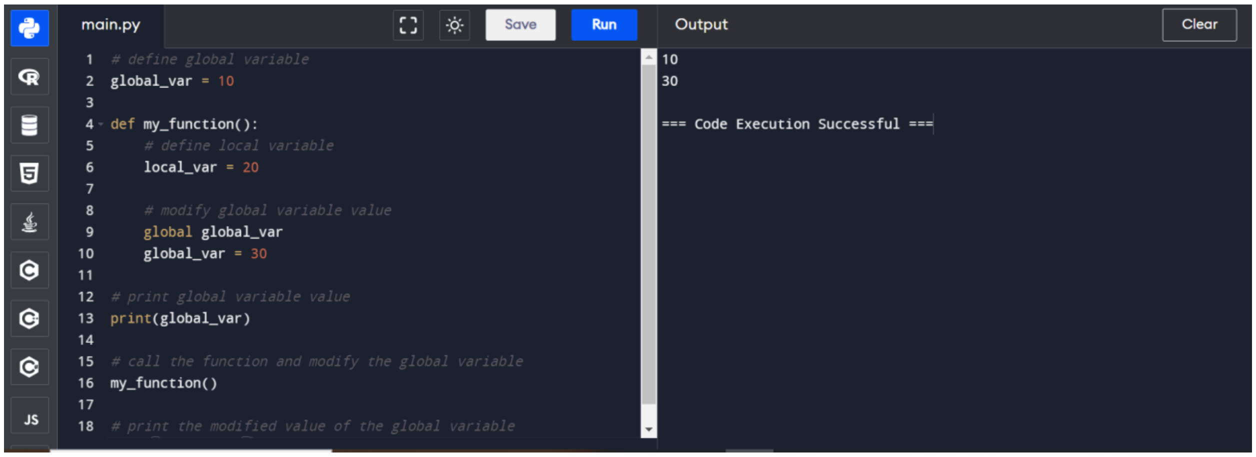
Reference Link: https://www.programiz.com/python-programming/namespace

Try out: Code 2:

```
# define global variable
```

Python Data Types

```
global_var = 10


def my_function():
    # define local variable
    local_var = 20


    # modify global variable value
    global global_var
    global_var = 30


# print global variable value
print(global_var)


# call the function and modify the global variable
my_function()


# print the modified value of the global variable
print(global_var)
```



```
# define global variable
global_var = 10

def my_function():
    # define local variable
    local_var = 20

    # modify global variable value
    global global_var
    global_var = 30

# print global variable value
print(global_var)

# call the function and modify the global variable
my_function()

# print the modified value of the global variable
```

```
10
30

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

Reference Link: https://www.programiz.com/python-programming/namespace

Python Data Types

Strings:

Strings in Python are sequences of characters, represented using single quotes (' ') , double quotes (" ") , or triple quotes (''' ''' or """ """) . Strings are versatile data types used for text processing, manipulation, and representation in Python programs.

Immutable Sequences:

Strings are immutable, meaning they cannot be modified once created. However, new strings can be generated based on existing ones.

Unicode Support:

Python strings support Unicode characters, allowing representation of diverse character sets and languages.

String Literals:

String literals can be enclosed in single quotes, double quotes, or triple quotes, providing flexibility in representing strings.

Key Characteristics of Strings:

String Manipulation:

Python offers various methods and operations for string manipulation, such as concatenation, slicing, formatting, and searching.

String Formatting:

String formatting allows inserting variable values, expressions, and formatting specifications into strings for dynamic content generation.

String Methods:

Python provides built-in string methods for common operations like conversion, case manipulation, splitting, joining, and more.

Working with Strings in Python:

Creating Strings:

Single-line strings:

```
message = "Hello, world!"
```

Python Data Types

- Single-line strings are enclosed in either single quotes (' ') or double quotes (" ").

Multi-line strings using triple quotes:

```
multiline_message = '''This is a
multi-line
string.'''
```

- Multi-line strings are enclosed in triple quotes (''' ''' or """ """) and can span across multiple lines.

String Manipulation:

Concatenating strings:

```
greeting = "Hello"
name = "Alice"
message = greeting + ", " + name + "!"
print(message)  # Output: Hello, Alice!
```

- Strings can be concatenated using the + operator to combine multiple strings into one.

String Slicing:

```
word = "Python"
print(word[0])  # Output: P
print(word[1:4])  # Output: yth
print(word[-1])  # Output: n
```

- String slicing allows extracting substrings or individual characters from a string using indexing and slicing syntax.

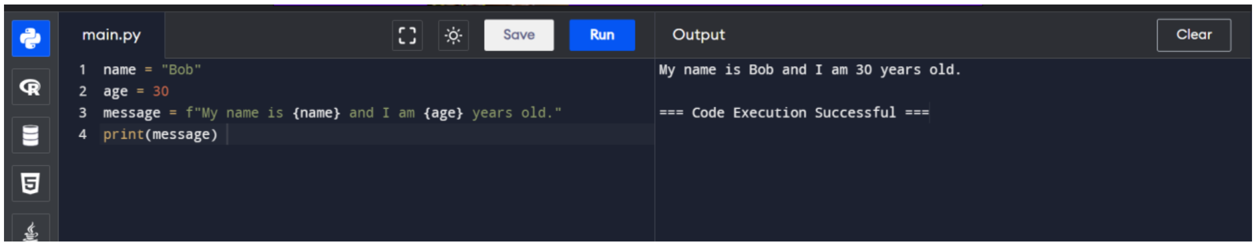
String Formatting:

Using f-strings (formatted strings):

```
name = "Bob"
age = 30
message = f"My name is {name} and I am {age} years old."
print(message)  # Output: My name is Bob and I am 30 years old.
```

Python Data Types

- f-strings allow embedding variables and expressions directly within strings for dynamic content generation.



```
name = "Bob"
age = 30
message = f"My name is {name} and I am {age} years old."
print(message)
```

```
My name is Bob and I am 30 years old.

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

Common String Operations:

String Length:

```
text = "Hello, world!"
length = len(text)
print(length)  # Output: 13
```

- The len() function returns the length of a string, which is the number of characters it contains.

String Methods:

```
sentence = "Python programming is fun!"
upper_case = sentence.upper()
lower_case = sentence.lower()
print(upper_case)  # Output: PYTHON PROGRAMMING  IS  FUN!
print(lower_case)  # Output: python programming is fun!
```

Casting:

Casting, also known as type conversion, is the process of converting one data type into another data type in Python.

Type conversions are essential for data manipulation, arithmetic operations, and ensuring compatibility between different data types.

Python Data Types

Using Casting (Type Conversion) in Python:

Implicit Casting:

Integer to Float Conversion:

```
x = 10
y = 3.5 + x  # Implicitly converts x to float for addition
print(y)  # Output: 13.5
```

- Python performs implicit casting when combining integers and floats in arithmetic operations, converting integers to floats as needed.

Boolean to Integer Conversion:

```
is_active = True
is_active_int = is_active + 10  # Implicitly converts True to 1 for addition
print(is_active_int)  # Output: 11
```

- Boolean values can be implicitly converted to integers, where True is equivalent to 1 and False is equivalent to 0.

Explicit Casting:

Integer to String Conversion:

```
x = 42
x_str = str(x)  # Explicitly converts integer to string
print("The answer is " + x_str)  # Output: The answer is 42
```

- The str() function is used for explicit casting from integers to strings, allowing concatenation with other strings.

String to Integer Conversion:

```
num_str = "123"
num_int = int(num_str)  # Explicitly converts string to integer
print(num_int + 5)  # Output: 128
```

- The int() function is used for explicit casting from strings to integers, enabling mathematical operations with numeric values.

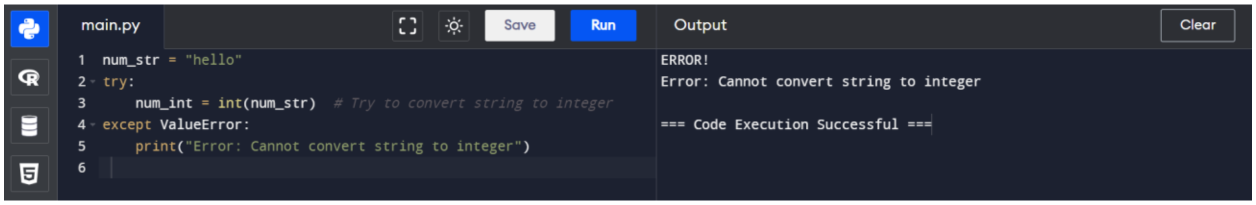
Python Data Types

Handling Type Errors:

Explicit casting with error handling:

```
num_str = "hello"
try:
    num_int = int(num_str)  # Try to convert string to integer
except ValueError:
    print("Error: Cannot convert string to integer")
```

- Error handling mechanisms such as try-except blocks can be used to handle type conversion errors gracefully and prevent program crashes.



```
num_str = "hello"
try:
    num_int = int(num_str)  # Try to convert string to integer
except ValueError:
    print("Error: Cannot convert string to integer")
```

```
ERROR!
Error: Cannot convert string to integer

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

Boolean:

- Booleans are used to represent the truthfulness or falsity of a statement or condition. True and False are the two Boolean literals in Python.
- Booleans are often used in conditional statements, loops, and logical expressions.

Creating Boolean Variables:

Assigning True and False to variables:

```
is_valid = True
is_active = False
```

- This code snippet creates Boolean variables is_valid and is_active with values True and False, respectively.

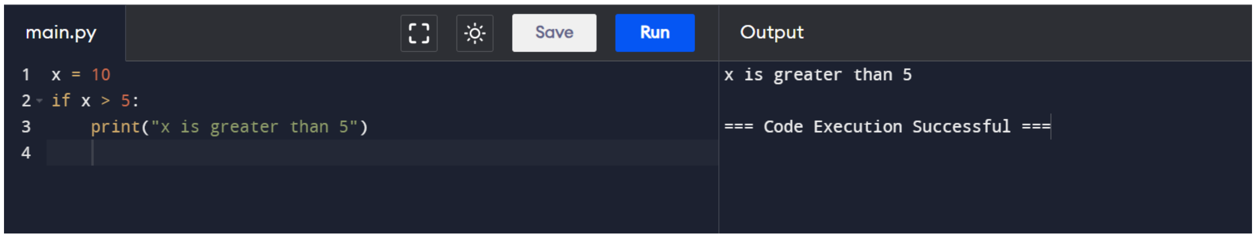
Using Boolean Values in Expressions:

Using Boolean values in if statements:

```
x = 10
if x > 5:
    print("x is greater than 5")  # Output: x is greater than 5
```

Python Data Types

- Booleans are commonly used in conditional statements like if statements to check conditions and execute code based on the result.



```
x = 10
if x > 5:
    print("x is greater than 5")
```

```
x is greater than 5

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

Logical Operations with Booleans:

Combining Boolean values with logical operators:

```
is_sunny = True
is_weekend = False

if is_sunny and not is_weekend:
   print("Go for a walk")
else:
   print("Stay indoors")  # Output: Stay indoors
```

- Logical operators (and, or, not) are used to combine Boolean values and perform logical operations based on the conditions.

Boolean Values in Control Flow:

Using Boolean values in loop conditions:

Try yourself:

```
# Python program to demonstrate

# while loop with True


N = 10

Sum = 0


# This loop will run forever

while True:
```

Sum += N

N -= 1

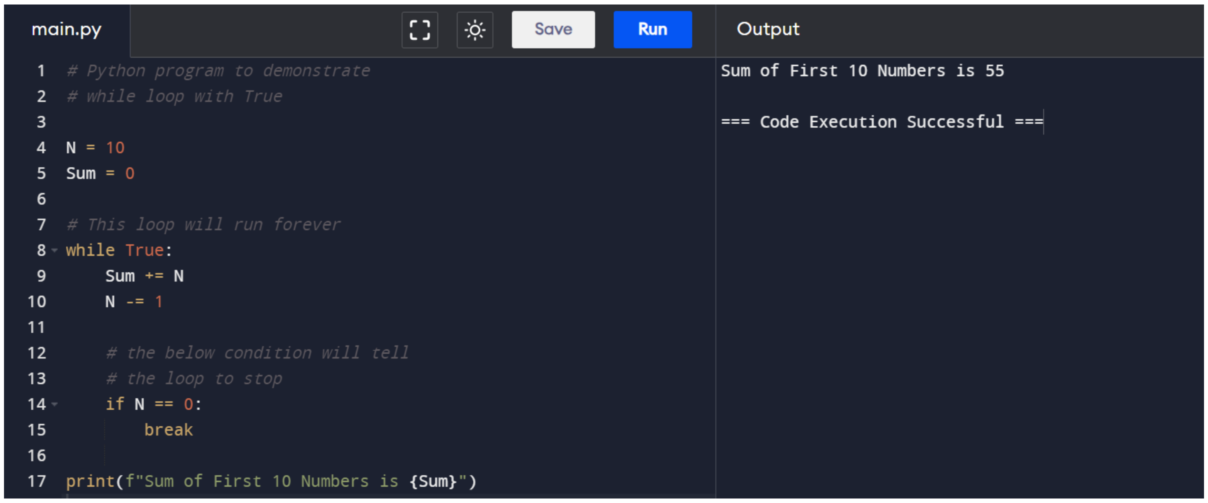
the below condition will tell

the loop to stop

if N == 0:

break

print(f"Sum of First 10 Numbers is {Sum}")



```
# Python program to demonstrate
# while loop with True

N = 10
Sum = 0

# This loop will run forever
while True:
    Sum += N
    N -= 1

    # the below condition will tell
    # the loop to stop
    if N == 0:
        break

print(f"Sum of First 10 Numbers is {Sum}")
```

```
Sum of First 10 Numbers is 55

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

List:

A List is an ordered and mutable group of values.

myList = [1, 2, 3, 4, 5]

Characteristics of a list:

- ordered by index
- mutable
- allows duplicate values

Python Data Types

- can contain values of different datatypes
- can contain other lists within it
- supports concatenation and slicing

List methods:

- **append()** - Adds an element at the end of the list
- **clear()** - Removes all the elements from the list
- **copy()** - Returns a copy of the list
- **count()** - Returns the number of elements with the specified value
- **extend()** - Add the elements of a list (or any iterable), to the end of the current list
- **index()** - Returns the index of the first element with the specified value
- **insert()** - Adds an element at the specified position
- **pop()** - Removes the element at the specified position
- **remove()** - Removes the item with the specified value
- **reverse()** - Reverses the order of the list
- **sort()** - Sorts the list

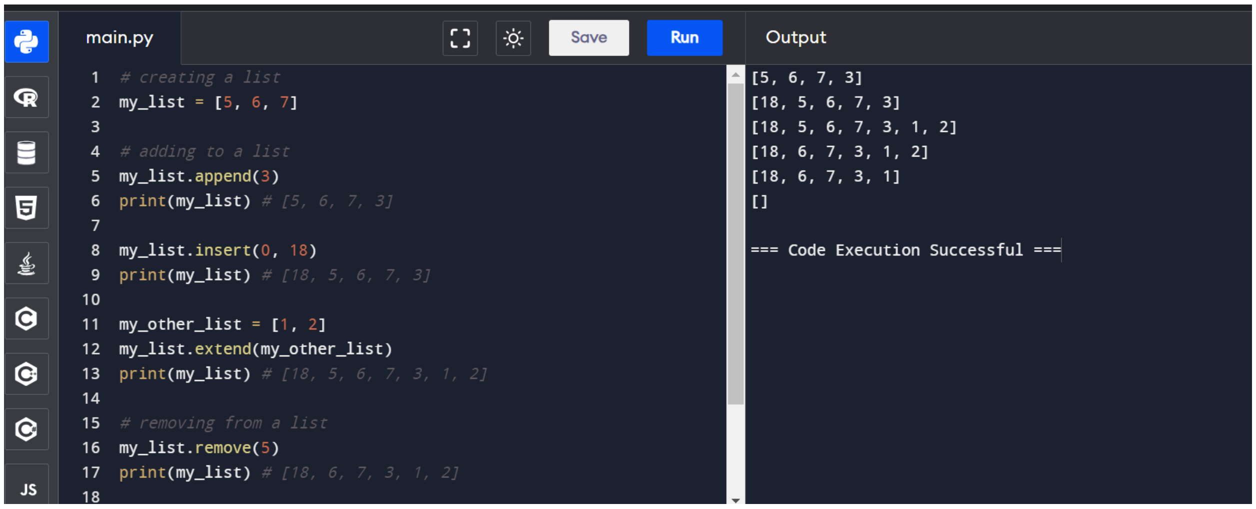
Try yourself:

```
# creating a list
my_list = [5, 6, 7]

# adding to a list
my_list.append(3)
print(my_list) # [5, 6, 7, 3]

my_list.insert(0, 18)
print(my_list) # [18, 5, 6, 7, 3]

my_other_list = [1, 2]
my_list.extend(my_other_list)
print(my_list) # [18, 5, 6, 7, 3, 1, 2]

# removing from a list
my_list.remove(5)
print(my_list) # [18, 6, 7, 3, 1, 2]

my_list.pop()
```

Python Data Types

print(my_list) # [18, 6, 7, 3, 1]

my_list.clear()
print(my_list) # []



```
# creating a list
my_list = [5, 6, 7]

# adding to a list
my_list.append(3)
print(my_list) # [5, 6, 7, 3]

my_list.insert(0, 18)
print(my_list) # [18, 5, 6, 7, 3]

my_other_list = [1, 2]
my_list.extend(my_other_list)
print(my_list) # [18, 5, 6, 7, 3, 1, 2]

# removing from a list
my_list.remove(5)
print(my_list) # [18, 6, 7, 3, 1, 2]
```

```
[5, 6, 7, 3]
[18, 5, 6, 7, 3]
[18, 5, 6, 7, 3, 1, 2]
[18, 6, 7, 3, 1, 2]
[18, 6, 7, 3, 1]
[]

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

my_list = ['apple', 'banana', 'orange', 4, 5.2]

value = my_list[0]

print(value) # apple

value = my_list[3]

print(value) # 4

value = my_list[-1]

print(value) # 5.2

result = my_list[:-1]

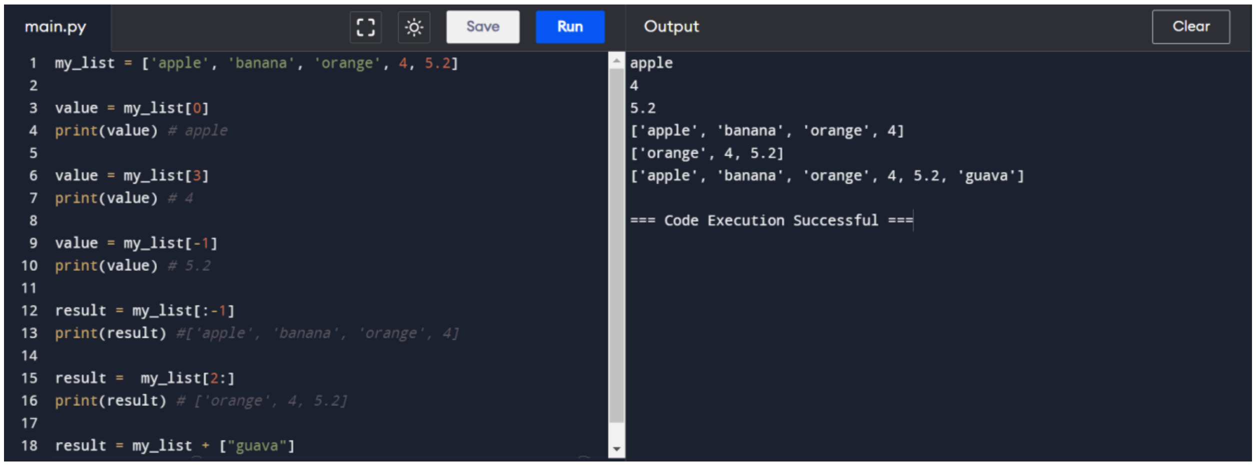
print(result) #['apple', 'banana', 'orange', 4]

result = my_list[2:]

print(result) # ['orange', 4, 5.2]

result = my_list + ["guava"]

print(result) # ['apple', 'banana', 'orange', 4, 5.2, 'guava']



```python
my_list = ['apple', 'banana', 'orange', 4, 5.2]

value = my_list[0]
print(value) # apple

value = my_list[3]
print(value) # 4

value = my_list[-1]
print(value) # 5.2

result = my_list[:-1]
print(result) #['apple', 'banana', 'orange', 4]

result =  my_list[2:]
print(result) # ['orange', 4, 5.2]

result = my_list + ["guava"]
```

```
apple
4
5.2
['apple', 'banana', 'orange', 4]
['orange', 4, 5.2]
['apple', 'banana', 'orange', 4, 5.2, 'guava']

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

Tuples:

A Tuple is an ordered and immutable group of values

Immutable means that after the creation of a tuple, we cannot edit its contents. Because tuples are sequences, or ordered groups of values, we can perform operations such as slicing or using the len() function.

Tuple literals are a comma-separated list of values that are commonly enclosed in parenthesis:

```python
my_tuple = (1, 2, 3, 4, 5)
```

Like lists, sets, and dictionaries, tuples can also be created using a function. For tuples the function is tuple():

```python
original = 1, 2, 3, 4, 5
copy = tuple(original)
```

Characteristics of a tuple:

- ordered by index

Python Data Types

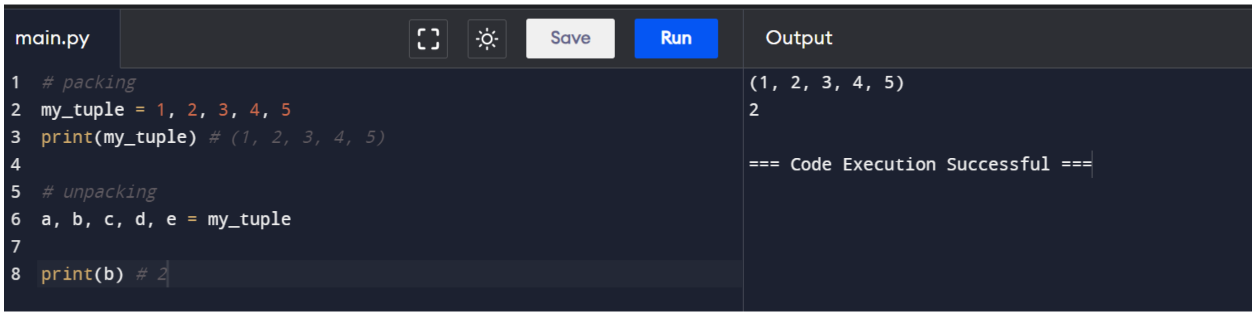
- immutable
- can be nested
- can contain values of different datatypes
- can contain mutable objects, such as lists
- supports slicing and the use of len()

Tuple methods:

- **len() -** Returns the amount of elements within the tuple
- **count() -** Returns the number of elements with the specified value
- **index() -** Returns the index of the first element with the specified value

Try yourself:

```
# packing
my_tuple = 1, 2, 3, 4, 5
print(my_tuple) # (1, 2, 3, 4, 5)

# unpacking
a, b, c, d, e = my_tuple

print(b) # 2
```



main.py

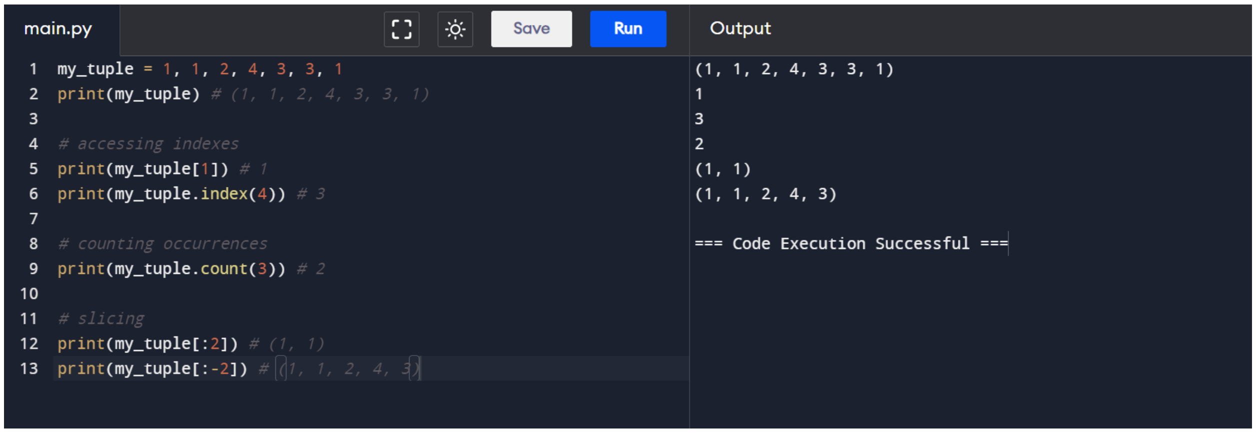
```
# packing
my_tuple = 1, 2, 3, 4, 5
print(my_tuple) # (1, 2, 3, 4, 5)

# unpacking
a, b, c, d, e = my_tuple

print(b) # 2
```

Output

```
(1, 2, 3, 4, 5)
2

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

```
my_tuple = 1, 1, 2, 4, 3, 3, 1
print(my_tuple) # (1, 1, 2, 4, 3, 3, 1)

# accessing indexes
print(my_tuple[1]) # 1
print(my_tuple.index(4)) # 3

# counting occurrences
```

```
print(my_tuple.count(3)) # 2

# slicing
print(my_tuple[:2]) # (1, 1)
print(my_tuple[:-2]) # (1, 1, 2, 4, 3)
```



```
my_tuple = 1, 1, 2, 4, 3, 3, 1
print(my_tuple) # (1, 1, 2, 4, 3, 3, 1)

# accessing indexes
print(my_tuple[1]) # 1
print(my_tuple.index(4)) # 3

# counting occurrences
print(my_tuple.count(3)) # 2

# slicing
print(my_tuple[:2]) # (1, 1)
print(my_tuple[:-2]) # (1, 1, 2, 4, 3)
```

```
(1, 1, 2, 4, 3, 3, 1)
1
3
2
(1, 1)
(1, 1, 2, 4, 3)

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

Range:

The range() function in Python is used to generate sequences of numbers efficiently. It is commonly used in loops and list comprehensions to iterate through a specific range of values.

Range generates a sequence of numbers based on start, stop, and step values. The syntax for range is range(start, stop, step), where start is the starting value, stop is the ending value (exclusive), and step is the increment (default is 1).

Syntax:

```
range(stop)
range(start, stop)
range(start, stop, step)

# start: The starting value of the range (optional, default is 0).
# stop: The ending value (exclusive) of the range.
# step: The increment (optional, default is 1).
```

Python Data Types

Creating a Range:

1. Create a range with a start, stop, and step:

```
numbers = range(1, 10, 2)
```

- This code snippet creates a range object named numbers starting from 1, ending before 10, and incrementing by 2. The range includes elements 1, 3, 5, 7, and 9.

2. Create a range with only stop value (implicitly starts from 0 and step is 1):

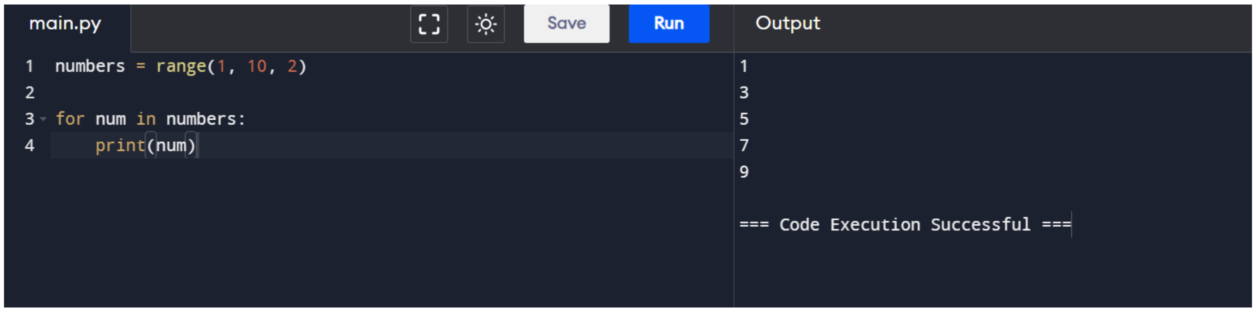
```
countdown = range(5)
```

- This code snippet creates a range object named countdown starting from 0, ending before 5, and incrementing by 1 (default step).
- The range includes elements 0, 1, 2, 3, and 4.

Try yourself:

```
numbers = range(1, 10, 2)

for num in numbers:
    print(num)
```

- A for loop is used to iterate through the elements of the range numbers, printing each element one by one.



```
numbers = range(1, 10, 2)

for num in numbers:
    print(num)
```

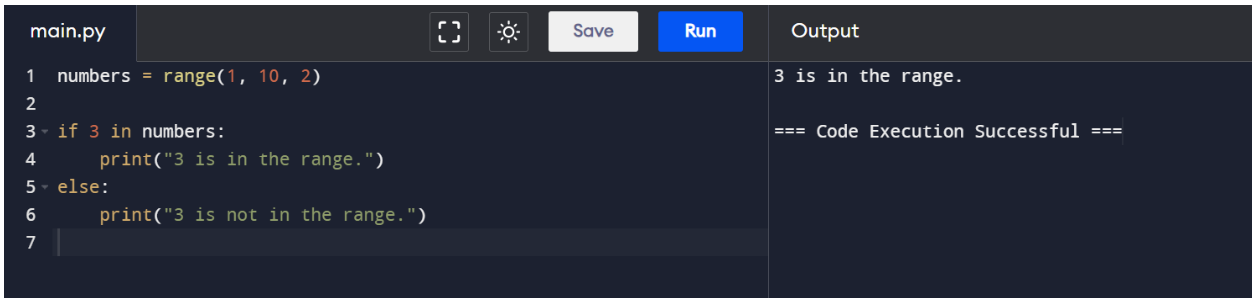
```
1
3
5
7
9

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

2. Checking range membership:

```
numbers = range(1, 10, 2)

if 3 in numbers:
    print("3 is in the range.")
else:
    print("3 is not in the range.")
```

```
# Output:
# 3 is in the range.
```

- The in keyword checks if a value (in this case, 3) is present in the range numbers.



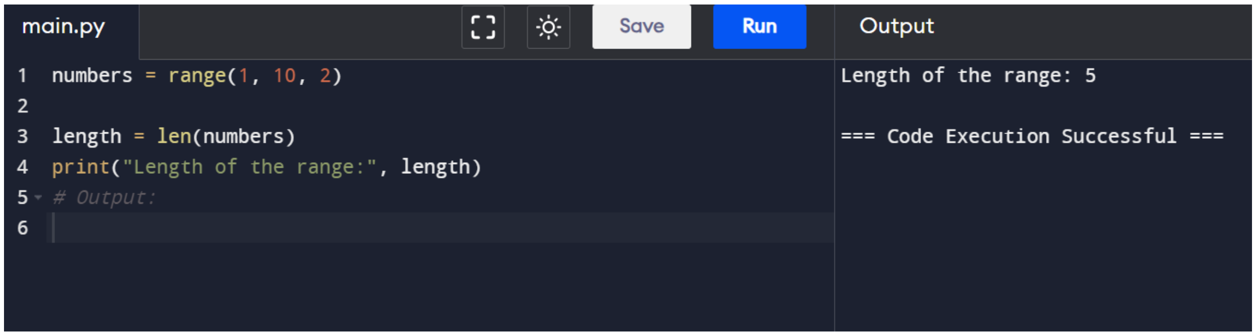
```
numbers = range(1, 10, 2)

if 3 in numbers:
    print("3 is in the range.")
else:
    print("3 is not in the range.")
```

```
3 is in the range.

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

3. Length of a range:

```
numbers = range(1, 10, 2)

length = len(numbers)
print("Length of the range:", length)
# Output:
# Length of the range: 5
```

- The len() function calculates and prints the length of the range numbers, which is the number of elements it contains.



```
numbers = range(1, 10, 2)

length = len(numbers)
print("Length of the range:", length)
# Output:
```

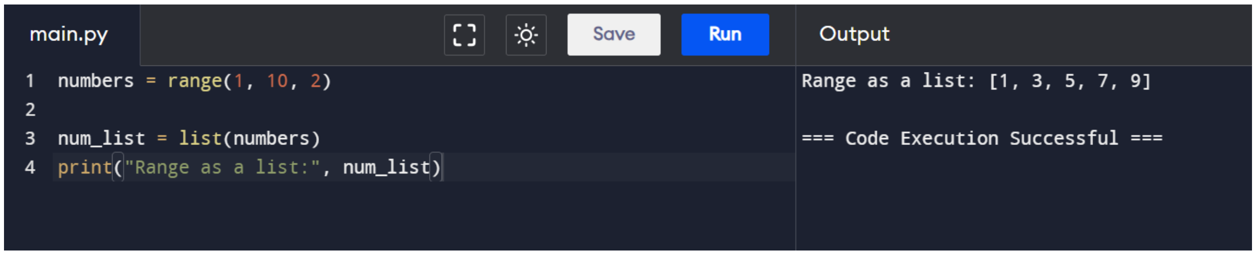
```
Length of the range: 5

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

4. Range as a list:

```
numbers = range(1, 10, 2)

num_list = list(numbers)
print("Range as a list:", num_list)
# Output:
# Range as a list: [1, 3, 5, 7, 9]
```

Python Data Types

- The list() function converts the range numbers into a list, which can be printed and manipulated like a regular list.



```python
numbers = range(1, 10, 2)

num_list = list(numbers)
print("Range as a list:", num_list)
```

```
Range as a list: [1, 3, 5, 7, 9]

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

Sets

Sets in Python are unordered collections of unique elements. They are designed to efficiently handle membership tests and eliminate duplicate entries.

- Sets are created using curly braces {} or the set() constructor.
- Unlike lists and tuples, sets do not maintain order, and elements are not indexed.
- Sets only contain unique elements; duplicate values are automatically removed.
- Python sets support mathematical operations like union, intersection, difference, and symmetric difference.

Creating Sets:

Create a set using curly braces

colors = {'red', 'green', 'blue'}

- This code snippet creates a set named colors containing the elements 'red', 'green', and 'blue' using curly braces {}.
- Sets in Python are unordered collections of unique elements, and the curly braces syntax is used to define sets.

Create a set using the set() constructor

fruits = set(['apple', 'banana', 'cherry'])

- This code snippet creates a set named fruits containing the elements 'apple', 'banana', and 'cherry' using the set() constructor.
- The constructor can take an iterable (like a list) as an argument to initialize the set.

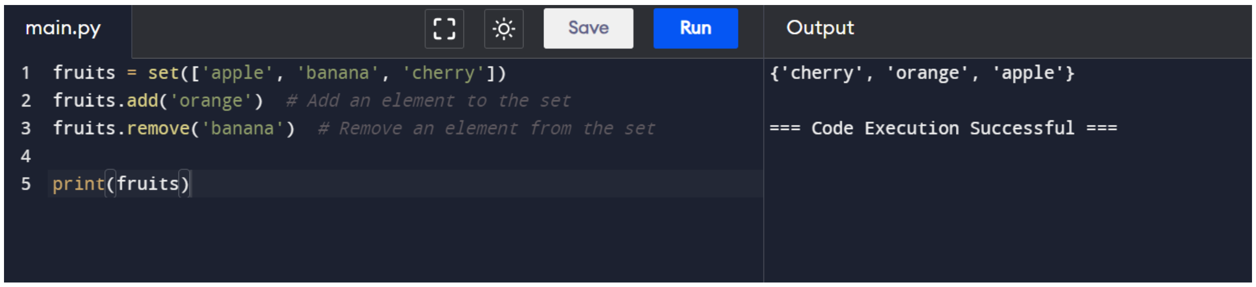
- Typecode: N/A (Sets do not have a specific typecode like arrays)

Try Yourself:

Adding and Removing Elements:

```
fruits = set(['apple', 'banana', 'cherry'])
fruits.add('orange')  # Add an element to the set
fruits.remove('banana')  # Remove an element from the set

print(fruits)
```

- Sets support methods like add() and remove() for adding and removing elements, respectively. In this code, 'orange' is added to the set fruits, and 'banana' is removed.



```
fruits = set(['apple', 'banana', 'cherry'])
fruits.add('orange')  # Add an element to the set
fruits.remove('banana')  # Remove an element from the set

print(fruits)
```

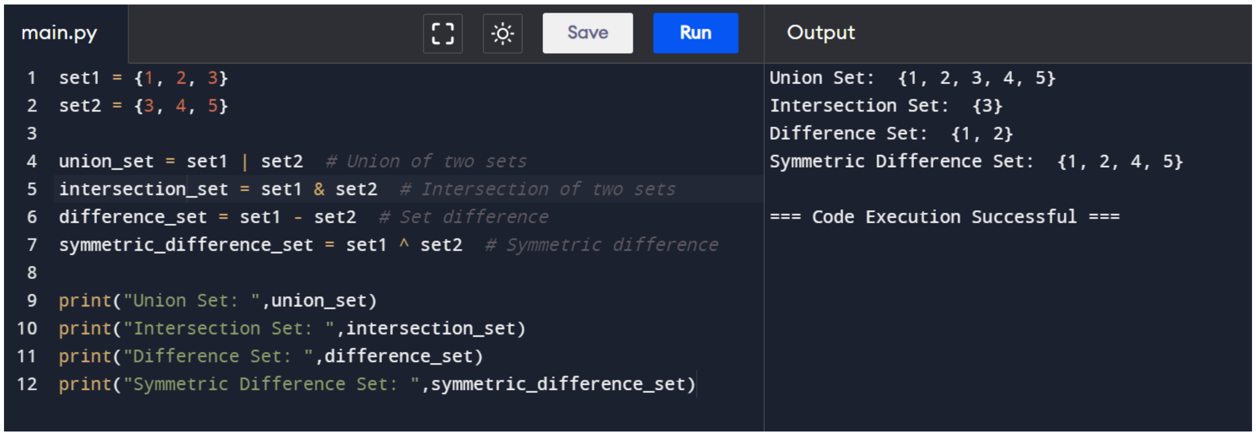
```
{'cherry', 'orange', 'apple'}

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

Python Data Types

Set Operations:set1 = {1, 2, 3}
set2 = {3, 4, 5}

union_set = set1 | set2 # Union of two sets
intersection_set = set1 **&** set2 # Intersection of two sets
difference_set = set1 - set2 # Set difference
symmetric_difference_set = set1 ^ set2 # Symmetric difference

print("Union Set: ",union_set)
print("Intersection Set: ",intersection_set)
print("Difference Set: ",difference_set)
print("Symmetric Difference Set: ",symmetric_difference_set)



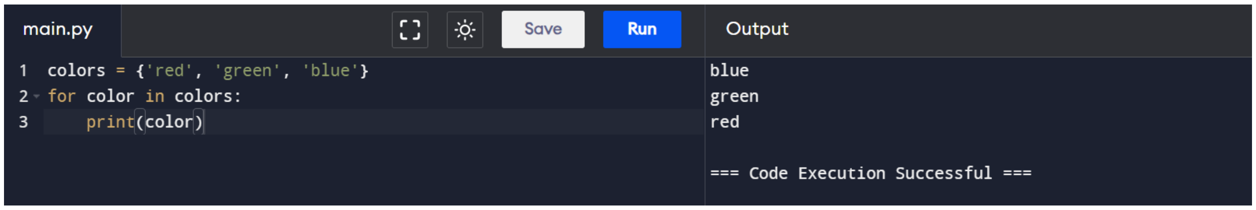
```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

union_set = set1 | set2  # Union of two sets
intersection_set = set1 & set2  # Intersection of two sets
difference_set = set1 - set2  # Set difference
symmetric_difference_set = set1 ^ set2  # Symmetric difference

print("Union Set: ",union_set)
print("Intersection Set: ",intersection_set)
print("Difference Set: ",difference_set)
print("Symmetric Difference Set: ",symmetric_difference_set)
```

```
Union Set:  {1, 2, 3, 4, 5}
Intersection Set:  {3}
Difference Set:  {1, 2}
Symmetric Difference Set:  {1, 2, 4, 5}

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

- Sets support various operations like union (|), intersection (&), set difference (-), and symmetric difference (^).
- These operations allow combining, comparing, and manipulating sets efficiently.

Iterating Through Sets:

colors = {'red', 'green', 'blue'}
for color **in** colors:
    print(color)



```
colors = {'red', 'green', 'blue'}
for color in colors:
    print(color)
```

```
blue
green
red

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

- Iterating through a set can be done using a for loop.
- Each element in the set colors is printed one by one in the loop.

Binary Type

Binary data in computer systems refers to data that is represented using only two possible values, typically 0 and 1.

In Python, binary data can be handled using the bytes and bytearray data types, allowing manipulation and conversion of binary data.

- Binary data represents information at the lowest level in a computer system, using binary digits (bits) 0 and 1.
- Binary data is commonly used for storing and transmitting data efficiently, especially for non-textual data like images, audio, and video.
- Python provides built-in data types bytes and bytearray for handling binary data.

Working with Binary Data in Python:

Creating Binary Data:

Create a bytes object from a sequence of integers:

```
binary_data = bytes([0b01000001, 0b01000010, 0b01000011])
```

- This code snippet creates a bytes object named binary_data containing binary values representing ASCII characters 'A', 'B', and 'C'.

Create a bytearray object from a binary string:

```
binary_string = "01010100 01001001 01001110 01011000"
binary_bytes = bytearray([int(byte, 2) for byte in binary_string.split()])
```

- This code snippet converts a binary string into a bytearray object named binary_bytes, representing ASCII characters in binary form.

Encoding and Decoding Binary Data:

Encoding binary data to a string:

```
binary_data = bytes([0b01000001, 0b01000010, 0b01000011])
binary_str = binary_data.decode('utf-8')
```

- The decode() method converts binary data (bytes) into a string using the UTF-8 encoding.

Decoding a binary string to binary data:

```
binary_string = "01010100 01001001 01001110 01011000"
binary_bytes = bytearray([int(byte, 2) for byte in binary_string.split()])
decoded_data = binary_bytes.decode('ascii')
```

- The decode() method converts a binary string to binary data (bytes) using the ASCII encoding.

Manipulating Binary Data:

Extracting bits from binary data:

```
binary_data = bytes([0b01010100, 0b01001001, 0b01001110, 0b01011000])
bit_mask = 0b00001111  # Mask to extract lower 4 bits
extracted_bits = bytes([byte & bit_mask for byte in binary_data])
```

- This code snippet extracts the lower 4 bits from each byte in binary_data using a bit mask.

Modifying binary data in-place:

```
binary_data = bytearray([0b01010100, 0b01001001, 0b01001110, 0b01011000])
binary_data[1] = 0b01000001  # Modify the second byte
```

- The bytearray type allows modifying binary data in-place, providing flexibility for data manipulation.

Non-Type

In Python, **None** is a special object that represents the absence of a value or a null value. It is an instance of the **NoneType** data type, which is a built-in data type in Python. The **NoneType** data type has only one possible value, which is None.

The **None** object is often used to represent the absence of a value, or as a default value when no other value is specified. It can be assigned to variables, returned from functions, or used in various operations and comparisons.

Syntax:

```
variable = None
```

Moreover, by assigning a variable to **None**, it depicts that no-value or a null value is represented by the specific variable.

For example:

```
my_var = None
print(type(my_var))
```

Output:

```
<class 'NoneType'>
```

Assigning None:

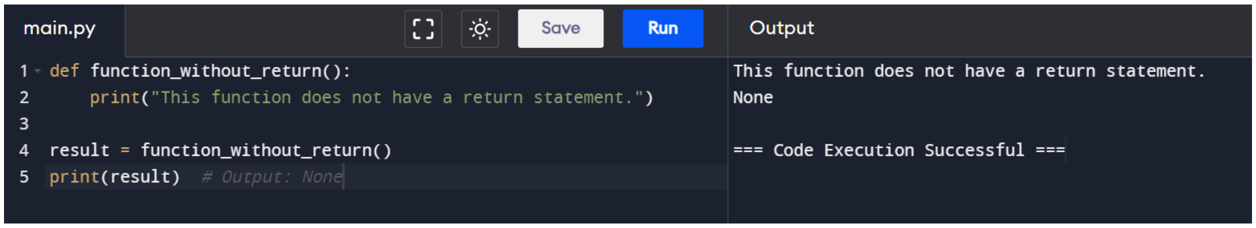
To visualize the **None** object and its data type **NoneType**, we can declare a variable with the value **None** and check its data type using the **type()** function:

```
# Declare a variable with the value None
variable = None

# Check the data type of the variable
print(type(variable))  # Output: <class 'NoneType'>

# Confirm that the variable has the value None
print(variable is None)  # Output: True
```

Try it yourself:

```
def function_without_return():
    print("This function does not have a return statement.")

result = function_without_return()
print(result)  # Output: None
```

Python Data Types



```python
def function_without_return():
    print("This function does not have a return statement.")

result = function_without_return()
print(result)  # Output: None
```

```
This function does not have a return statement.
None

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

a = None

if a is None:

    print("a is None!") # This will be printed

else:

    print("a is not None!")

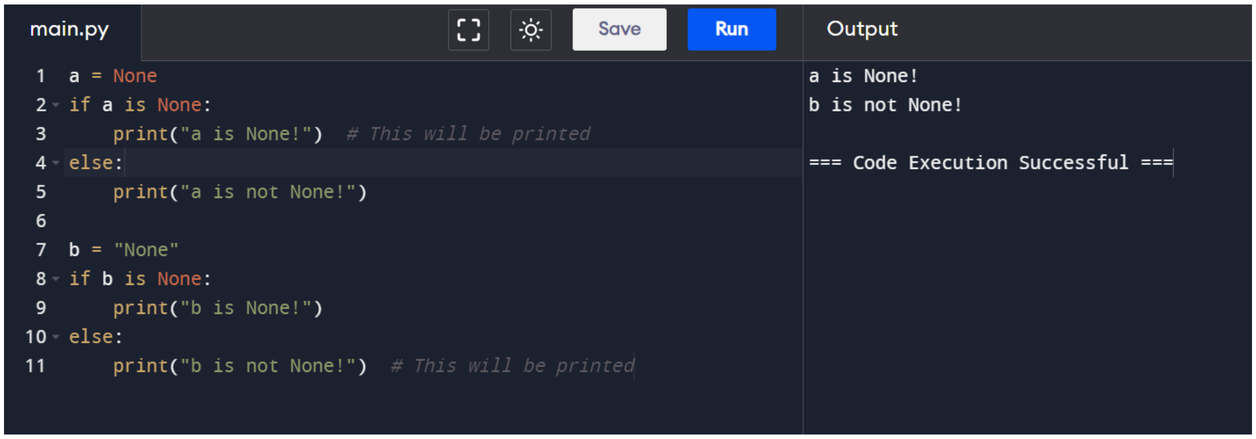
b = "None"

if b is None:

    print("b is None!")

else:

    print("b is not None!") # This will be printed



```python
a = None
if a is None:
    print("a is None!")  # This will be printed
else:
    print("a is not None!")

b = "None"
if b is None:
    print("b is None!")
else:
    print("b is not None!")  # This will be printed
```

```
a is None!
b is not None!

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

Dictionaries

Dictionaries in Python are a built-in data structure that allow you to store and organize data in the form of key-value pairs. They provide an efficient way to associate values with unique keys, making it easy to retrieve and manipulate data.

Python Data Types

Dictionaries are useful for storing related data, such as information contained in an ID or a user profile. They are constructed using curly braces **{}**, with each key-value pair separated by a colon (**:**), and pairs separated by commas (**,**).

Syntax:

```
my_dict = {
  "key1":"value1",
  "key2":"value2"
 }
```

Here's an example of a dictionary in Python:

```
user_profile = {'name': 'Alice', 'age': 28, 'is_active': True, 'email': 'alice@example.com'}
```

What are Keys?

In dictionaries, keys play a crucial role in establishing a mapping between unique identifiers and their associated values, creating a map-like structure. Keys must be immutable data types, such as strings or numbers, because their values cannot change once assigned. This is a requirement to maintain the integrity of the mapping.

Mutable data types like lists cannot be used as keys because their values can change, violating the principle of immutability and making the mapping unreliable.

Dictionary Methods:

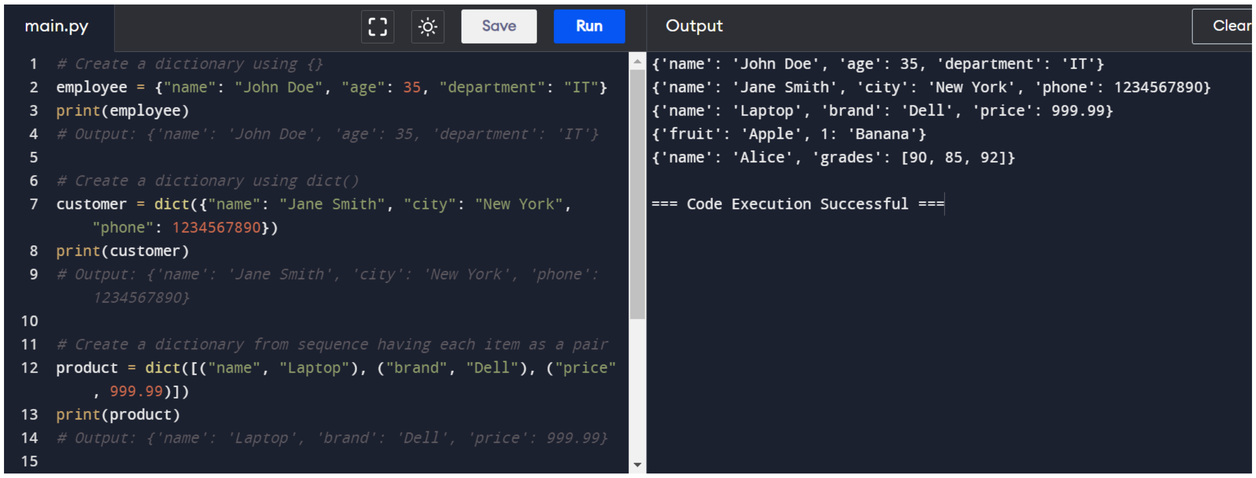
The Python dictionary provides a variety of methods and functions that can be used to easily perform operations on the key-value pairs. Python dictionary methods are listed below:

- clear() -> Removes all the elements from the dictionary
- copy() -> Returns a shallow copy of the specified dictionary
- fromkeys(seq, val) -> Creates a new dictionary with keys from seq and val assigned to all the keys
- get(key) -> Returns the value of the specified key
- has_key() -> Returns True if the key exists in the dictionary, else returns False
- items() -> Returns a list of dictionary’s items in (key, value) format pairs
- keys() -> Returns a list containing all the keys in the dictionary
- pop(key) -> Removes and returns an element from a dictionary having the given key

- popitem() -> Removes and returns an arbitrary item (key, value).
- setdefault(key, val) -> Returns the corresponding value if the key is in the dictionary. If not, inserts the key with a value of val
- update() -> Updates the dictionary by adding key-value pair
- values() -> Returns a list containing all the values in the dictionary

```
# Create a dictionary using {}
employee = {"name": "John Doe", "age": 35, "department": "IT"}
print(employee)
# Output: {'name': 'John Doe', 'age': 35, 'department': 'IT'}

# Create a dictionary using dict()
customer = dict({"name": "Jane Smith", "city": "New York", "phone": 1234567890})
print(customer)
# Output: {'name': 'Jane Smith', 'city': 'New York', 'phone': 1234567890}

# Create a dictionary from sequence having each item as a pair
product = dict([("name", "Laptop"), ("brand", "Dell"), ("price", 999.99)])
print(product)
# Output: {'name': 'Laptop', 'brand': 'Dell', 'price': 999.99}

# Create dictionary with mixed keys
# First key is a string, and second is an integer
mixed_dict = {"fruit": "Apple", 1: "Banana"}
print(mixed_dict)
# Output: {'fruit': 'Apple', 1: 'Banana'}

# Create dictionary with value as a list
student = {"name": "Alice", "grades": [90, 85, 92]}
print(student)
# Output: {'name': 'Alice', 'grades': [90, 85, 92]}
```

Python Data Types



```python
# Create a dictionary using {}
employee = {"name": "John Doe", "age": 35, "department": "IT"}
print(employee)
# Output: {'name': 'John Doe', 'age': 35, 'department': 'IT'}

# Create a dictionary using dict()
customer = dict({"name": "Jane Smith", "city": "New York",
    "phone": 1234567890})
print(customer)
# Output: {'name': 'Jane Smith', 'city': 'New York', 'phone':
    1234567890}

# Create a dictionary from sequence having each item as a pair
product = dict([("name", "Laptop"), ("brand", "Dell"), ("price"
    , 999.99)])
print(product)
# Output: {'name': 'Laptop', 'brand': 'Dell', 'price': 999.99}
```

```
{'name': 'John Doe', 'age': 35, 'department': 'IT'}
{'name': 'Jane Smith', 'city': 'New York', 'phone': 1234567890}
{'name': 'Laptop', 'brand': 'Dell', 'price': 999.99}
{'fruit': 'Apple', 1: 'Banana'}
{'name': 'Alice', 'grades': [90, 85, 92]}

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

Numbers

Python provides built-in support for working with different kinds of numeric data types - integers, floating-point numbers, and complex numbers.

In Python, numbers are treated as objects, and you can create them directly by assigning literal values to variables, like x = 10 (integer), y = 3.14 (float), or z = 2 + 3j (complex). Alternatively, Python offers constructor functions int(), float(), and complex() that you can use to create numbers from other data.

Types of Python Number

There are mainly three types of Python Number:

- Integer
- Floating-point Numbers/ Decimal point number
- Complex Numbers

How to Create a Number Variable in Python?

```
x = 10;
 y = 12.5;
 z = 1 + 2j
```

How to find the type of a Number?

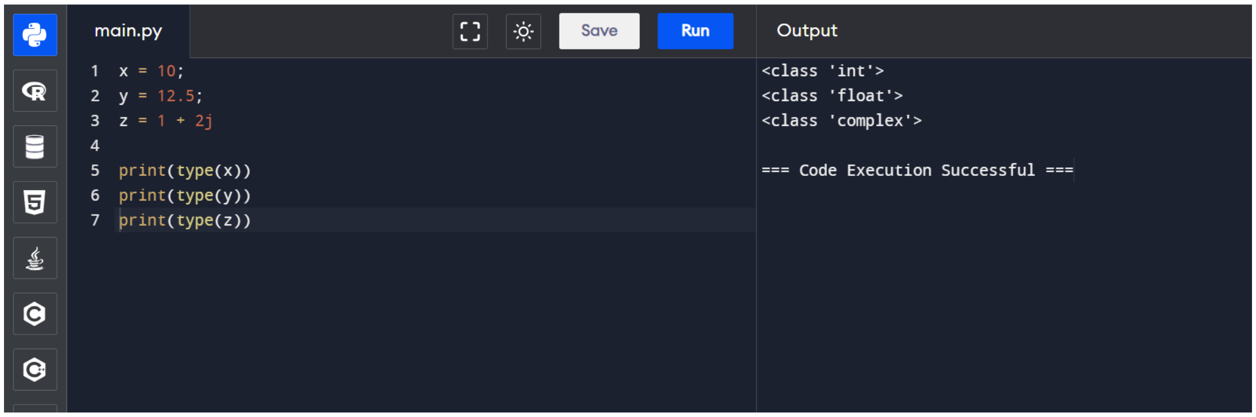
We can find the type of number using the **type()** function.

Try Yourself:

```
x = 10;
y = 12.5;
```

Python Data Types

```
z = 1 + 2j

print(type(x))
print(type(y))
print(type(z))
```



```
x = 10;
y = 12.5;
z = 1 + 2j

print(type(x))
print(type(y))
print(type(z))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>

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

PEMDAS:

Mathematical operations do not strictly follow the PEMDAS (Parentheses, Exponents, Multiplication, Division, Addition, Subtraction) order of operations that is commonly taught in basic mathematics. Instead, Python follows a specific order of operations known as the Python operator precedence.

The order of operations in Python is as follows:

1. Parentheses
2. Exponentiation (**) (right-to-left)
3. Multiplication, Division, Modulus, and Floor Division (*, /, %, //) (left-to-right)
4. Addition and Subtraction (+, -) (left-to-right)

Here are a few key differences from the PEMDAS order:

- Exponentiation is evaluated from right to left, rather than left to right.
- Multiplication, division, modulus, and floor division have the same precedence and are evaluated from left to right.
- There is no separate precedence level for multiplication and division.

For example, consider the following expression:

```
2 + 3 * 4 ** 2
```

In python, this expression will be evaluated as follows:

5. **4 ** 2** is evaluated first (right-to-left), resulting in 16.
6. **3 * 16** is evaluated next (left-to-right), resulting in 48.
7. **2 + 48** is evaluated last (left-to-right), resulting in 50.

So, the final result of this expression in Python is **50**.

If you want to override the default operator precedence, you can use parentheses to explicitly group operations and control the order of evaluation.

It's important to note that while Python's operator precedence differs slightly from the traditional PEMDAS order, it is consistent and well-defined. As long as you are aware of Python's specific order of operations, you can write correct mathematical expressions in your code.

Try Yourself:

b = 0b11011000 # binary

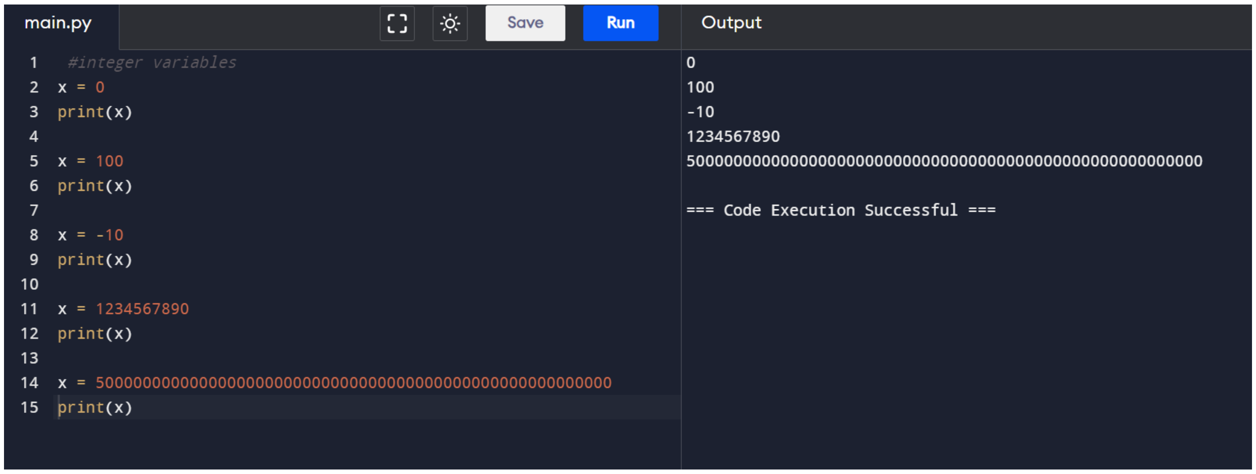
print(b)

o = 0o12 # octal

print(o)

h = 0x12 # hexadecimal

print(h)



```python
#integer variables
x = 0
print(x)

x = 100
print(x)

x = -10
print(x)

x = 1234567890
print(x)

x = 5000000000000000000000000000000000000000000000000000000
print(x)
```

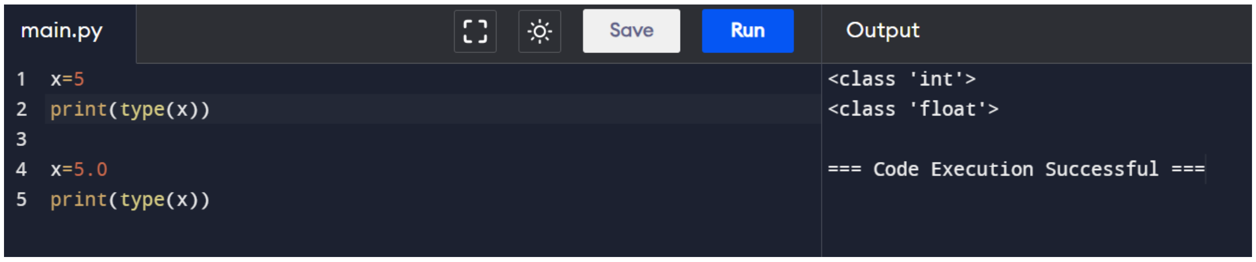
```
0
100
-10
1234567890
5000000000000000000000000000000000000000000000000000000

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

Python Data Types

Python does not allow comma as number delimiter. Use underscore _ as a delimiter instead.

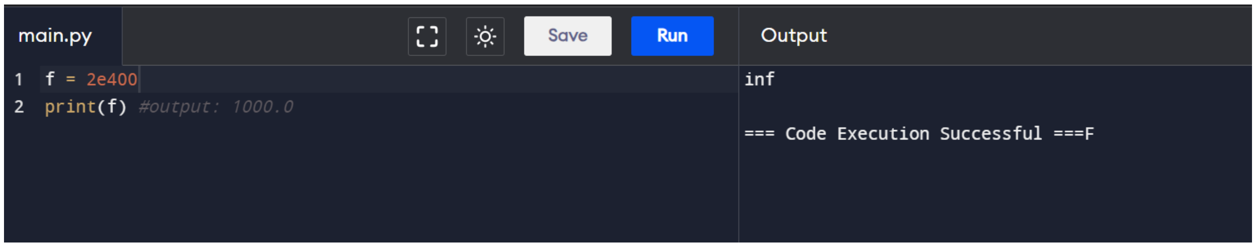
x = 1_234_567_890

print(x) #output: 1234567890



```
x=5
print(type(x))

x=5.0
print(type(x))
```

```
<class 'int'>
<class 'float'>

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



```
f = 2e400
print(f) #output: 1000.0
```

```
inf

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

f = 1e3
print(f) #output: 1000.0

f = 1e5
print(f) #output:100000.0

f = 3.4556789e2
print(f) #output:
print(type(f)) #output:345.56789

Exercise: a = 10 #int

b = 31.54 #float

c = 3j #complex

#convert from int to float
x = float(a)

Python Data Types

```
#convert from float to int
y = int(b)

#convert from int to complex
z = complex(b)

print(x)
print(y)
print(z)

print("x type: ", type(x))
print("y type: ", type(y))
print("z type: ", type(z))
```

Share the output

Thank you