

Python Fundamentals

XE105

v1.4



THINKCYBER

Table of Contents

Introduction to Python.....	4
History of Python	4
Why Choose Python?.....	4
Python 2 vs Python 3	5
Installing Python	5
First Python Program	5
Variables.....	7
What is a Variable?	7
Assigning Values to Variables.....	7
Data Types in Python	9
Type Conversion.....	13
Data Structures	17
Lists: Creation, Methods, and List Comprehensions.....	17
Tuples: Differences from Lists	23
Dictionaries: Creation, Methods, and Dictionary Comprehensions	24
Sets: Operations, Use-Cases.....	28
Basic Functions.....	33
Built-in Functions	33
Print Function.....	37
Slicing and Casting.....	40
Basic String Operations	40
String Methods (split, join, replace, etc.).....	44
String Formatting	49
Loops.....	53
The for Loop	53
The while Loop.....	56
Loop Control Statements (break, continue, pass).....	58
Nested Loops.....	61
Conditions	62
Boolean Expressions	62
Comparison Operators.....	65
Logical Operators	67
The if, elif, and else Statements	69
Nested Conditions.....	71
Input and Output	73

Reading input with input()	73
Writing output with print()	75
Reading and Writing Files.....	77
Working with File Modes	80
Error Handling	83
Common Python Errors.....	83
The try, except Blocks.....	87
The finally Block	90
Raising Exceptions.....	92
Modules	94
What is a Module?	94
Importing Modules	94
Common built-in Modules	97
Working with JSON, CSV	100
Functions.....	102
Defining a Function	102
Function Parameters and Arguments	104
Return Statement.....	107
Variable Scope (local vs global)	110
Lambda Functions.....	112

Introduction to Python

History of Python

Python, a high-level, interpreted programming language, was conceived in the late 1980s by Guido van Rossum in the Netherlands. The language's inception was during the Christmas holidays of 1989, and its implementation began in December. Python was officially released as Python 0.9.0 in February 1991.

The name "Python" was inspired by the British comedy series "Monty Python's Flying Circus," a favorite of van Rossum. This influence can be seen in the playful approach to tutorials and reference materials, which often contain humorous examples.



Python has seen multiple versions since its inception:

- **Python 1.0 (1994):** The first official version.
- **Python 2.0 (2000):** Introduced new features like garbage collection and Unicode support. The 2.x series continued until 2.7.
- **Python 3.0 (2008):** A major overhaul, it was designed to rectify fundamental design flaws. This version was not backward compatible with Python 2.

Why Choose Python?

Python has grown in popularity for several reasons:

1. **Readability:** Python's syntax is clear and concise, which makes it easy to read and write.
2. **Versatility:** It's used in web development, data analysis, artificial intelligence, scientific computing, and more.
3. **Community:** A vast and active community means abundant resources, libraries, and frameworks.
4. **Open Source:** Being open-source encourages collaboration and ensures the language is continually improving.
5. **Cross-Platform:** Python is portable and can run on various operating systems.

Python 2 vs Python 3

While both versions coexisted for a time, it's essential to understand their differences:

- **Print Statement vs. Function:** In Python 2, **print** is a statement. In Python 3, it's a function: **print()**.
- **Integer Division:** In Python 2, dividing two integers results in an integer. In Python 3, it results in a float.
- **Unicode Support:** Python 3 supports Unicode by default, making it easier to work with non-ASCII text.
- **Standard Library Changes:** Some libraries in Python 2 were reorganized in Python 3, leading to different import paths.

It's worth noting that Python 2 reached its end of life on January 1, 2020. This means no more updates, not even security patches. As such, it's recommended to use Python 3 for all new projects.

Installing Python


To install Python:

1. **Visit the Official Website:** Go to [Python's official website](#).
2. **Download the Installer:** Choose the version suitable for your OS (Windows, macOS, Linux).
3. **Run the Installer:** Follow the on-screen instructions. Ensure you check the box that says "Add Python to PATH" to access Python from the command line.
4. **Verify Installation:** Open a terminal or command prompt and type **python --version**. This should display the installed Python version.

First Python Program

Once Python is installed, you can write your first program:

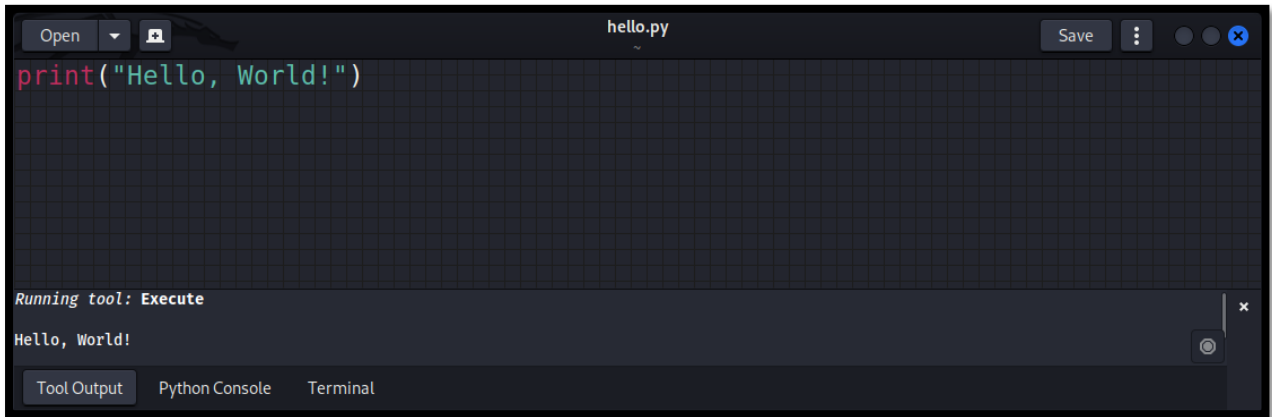
1. **Using the Interactive Shell:** Open the Python shell by typing **python** in the command prompt or terminal. Then type **print("Hello, World!")** and press Enter.



```
kali@kali: ~  
File Actions Edit View Help  
(kali@kali)~  
$ python  
Python 3.11.6 (main, Oct 8 2023, 05:06:43) [GCC 13.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("Hello, World!")  
Hello, World!  
>>> 
```

2. Using a Script:

- Create a new file named **hello.py**.
- Open it in a text editor and type **print("Hello, World!")**.
- Save and close the file.
- In the terminal or command prompt, navigate to the directory containing the file and type **python hello.py**.



The screenshot shows a code editor window titled "hello.py". The editor contains the following Python code:

```
print("Hello, World!")
```

Below the code editor, there is a "Running tool: Execute" section. The output of the execution is displayed as:

```
Hello, World!
```

At the bottom of the window, there are three tabs: "Tool Output", "Python Console", and "Terminal".

You should see the message "Hello, World!" displayed, indicating that your program ran successfully.

Variables

What is a Variable?

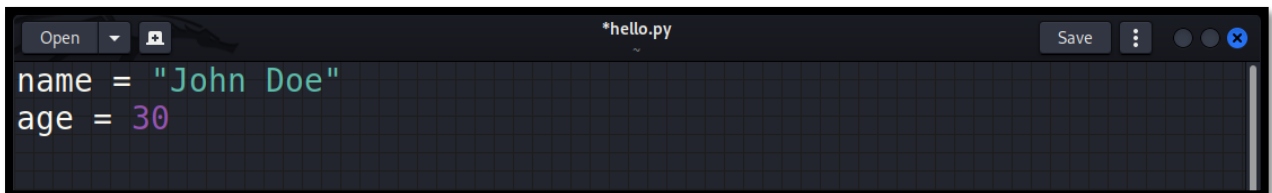
In programming, a variable is a symbolic name associated with a value. Think of it as a container or storage location that holds data. This data can be of various types, such as numbers, strings, lists, or more complex types. The primary purpose of a variable is to store information for later use, allowing for manipulation, retrieval, and other operations.

Characteristics of Variables:

1. **Mutable:** Variables can change their value over time.
2. **Typed:** Every variable in Python has a data type, which determines the kind of value it can hold (e.g., integer, string, list).
3. **Identified by Name:** Variables have unique names, often referred to as "identifiers," which are used to access their stored values.

Example:

Consider the analogy of a mailbox. A mailbox holds letters (data) and has a unique identifier (its address). Similarly, a variable holds data and has a unique name.

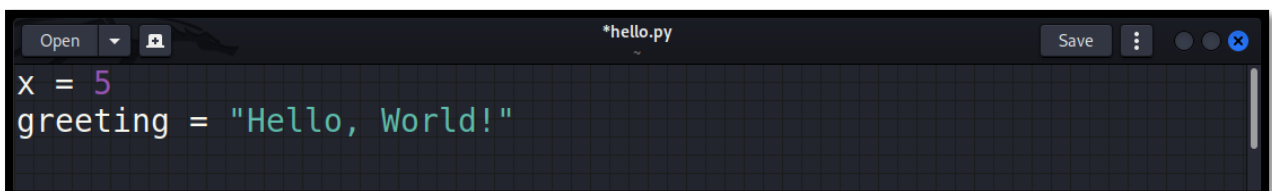
A screenshot of a code editor window titled '*hello.py'. The editor contains two lines of Python code: 'name = "John Doe"' and 'age = 30'. The text is color-coded: 'name' is blue, 'John Doe' is green, 'age' is blue, and '30' is purple. The window has a dark background and standard window controls (Open, Save, Close) at the top.

In the above example, **name** and **age** are variables. The variable **name** holds the string "John Doe", and the variable **age** holds the number 30.

Assigning Values to Variables

Assigning a value to a variable is a fundamental operation in programming. In Python, the equals sign (=) is used for assignment. The variable name is placed on the left, and the value to be assigned is placed on the right.

Basic Assignment:

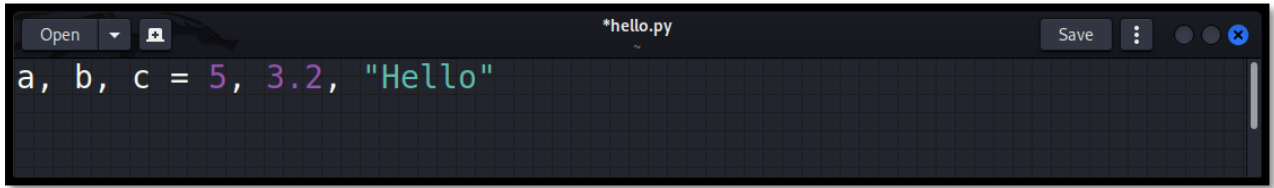
A screenshot of a code editor window titled '*hello.py'. The editor contains two lines of Python code: 'x = 5' and 'greeting = "Hello, World!"'. The text is color-coded: 'x' is blue, '5' is purple, 'greeting' is blue, and 'Hello, World!' is green. The window has a dark background and standard window controls (Open, Save, Close) at the top.

Assigns the integer value 5 to the variable x.

Assigns the string "Hello, World!" to the variable greeting.

Multiple Assignment:

Python allows you to assign values to multiple variables in a single line:

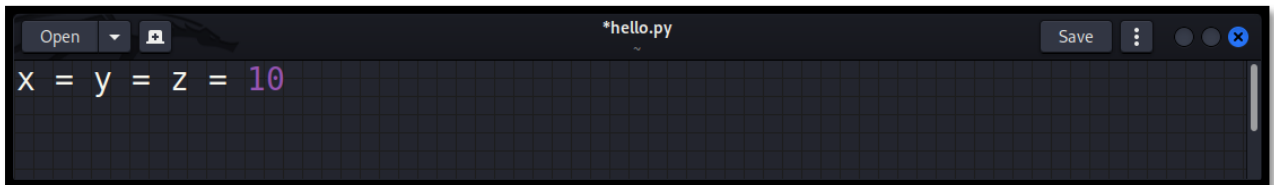
A screenshot of a code editor window titled '*hello.py'. The editor contains the Python code: `a, b, c = 5, 3.2, "Hello"`. The code is color-coded: 'a, b, c' is in white, '=' is in light blue, '5' is in purple, '3.2' is in green, and '"Hello"' is in light green. The editor has a dark background and a grid pattern.

This is equivalent to:

- `a = 5`
- `b = 3.2`
- `c = "Hello"`

Assigning the Same Value to Multiple Variables:

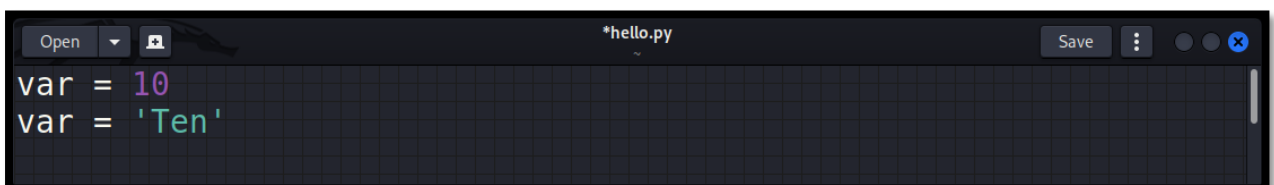
You can also assign the same value to multiple variables:

A screenshot of a code editor window titled '*hello.py'. The editor contains the Python code: `x = y = z = 10`. The code is color-coded: 'x = y = z' is in white and '10' is in purple. The editor has a dark background and a grid pattern.

This is equivalent to: `# x = 10 # y = 10 # z = 10`

Dynamic Typing:

One of Python's features is dynamic typing, which means you don't have to declare a variable's type explicitly. The type is determined at runtime based on the assigned value:

A screenshot of a code editor window titled '*hello.py'. The editor contains two lines of Python code: `var = 10` and `var = 'Ten'`. The code is color-coded: 'var = 10' is in white and '10' is in purple; 'var = 'Ten'' is in white and 'Ten' is in light green. The editor has a dark background and a grid pattern.

```
var = 10      # 'var' is an integer
var = "Ten"   # Now, 'var' is a string
```


Data Types in Python

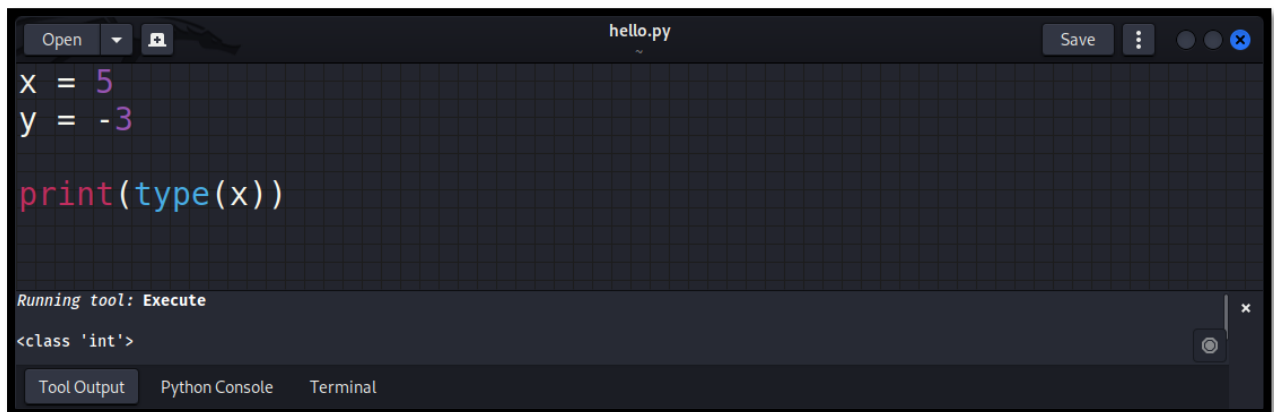
In Python, every value has an associated data type. A data type defines the kind of value a variable can hold, such as integer, float, string, etc. Understanding these data types is crucial as they dictate the operations that can be performed on the values and how these values are stored in memory.

Fundamental Data Types

Integer (int)

Integers are whole numbers, both positive and negative.

Example:



```
hello.py
x = 5
y = -3

print(type(x))

Running tool: Execute
<class 'int'>
```

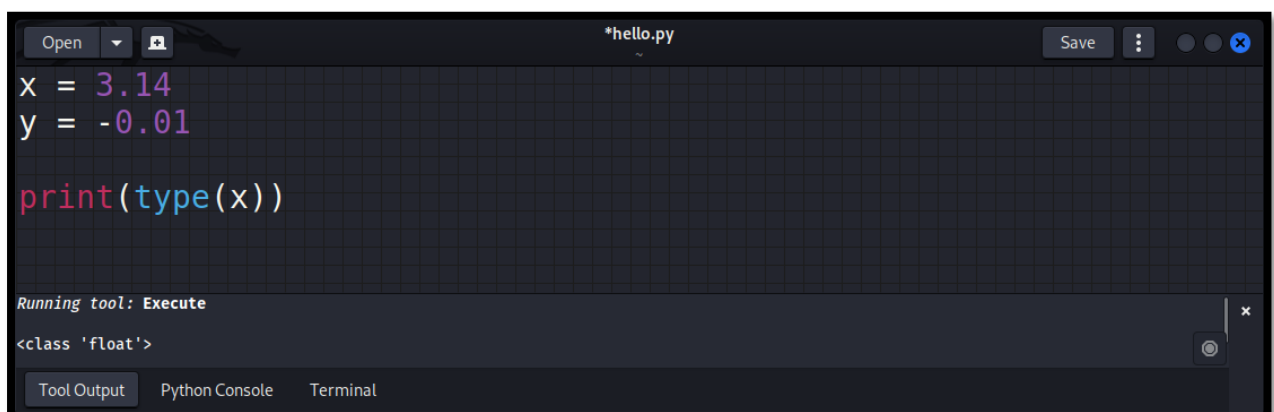
The screenshot shows a code editor window titled 'hello.py' with the following code: `x = 5`, `y = -3`, and `print(type(x))`. Below the code, a terminal window displays the output `<class 'int'>`. The interface includes 'Open', 'Save', and 'Terminal' buttons.

Output: <class 'int'>

Floating Point (float)

Floating-point numbers represent real numbers and are written with a decimal point.

Example:



```
*hello.py
x = 3.14
y = -0.01

print(type(x))

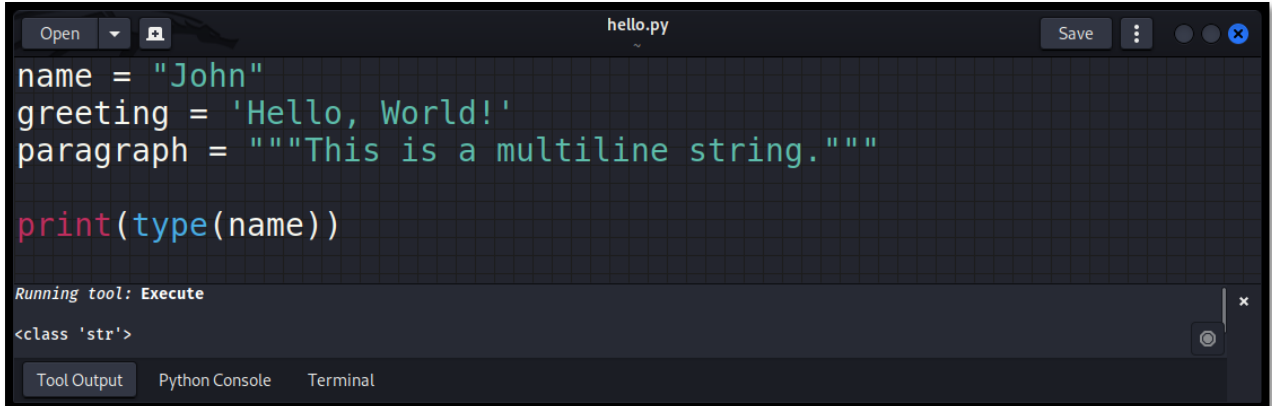
Running tool: Execute
<class 'float'>
```

The screenshot shows a code editor window titled '*hello.py' with the following code: `x = 3.14`, `y = -0.01`, and `print(type(x))`. Below the code, a terminal window displays the output `<class 'float'>`. The interface includes 'Open', 'Save', and 'Terminal' buttons.

String (str)

Strings are sequences of characters. In Python, strings can be enclosed in single ('), double ("), or triple ("'' or ''''') quotes.

Example:



```
name = "John"
greeting = 'Hello, World!'
paragraph = """This is a multiline string."""

print(type(name))
```

Running tool: Execute

<class 'str'>

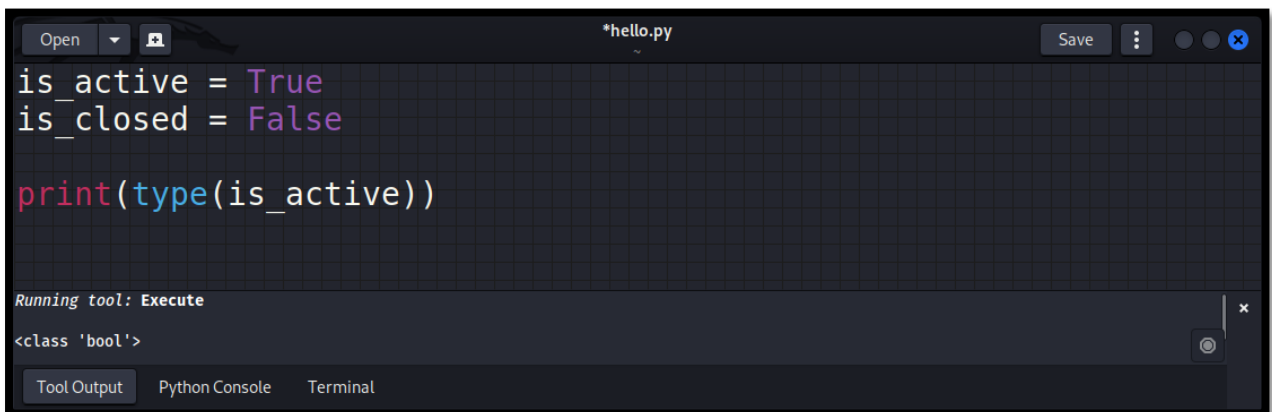
Tool Output Python Console Terminal

Output: <class 'str'>

Boolean (bool)

Booleans represent one of two values: **True** or **False**.

Example:



```
is_active = True
is_closed = False

print(type(is_active))
```

Running tool: Execute

<class 'bool'>

Tool Output Python Console Terminal

Output: <class 'bool'>

Compound Data Types

List

A list is an ordered collection of items, which can be of any type. Lists are mutable, meaning their content can change.

Example:



```
hello.py
Open Save
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]

print(type(fruits))

Running tool: Execute
<class 'list'>
Tool Output Python Console Terminal
```

Output: <class 'list'>

Tuple

A tuple is similar to a list but is immutable, meaning its content cannot be changed after creation.

Example:



```
hello.py
Open Save
coordinates = (4.0, 5.0)
colors = ("red", "green", "blue")

print(type(coordinates))

Running tool: Execute
<class 'tuple'>
Tool Output Python Console Terminal
```

Output: <class 'tuple'>

Dictionary (dict)

A dictionary is an unordered collection of key-value pairs. Keys must be unique.

Example:

```
person = { "name": "Alice", "age": 30, "city": "New York" }  
  
print(type(person))
```

Running tool: Execute

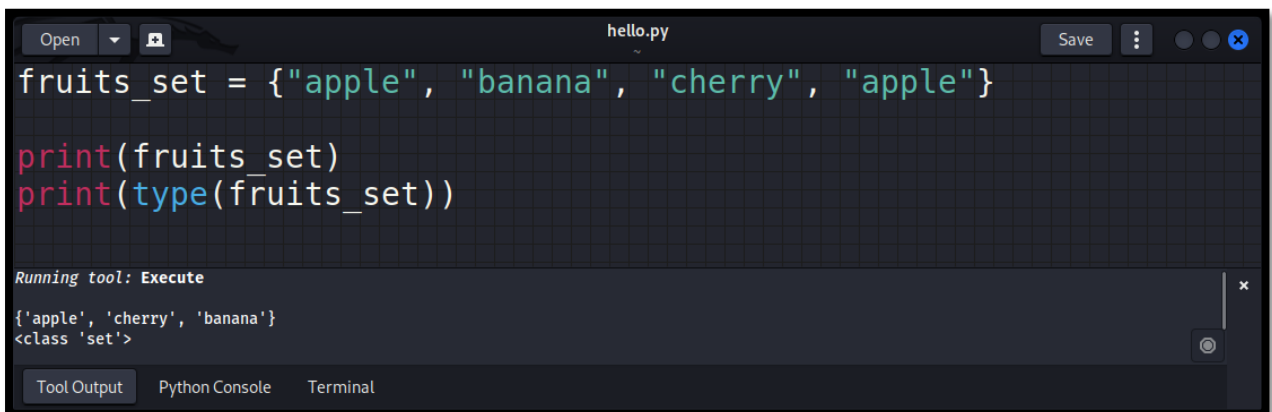
```
<class 'dict'>
```

Tool Output Python Console Terminal

Output: <class 'dict'>

Set

A set is an unordered collection of unique items.

Example:

```
fruits_set = {"apple", "banana", "cherry", "apple"}  
  
print(fruits_set)  
print(type(fruits_set))
```

Running tool: Execute

```
{'apple', 'cherry', 'banana'}  
<class 'set'>
```

Tool Output Python Console Terminal

Output:

```
{'apple', 'banana', 'cherry'}  
<class 'set'>
```

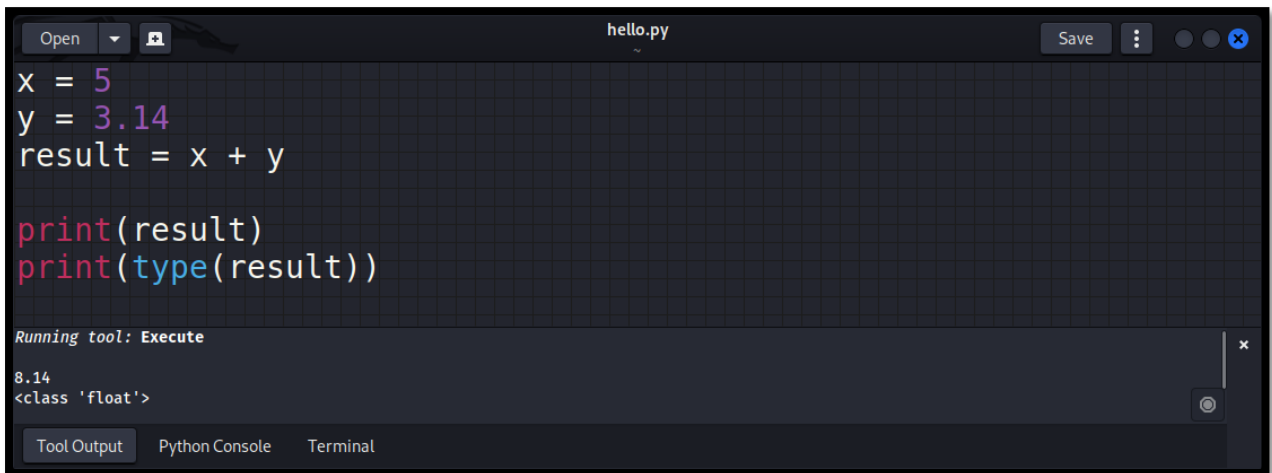
Type Conversion

Type conversion, often referred to as type casting, is the process of converting a value from one data type to another. In Python, while some conversions happen implicitly (automatic type conversion), there are times when you'll need to perform explicit type conversion to meet specific requirements.

Implicit Type Conversion

Python automatically converts one data type to another without the programmer's intervention in certain scenarios. This is known as implicit type conversion or coercion.

Example:



```
hello.py
x = 5
y = 3.14
result = x + y

print(result)
print(type(result))

Running tool: Execute
8.14
<class 'float'>
```

In the above example, the integer `x` is automatically converted to a float to perform the addition with another float `y`.

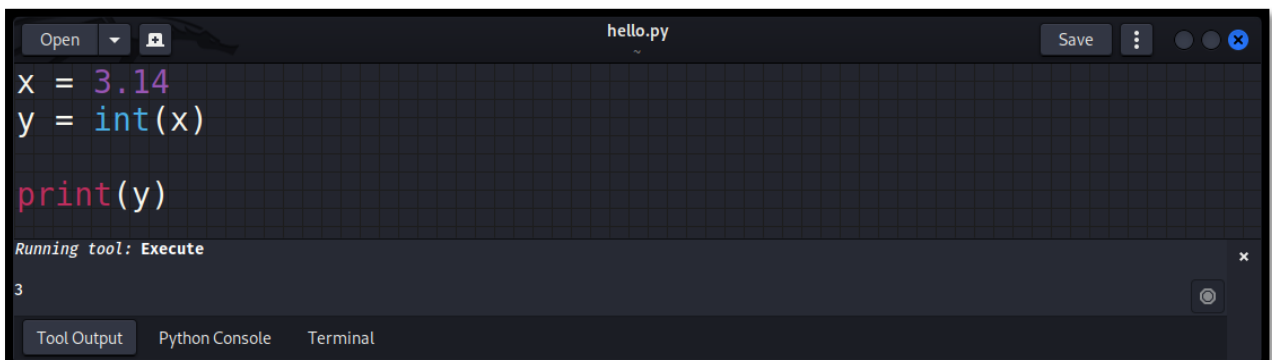
Explicit Type Conversion

When you manually change the data type of a value, it's called explicit type conversion. Python provides built-in functions for this purpose.

`int()`

Converts a value to an integer. It can't convert complex numbers or strings with non-numeric values.

Example:



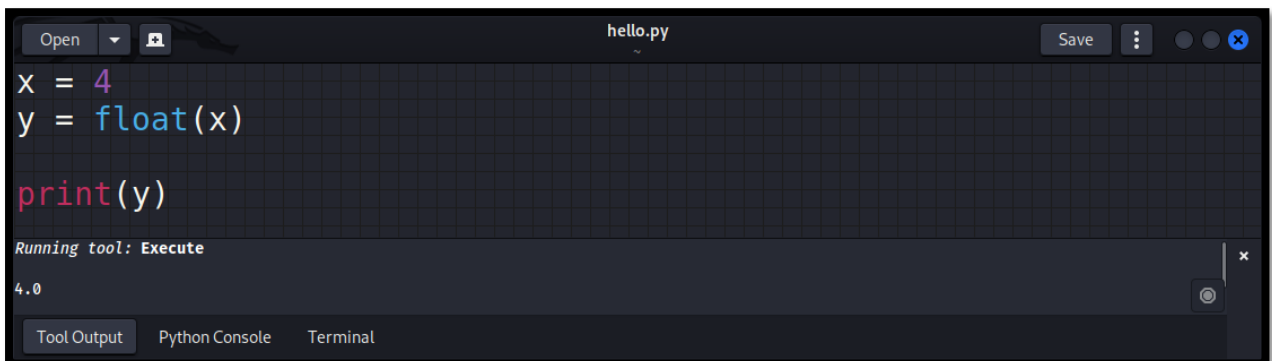
```
hello.py
x = 3.14
y = int(x)

print(y)

Running tool: Execute
3
```

float()

Converts a value to a floating-point number.

Example:

```
Open hello.py Save
```

```
x = 4  
y = float(x)  
  
print(y)
```

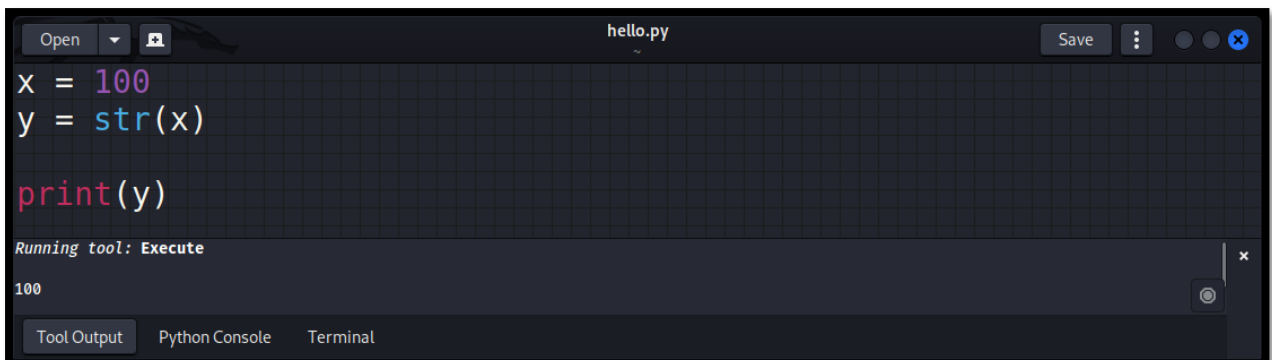
Running tool: Execute

```
4.0
```

Tool Output Python Console Terminal

str()

Converts a value to a string.

Example:

```
Open hello.py Save
```

```
x = 100  
y = str(x)  
  
print(y)
```

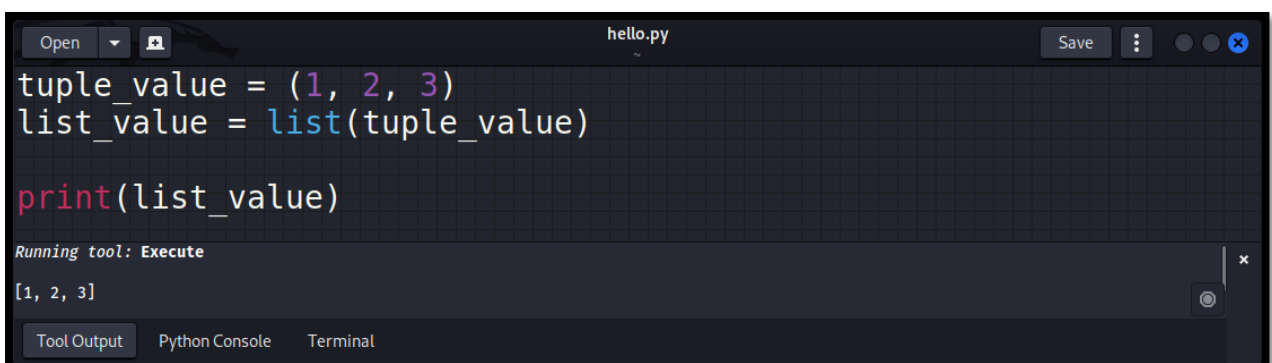
Running tool: Execute

```
100
```

Tool Output Python Console Terminal

list()

Converts a value to a list.

Example:

```
Open hello.py Save
```

```
tuple_value = (1, 2, 3)  
list_value = list(tuple_value)  
  
print(list_value)
```

Running tool: Execute

```
[1, 2, 3]
```

Tool Output Python Console Terminal

tuple()

Converts a value to a tuple.

Example:

```
Open hello.py Save
list_value = [4, 5, 6]
tuple_value = tuple(list_value)

print(tuple_value)

Running tool: Execute
(4, 5, 6)
Tool Output Python Console Terminal
```

The screenshot shows a code editor window titled 'hello.py'. The code defines a list 'list_value' with elements [4, 5, 6], converts it to a tuple 'tuple_value' using the 'tuple()' function, and prints the result. The output in the console is '(4, 5, 6)'. The editor interface includes 'Open', 'Save', and window control buttons at the top, and tabs for 'Tool Output', 'Python Console', and 'Terminal' at the bottom.

set()

Converts a value to a set.

Example:

```
Open hello.py Save
list_value = [1, 2, 2, 3, 4]
set_value = set(list_value)

print(set_value)

Running tool: Execute
{1, 2, 3, 4}
Tool Output Python Console Terminal
```

The screenshot shows a code editor window titled 'hello.py'. The code defines a list 'list_value' with elements [1, 2, 2, 3, 4], converts it to a set 'set_value' using the 'set()' function, and prints the result. The output in the console is '{1, 2, 3, 4}'. The editor interface includes 'Open', 'Save', and window control buttons at the top, and tabs for 'Tool Output', 'Python Console', and 'Terminal' at the bottom.

dict()

Converts a sequence of key-value pairs to a dictionary.

Example:

```
Open hello.py Save
key_value_pairs = [("name", "Alice"), ("age", 30)]
dict_value = dict(key_value_pairs)

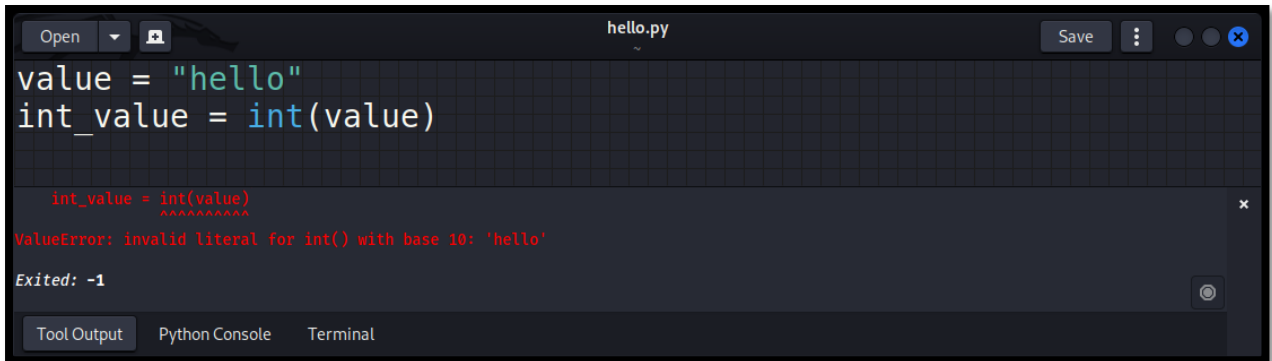
print(dict_value)

Running tool: Execute
{'name': 'Alice', 'age': 30}
Tool Output Python Console Terminal
```

The screenshot shows a code editor window titled 'hello.py'. The code defines a list of key-value pairs 'key_value_pairs' as [("name", "Alice"), ("age", 30)], converts it to a dictionary 'dict_value' using the 'dict()' function, and prints the result. The output in the console is {'name': 'Alice', 'age': 30}. The editor interface includes 'Open', 'Save', and window control buttons at the top, and tabs for 'Tool Output', 'Python Console', and 'Terminal' at the bottom.

Caution with Type Conversion

While type conversion is powerful, it's essential to be cautious. Not all values can be converted to all data types. For instance, trying to convert a non-numeric string to an integer or float will raise an error.

Example:

```
hello.py
value = "hello"
int_value = int(value)

int_value = int(value)
^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'hello'

Exited: -1
```

The screenshot shows a code editor window titled 'hello.py' with the following code: `value = "hello"` and `int_value = int(value)`. Below the code, a red error message is displayed: `int_value = int(value)` followed by `^^^^^^^^^^` and `ValueError: invalid literal for int() with base 10: 'hello'`. At the bottom, it says `Exited: -1`. The IDE interface includes a menu bar with 'Open' and 'Save', and a status bar with 'Tool Output', 'Python Console', and 'Terminal'.

This will raise a ValueError.

Data Structures

Lists: Creation, Methods, and List Comprehensions

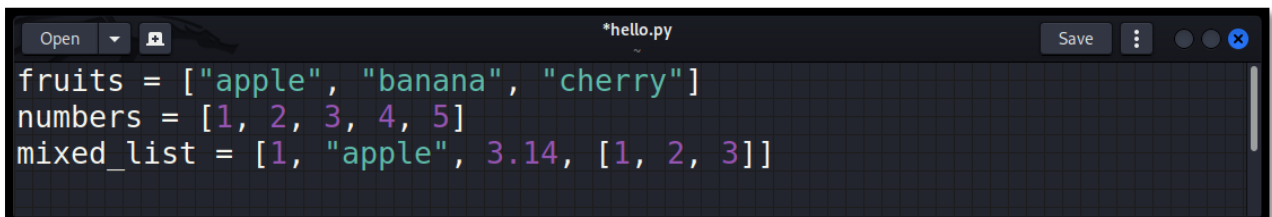
In Python, a list is a mutable, ordered collection of items. Lists can contain elements of different data types, including other lists. Being one of the most versatile data structures in Python, lists find applications in numerous scenarios, from simple data storage to complex data manipulation tasks.

Creating Lists

Using Square Brackets

The most common way to create a list is by enclosing a comma-separated sequence of values in square brackets [].

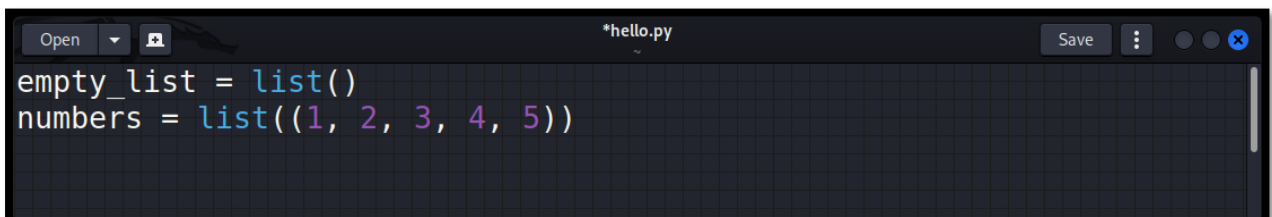
Example:

A screenshot of a code editor window titled '*hello.py'. The editor contains three lines of Python code: 'fruits = ["apple", "banana", "cherry"]', 'numbers = [1, 2, 3, 4, 5]', and 'mixed_list = [1, "apple", 3.14, [1, 2, 3]]'. The code is syntax-highlighted with colors: strings are green, integers are purple, and floats are blue. The editor has a dark background and standard window controls (Open, Save, Close) at the top.

Using the list() Constructor

You can also create a list using the **list()** constructor.

Example: Creating a list from a tuple

A screenshot of a code editor window titled '*hello.py'. The editor contains two lines of Python code: 'empty_list = list()' and 'numbers = list((1, 2, 3, 4, 5))'. The code is syntax-highlighted with colors: the list constructor 'list()' is blue, and the tuple elements are purple. The editor has a dark background and standard window controls (Open, Save, Close) at the top.

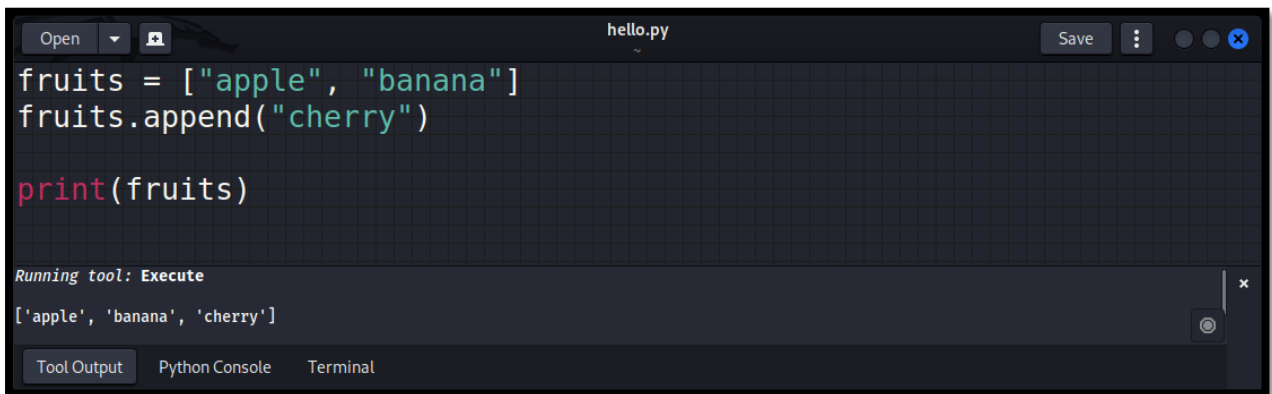
List Methods

Python lists come with a variety of built-in methods that allow for easy manipulation and querying.

append()

Adds an item to the end of the list.

Example:



```
hello.py
Open Save
fruits = ["apple", "banana"]
fruits.append("cherry")

print(fruits)

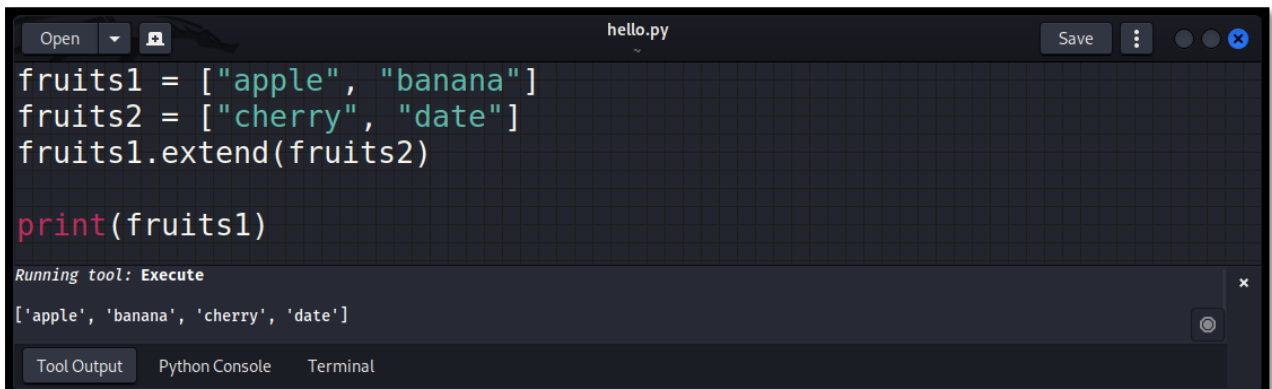
Running tool: Execute
['apple', 'banana', 'cherry']
Tool Output Python Console Terminal
```

Output: ['apple', 'banana', 'cherry']

extend()

Adds the elements of a list (or any iterable) to the end of the current list.

Example:



```
hello.py
Open Save
fruits1 = ["apple", "banana"]
fruits2 = ["cherry", "date"]
fruits1.extend(fruits2)

print(fruits1)

Running tool: Execute
['apple', 'banana', 'cherry', 'date']
Tool Output Python Console Terminal
```

Output: ['apple', 'banana', 'cherry', 'date']

insert()

Inserts an item at a specified position.

Example:

```
hello.py
fruits = ["apple", "cherry"]
fruits.insert(1, "banana")

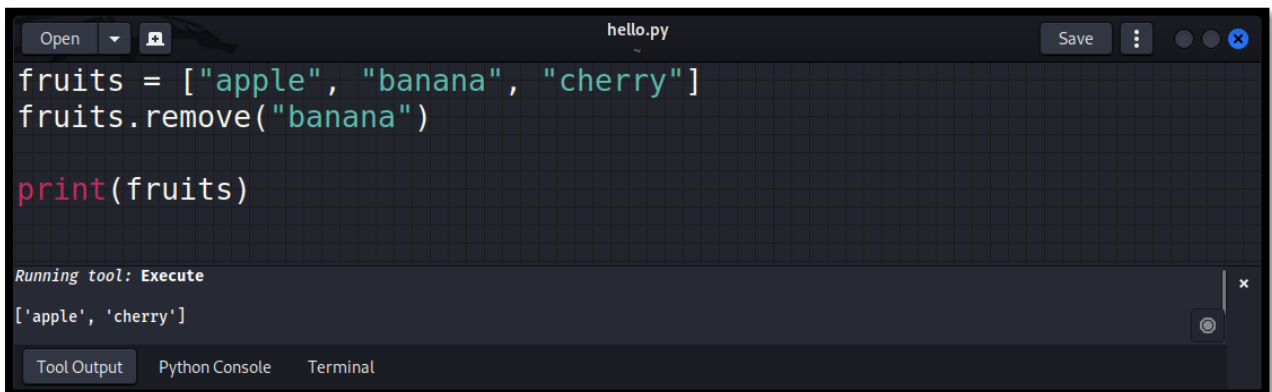
print(fruits)

Running tool: Execute
['apple', 'banana', 'cherry']
```

Output: ['apple', 'banana', 'cherry']

remove()

Removes the first occurrence of a specified item.

Example:

```
hello.py
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")

print(fruits)

Running tool: Execute
['apple', 'cherry']
```

Output: ['apple', 'cherry']

pop()

Removes and returns the item at a specified position. If no index is specified, it removes the last item.

Example:

```
hello.py
fruits = ["apple", "banana", "cherry"]
popped_fruit = fruits.pop(1)

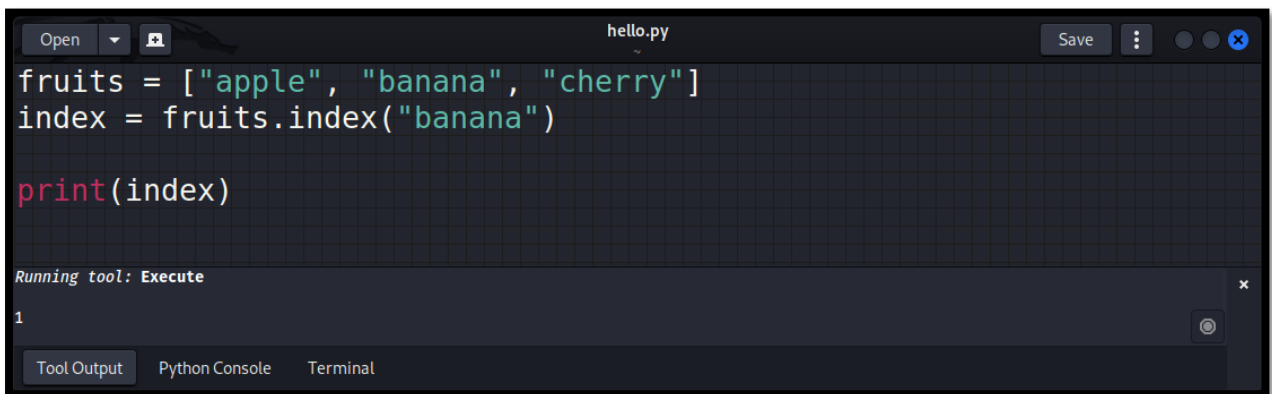
print(popped_fruit)

Running tool: Execute
banana
```

Output: 'banana' print(fruits) # Output: ['apple', 'cherry']

index()

Returns the index of the first occurrence of a specified item.

Example:

```
hello.py
fruits = ["apple", "banana", "cherry"]
index = fruits.index("banana")

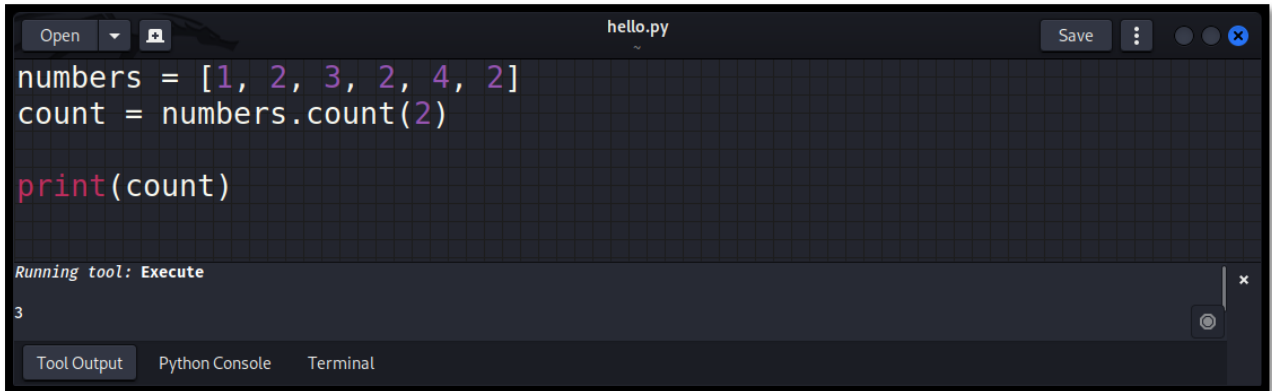
print(index)

Running tool: Execute
1
```

Output: 1

count()

Returns the number of occurrences of a specified item.

Example:A screenshot of a Python IDE window titled 'hello.py'. The code in the editor is:

```
numbers = [1, 2, 3, 2, 4, 2]
count = numbers.count(2)

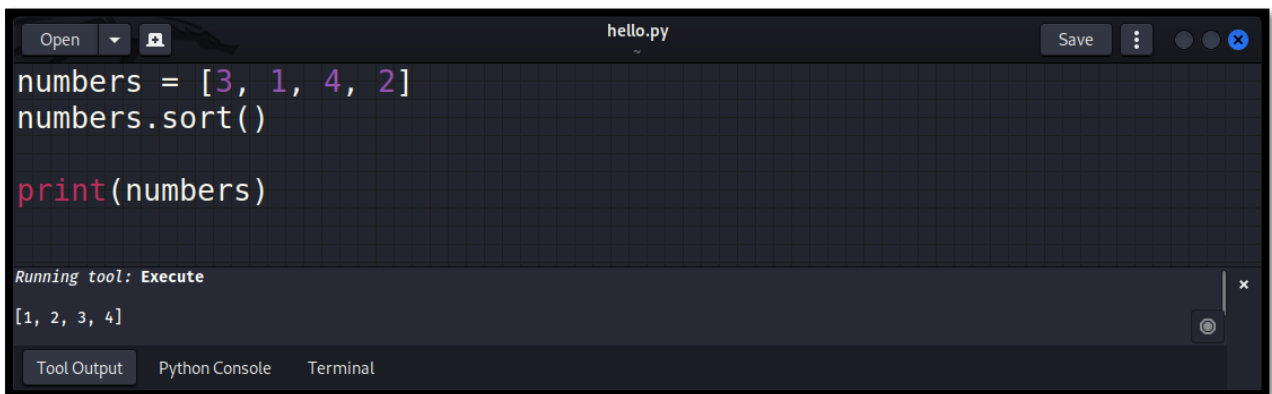
print(count)
```

The IDE has a 'Running tool: Execute' button. Below the code, the output '3' is displayed in a console area. At the bottom, there are tabs for 'Tool Output', 'Python Console', and 'Terminal'.

Output: 3

sort()

Sorts the list in ascending order. For descending order, you can use the **reverse=True** argument.

Example:A screenshot of a Python IDE window titled 'hello.py'. The code in the editor is:

```
numbers = [3, 1, 4, 2]
numbers.sort()

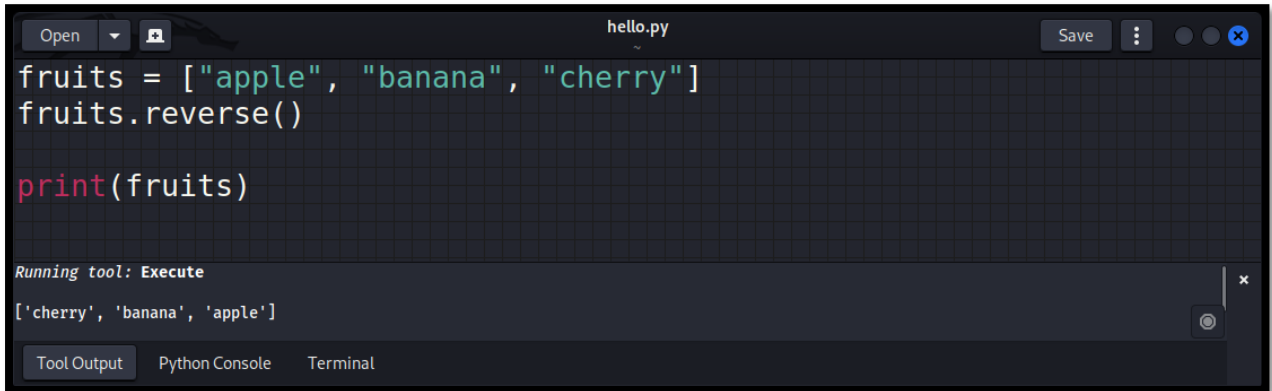
print(numbers)
```

The IDE has a 'Running tool: Execute' button. Below the code, the output '[1, 2, 3, 4]' is displayed in a console area. At the bottom, there are tabs for 'Tool Output', 'Python Console', and 'Terminal'.

Output: [1, 2, 3, 4]

reverse()

Reverses the order of the list.

Example:A screenshot of a Python IDE window titled 'hello.py'. The code in the editor is:

```
fruits = ["apple", "banana", "cherry"]
fruits.reverse()

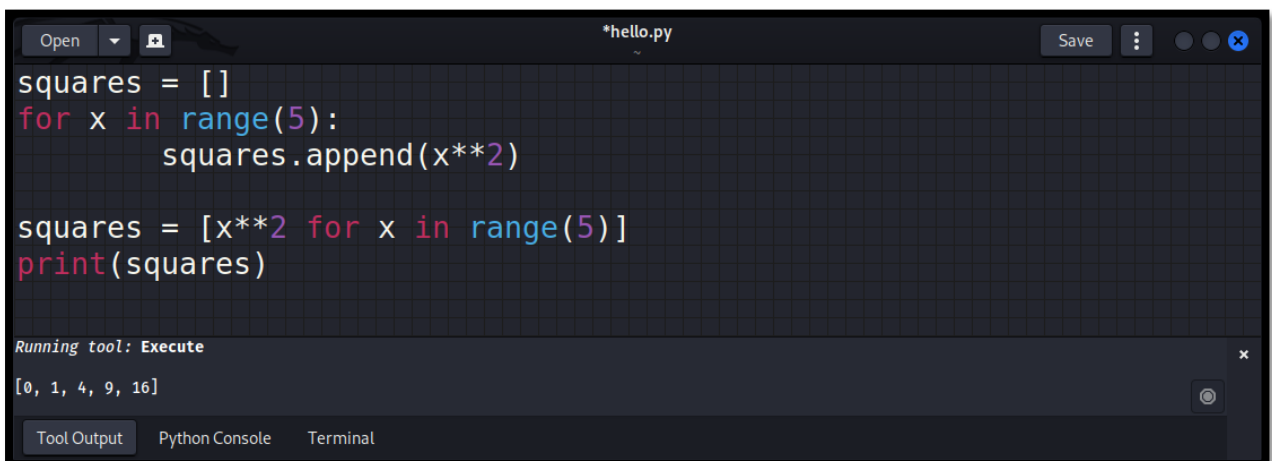
print(fruits)
```

The IDE's output console shows the result of running the code: `['cherry', 'banana', 'apple']`. The console also includes a 'Running tool: Execute' label and tabs for 'Tool Output', 'Python Console', and 'Terminal'.

Output: ['cherry', 'banana', 'apple']

List Comprehensions

List comprehensions provide a concise way to create lists based on existing lists or iterables. They can also incorporate conditionals, making them a powerful tool for list generation.

Example:A screenshot of a Python IDE window titled '*hello.py'. The code in the editor is:

```
squares = []
for x in range(5):
    squares.append(x**2)

squares = [x**2 for x in range(5)]
print(squares)
```

The IDE's output console shows the result of running the code: `[0, 1, 4, 9, 16]`. The console also includes a 'Running tool: Execute' label and tabs for 'Tool Output', 'Python Console', and 'Terminal'.

Output: [0, 1, 4, 9, 16]

Tuples: Differences from Lists

In Python, a tuple is an ordered collection of items, similar to a list. However, unlike lists, tuples are immutable, meaning their content cannot be modified after creation. Tuples are often used to represent fixed collections of items or to ensure data integrity.

Creating Tuples

Tuples are created by enclosing a comma-separated sequence of values in parentheses ().

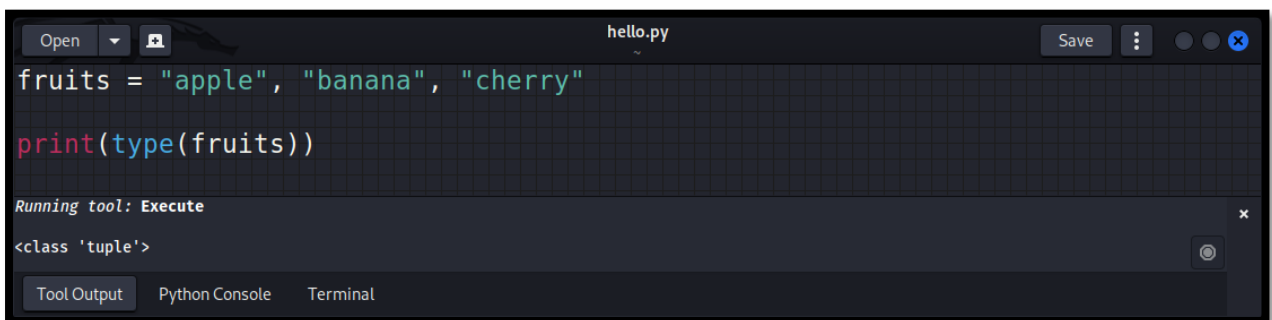
Example:

A screenshot of a code editor window titled '*hello.py'. The code defines three tuples: 'fruits' containing three strings, 'coordinates' containing two floats, and 'single_element_tuple' containing one integer with a trailing comma.

```
fruits = ("apple", "banana", "cherry")
coordinates = (4.0, 5.0)
single_element_tuple = (5,)
```

Note the trailing comma for single-element tuples.

You can also create a tuple without parentheses, known as tuple packing:

A screenshot of a code editor window titled 'hello.py'. The code defines a tuple 'fruits' without parentheses and prints its type. Below the code, a terminal window shows the output: '<class 'tuple''.

```
fruits = "apple", "banana", "cherry"
print(type(fruits))
```

Running tool: Execute

```
<class 'tuple'>
```

Differences Between Tuples and Lists

1. Mutability:

- **List:** Lists are mutable, meaning you can modify their content (add, remove, or change items).
- **Tuple:** Tuples are immutable, meaning once they are created, you cannot modify their content.

2. Syntax:

- **List:** Lists are defined using square brackets [].
- **Tuple:** Tuples are defined using parentheses ().

3. Methods:

- **List:** Lists have several built-in methods like **append()**, **remove()**, and **extend()**.
- **Tuple:** Tuples have a limited set of methods. The most commonly used are **count()** and **index()**.

4. Performance:

- **List:** Due to their mutability, lists generally have a slightly larger memory overhead.

- **Tuple:** Tuples are more memory-efficient and can be faster for iteration due to their immutability.

5. Use Cases:

- **List:** Lists are more common for collections that might need to change over the lifetime of the program.
- **Tuple:** Tuples are used when the data shouldn't change, such as representing fixed collections or as dictionary keys.

When to Use Tuples

1. **Data Integrity:** If you want to ensure data remains unchanged throughout the program, use a tuple. Its immutability guarantees the data's integrity.
2. **Dictionary Keys:** Since dictionary keys need to be hashable and immutable, tuples can be used as dictionary keys, while lists cannot.
3. **Function Arguments and Return Values:** Tuples are often used in function arguments and return values when a function needs to accept or return a fixed collection of items.
4. **Performance:** If you're defining a constant set of values and all you're ever going to do is iterate through it, a tuple can be more efficient than a list.

Dictionaries: Creation, Methods, and Dictionary Comprehensions

In Python, a dictionary is an unordered collection of key-value pairs. Each key must be unique, and it can be of any immutable type, such as strings, numbers, or tuples. Dictionaries are mutable, which means you can add, remove, or modify key-value pairs. They are often used for tasks like data retrieval, where you can quickly find a value based on its key.

Creating Dictionaries

Using Curly Braces

The most common way to create a dictionary is by enclosing a comma-separated sequence of key-value pairs in curly braces `{}`. Key-value pairs are separated by colons.

Example:

A screenshot of a code editor window titled '*hello.py'. The window has a dark background and a light-colored text area. The code displayed is:

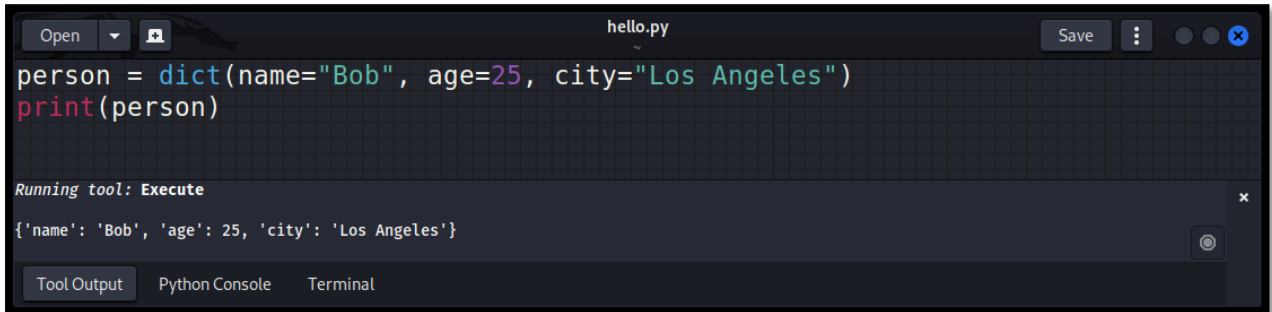
```
person = { "name": "Alice", "age": 30, "city": "New York" }
```

 The code is color-coded: strings are in green, integers are in purple, and the dictionary structure is in white. The window has standard OS window controls (Open, Save, Close) and a search icon.

Using the dict() Constructor

You can also create a dictionary using the `dict()` constructor.

Example:



```
Open hello.py Save
```

```
person = dict(name="Bob", age=25, city="Los Angeles")
print(person)
```

Running tool: Execute

```
{'name': 'Bob', 'age': 25, 'city': 'Los Angeles'}
```

Tool Output Python Console Terminal

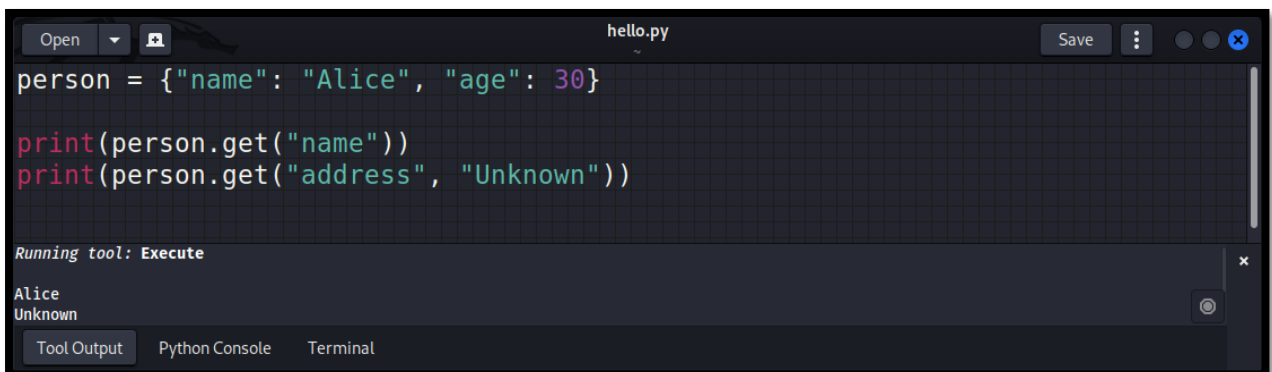
Dictionary Methods

Dictionaries in Python come with a variety of built-in methods for manipulation and querying.

get()

Returns the value for a given key. If the key is not found, it returns a default value.

Example:



```
Open hello.py Save
```

```
person = {"name": "Alice", "age": 30}
print(person.get("name"))
print(person.get("address", "Unknown"))
```

Running tool: Execute

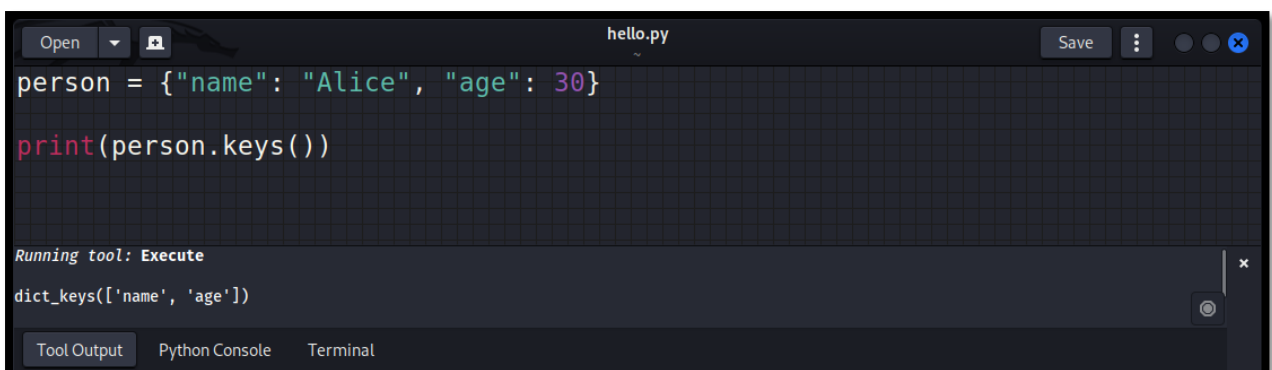
```
Alice
Unknown
```

Tool Output Python Console Terminal

keys()

Returns a list of all the keys in the dictionary.

Example:



```
Open hello.py Save
```

```
person = {"name": "Alice", "age": 30}
print(person.keys())
```

Running tool: Execute

```
dict_keys(['name', 'age'])
```

Tool Output Python Console Terminal

values()

Returns a list of all the values in the dictionary.

Example:

```
Open hello.py Save
```

```
person = {"name": "Alice", "age": 30}
print(person.values())
```

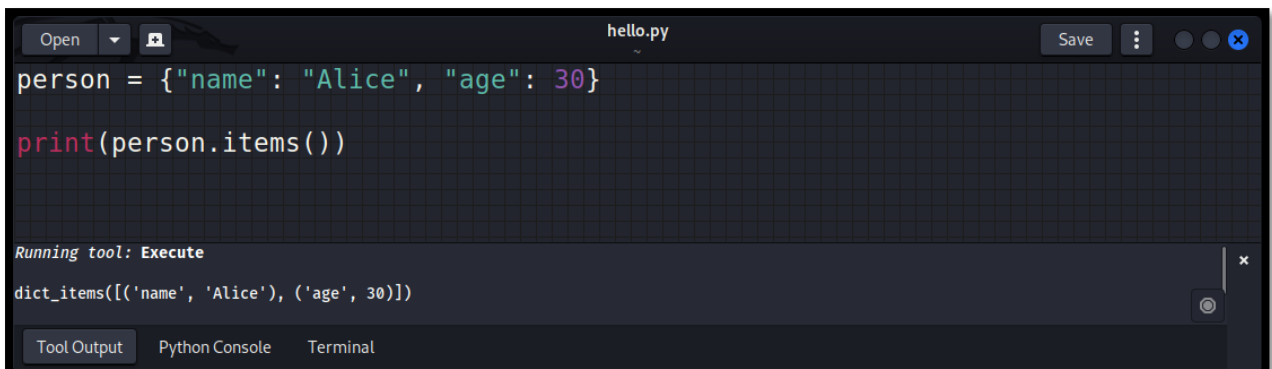
Running tool: Execute

```
dict_values(['Alice', 30])
```

Tool Output Python Console Terminal

items()

Returns a list of all the key-value pairs in the dictionary.

Example:

```
Open hello.py Save
```

```
person = {"name": "Alice", "age": 30}
print(person.items())
```

Running tool: Execute

```
dict_items([('name', 'Alice'), ('age', 30)])
```

Tool Output Python Console Terminal

update()

Merges a dictionary with another dictionary or with an iterable of key-value pairs.

Example:

```
Open hello.py Save
```

```
person = {"name": "Alice", "age": 30}
additional_info = {"city": "New York", "job": "Engineer"}
person.update(additional_info)

print(person)
```

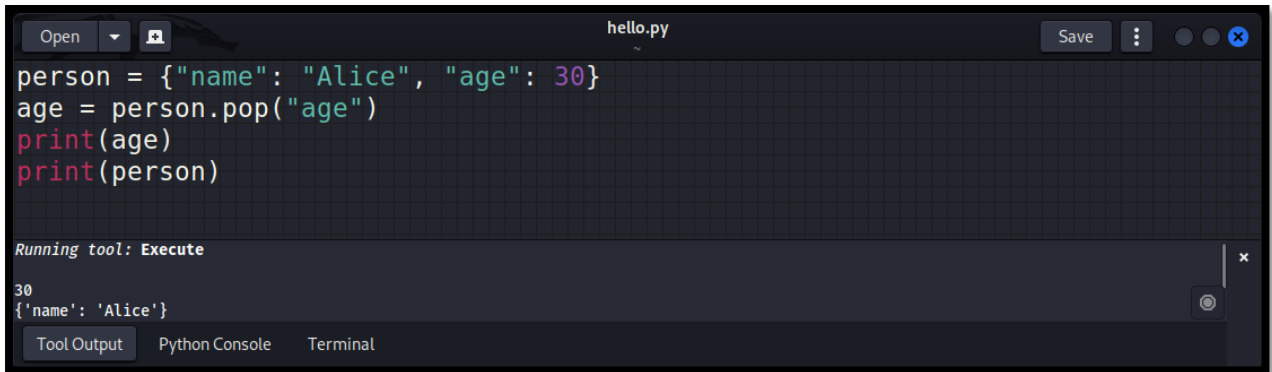
Running tool: Execute

```
{'name': 'Alice', 'age': 30, 'city': 'New York', 'job': 'Engineer'}
```

Tool Output Python Console Terminal

pop()

Removes and returns the value for a given key. Raises a `KeyError` if the key is not found, unless a default value is provided.

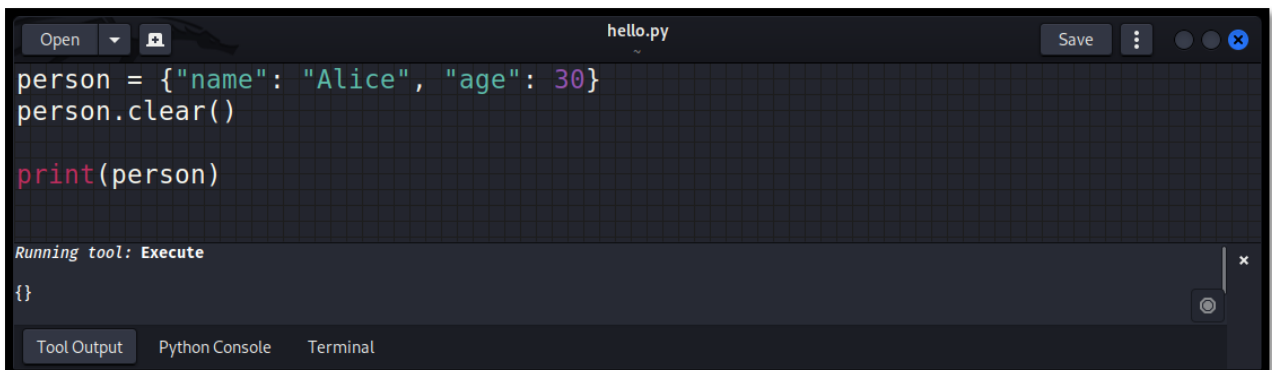
Example:

```
hello.py
person = {"name": "Alice", "age": 30}
age = person.pop("age")
print(age)
print(person)

Running tool: Execute
30
{'name': 'Alice'}
```

clear()

Removes all items from the dictionary.

Example:

```
hello.py
person = {"name": "Alice", "age": 30}
person.clear()

print(person)

Running tool: Execute
{}
```

Dictionary Comprehensions

Dictionary comprehensions provide a concise way to create dictionaries. They are similar to list comprehensions but produce dictionaries.

Example:

```
hello.py
squares = {x: x**2 for x in (2, 3, 4, 5)}

print(squares)

Running tool: Execute
{2: 4, 3: 9, 4: 16, 5: 25}
```

Sets: Operations, Use-Cases

In Python, a set is an unordered collection of unique items. Sets are similar to lists and tuples but do not allow duplicate values. They are particularly useful for membership testing, eliminating duplicate entries, and performing mathematical set operations.

Creating Sets

Sets can be created using curly braces `{}` or the `set()` constructor.

Example:



```
hello.py
# Using curly braces
fruits = {"apple", "banana", "cherry"}
# Using the set() constructor
colors = set(["red", "green", "blue"])
```

Using the `set()` constructor

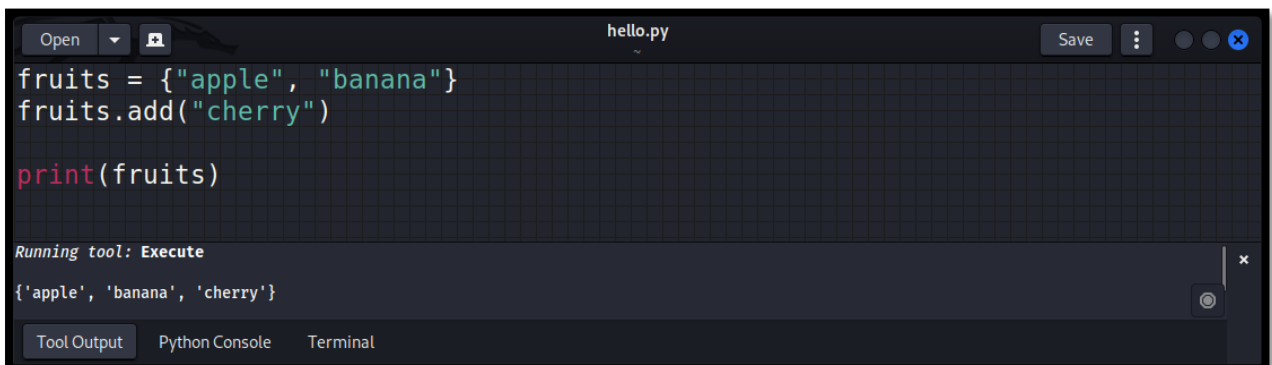
Note: An empty set must be created using the `set()` constructor. Using `{}` creates an empty dictionary.

Set Operations

`add()`

Adds an element to the set.

Example:



```
hello.py
fruits = {"apple", "banana"}
fruits.add("cherry")

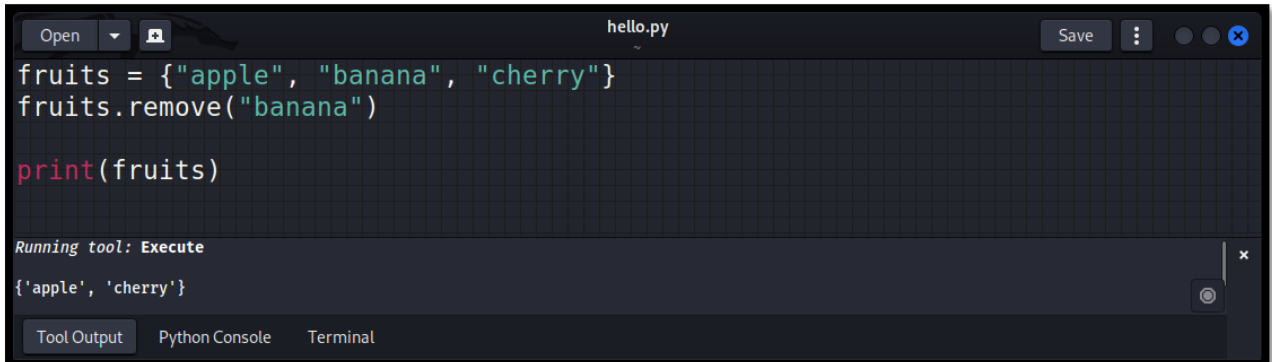
print(fruits)

Running tool: Execute
{'apple', 'banana', 'cherry'}
```

Output: {'apple', 'banana', 'cherry'}

remove()

Removes a specified element from the set. Raises a KeyError if the element is not found.

Example:

```
hello.py
Open Save
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")

print(fruits)

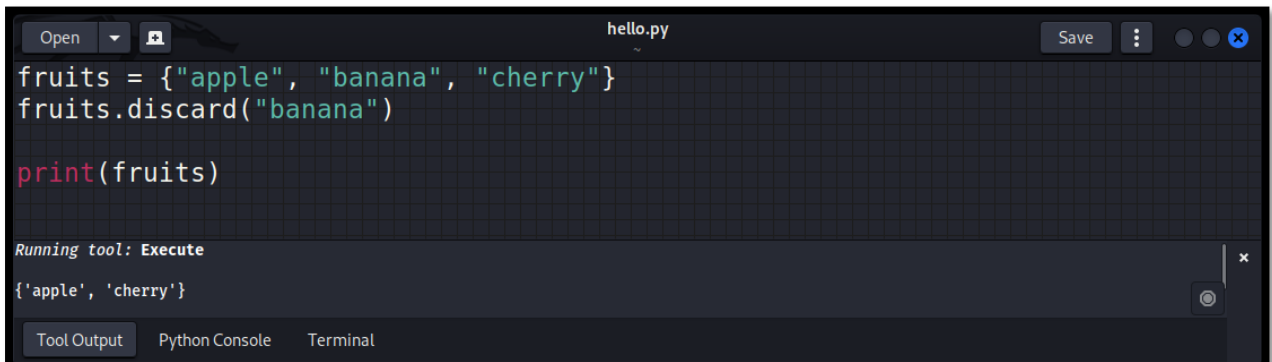
Running tool: Execute
{'apple', 'cherry'}
```

The screenshot shows a code editor window titled 'hello.py'. The code defines a set 'fruits' containing 'apple', 'banana', and 'cherry'. The 'remove()' method is used to delete 'banana'. The output shows the set after execution: {'apple', 'cherry'}.

Output: {'apple', 'cherry'}

discard()

Removes a specified element from the set. Does nothing if the element is not found.

Example:

```
hello.py
Open Save
fruits = {"apple", "banana", "cherry"}
fruits.discard("banana")

print(fruits)

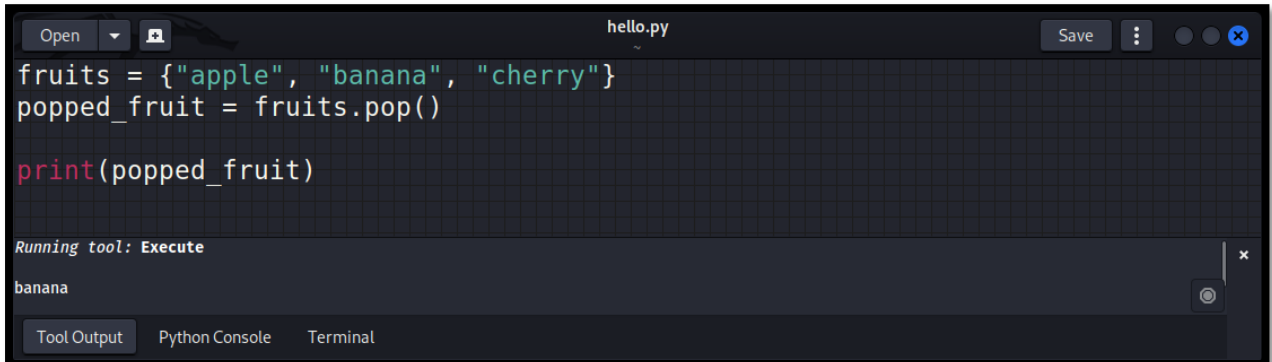
Running tool: Execute
{'apple', 'cherry'}
```

The screenshot shows a code editor window titled 'hello.py'. The code defines a set 'fruits' containing 'apple', 'banana', and 'cherry'. The 'discard()' method is used to delete 'banana'. The output shows the set after execution: {'apple', 'cherry'}.

Output: {'apple', 'cherry'}

pop()

Removes and returns an arbitrary element from the set. Raises a KeyError if the set is empty.

Example:

```
hello.py
fruits = {"apple", "banana", "cherry"}
popped_fruit = fruits.pop()

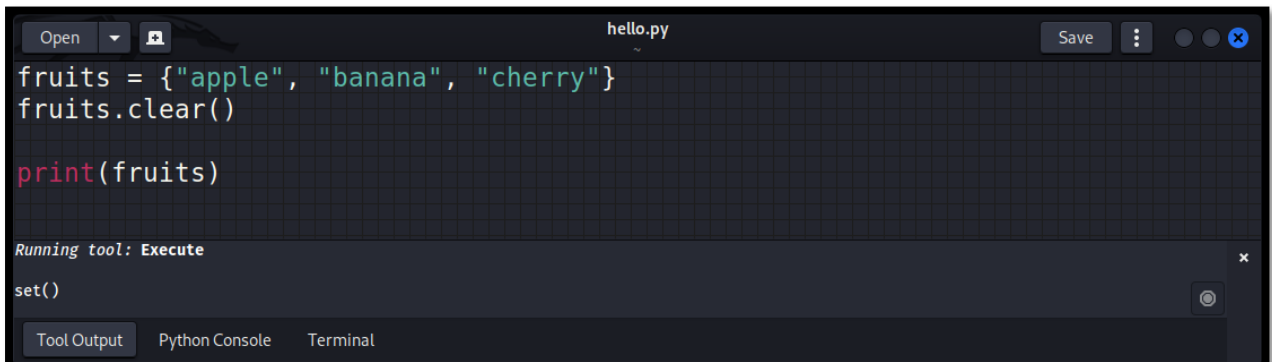
print(popped_fruit)

Running tool: Execute
banana
Tool Output Python Console Terminal
```

Output can be 'apple', 'banana', or 'cherry' since sets are unordered.

clear()

Removes all elements from the set.

Example:

```
hello.py
fruits = {"apple", "banana", "cherry"}
fruits.clear()

print(fruits)

Running tool: Execute
set()
Tool Output Python Console Terminal
```

Output: set()

Mathematical Set Operations

Python sets support various mathematical operations like union, intersection, difference, and symmetric difference.

Example:



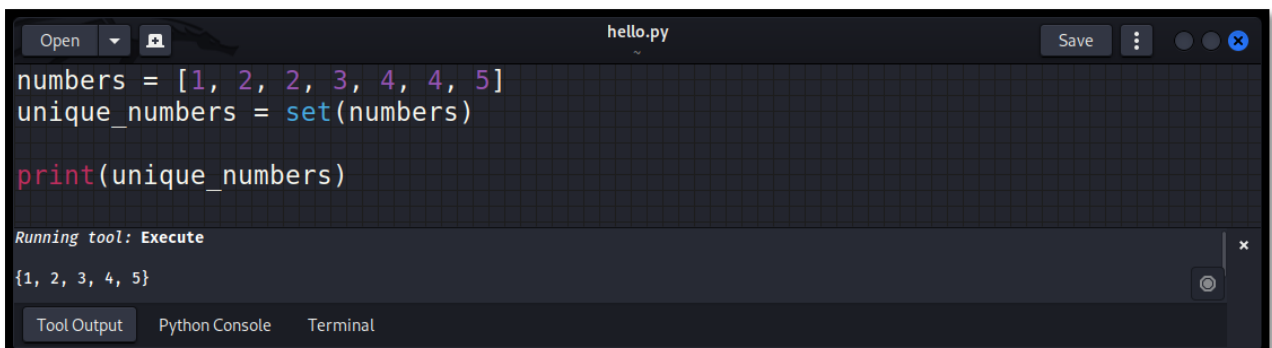
```
hello.py
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
print(A | B) # Output: {1, 2, 3, 4, 5, 6}
print(A.union(B)) # Intersection
print(A & B) # Output: {3, 4}
print(A.intersection(B)) # Difference
print(A - B) # Output: {1, 2}
print(A.difference(B)) # Symmetric Difference
print(A ^ B) # Output: {1, 2, 5, 6}
print(A.symmetric_difference(B))

Running tool: Execute
{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 6}
{3, 4}
{3, 4}
{1, 2}
{1, 2}
{1, 2, 5, 6}
{1, 2, 5, 6}
```

Use-Cases for Sets

1. **Removing Duplicates:** Since sets do not allow duplicate values, they can be used to remove duplicates from a list or other iterable.

Example:



```
hello.py
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = set(numbers)

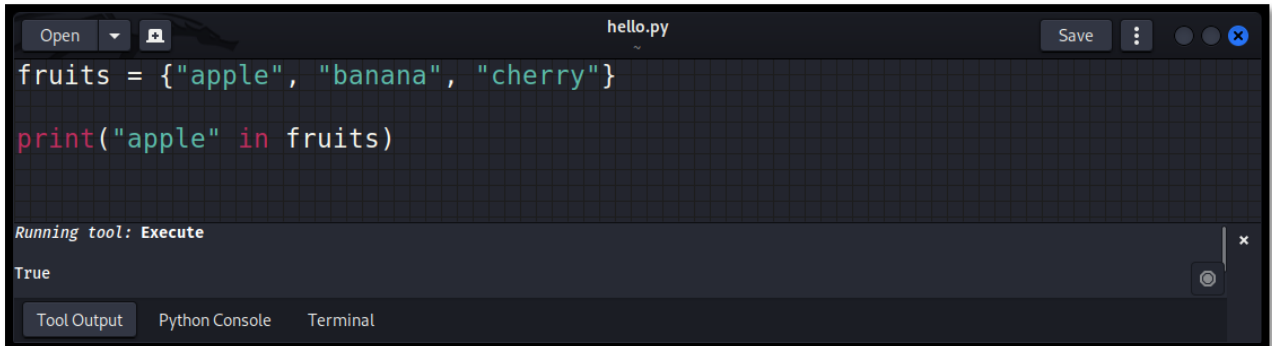
print(unique_numbers)

Running tool: Execute
{1, 2, 3, 4, 5}
```

Output: {1, 2, 3, 4, 5}

2. **Membership Testing:** Checking if an item exists in a set is faster than checking if it exists in a list or tuple.

Example:



```
hello.py
fruits = {"apple", "banana", "cherry"}
print("apple" in fruits)

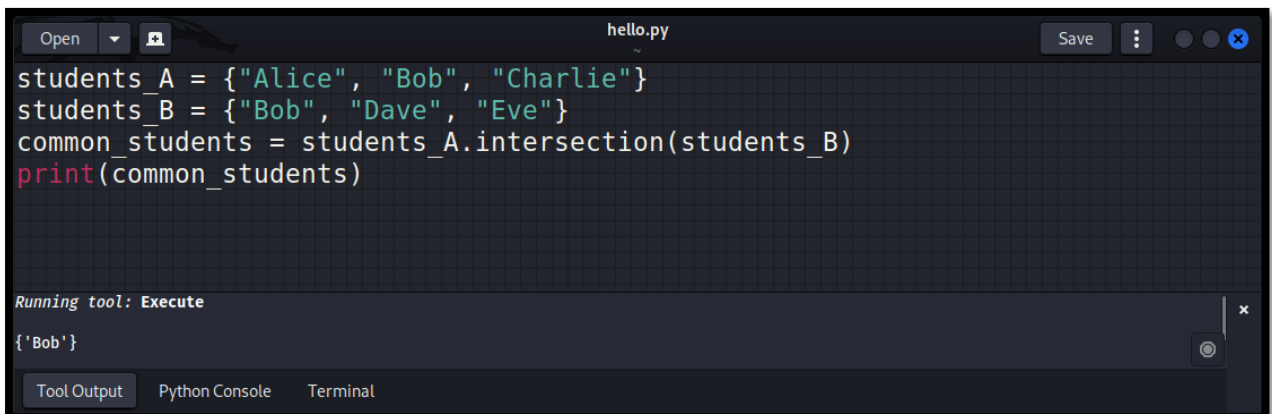
Running tool: Execute
True
```

The screenshot shows a Python IDE window titled 'hello.py'. The code defines a set 'fruits' containing 'apple', 'banana', and 'cherry', and then prints the result of 'apple' in fruits. The output console shows 'True'.

Output: True

3. **Mathematical Operations:** Sets can be used to perform mathematical set operations like union, intersection, and difference.

Example:



```
hello.py
students_A = {"Alice", "Bob", "Charlie"}
students_B = {"Bob", "Dave", "Eve"}
common_students = students_A.intersection(students_B)
print(common_students)

Running tool: Execute
{'Bob'}
```

The screenshot shows a Python IDE window titled 'hello.py'. The code defines two sets, 'students_A' and 'students_B', and then uses the 'intersection' method to find common students. The output console shows {'Bob'}.

Output: {'Bob'}

Basic Functions

Built-in Functions

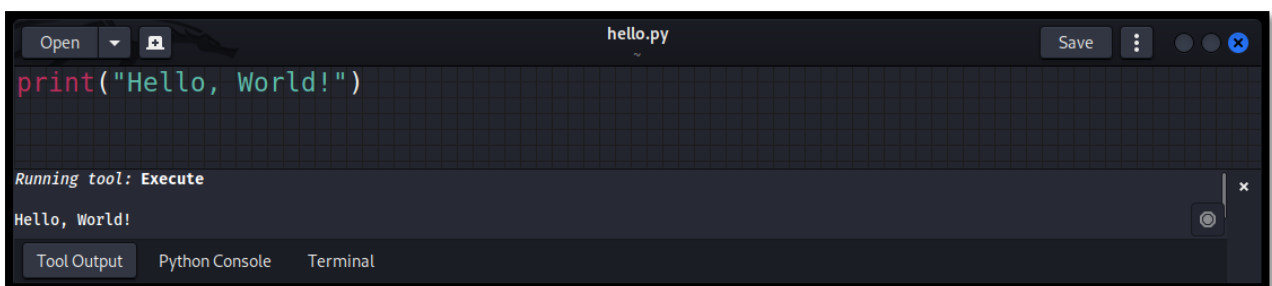
Python comes with a rich set of built-in functions that are always available for use without the need for any imports. These functions provide essential functionality and are optimized for performance, making them a fundamental part of the Python programming experience.

Common Built-in Functions

print()

Displays the specified message or object to the console.

Example:



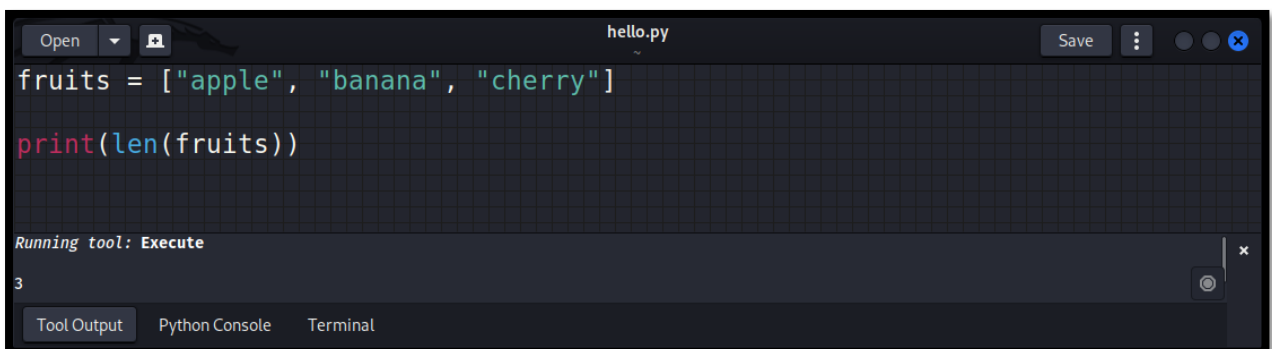
```
hello.py
print("Hello, World!")

Running tool: Execute
Hello, World!
```

len()

Returns the number of items in an object.

Example:



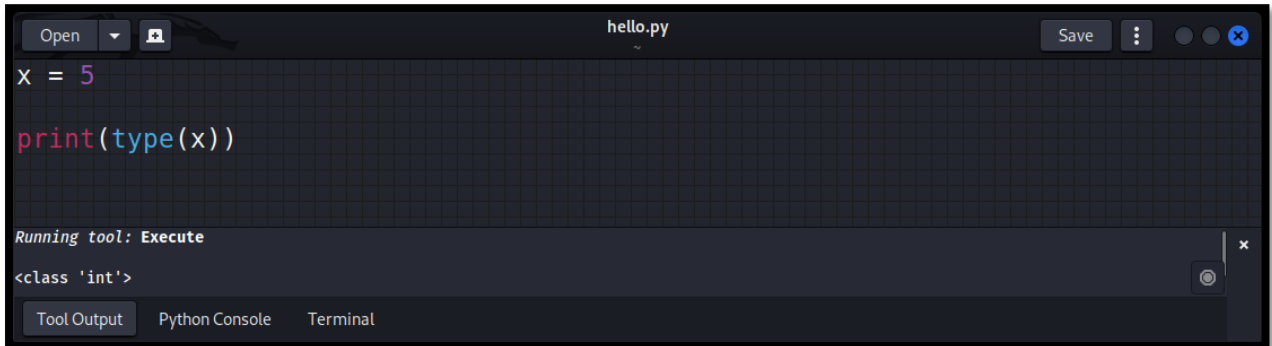
```
hello.py
fruits = ["apple", "banana", "cherry"]
print(len(fruits))

Running tool: Execute
3
```

Output: 3

type()

Returns the type of an object.

Example:

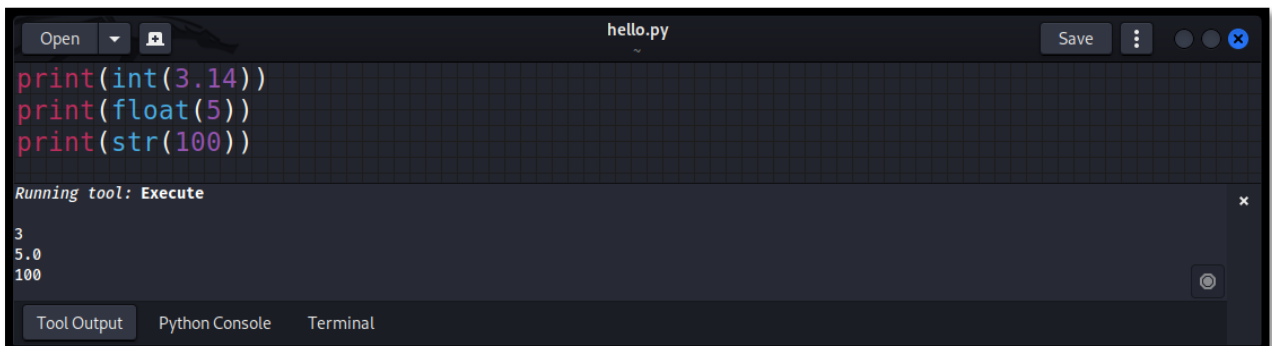
```
Open hello.py Save
X = 5
print(type(x))

Running tool: Execute
<class 'int'>
Tool Output Python Console Terminal
```

Output: <class 'int'>

int(), float(), str()

Converts a value to an integer, floating-point number, or string, respectively.

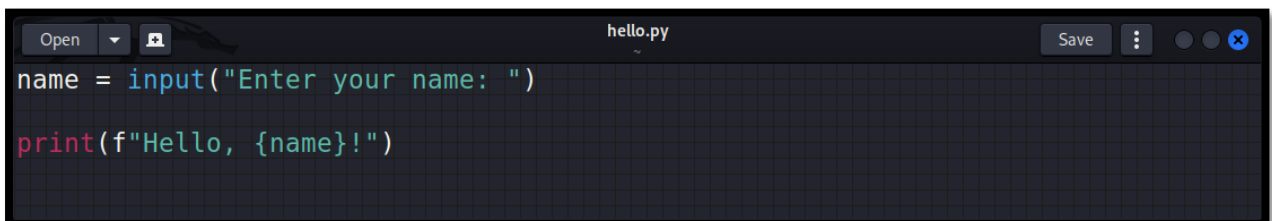
Example:

```
Open hello.py Save
print(int(3.14))
print(float(5))
print(str(100))


Running tool: Execute
3
5.0
100
Tool Output Python Console Terminal
```

input()

Reads a line from input and returns it as a string.

Example:

```
Open hello.py Save
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

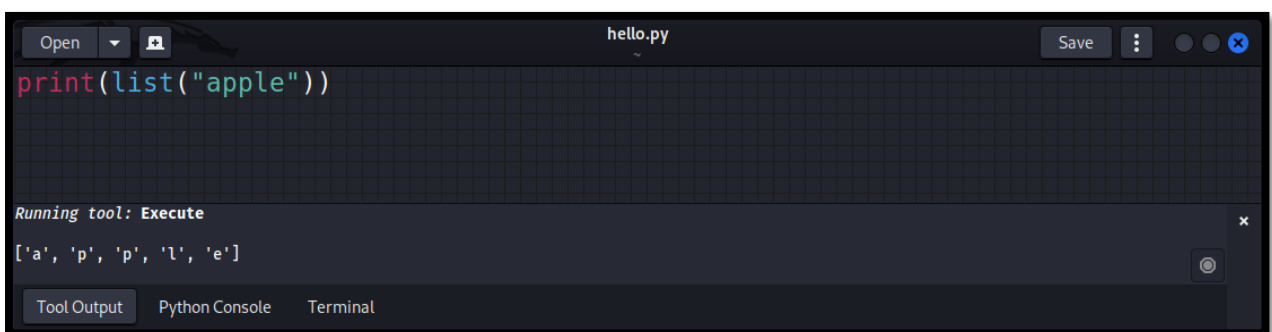


```
kali@kali: ~  
File Actions Edit View Help  
~(kali@kali)-[~]  
└─$ python hello.py  
Enter your name: Mike  
Hello, Mike!
```

list(), tuple(), set(), dict()

Converts a value to a list, tuple, set, or dictionary, respectively.

Example:



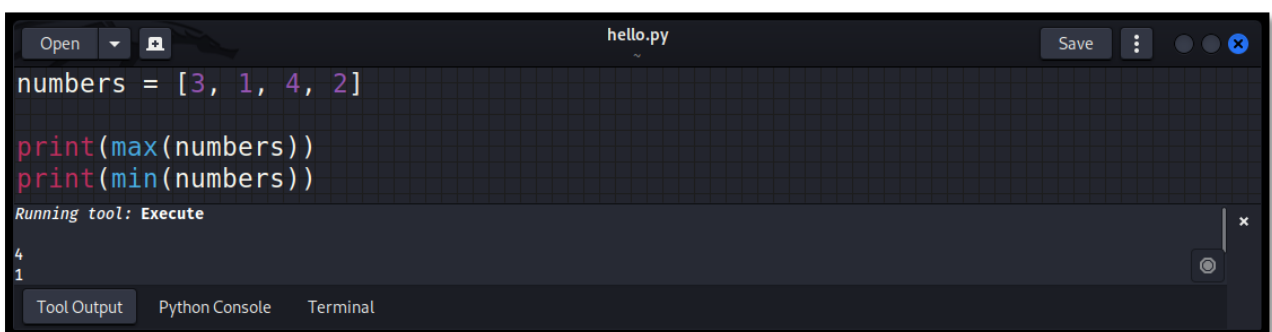
```
hello.py  
print(list("apple"))  
  
Running tool: Execute  
['a', 'p', 'p', 'l', 'e']  
Tool Output Python Console Terminal
```

Output: ['a', 'p', 'p', 'l', 'e']

max() and min()

Returns the largest or smallest item from an iterable or from two or more arguments.

Example:



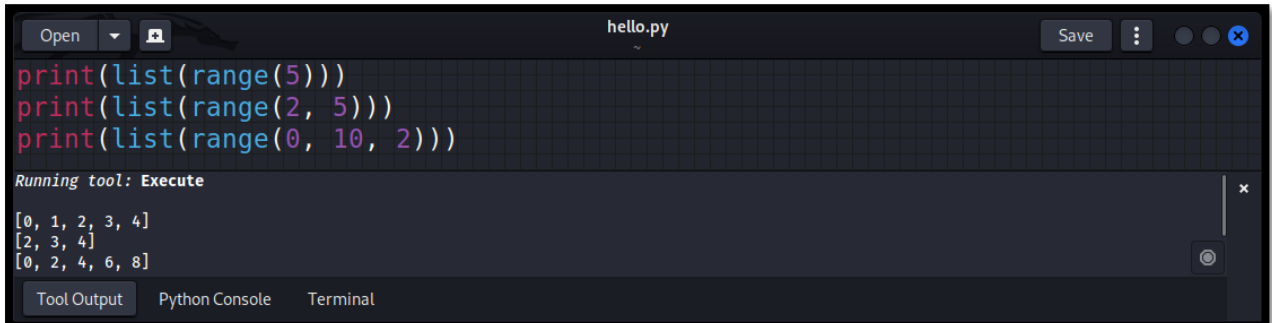
```
hello.py  
numbers = [3, 1, 4, 2]  
  
print(max(numbers))  
print(min(numbers))  
  
Running tool: Execute  
4  
1  
Tool Output Python Console Terminal
```

Output:

4
1

range()

Generates a sequence of numbers.

Example:A screenshot of a code editor window titled 'hello.py'. The code contains three print statements: print(list(range(5))), print(list(range(2, 5))), and print(list(range(0, 10, 2))). Below the code, the output shows three lists: [0, 1, 2, 3, 4], [2, 3, 4], and [0, 2, 4, 6, 8]. The editor has tabs for 'Tool Output', 'Python Console', and 'Terminal'.

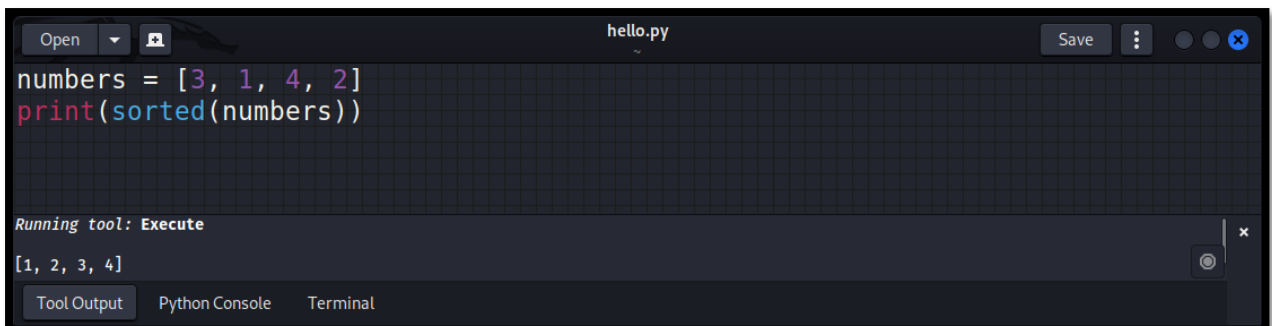
```
print(list(range(5)))
print(list(range(2, 5)))
print(list(range(0, 10, 2)))

Running tool: Execute

[0, 1, 2, 3, 4]
[2, 3, 4]
[0, 2, 4, 6, 8]
```

sorted()

Returns a sorted list from the specified iterable.

Example:A screenshot of a code editor window titled 'hello.py'. The code defines a list 'numbers = [3, 1, 4, 2]' and prints the result of sorted(numbers). The output shows the sorted list [1, 2, 3, 4]. The editor has tabs for 'Tool Output', 'Python Console', and 'Terminal'.

```
numbers = [3, 1, 4, 2]
print(sorted(numbers))

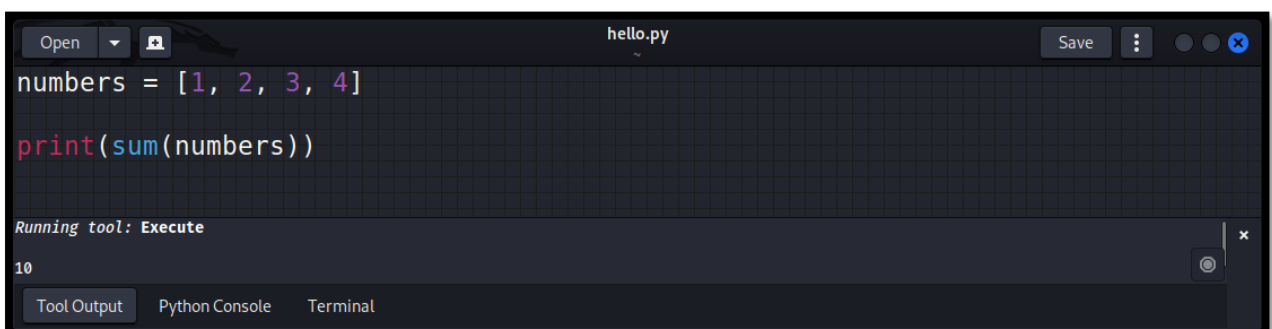
Running tool: Execute

[1, 2, 3, 4]
```

Output: [1, 2, 3, 4]

sum()

Returns the sum of all items in an iterable.

Example:A screenshot of a code editor window titled 'hello.py'. The code defines a list 'numbers = [1, 2, 3, 4]' and prints the result of sum(numbers). The output shows the number 10. The editor has tabs for 'Tool Output', 'Python Console', and 'Terminal'.

```
numbers = [1, 2, 3, 4]
print(sum(numbers))

Running tool: Execute

10
```

Print Function

The **print()** function is one of the most frequently used built-in functions in Python. It allows developers to output data to the console, making it an invaluable tool for debugging, data display, and user interaction.

Basic Usage of print()

At its simplest, the **print()** function outputs the provided arguments to the console.

Example:

```
print("Hello, World!")
```

Printing Multiple Arguments

The **print()** function can accept multiple arguments and will concatenate them with a space by default.

Example:



The screenshot shows a code editor window titled 'hello.py'. The code contains the following lines:

```
name = "Alice"  
age = 30  
print("Name:", name, "| Age:", age)
```

Below the code, there is a 'Running tool: Execute' section. The output of the code is displayed as:

```
Name: Alice | Age: 30
```

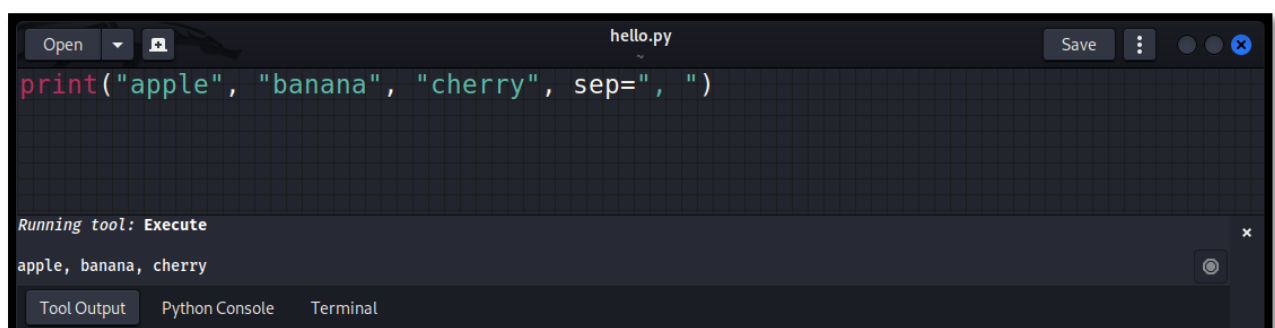
The editor also shows tabs for 'Tool Output', 'Python Console', and 'Terminal'.

Output: Name: Alice | Age: 30

The sep Parameter

The **sep** parameter defines the character or characters used to separate multiple arguments. By default, **sep** is set to a space.

Example:



The screenshot shows a code editor window titled 'hello.py'. The code contains the following line:

```
print("apple", "banana", "cherry", sep=", ")
```

Below the code, there is a 'Running tool: Execute' section. The output of the code is displayed as:

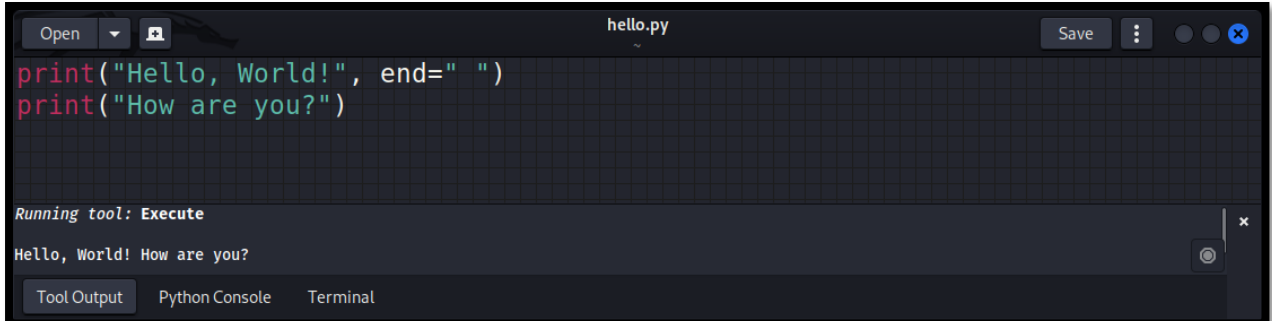
```
apple, banana, cherry
```

The editor also shows tabs for 'Tool Output', 'Python Console', and 'Terminal'.

The end Parameter

The **end** parameter specifies what to print at the end. By default, **end** is set to a newline character (`\n`), which means after the **print()** function is executed, the next print will be on a new line.

Example:



```
hello.py
print("Hello, World!", end=" ")
print("How are you?")

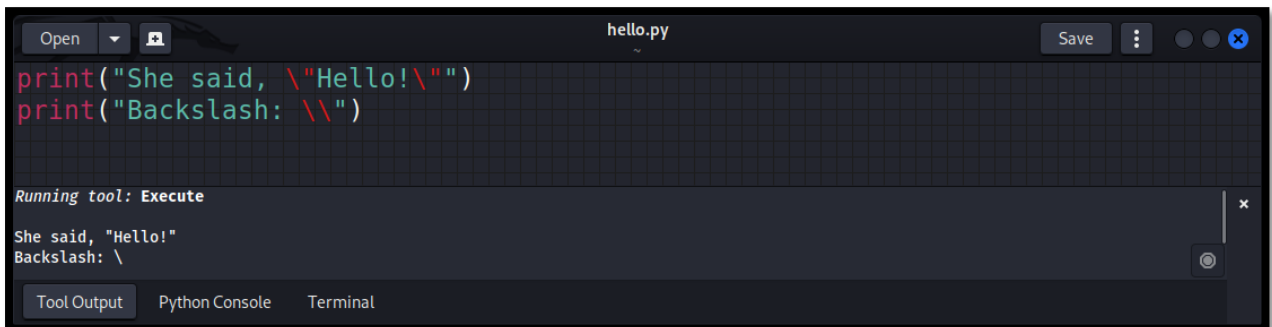
Running tool: Execute
Hello, World! How are you?
```

The screenshot shows a code editor window titled 'hello.py' with two lines of Python code: `print("Hello, World!", end=" ")` and `print("How are you?")`. Below the code, the output of the script is displayed as 'Hello, World! How are you?' on a single line. The interface includes 'Open', 'Save', and 'Terminal' buttons.

Printing Special Characters

To print special characters, such as a double quote or a backslash, you can use escape sequences.

Example:



```
hello.py
print("She said, \"Hello!\")
print("Backslash: \\")

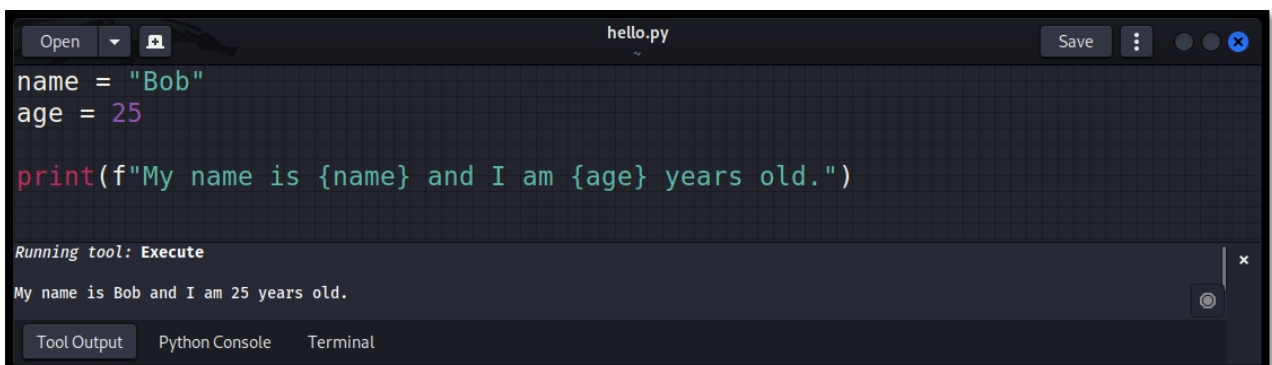
Running tool: Execute
She said, "Hello!"
Backslash: \
```

The screenshot shows a code editor window titled 'hello.py' with two lines of Python code: `print("She said, \"Hello!\")` and `print("Backslash: \\")`. Below the code, the output is displayed as 'She said, "Hello!"' and 'Backslash: \' on separate lines. The interface includes 'Open', 'Save', and 'Terminal' buttons.

String Formatting with print()

The **print()** function can be combined with string formatting techniques to produce formatted output.

Example:



```
hello.py
name = "Bob"
age = 25

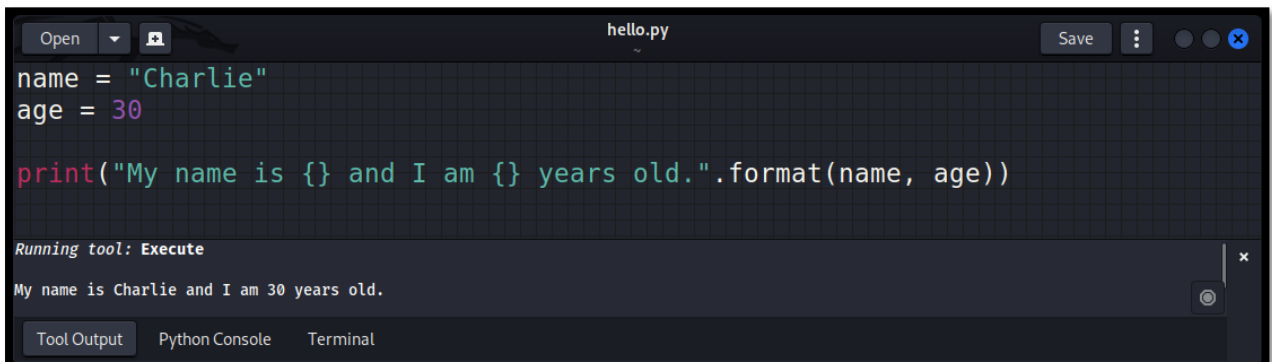
print(f"My name is {name} and I am {age} years old.")

Running tool: Execute
My name is Bob and I am 25 years old.
```

The screenshot shows a code editor window titled 'hello.py' with three lines of Python code: `name = "Bob"`, `age = 25`, and `print(f"My name is {name} and I am {age} years old.")`. Below the code, the output is displayed as 'My name is Bob and I am 25 years old.' The interface includes 'Open', 'Save', and 'Terminal' buttons.

Using the format() method

Example:



The screenshot shows a code editor window titled 'hello.py'. The code defines variables 'name' and 'age', and uses the 'format()' method to print a string. The output of the code is displayed in the 'Tool Output' pane below the editor.

```
name = "Charlie"
age = 30

print("My name is {} and I am {} years old.".format(name, age))
```

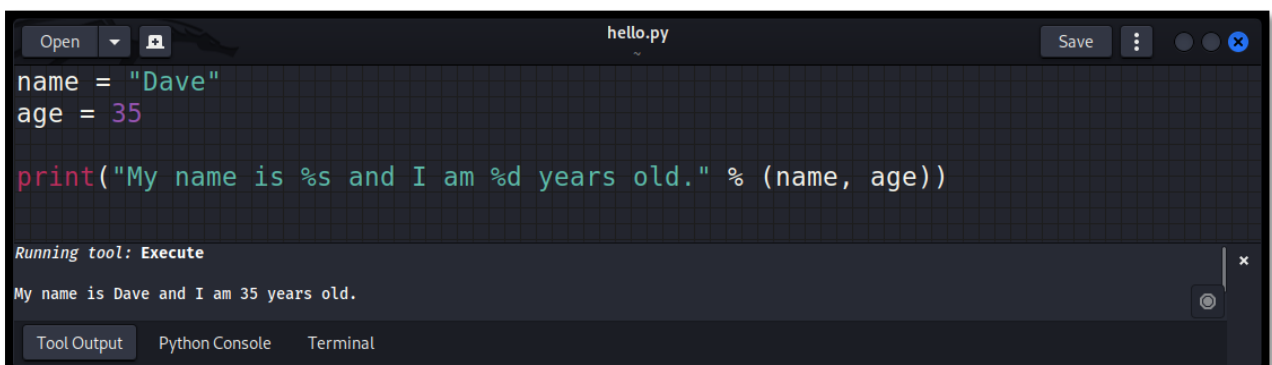
Running tool: Execute

My name is Charlie and I am 30 years old.

Tool Output Python Console Terminal

Using %-formatting

Example:



The screenshot shows a code editor window titled 'hello.py'. The code defines variables 'name' and 'age', and uses %-formatting to print a string. The output of the code is displayed in the 'Tool Output' pane below the editor.

```
name = "Dave"
age = 35

print("My name is %s and I am %d years old." % (name, age))
```

Running tool: Execute

My name is Dave and I am 35 years old.

Tool Output Python Console Terminal

Slicing and Casting

Basic String Operations

Strings are among the most commonly used data types in Python. They are sequences of characters and can be manipulated in various ways to achieve desired results.

String Concatenation

Strings can be joined or concatenated using the + operator.

Example:



```
hello.py
Open Save
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name

print(full_name)

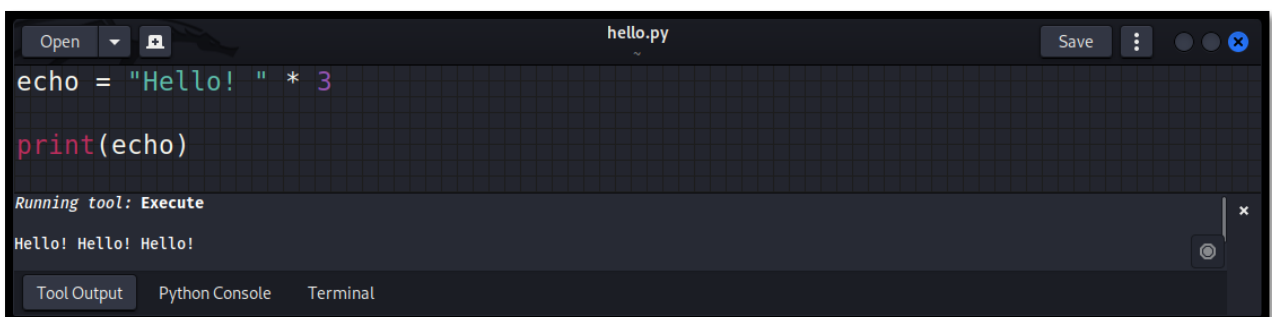
Running tool: Execute
John Doe
Tool Output Python Console Terminal
```

Output: John Doe

String Repetition

You can repeat a string a specified number of times using the * operator.

Example:



```
hello.py
Open Save
echo = "Hello! " * 3

print(echo)

Running tool: Execute
Hello! Hello! Hello!
Tool Output Python Console Terminal
```

Output: Hello! Hello! Hello!

String Slicing

Slicing allows you to extract a portion of a string. It's done by specifying a start index and an end index. The slice will include characters from the start index up to, but not including, the end index.

Example:



```
Open  *hello.py  Save  x
text = "Python programming"
slice1 = text[0:6]      # Extracts characters from index 0 to 5
slice2 = text[7:]      # Extracts characters from index 7 to the end

print(slice2)

Running tool: Execute
programming
Tool Output  Python Console  Terminal
```

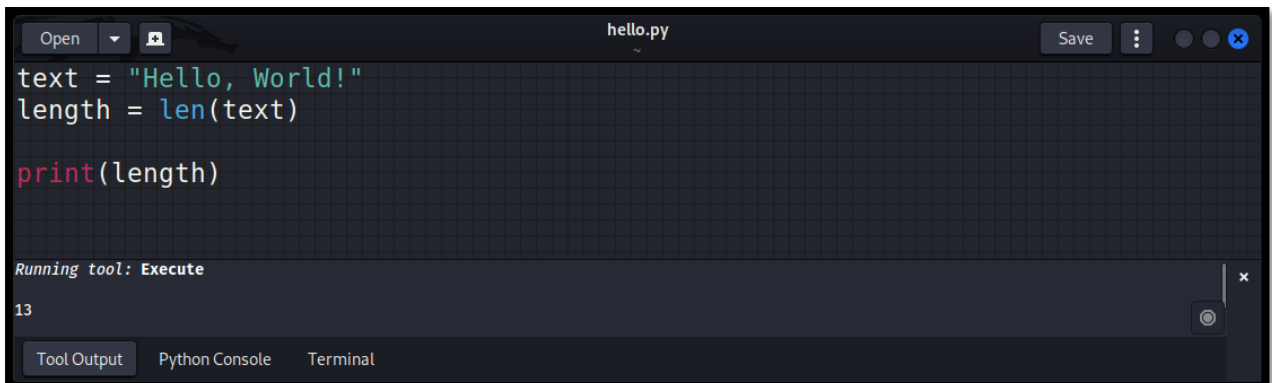
Output:

```
Python
programming
```

String Length

The `len()` function returns the number of characters in a string.

Example:



```
Open  hello.py  Save  x
text = "Hello, World!"
length = len(text)

print(length)

Running tool: Execute
13
Tool Output  Python Console  Terminal
```

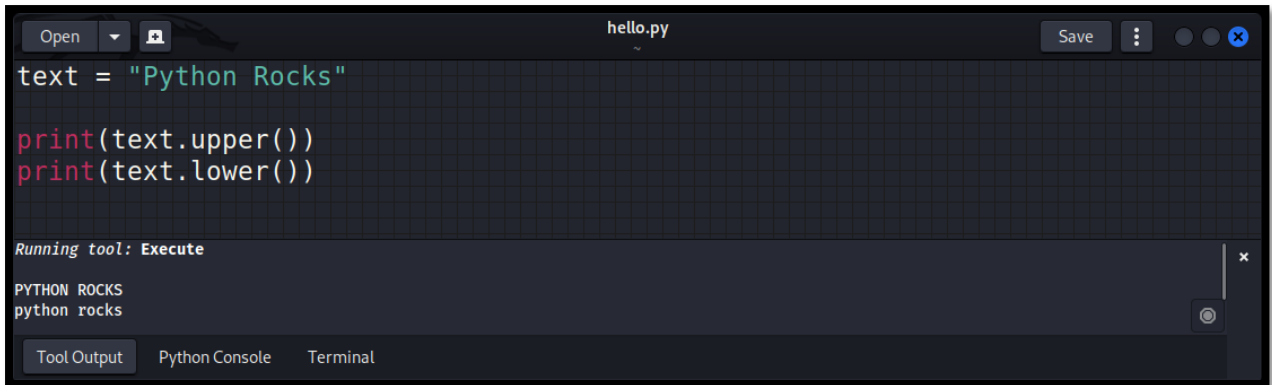
Output: 13

String Transformation

Strings in Python come with built-in methods for common transformations.

Uppercase and Lowercase

Example:



```
Open hello.py Save
```

```
text = "Python Rocks"  
  
print(text.upper())  
print(text.lower())
```

Running tool: Execute

```
PYTHON ROCKS  
python rocks
```

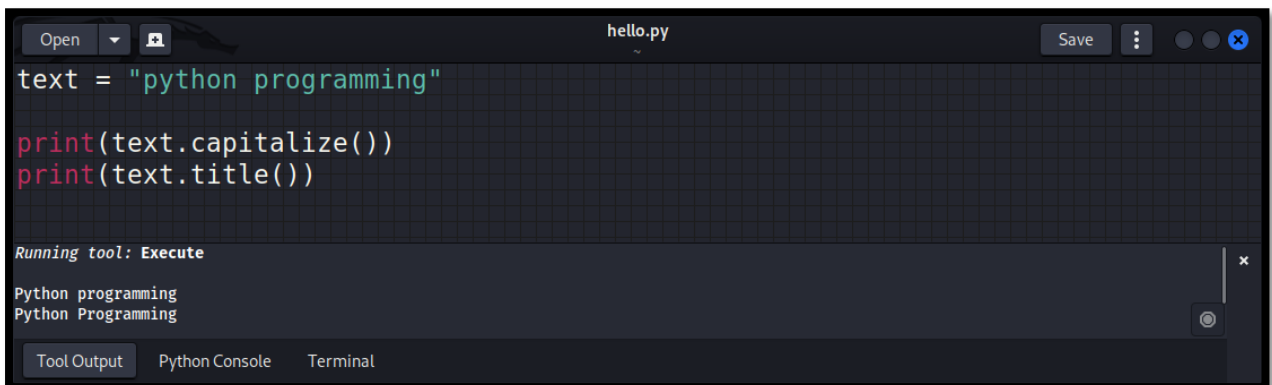
Tool Output Python Console Terminal

Output:

```
PYTHON ROCKS  
python rocks
```

Capitalization

Example:



```
Open hello.py Save
```

```
text = "python programming"  
  
print(text.capitalize())  
print(text.title())
```

Running tool: Execute

```
Python programming  
Python Programming
```

Tool Output Python Console Terminal

Output:

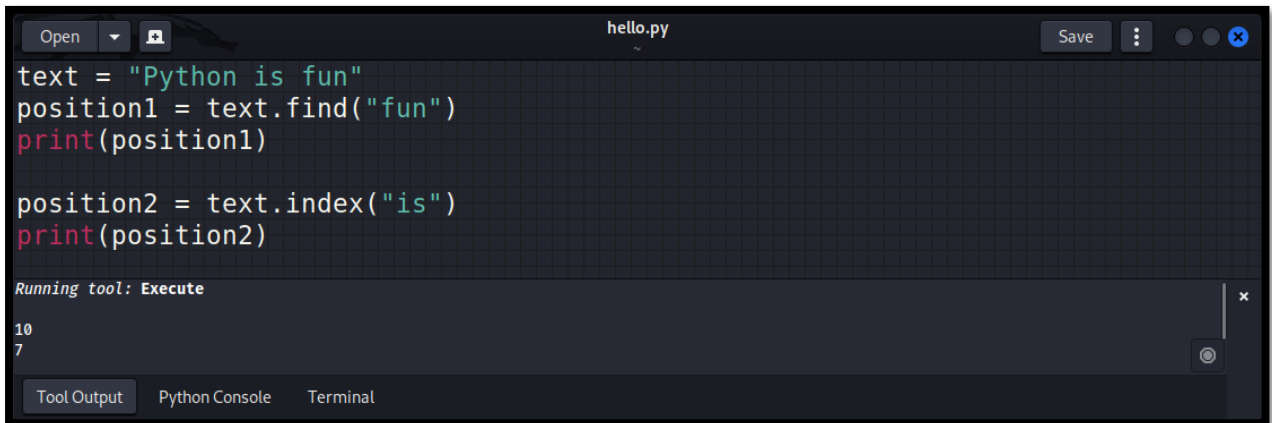
```
Python programming  
Python Programming
```

Searching in Strings

find() and index()

Both methods search for a substring and return the starting index of the first occurrence. While **find()** returns **-1** if the substring is not found, **index()** raises a **ValueError**.

Example:



```
hello.py
text = "Python is fun"
position1 = text.find("fun")
print(position1)

position2 = text.index("is")
print(position2)

Running tool: Execute
10
7
```

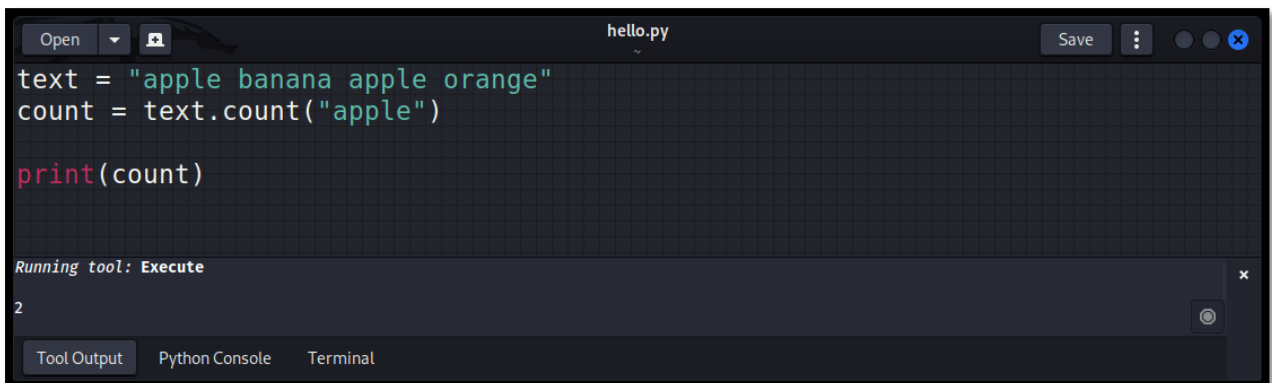
Output:

```
10
7
```

count()

Returns the number of occurrences of a substring.

Example:



```
hello.py
text = "apple banana apple orange"
count = text.count("apple")

print(count)

Running tool: Execute
2
```

Output: 2

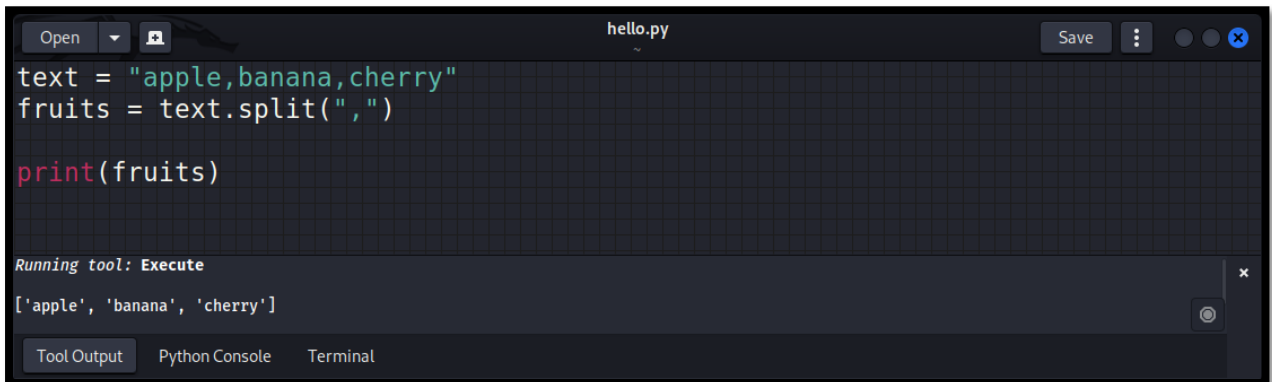
String Methods (split, join, replace, etc.)

Strings in Python are equipped with a variety of built-in methods that allow for efficient manipulation and transformation. These methods provide solutions for common tasks, making string handling in Python both powerful and user-friendly.

The `split()` Method

The `split()` method divides a string into a list based on a specified delimiter.

Example:



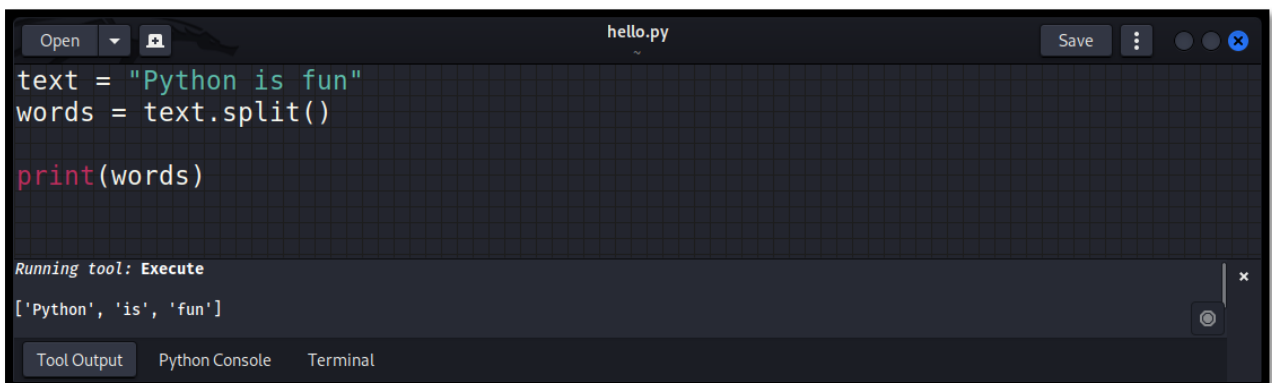
```
hello.py
text = "apple,banana,cherry"
fruits = text.split(",")

print(fruits)

Running tool: Execute
['apple', 'banana', 'cherry']
```

Output: ['apple', 'banana', 'cherry']

By default, if no delimiter is specified, `split()` divides the string at each space.



```
hello.py
text = "Python is fun"
words = text.split()

print(words)

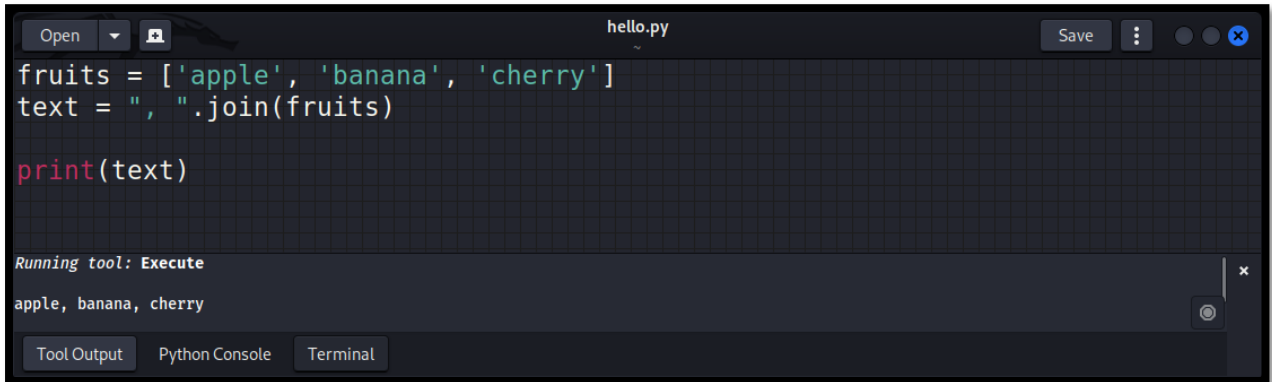
Running tool: Execute
['Python', 'is', 'fun']
```

Output: ['Python', 'is', 'fun']

The join() Method

The `join()` method combines a list of strings into a single string using a specified delimiter.

Example:

A screenshot of a Python IDE window titled 'hello.py'. The code in the editor is:

```
fruits = ['apple', 'banana', 'cherry']
text = ", ".join(fruits)

print(text)
```

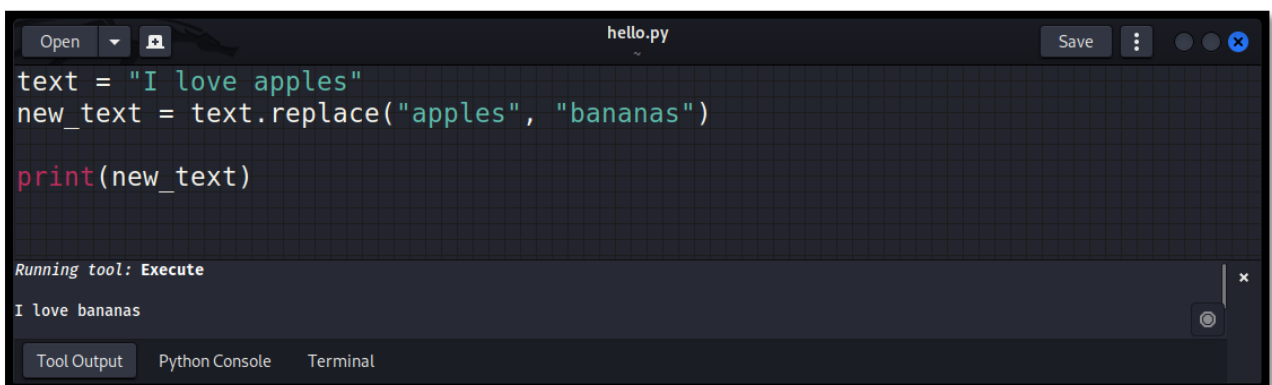
The output console below the code shows the result: 'apple, banana, cherry'. The IDE interface includes 'Open', 'Save', and 'Terminal' buttons.

Output: apple, banana, cherry

The replace() Method

The `replace()` method replaces a specified substring with another substring.

Example:

A screenshot of a Python IDE window titled 'hello.py'. The code in the editor is:

```
text = "I love apples"
new_text = text.replace("apples", "bananas")

print(new_text)
```

The output console below the code shows the result: 'I love bananas'. The IDE interface includes 'Open', 'Save', and 'Terminal' buttons.

Output: I love bananas

The startswith() and endswith() Methods

These methods check if a string starts or ends with a specified substring, respectively, returning **True** or **False**.

Example:



```
Open | hello.py | Save | [Icons]
text = "Python programming"
print(text.startswith("Python"))
print(text.endswith("Java"))

Running tool: Execute
True
False
Tool Output | Python Console | Terminal
```

Output:

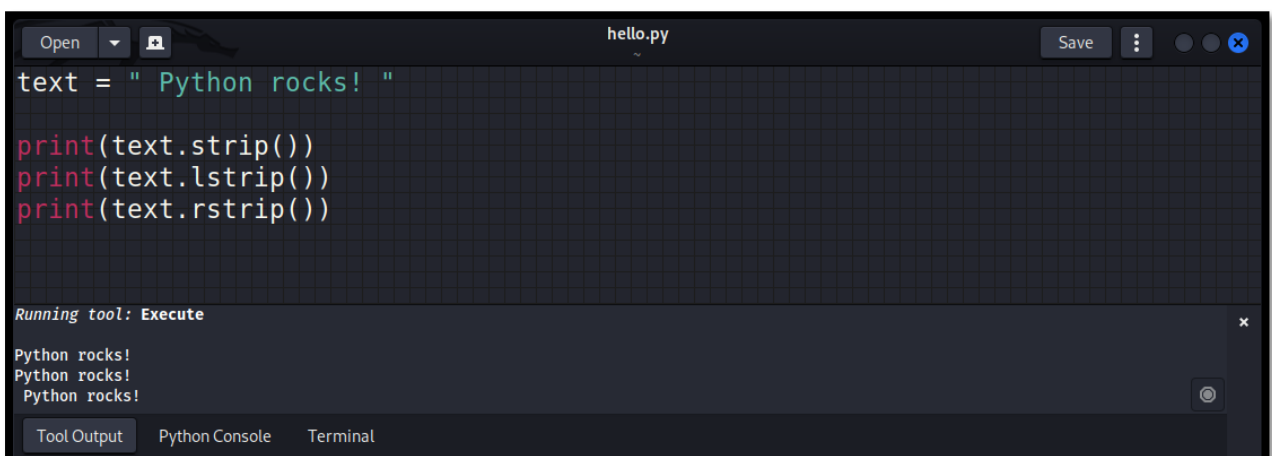
```
True
False
```

The strip(),rstrip(), and lstrip() Methods

These methods remove whitespace characters (like spaces, tabs, and newlines) from the beginning and/or end of a string.

- **strip()** removes from both ends.
- **lstrip()** removes from the left.
- **rstrip()** removes from the right.

Example:



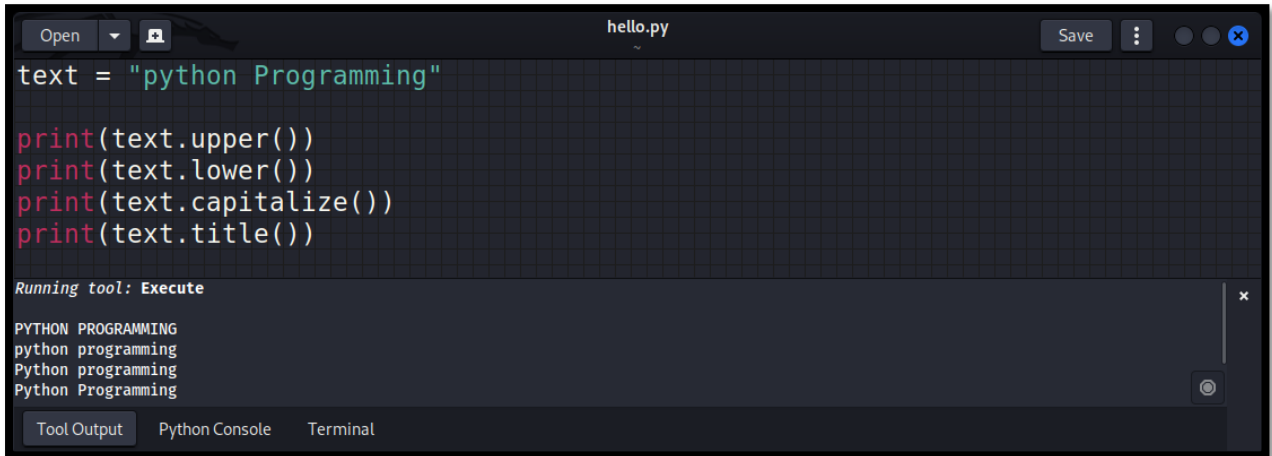
```
Open | hello.py | Save | [Icons]
text = " Python rocks! "
print(text.strip())
print(text.lstrip())
print(text.rstrip())

Running tool: Execute
Python rocks!
Python rocks!
Python rocks!
Tool Output | Python Console | Terminal
```

The upper(), lower(), capitalize(), and title() Methods

These methods are used for case conversion.

Example:



```
Open hello.py Save
text = "python Programming"
print(text.upper())
print(text.lower())
print(text.capitalize())
print(text.title())
Running tool: Execute
PYTHON PROGRAMMING
python programming
Python programming
Python Programming
Tool Output Python Console Terminal
```

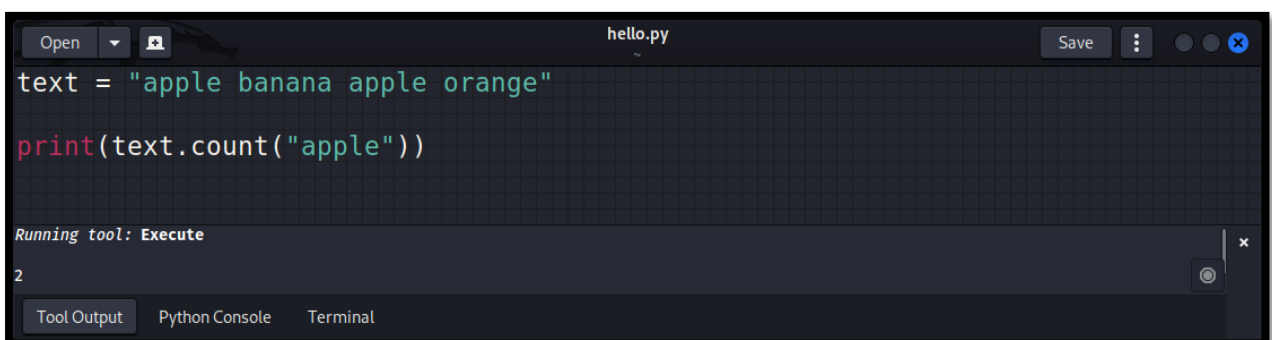
Output:

```
PYTHON PROGRAMMING
python programming
Python programming
Python Programming
```

The count() Method

This method returns the number of occurrences of a specified substring in the given string.

Example:



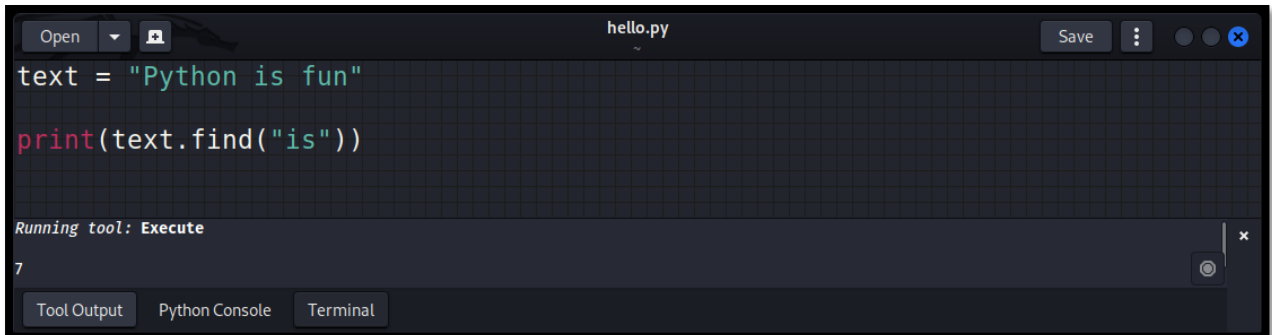
```
Open hello.py Save
text = "apple banana apple orange"
print(text.count("apple"))
Running tool: Execute
2
Tool Output Python Console Terminal
```

Output: 2

The find() and index() Methods

Both methods return the index of the first occurrence of a specified substring. If the substring is not found, **find()** returns **-1**, while **index()** raises a **ValueError**.

Example:



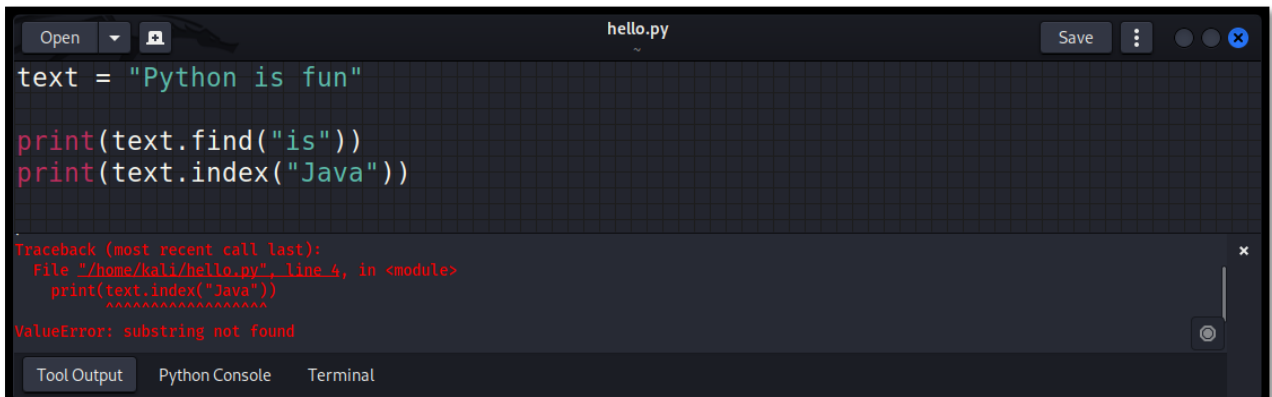
```
hello.py
text = "Python is fun"
print(text.find("is"))
```

Running tool: Execute

7

Tool Output Python Console Terminal

Output: 7



```
hello.py
text = "Python is fun"
print(text.find("is"))
print(text.index("Java"))
```

Traceback (most recent call last):
File "/home/kali/hello.py", line 4, in <module>
print(text.index("Java"))
^^^^^^^^^^^^^^^^^^^^
ValueError: substring not found

Tool Output Python Console Terminal

This will raise a ValueError.

String Formatting

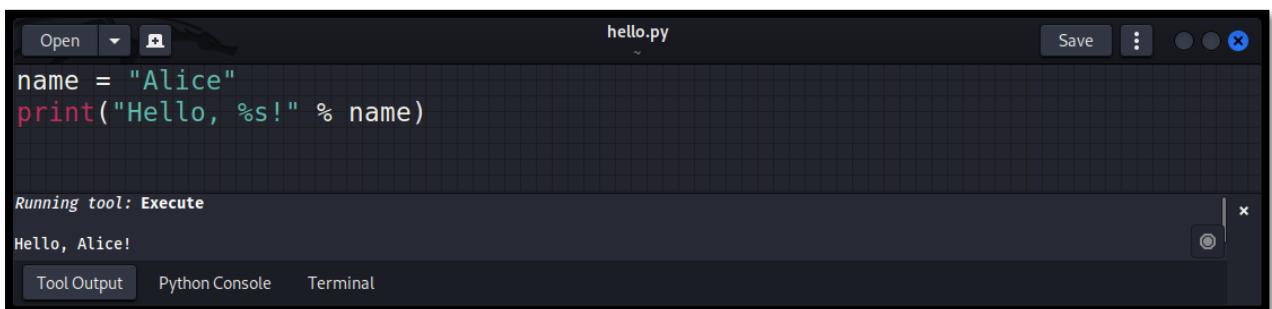
String formatting is a powerful feature in Python that allows for the creation of strings by embedding expressions inside string literals. This capability is essential for tasks like data presentation, logging, and generating user-friendly messages.

The % Operator

One of the oldest ways to format strings in Python is using the % operator, reminiscent of the printf-style string formatting found in languages like C.

Basic Usage

Example:



```
Open hello.py Save
```

```
name = "Alice"
print("Hello, %s!" % name)
```

Running tool: Execute

Hello, Alice!

Tool Output Python Console Terminal

Output: Hello, Alice!

Multiple Substitutions

Example:



```
Open hello.py Save
```

```
name = "Alice"
age = 30
print("%s is %d years old." % (name, age))
```

Running tool: Execute

Alice is 30 years old.

Tool Output Python Console Terminal

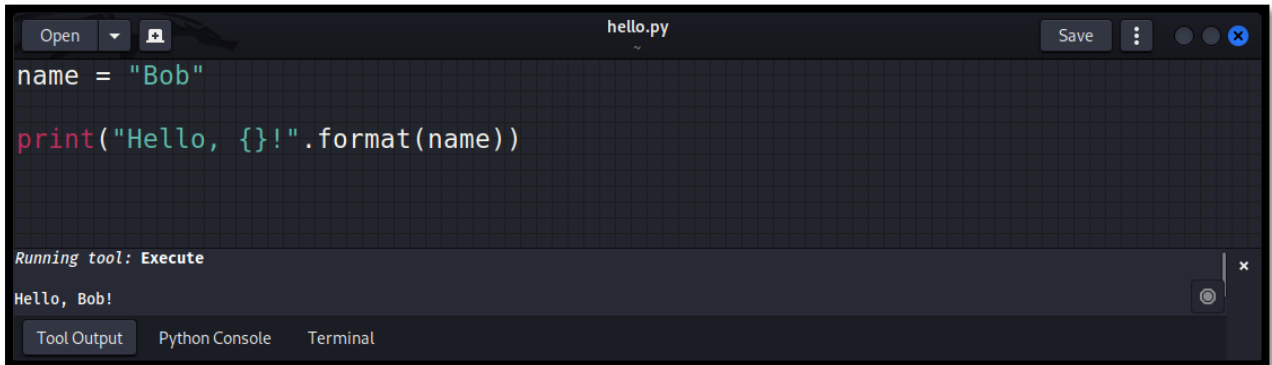
Output: Alice is 30 years old.

The str.format() Method

Introduced in Python 2.6, the `str.format()` method offers more flexibility compared to the `%` operator.

Basic Formatting

Example:



```
name = "Bob"
print("Hello, {}".format(name))
```

Running tool: Execute

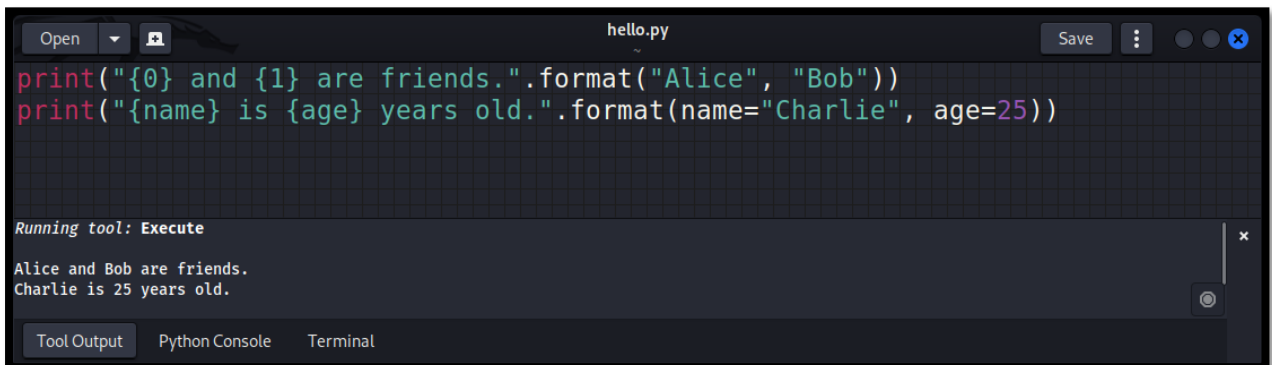
Hello, Bob!

Tool Output Python Console Terminal

Output: Hello, Bob!

Positional and Keyword Arguments

Example:



```
print("{0} and {1} are friends.".format("Alice", "Bob"))
print("{name} is {age} years old.".format(name="Charlie", age=25))
```

Running tool: Execute

Alice and Bob are friends.
Charlie is 25 years old.

Tool Output Python Console Terminal

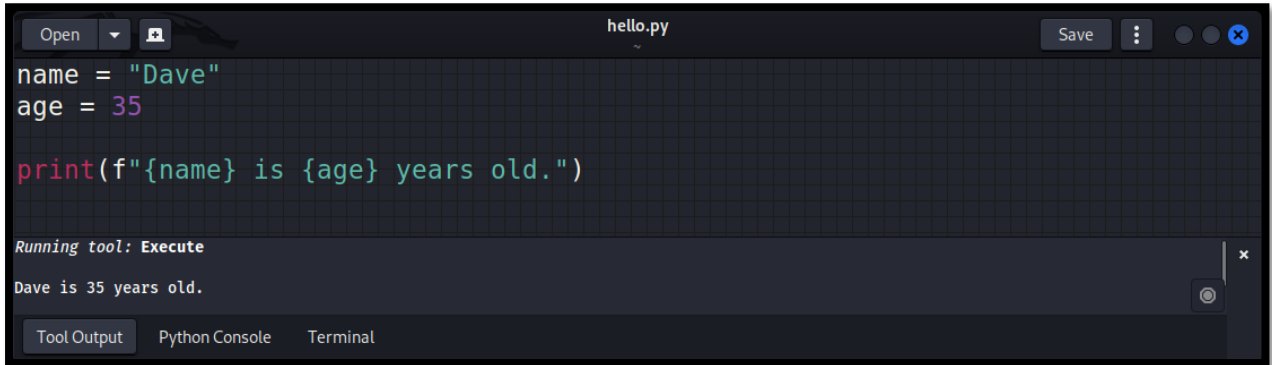
Output:

Alice and Bob are friends.
Charlie is 25 years old.

F-Strings (Formatted String Literals)

Introduced in Python 3.6, f-strings offer a concise and convenient way to embed expressions inside string literals.

Example:



```
name = "Dave"
age = 35

print(f"{name} is {age} years old.")
```

Running tool: Execute

Dave is 35 years old.

Tool Output Python Console Terminal

Output: Dave is 35 years old.

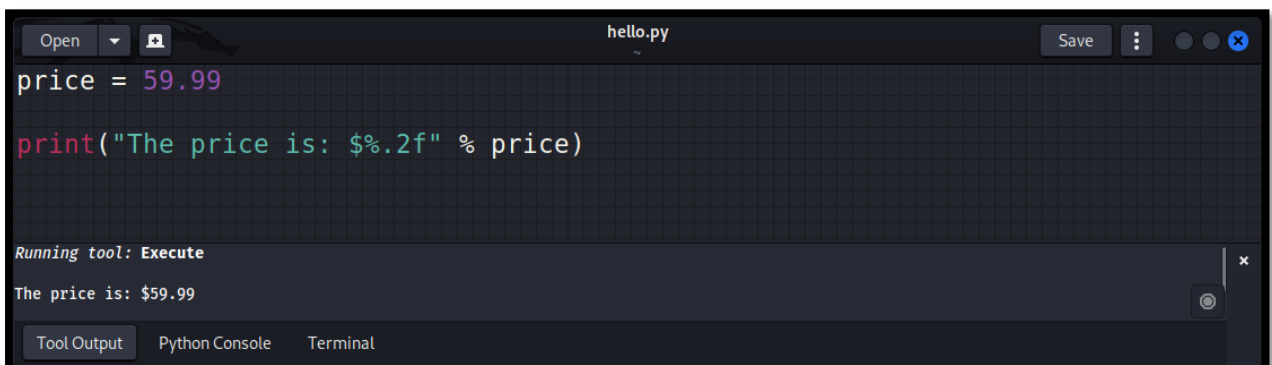
Expressions inside the curly braces are evaluated at runtime.

Formatting Numbers

Python provides ways to format numbers as strings, which is particularly useful for displaying monetary values, percentages, or fixed decimal places.

Using the % Operator

Example:



```
price = 59.99

print("The price is: $%.2f" % price)
```

Running tool: Execute

The price is: \$59.99

Tool Output Python Console Terminal

Output: The price is: \$59.99

Using the str.format() Method

Example:



```
hello.py
price = 59.99
print("The price is: ${:.2f}".format(price))
```

Running tool: Execute

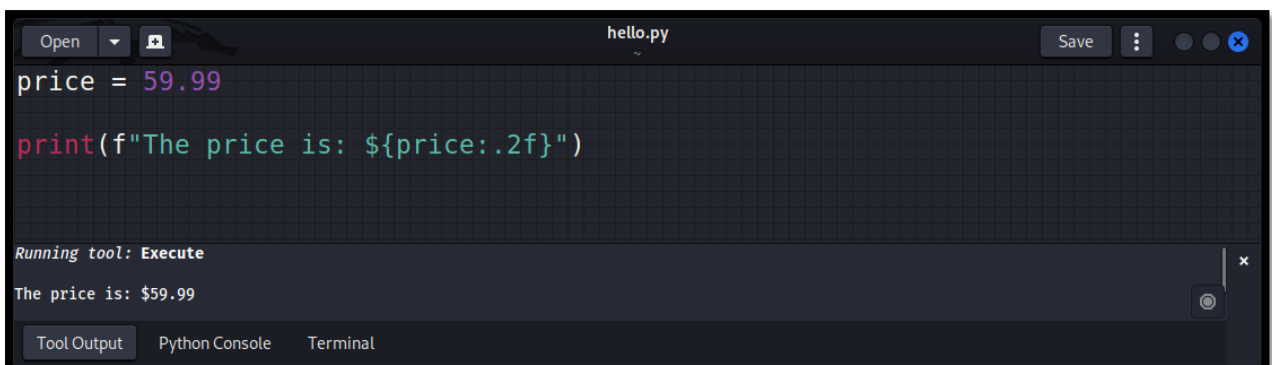
The price is: \$59.99

Tool Output Python Console Terminal

Output: The price is: \$59.99

Using F-Strings

Example:



```
hello.py
price = 59.99
print(f"The price is: ${price:.2f}")
```

Running tool: Execute

The price is: \$59.99

Tool Output Python Console Terminal

Output: The price is: \$59.99

Loops

The for Loop

The **for** loop in Python is a powerful control flow tool that allows developers to iterate over items in a sequence (such as a list, tuple, or string) or other iterable objects.

Basic Structure of the for Loop

The **for** loop works by iterating over each item in a sequence, executing a block of code for each item.

Syntax:

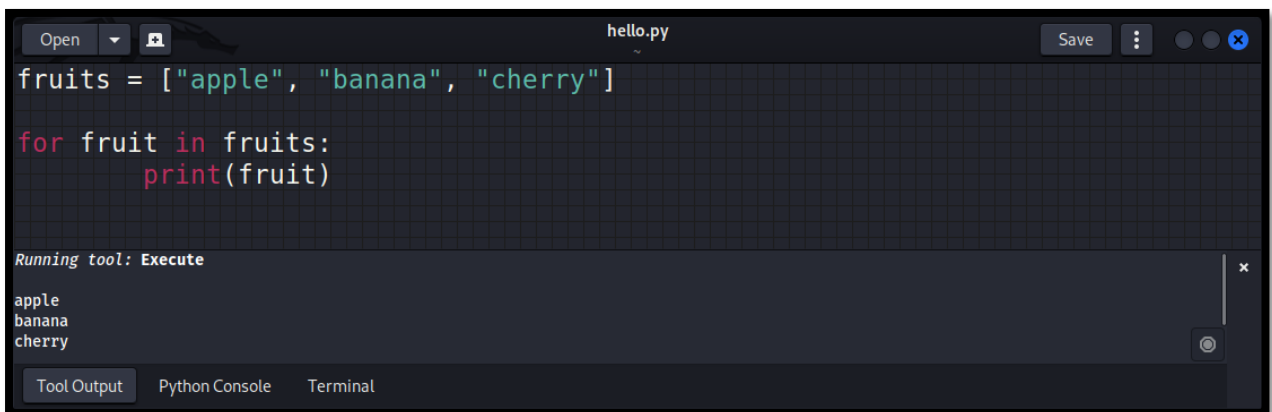
for item in sequence:

```
    # Code to execute for each item
```

Iterating Over Lists

One of the most common uses of the **for** loop is to iterate over lists.

Example:



```
hello.py
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

Running tool: Execute

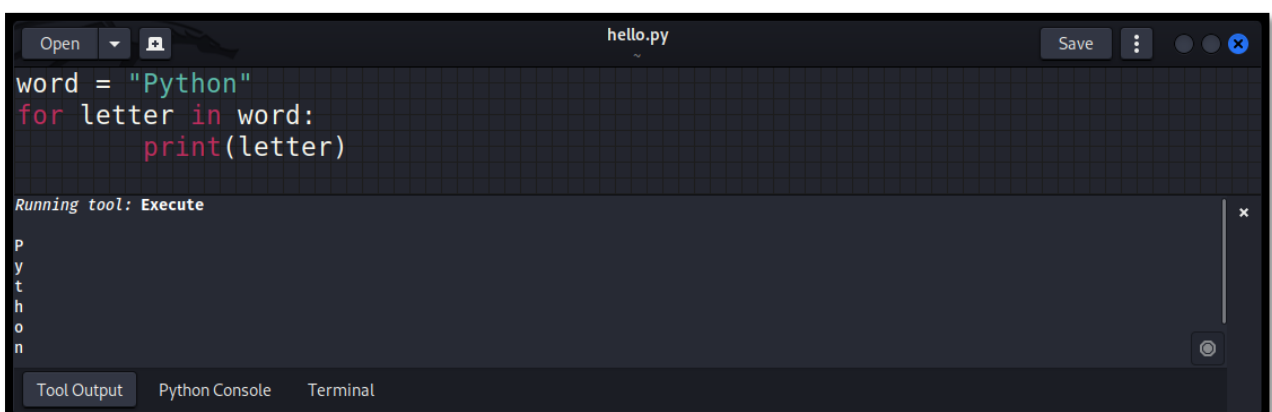
```
apple
banana
cherry
```

Tool Output Python Console Terminal

Iterating Over Strings

Strings are sequences of characters, so you can use a **for** loop to iterate over each character.

Example:



```
hello.py
word = "Python"

for letter in word:
    print(letter)
```

Running tool: Execute

```
P
y
t
h
o
n
```

Tool Output Python Console Terminal

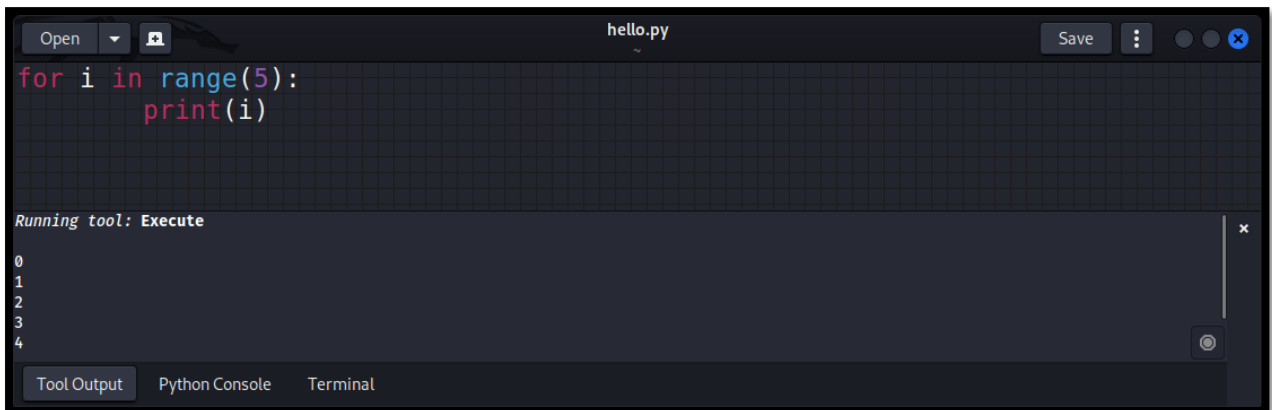
The range() Function

The `range()` function is often used with the `for` loop to generate a sequence of numbers.

Syntax:

- `range(stop)`: Generates numbers from 0 up to (but not including) `stop`.
- `range(start, stop)`: Generates numbers from `start` up to (but not including) `stop`.
- `range(start, stop, step)`: Generates numbers from `start` up to (but not including) `stop`, incremented by `step`.

Example:



```
Open hello.py Save
```

```
for i in range(5):  
    print(i)
```

Running tool: Execute

```
0  
1  
2  
3  
4
```

Tool Output Python Console Terminal

Nested for Loops

You can nest `for` loops inside other `for` loops to create more complex iterations.

Example:



```
Open hello.py Save
```

```
for i in range(3):  
    for j in range(2):  
        print(f"i = {i}, j = {j}")
```

Running tool: Execute

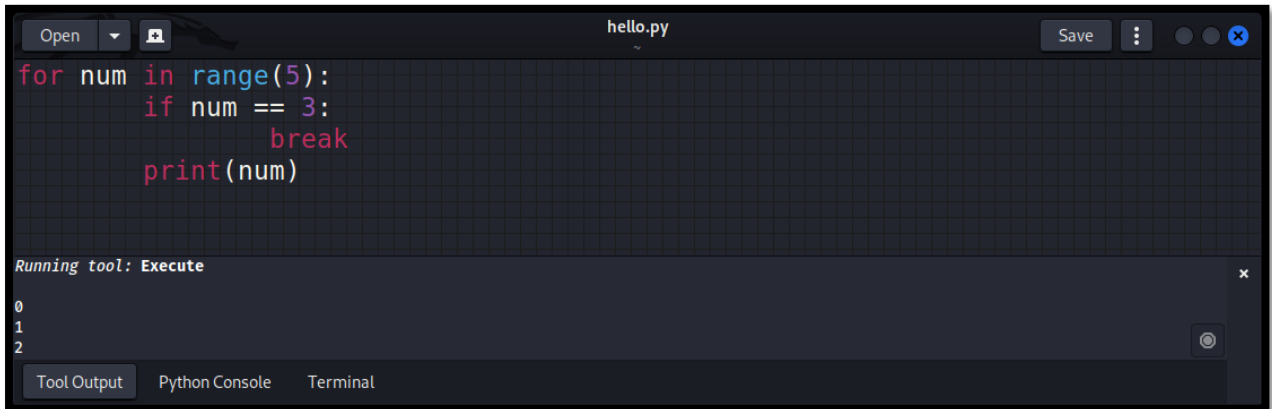
```
i = 0, j = 0  
i = 0, j = 1  
i = 1, j = 0  
i = 1, j = 1  
i = 2, j = 0  
i = 2, j = 1
```

Tool Output Python Console Terminal

The break and continue Statements

- The **break** statement is used to exit the **for** loop prematurely.
- The **continue** statement skips the rest of the current iteration and moves to the next one.

Example:



```
Open hello.py Save
```

```
for num in range(5):  
    if num == 3:  
        break  
    print(num)
```

Running tool: Execute

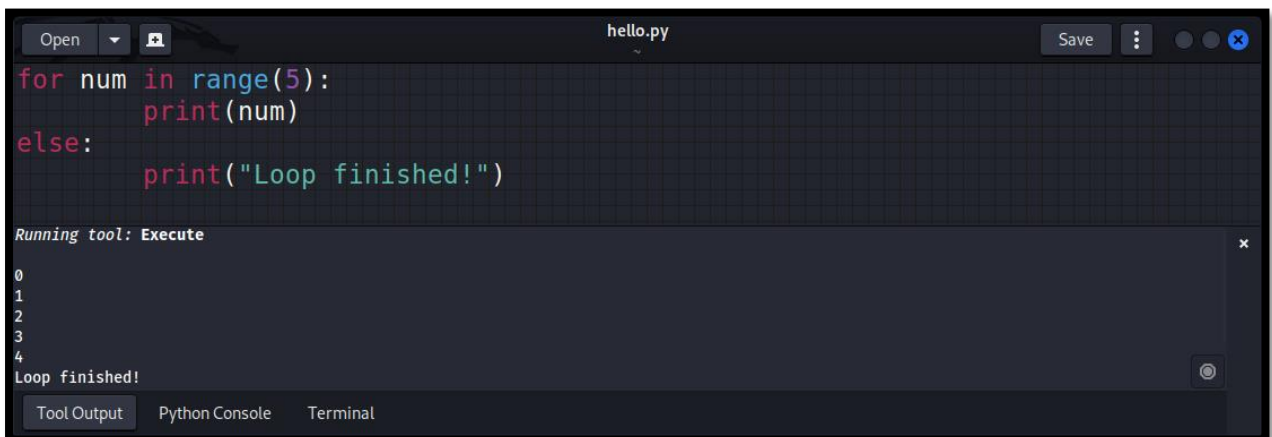
```
0  
1  
2
```

Tool Output Python Console Terminal

The else Clause in for Loops

In Python, the **for** loop can have an optional **else** clause, which is executed after the loop finishes, but only if the loop wasn't terminated by a **break** statement.

Example:



```
Open hello.py Save
```

```
for num in range(5):  
    print(num)  
else:  
    print("Loop finished!")
```

Running tool: Execute

```
0  
1  
2  
3  
4  
Loop finished!
```

Tool Output Python Console Terminal

Output:

```
0  
1  
2  
3  
4  
Loop finished!
```

The while Loop

The **while** loop in Python is used to repeatedly execute a block of code as long as a specified condition is **True**. Unlike the **for** loop, which runs a predetermined number of times based on a sequence, the **while** loop runs until its condition becomes **False**.

Basic Structure of the while Loop

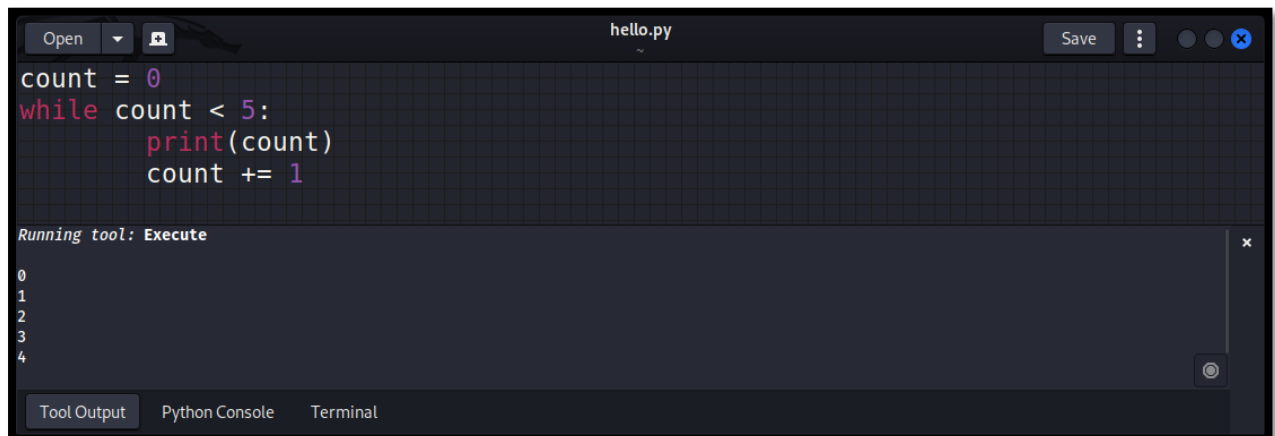
The **while** loop tests a condition and executes its block of code if the condition is **True**. After each iteration, the condition is re-evaluated.

Syntax:

```
while condition:  
    # Code to execute while condition is True
```

Basic while Loop Example

Example:

A screenshot of a Python IDE window titled 'hello.py'. The code in the editor is:

```
count = 0  
while count < 5:  
    print(count)  
    count += 1
```

Below the code editor is a terminal window titled 'Running tool: Execute' showing the output of the program:

```
0  
1  
2  
3  
4
```

The IDE interface includes 'Open', 'Save', and window control buttons.

Output:

```
0  
1  
2  
3  
4
```

Infinite while Loops

If the condition in a **while** loop always evaluates to **True**, the loop will run indefinitely, creating an infinite loop.

Example:

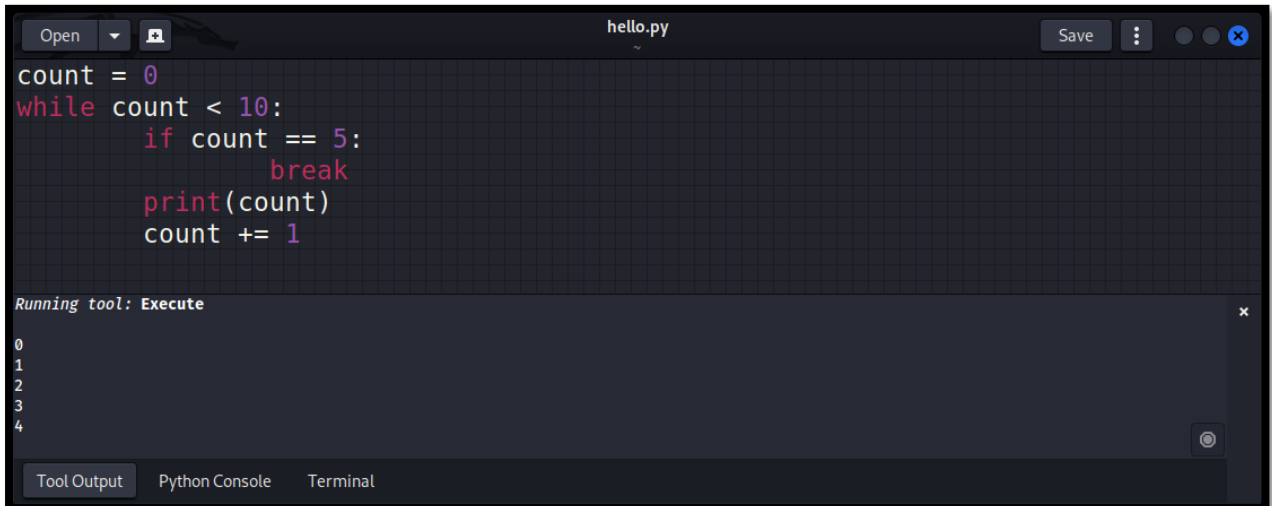
```
# while True:  
    # print("This will print forever!")
```

It's crucial to ensure that the loop's condition will eventually become **False** to avoid unintentional infinite loops.

The break and continue Statements in while Loops

- The **break** statement exits the **while** loop prematurely.
- The **continue** statement skips the rest of the current iteration and returns to the loop's condition.

Example:



```
count = 0
while count < 10:
    if count == 5:
        break
    print(count)
    count += 1
```

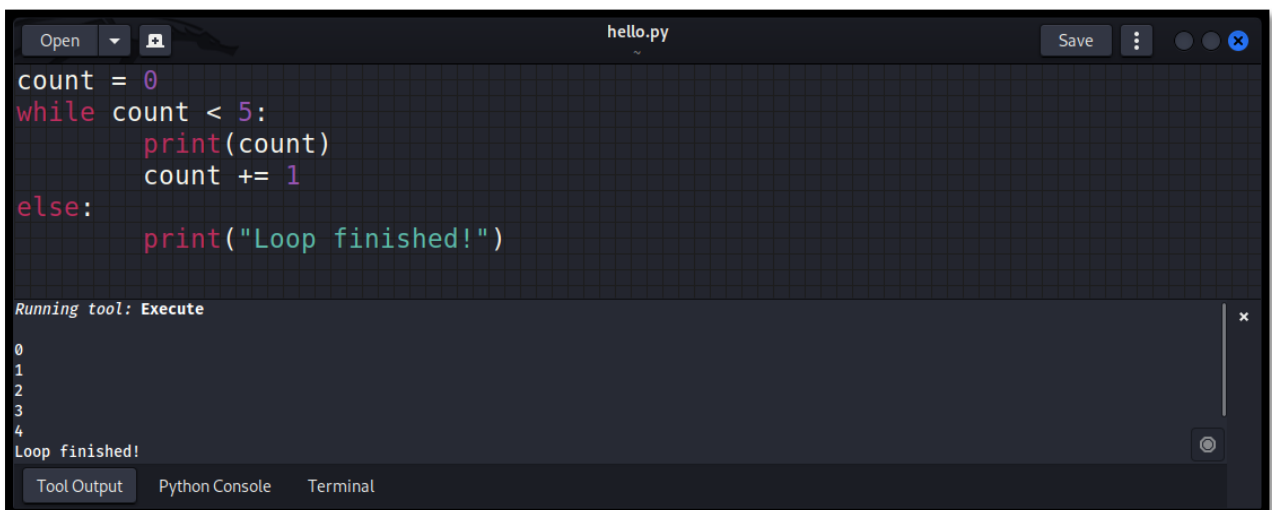
Running tool: Execute

```
0
1
2
3
4
```

The else Clause in while Loops

Similar to the **for** loop, the **while** loop in Python can also have an optional **else** clause. It is executed after the loop finishes, but only if the loop wasn't terminated by a **break** statement.

Example:



```
count = 0
while count < 5:
    print(count)
    count += 1
else:
    print("Loop finished!")
```

Running tool: Execute

```
0
1
2
3
4
Loop finished!
```

Practical Applications of the while Loop

The **while** loop is particularly useful in scenarios where the number of iterations is not known in advance, such as reading user input until a valid response is given.

Example:

A screenshot of a code editor window titled 'hello.py'. The code is as follows:

```
user_input = ""
while user_input != "exit":
    user_input = input("Enter a command (type 'exit' to quit): ")
print(f"You entered: {user_input}")
```

Loop Control Statements (break, continue, pass)

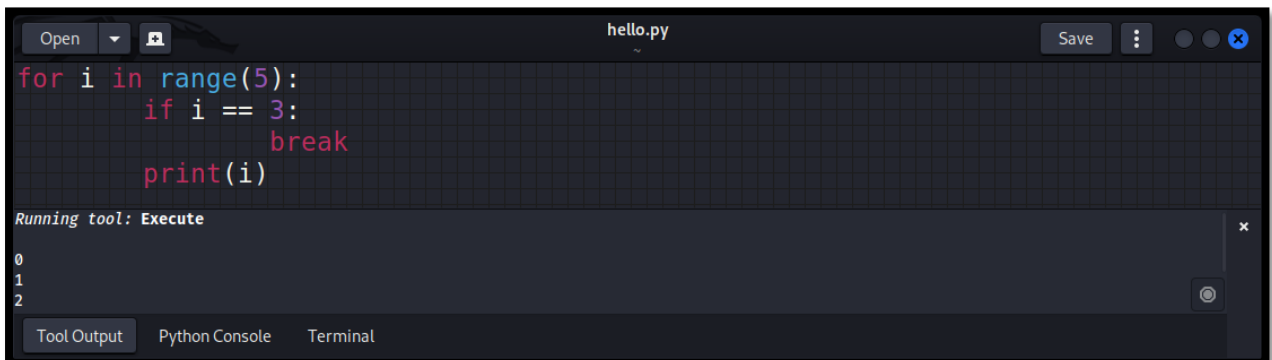
Introduction to Loop Control Statements

Loop control statements modify the behavior of loops in Python. They allow developers to manage the flow of execution within loops, making them more dynamic and responsive to specific conditions.

The break Statement

The **break** statement is used to exit a loop prematurely, terminating its execution before it would naturally finish.

Example:

A screenshot of a code editor window titled 'hello.py'. The code is as follows:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Below the code, there is a terminal window titled 'Running tool: Execute' showing the output:

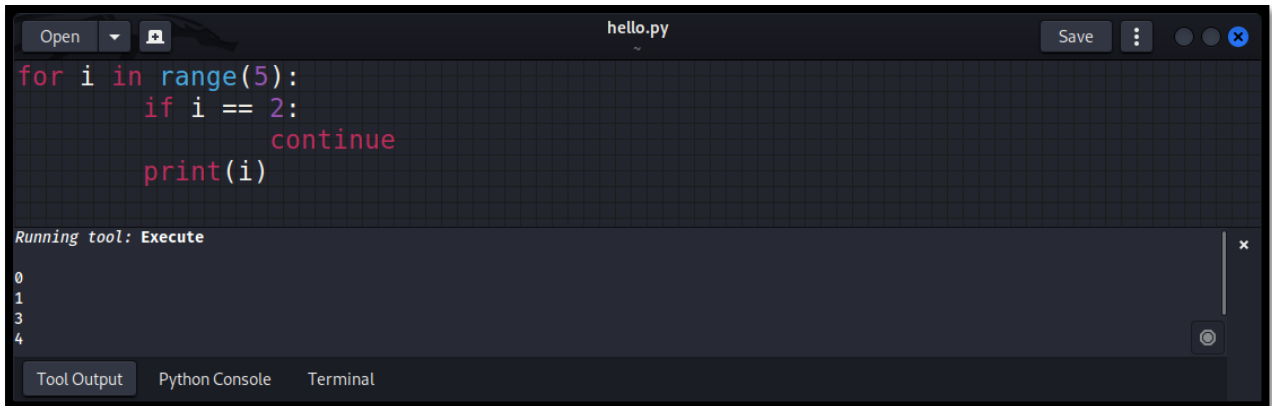
```
0
1
2
```

In the example above, the loop is terminated when *i* equals 3, and thus only the numbers 0, 1, and 2 are printed.

The **continue** Statement

The **continue** statement skips the current iteration of a loop and moves to the next one. It's useful for bypassing specific parts of the loop for certain conditions.

Example:



```
Open hello.py Save
for i in range(5):
    if i == 2:
        continue
    print(i)

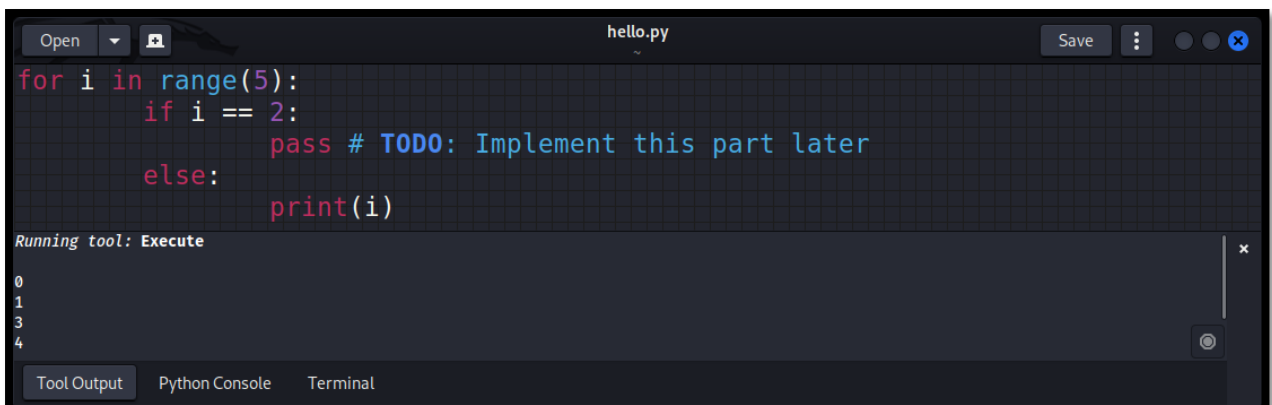
Running tool: Execute
0
1
3
4
Tool Output Python Console Terminal
```

In the example above, the number 2 is skipped due to the **continue** statement, and the loop continues with the next iteration.

The **pass** Statement

The **pass** statement is a null operation — nothing happens when it's executed. It's often used as a placeholder, allowing developers to define a loop or a function syntactically but delay the implementation of its content.

Example:



```
Open hello.py Save
for i in range(5):
    if i == 2:
        pass # TODO: Implement this part later
    else:
        print(i)

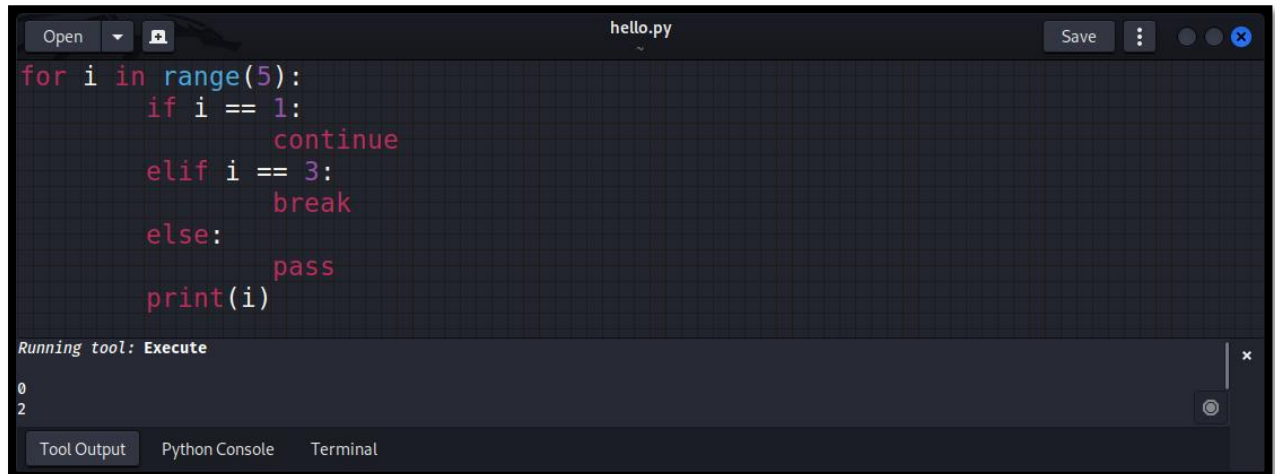
Running tool: Execute
0
1
3
4
Tool Output Python Console Terminal
```

In the example above, the **pass** statement serves as a placeholder for future code related to the condition `i == 2`. The loop continues to execute normally for all other values of `i`.

Combining Loop Control Statements

Loop control statements can be combined in various ways to achieve more complex control flows within loops.

Example:



```
hello.py
for i in range(5):
    if i == 1:
        continue
    elif i == 3:
        break
    else:
        pass
    print(i)

Running tool: Execute
0
2
Tool Output Python Console Terminal
```

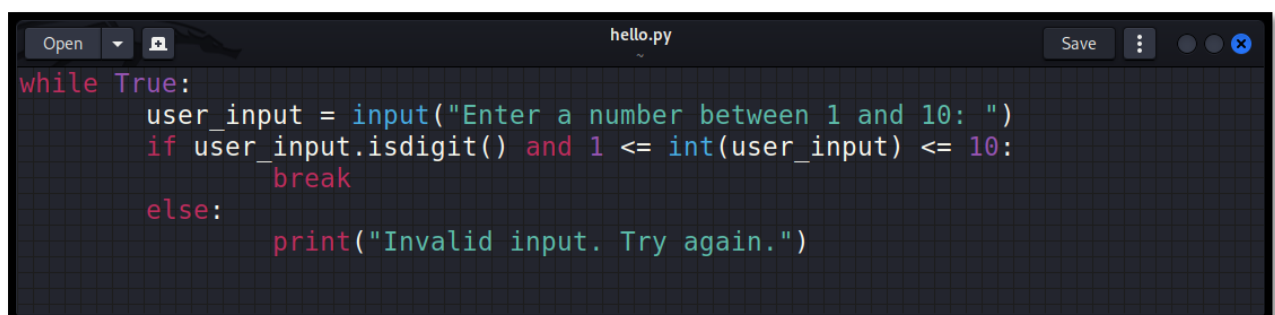
In the example above, the number 1 is skipped due to the **continue** statement, the loop is terminated when *i* equals 3 because of the **break** statement, and the **pass** statement serves as a placeholder for potential future code.

Practical Applications of Loop Control Statements

Loop control statements are invaluable when dealing with dynamic data or user input, where the behavior of the loop needs to adjust based on specific conditions.

Example:

Imagine a program that prompts users for input until they provide a valid response:



```
hello.py
while True:
    user_input = input("Enter a number between 1 and 10: ")
    if user_input.isdigit() and 1 <= int(user_input) <= 10:
        break
    else:
        print("Invalid input. Try again.")
```

In this example, the **break** statement ensures that the loop terminates once the user provides a valid input.

Nested Loops

Nested loops refer to loops placed inside other loops. They allow developers to iterate over multiple dimensions or levels, making them essential for tasks like iterating over multi-dimensional arrays or generating patterns.

Basics of Nested Loops

A loop becomes nested when another loop is defined within its block of code. Both **for** and **while** loops can be nested within each other in any combination.

Syntax:

```
for outer_variable in outer_sequence:
    for inner_variable in inner_sequence:
        # Code to execute for each combination of outer and inner loop
```

Example:



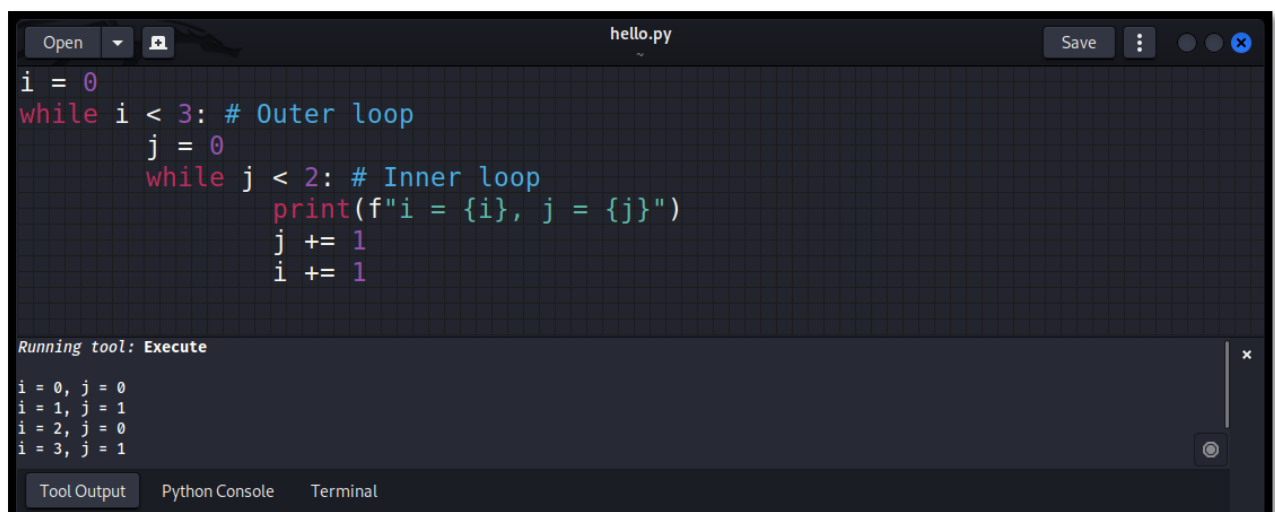
```
Open hello.py Save
for i in range(3): # Outer loop
    for j in range(2): # Inner loop
        print(f"i = {i}, j = {j}")

Running tool: Execute
i = 0, j = 0
i = 0, j = 1
i = 1, j = 0
i = 1, j = 1
i = 2, j = 0
i = 2, j = 1
Tool Output Python Console Terminal
```

Nested while Loops

Just like **for** loops, **while** loops can also be nested to create more complex iterations.

Example:



```
Open hello.py Save
i = 0
while i < 3: # Outer loop
    j = 0
    while j < 2: # Inner loop
        print(f"i = {i}, j = {j}")
        j += 1
    i += 1

Running tool: Execute
i = 0, j = 0
i = 1, j = 1
i = 2, j = 0
i = 3, j = 1
Tool Output Python Console Terminal
```

Conditions

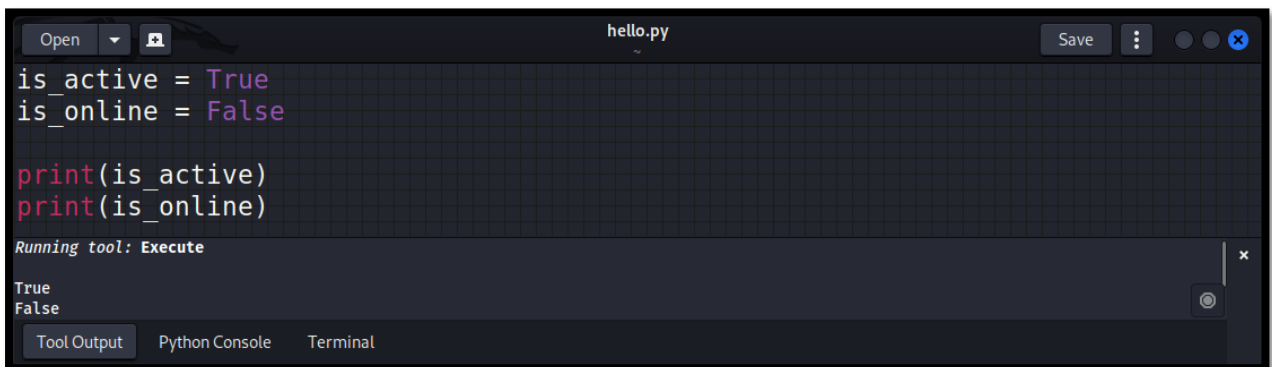
Boolean Expressions

Boolean expressions are foundational to programming, allowing developers to make decisions and control the flow of execution. In Python, a boolean expression evaluates to one of two values: **True** or **False**.

Basic Boolean Values

In Python, the two boolean values are represented by the keywords **True** and **False**.

Example:



```

Open | hello.py | Save
is_active = True
is_online = False

print(is_active)
print(is_online)

Running tool: Execute
True
False
Tool Output | Python Console | Terminal

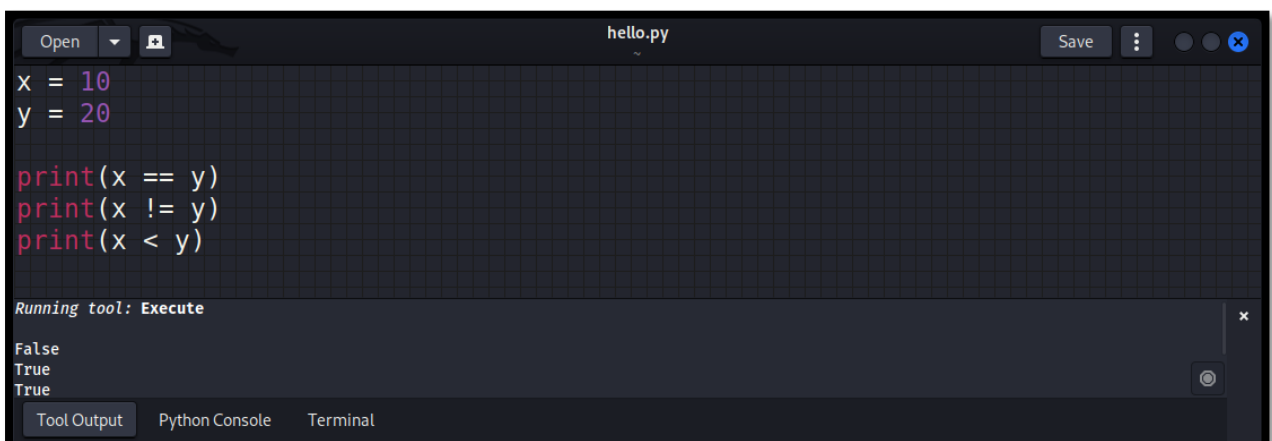
```

Comparison Operators

Comparison operators are used to compare two values and return a boolean result.

Operator	Description	Example	Result
==	Equal to	5 == 3	False
!=	Not equal to	5 != 3	True
>	Greater than	5 > 3	True
<	Less than	5 < 3	False
>=	Greater than or equal	5 >= 5	True
<=	Less than or equal	5 <= 3	False

Example:



```

Open | hello.py | Save
x = 10
y = 20

print(x == y)
print(x != y)
print(x < y)

Running tool: Execute
False
True
True
Tool Output | Python Console | Terminal

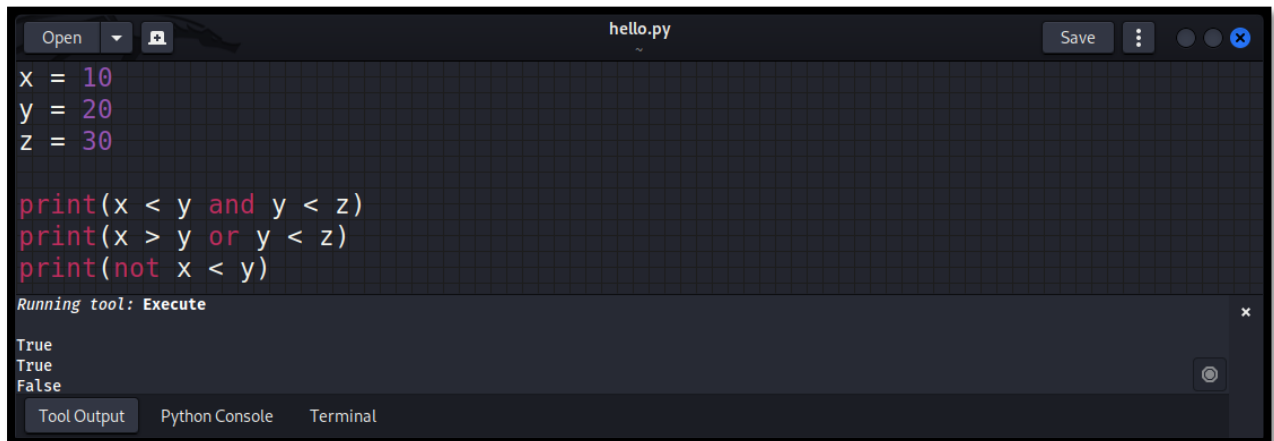
```

Logical Operators

Logical operators are used to combine multiple boolean expressions.

Operator	Description	Example	Result
and	Logical AND	(5 > 3) and (5 > 4)	True
or	Logical OR	(5 > 3) or (5 < 4)	True
not	Logical NOT	not (5 > 3)	False

Example:



```
hello.py
x = 10
y = 20
z = 30

print(x < y and y < z)
print(x > y or y < z)
print(not x < y)

Running tool: Execute
True
True
False
```

The in and not in Operators

These operators are used to test if a value is present in a sequence (like a list, tuple, or string).

Example:



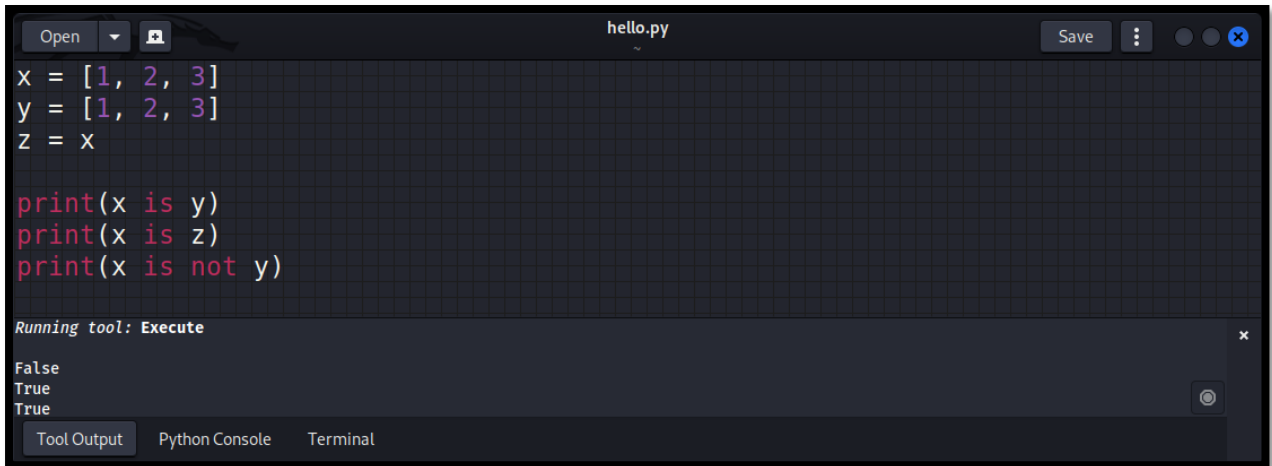
```
hello.py
fruits = ["apple", "banana", "cherry"]
print("apple" in fruits)
print("orange" not in fruits)

Running tool: Execute
True
True
```

The is and is not Operators

These operators compare the memory locations of two objects, not their content. They are used to check if two variables refer to the same object.

Example:



```
hello.py
Open Save
x = [1, 2, 3]
y = [1, 2, 3]
z = x

print(x is y)
print(x is z)
print(x is not y)

Running tool: Execute
False
True
True
Tool Output Python Console Terminal
```

Truthy and Falsy Values

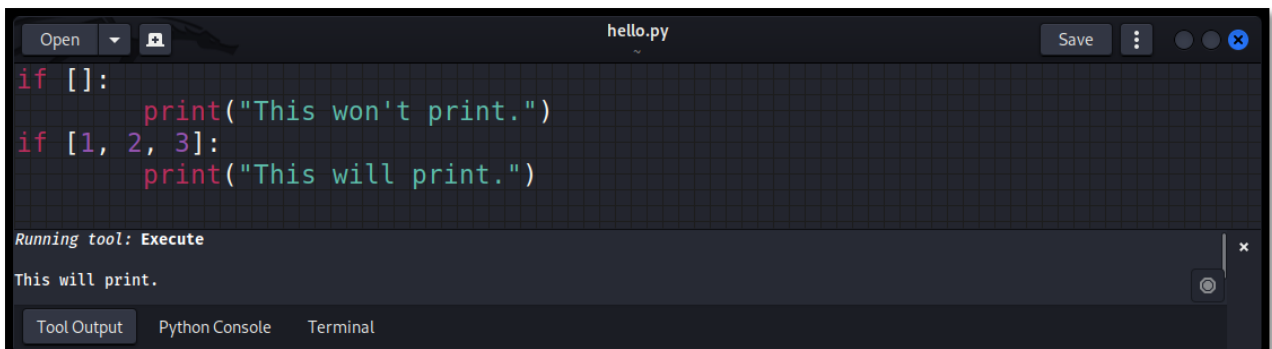
In contexts where a boolean value is expected, Python will treat certain values as "truthy" (evaluating to **True**) and others as "falsy" (evaluating to **False**).

Falsy values include:

- **None**
- **False**
- Zero of any numeric type (**0**, **0.0**, **0j**, etc.)
- Any empty sequence (**[]**, **()**, **""**, etc.)
- Any empty mapping (**{}**)
- Custom objects that implement a **__bool__()** or **__len__()** method that returns **False** or **0**, respectively.

All other values are considered truthy.

Example:



```
hello.py
Open Save
if []:
    print("This won't print.")
if [1, 2, 3]:
    print("This will print.")

Running tool: Execute
This will print.
Tool Output Python Console Terminal
```


Comparison Operators

Comparison operators are fundamental to programming, allowing developers to compare values and make decisions based on the results of those comparisons. In Python, comparison operators return a boolean value, either **True** or **False**, depending on the outcome of the comparison.

List of Comparison Operators

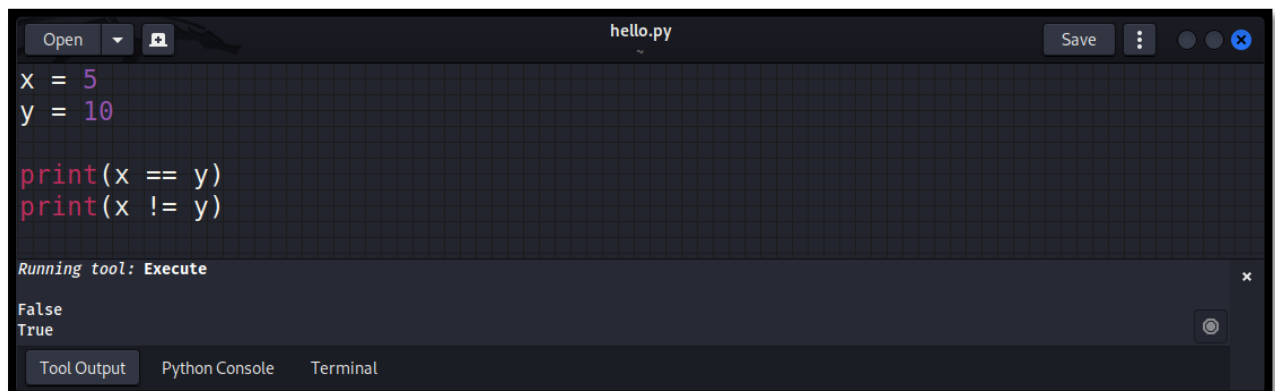
Here's a list of the primary comparison operators in Python:

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Using the == and != Operators

The == operator checks if two values are equal, while the != operator checks if they are not equal.

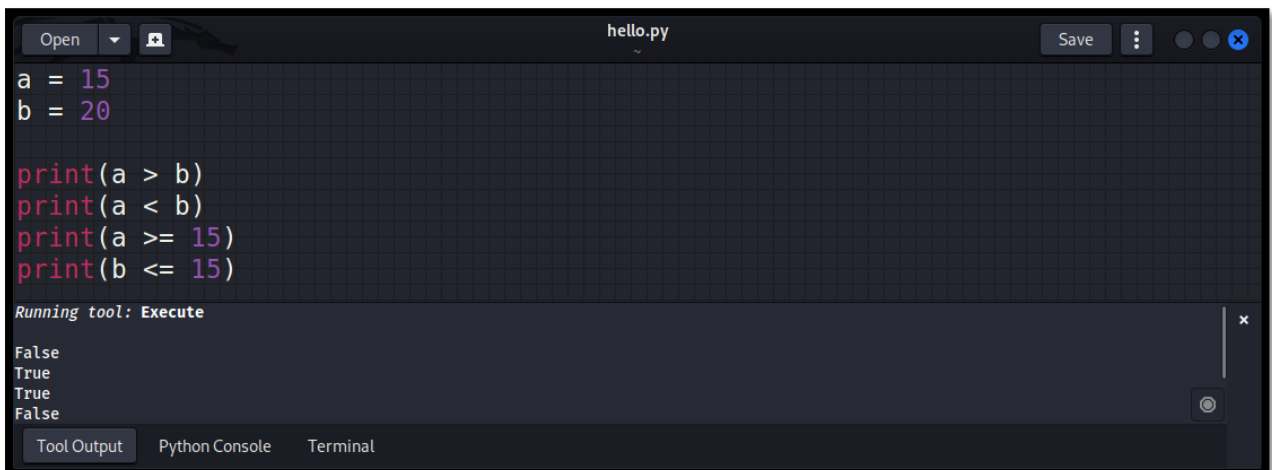
Example:



```
hello.py
Open Save
x = 5
y = 10
print(x == y)
print(x != y)
Running tool: Execute
False
True
Tool Output Python Console Terminal
```

Using the >, <, >=, and <= Operators

These operators are used to compare the magnitude of values.

Example:

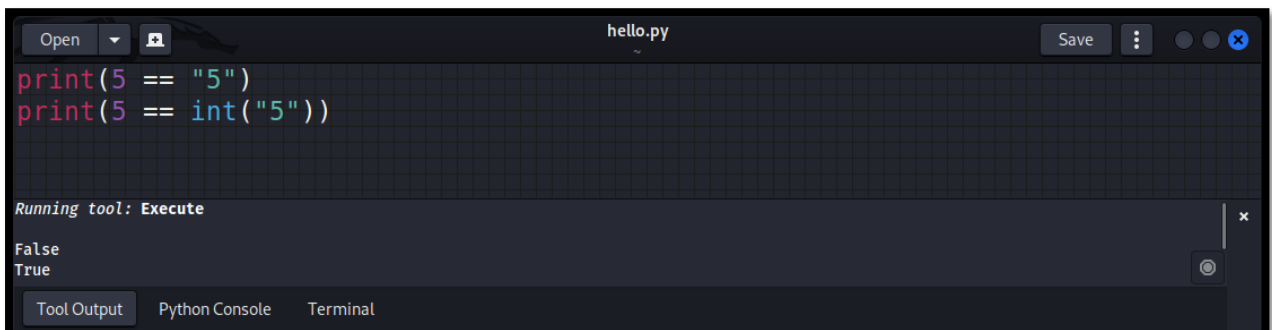
```
Open hello.py Save
a = 15
b = 20

print(a > b)
print(a < b)
print(a >= 15)
print(b <= 15)

Running tool: Execute
False
True
True
False
Tool Output Python Console Terminal
```

Comparing Different Data Types

In Python, you can compare different data types. However, doing so can lead to unexpected results if not done with caution.

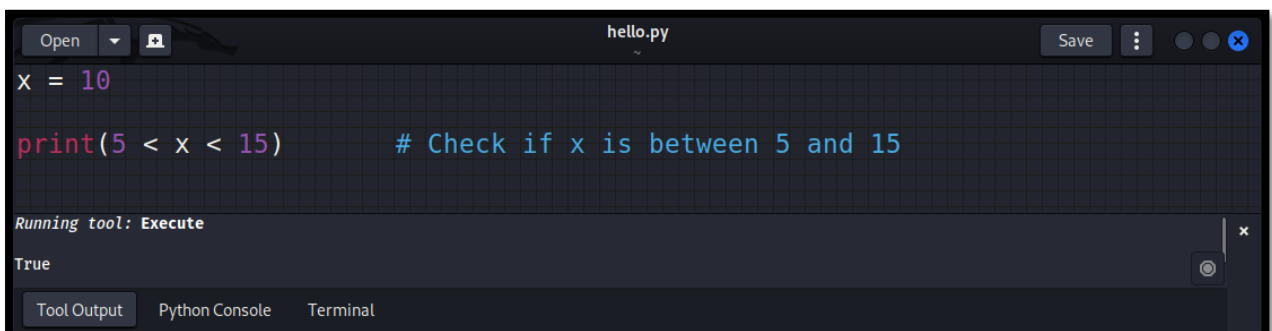
Example:

```
Open hello.py Save
print(5 == "5")
print(5 == int("5"))

Running tool: Execute
False
True
Tool Output Python Console Terminal
```

Chaining Comparison Operators

Python allows for the chaining of comparison operators, which can lead to more concise and readable conditions.

Example:

```
Open hello.py Save
x = 10

print(5 < x < 15)      # Check if x is between 5 and 15

Running tool: Execute
True
Tool Output Python Console Terminal
```

Logical Operators

Logical operators are used to combine multiple conditions or boolean expressions, allowing developers to create more complex and nuanced decision-making structures in their programs. In Python, logical operators return a boolean value: **True** or **False**, based on the evaluation of the combined conditions.

List of Logical Operators

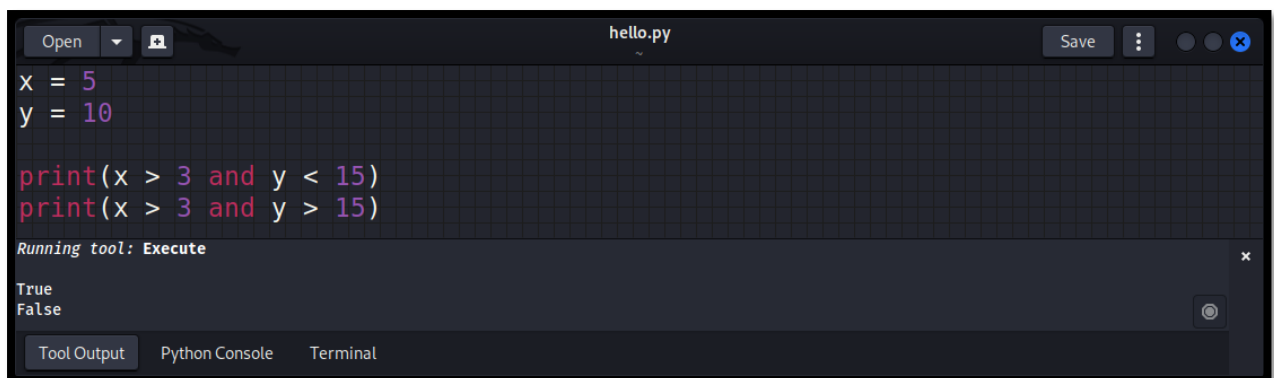
Here's a list of the primary logical operators in Python:

Operator	Description
and	Logical AND
or	Logical OR
not	Logical NOT

The and Operator

The **and** operator returns **True** if both the conditions it combines are true. Otherwise, it returns **False**.

Example:

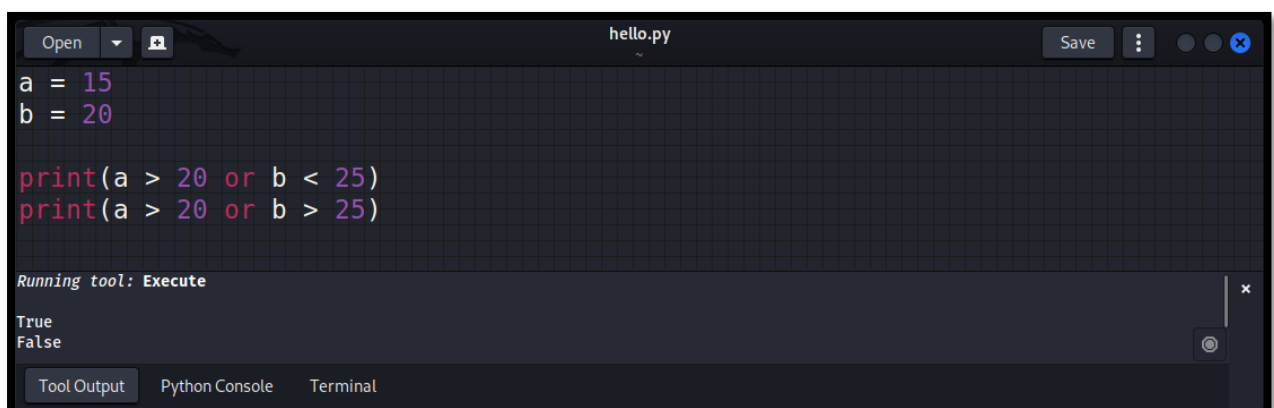


```
hello.py
Open Save
x = 5
y = 10
print(x > 3 and y < 15)
print(x > 3 and y > 15)
Running tool: Execute
True
False
Tool Output Python Console Terminal
```

The or Operator

The **or** operator returns **True** if at least one of the conditions it combines is true. If both conditions are false, it returns **False**.

Example:

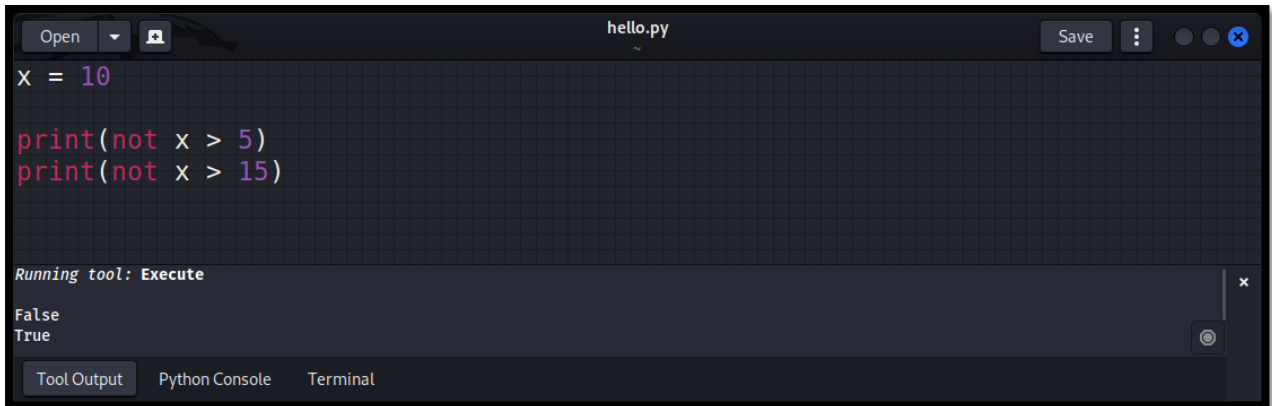


```
hello.py
Open Save
a = 15
b = 20
print(a > 20 or b < 25)
print(a > 20 or b > 25)
Running tool: Execute
True
False
Tool Output Python Console Terminal
```

The not Operator

The **not** operator inverts the result of the condition it precedes. If the condition is **True**, **not** will return **False**, and vice versa.

Example:



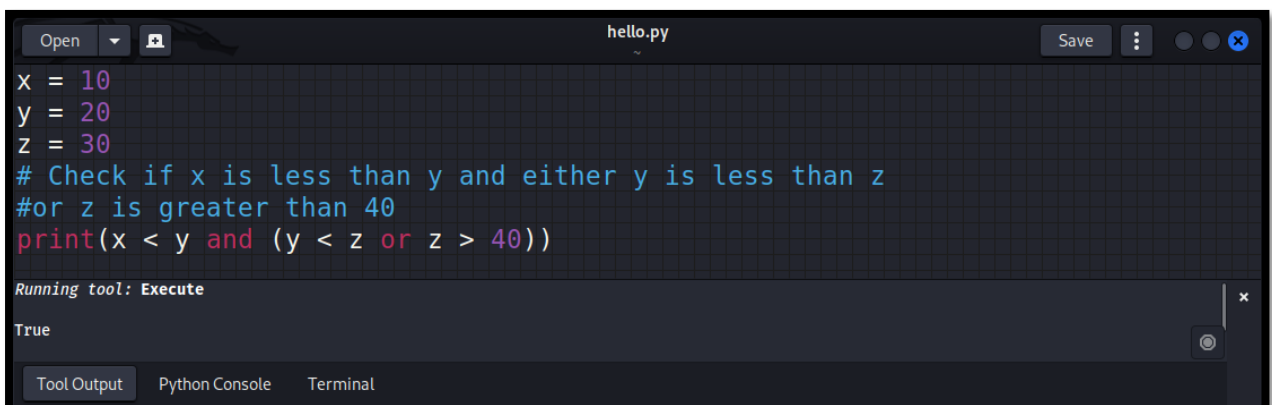
```
hello.py
Open Save
X = 10
print(not x > 5)
print(not x > 15)

Running tool: Execute
False
True
Tool Output Python Console Terminal
```

Combining Logical Operators

Logical operators can be combined to form more complex conditions.

Example:



```
hello.py
Open Save
X = 10
y = 20
z = 30
# Check if x is less than y and either y is less than z
# or z is greater than 40
print(x < y and (y < z or z > 40))

Running tool: Execute
True
Tool Output Python Console Terminal
```

The if, elif, and else Statements

Introduction to Conditional Statements

Conditional statements are a cornerstone of programming, allowing developers to execute specific blocks of code based on whether certain conditions are met. In Python, the **if**, **elif**, and **else** statements provide a flexible framework for handling these conditions.

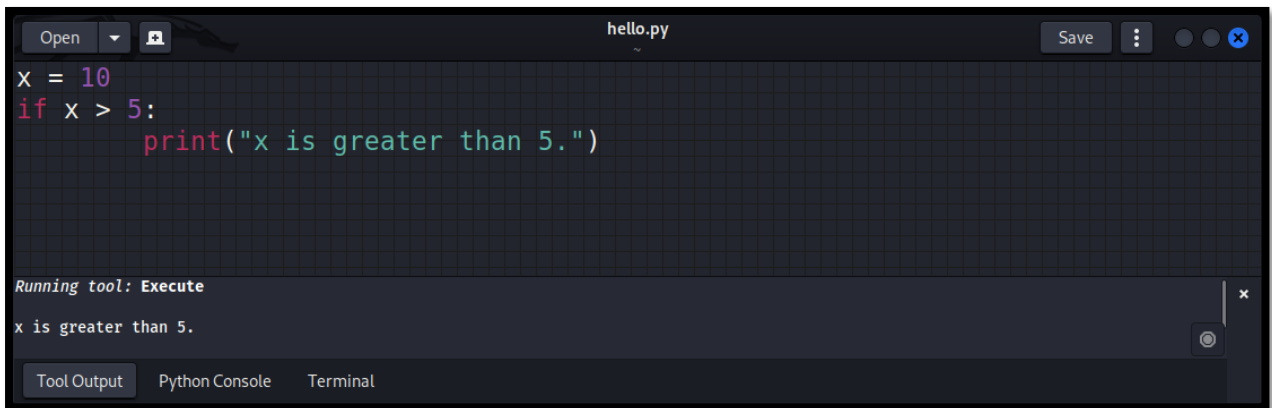
The if Statement

The **if** statement is used to test a condition and execute a block of code if that condition is **True**.

Syntax:

```
if condition: # code to execute if condition is True
```

Example:



The screenshot shows a code editor window titled 'hello.py' with the following Python code:

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

Below the code editor, the output console shows the result of running the code:

```
Running tool: Execute
x is greater than 5.
```

The IDE interface includes buttons for 'Tool Output', 'Python Console', and 'Terminal' at the bottom.

The else Statement

The **else** statement follows an **if** statement and defines a block of code to be executed if the condition in the **if** statement is **False**.

Syntax:

```
if condition: # code to execute if condition is True else: # code to execute if condition is False
```

Example:



The screenshot shows a code editor window titled 'hello.py' with the following Python code:

```
x = 3
if x > 5:
    print("x is greater than 5.")
else:
    print("x is not greater than 5.")
```

Below the code editor, the output console shows the result of running the code:

```
Running tool: Execute
x is not greater than 5.
```

The IDE interface includes buttons for 'Tool Output', 'Python Console', and 'Terminal' at the bottom.

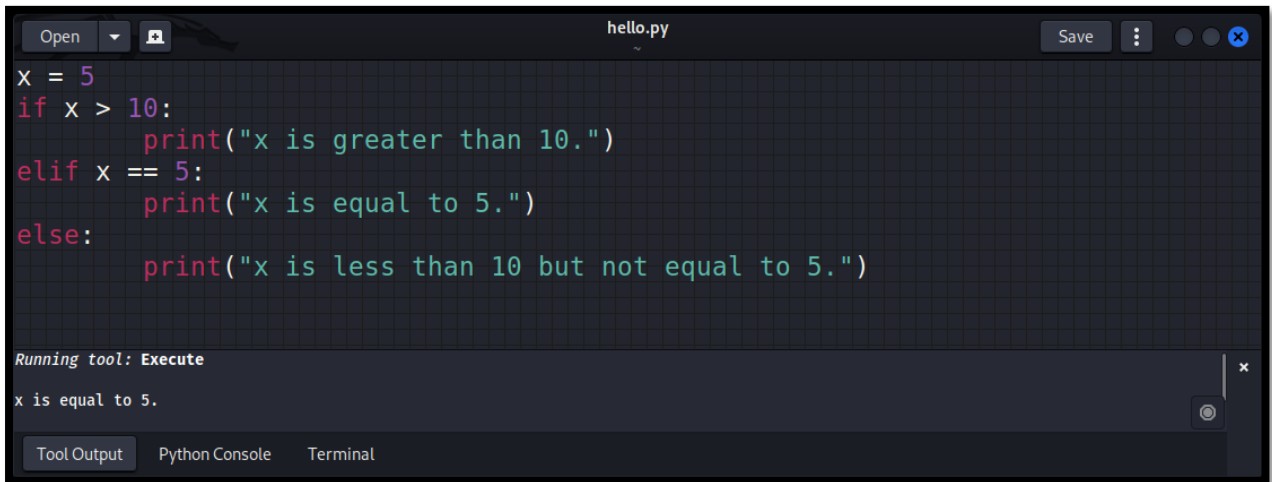
The elif Statement

The **elif** (short for "else if") statement allows for checking multiple conditions in sequence. If the condition in an **if** statement is **False**, the **elif** condition is checked next. You can have multiple **elif** statements to handle various conditions.

Syntax:

```
if condition1: # code to execute if condition1 is True
elif condition2: # code to execute if condition2 is True
else: # code to execute if no conditions are True
```

Example:



```
hello.py
x = 5
if x > 10:
    print("x is greater than 10.")
elif x == 5:
    print("x is equal to 5.")
else:
    print("x is less than 10 but not equal to 5.")
```

Running tool: Execute

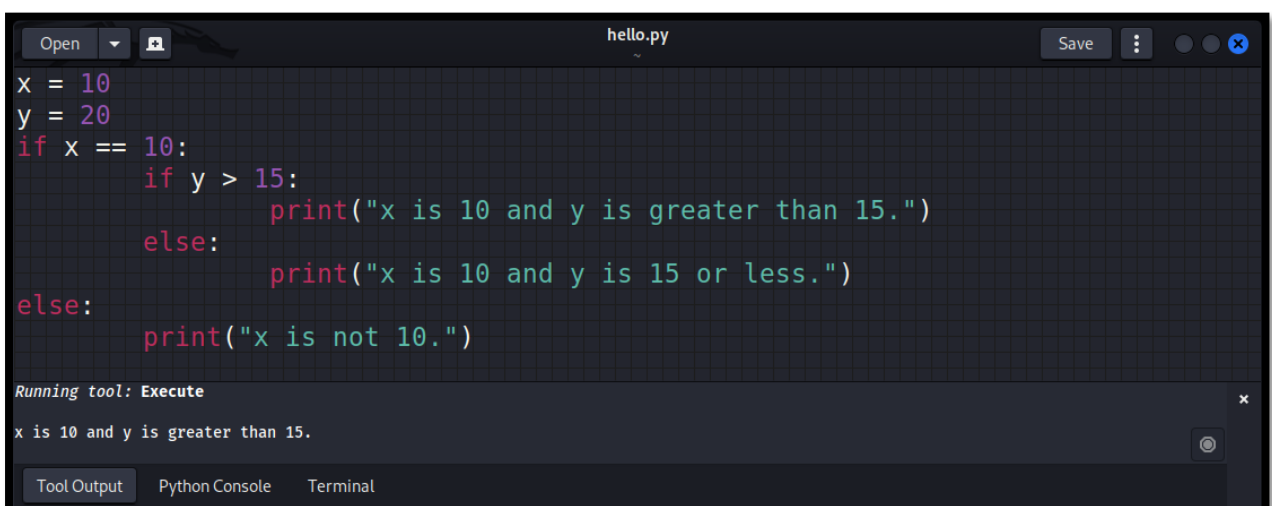
x is equal to 5.

Tool Output Python Console Terminal

Nested Conditional Statements

You can nest **if** statements within other **if**, **elif**, or **else** blocks, allowing for more complex decision-making structures.

Example:



```
hello.py
x = 10
y = 20
if x == 10:
    if y > 15:
        print("x is 10 and y is greater than 15.")
    else:
        print("x is 10 and y is 15 or less.")
else:
    print("x is not 10.")
```

Running tool: Execute

x is 10 and y is greater than 15.

Tool Output Python Console Terminal

Nested Conditions

Nested conditions refer to conditional statements placed inside other conditional statements. This allows for more intricate decision-making structures, enabling developers to test multiple conditions in a hierarchical manner.

Basics of Nested Conditions

A nested condition arises when an **if**, **elif**, or **else** statement contains another **if** or **elif** statement within its block of code.

Syntax:

```
if outer_condition: # Outer condition code
    if inner_condition: # Inner condition code
```

Using Nested if Statements

Nested **if** statements allow for checking a secondary condition only if the primary condition is met.

Example:



```
hello.py
x = 10
y = 20

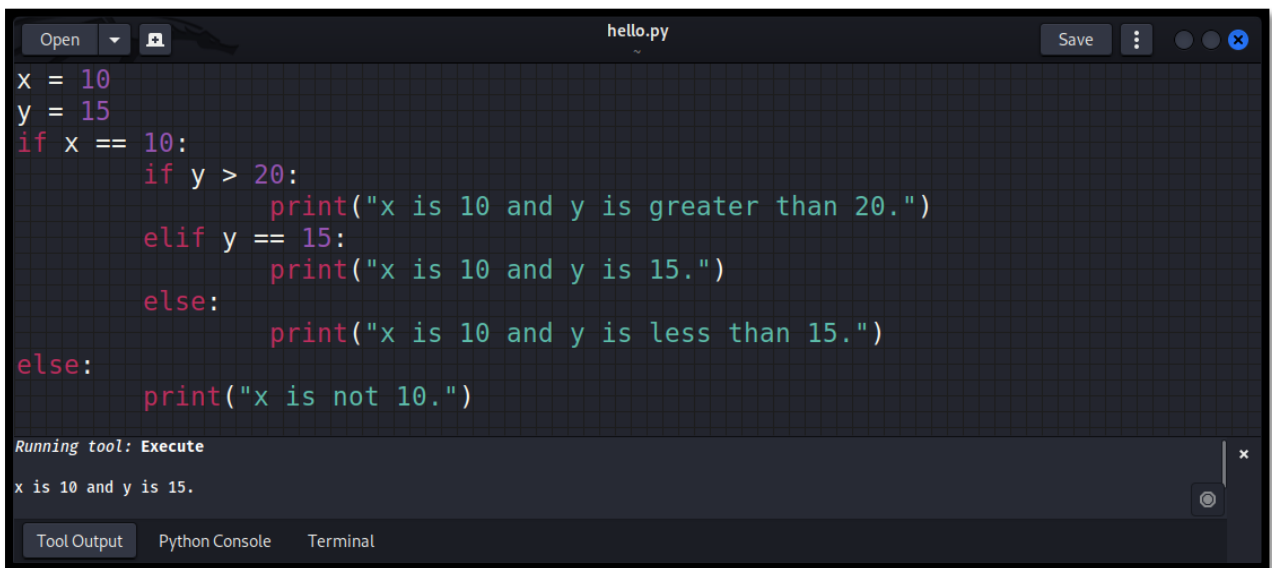
if x == 10:
    if y > 15:
        print("x is 10 and y is greater than 15.")

Running tool: Execute
x is 10 and y is greater than 15.
```

The screenshot shows a code editor window titled 'hello.py' with the following Python code: `x = 10`, `y = 20`, `if x == 10:`, `if y > 15:`, and `print("x is 10 and y is greater than 15.")`. Below the code, a terminal window shows the output: `x is 10 and y is greater than 15.`

Combining Nested Conditions with elif and else

You can combine nested conditions with **elif** and **else** statements for more complex decision-making structures.

Example:

```
hello.py
x = 10
y = 15
if x == 10:
    if y > 20:
        print("x is 10 and y is greater than 20.")
    elif y == 15:
        print("x is 10 and y is 15.")
    else:
        print("x is 10 and y is less than 15.")
else:
    print("x is not 10.")
```

Running tool: Execute

x is 10 and y is 15.


Tool Output Python Console Terminal

Practical Applications of Nested Conditions

Nested conditions are particularly useful in scenarios where decisions depend on a series of criteria being met.

Example:

Imagine an application process where an applicant must be over 18 and must also pass a test to qualify:



```
hello.py
age = 20
test_score = 85
if age > 18:
    if test_score >= 80:
        print("Application accepted.")
    else:
        print("Application rejected due to low test score.")
else:
    print("Application rejected due to age.")
```

Running tool: Execute

Application accepted.

Tool Output Python Console Terminal

Input and Output

Reading input with input()

In Python, the **input()** function is a built-in function used to read input from the user. Whether you're building interactive scripts or simple command-line tools, understanding how to effectively use **input()** is crucial.

Basics of the input() Function

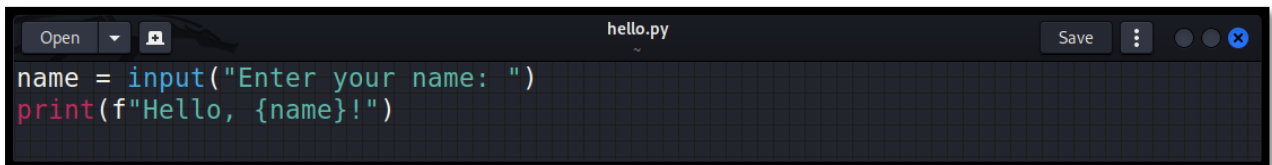
The **input()** function reads a line from the input (usually from the user's keyboard) and returns it as a string (excluding the trailing newline).

Syntax: `input([prompt])`

Where **prompt** is an optional parameter that specifies a message to display before reading the input.

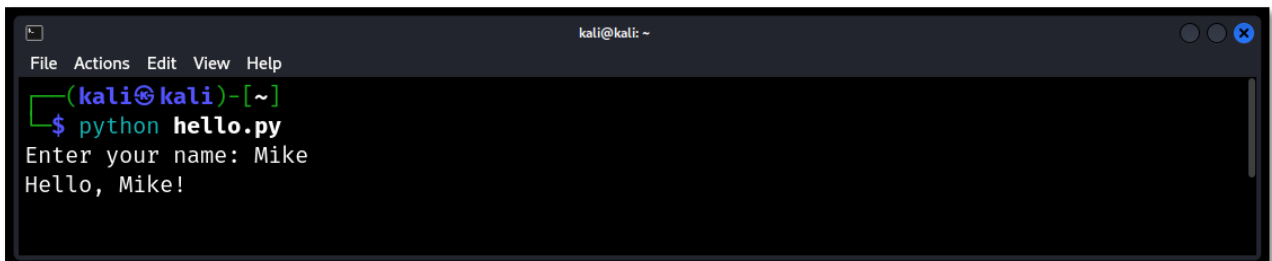
Simple input() Example

Example:



```
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

In this example, the program prompts the user to enter their name and then prints a greeting using the provided name.



```
(kali@kali)~$ python hello.py
Enter your name: Mike
Hello, Mike!
```

Type Conversion with input()

Since the **input()** function always returns a string, it's often necessary to convert this string to other data types, such as integers or floats, especially when dealing with numerical input.

Example:



```
age_str = input("Enter your age: ")
age = int(age_str)

print(f"Next year, you will be {age + 1} years old.")
```

In the example above, the user's age is read as a string and then converted to an integer using the `int()` function.



```
kali@kali: ~  
File Actions Edit View Help  
(kali@kali)~[~]  
$ python hello.py  
Enter your age: 22  
Next year, you will be 23 years old.
```

Handling Invalid Input

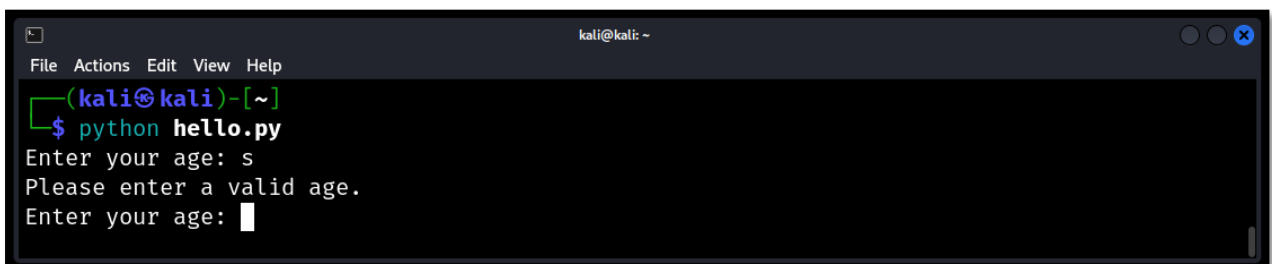
When reading input from users, there's always a chance they might provide invalid data. It's essential to handle such scenarios gracefully.

Example:



```
hello.py  
Save  
while True:  
    try:  
        age_str = input("Enter your age: ")  
        age = int(age_str)  
        print(f"Next year, you will be {age + 1} years old.")  
        break  
    except ValueError:  
        print("Please enter a valid age.")
```

In this example, the program keeps prompting the user for their age until they provide a valid integer. If the conversion to an integer fails, a **ValueError** exception is raised, and the user is informed of the error.

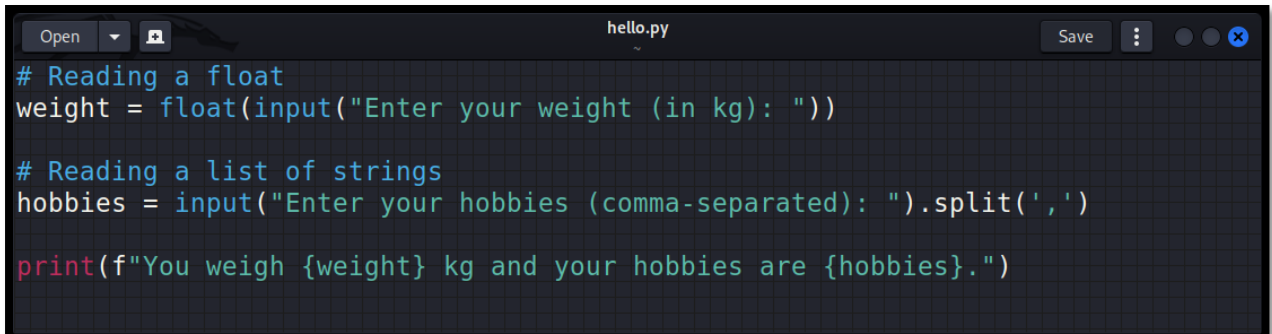


```
kali@kali: ~  
File Actions Edit View Help  
(kali@kali)~[~]  
$ python hello.py  
Enter your age: s  
Please enter a valid age.  
Enter your age: █
```

Using input() with Different Data Types

The `input()` function can be combined with various data type conversions to read different kinds of input.

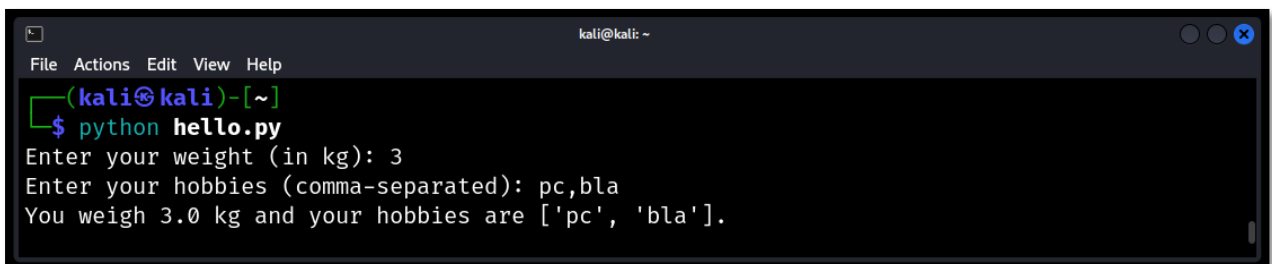
Example:



```
hello.py
# Reading a float
weight = float(input("Enter your weight (in kg): "))

# Reading a list of strings
hobbies = input("Enter your hobbies (comma-separated): ").split(',')

print(f"You weigh {weight} kg and your hobbies are {hobbies}.")
```



```
kali@kali: ~
File Actions Edit View Help
(kali@kali)~[~]
$ python hello.py
Enter your weight (in kg): 3
Enter your hobbies (comma-separated): pc,bla
You weigh 3.0 kg and your hobbies are ['pc', 'bla'].
```

Security Considerations with input()

In older versions of Python (2.x), there was a function called `raw_input()` which behaved like Python 3's `input()`. However, Python 2 also had a function called `input()` which evaluated the user's input as Python code. This could lead to potential security issues. In Python 3, this behavior was removed, and the `input()` function always returns user input as a string, making it safer.

Writing output with print()

The `print()` function is one of the most frequently used functions in Python. It provides a way to display information to the user, making it essential for debugging, data visualization, and user interaction.

Basics of the print() Function

The primary purpose of the `print()` function is to display text to the console. By default, it outputs text followed by a newline.

Syntax:

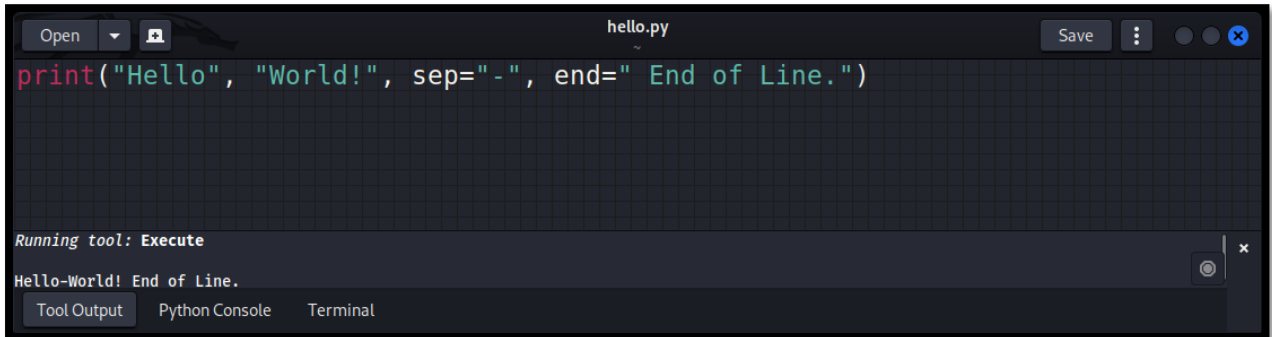
```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- **objects:** The values to be printed.
- **sep:** Separator between values (default is a space).
- **end:** Specifies what to print at the end (default is a newline).
- **file:** The file where the values are printed (default is `sys.stdout` which means the console).
- **flush:** Whether to flush the output buffer (default is `False`).

Using Separators and End Arguments

The `sep` and `end` arguments allow for customization of the output format.

Example:



```
hello.py
print("Hello", "World!", sep="-", end=" End of Line.")

Running tool: Execute
Hello-World! End of Line.
```

Printing Multiple Objects

You can print multiple objects of different data types in a single `print()` call.

Example:



```
hello.py
name = "Alice"
age = 30

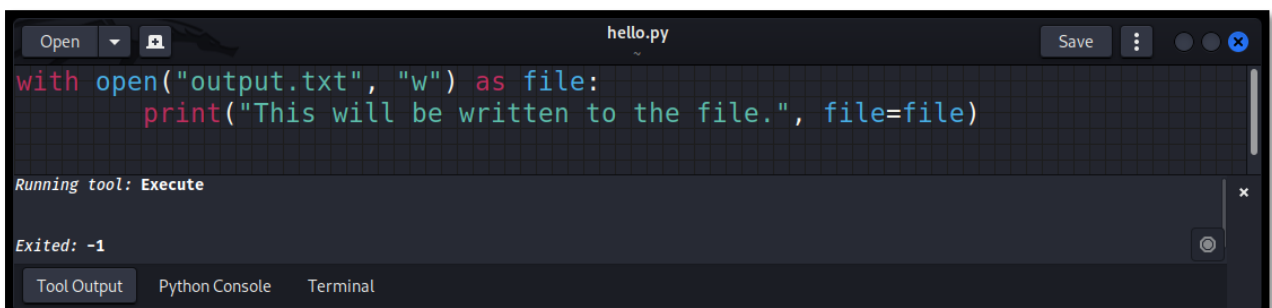
print("Name:", name, "| Age:", age)

Running tool: Execute
Name: Alice | Age: 30
```

Redirecting Output to a File

Using the `file` argument, you can redirect the output of the `print()` function to a file.

Example:



```
hello.py
with open("output.txt", "w") as file:
    print("This will be written to the file.", file=file)

Running tool: Execute
Exited: -1
```

In this example, the text is written to `output.txt` instead of being displayed on the console.

Reading and Writing Files

File handling is a fundamental aspect of many applications, allowing for data persistence, logging, configuration, and more. Python provides built-in functions and methods to read from and write to files with ease.

Opening Files

The **open()** function is used to open a file and returns a file object. It takes two main arguments: the filename and the mode.

Syntax:

```
open(filename, mode)
```

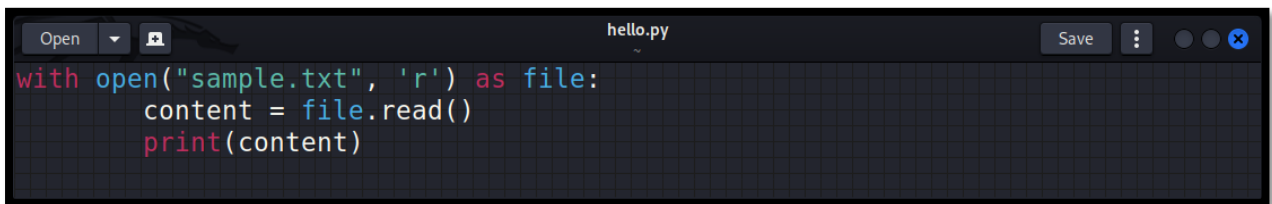
Common modes include:

- **'r'**: Read (default)
- **'w'**: Write (overwrites the file)
- **'a'**: Append (adds to the end of the file)
- **'b'**: Binary mode

Reading from Files

Once a file is opened in read mode, you can read its contents using various methods.

Example:

A screenshot of a code editor window titled 'hello.py'. The window has a dark background and a light-colored border. The code inside is:

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

The code is color-coded: 'with' is blue, 'open' is red, 'sample.txt' is green, 'r' is red, 'as' is blue, 'file' is green, 'content' is green, '=' is blue, 'file.read()' is green, and 'print' is red.

The **with** statement ensures that the file is properly closed after its suite finishes.

Writing to Files

To write to a file, you open it in write or append mode and use the **write()** method.

Example:

A screenshot of a code editor window titled 'hello.py'. The window has a dark background and a light-colored border. The code inside is:

```
with open("output.txt", 'w') as file:  
    file.write("Hello, World!")
```

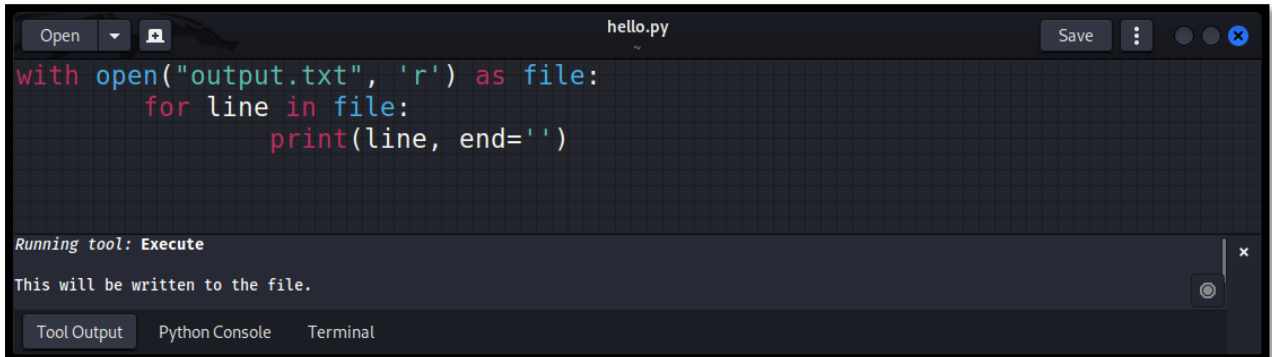
The code is color-coded: 'with' is blue, 'open' is red, 'output.txt' is green, 'w' is red, 'as' is blue, 'file' is green, '=' is blue, 'file.write' is green, and '"Hello, World!'" is green.

This will create (or overwrite) **output.txt** and write "Hello, World!" to it.

Reading Lines

Files can be read line-by-line using the `readline()` method or by iterating over the file object.

Example:



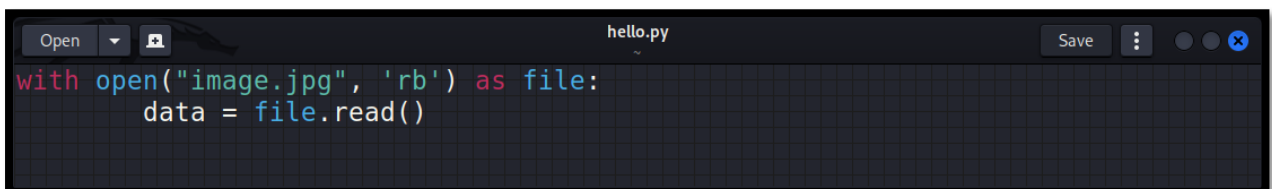
```
hello.py
with open("output.txt", 'r') as file:
    for line in file:
        print(line, end='')

Running tool: Execute
This will be written to the file.
```

Working with Binary Files

To read or write binary data, use the 'b' mode with 'r' or 'w'.

Example:

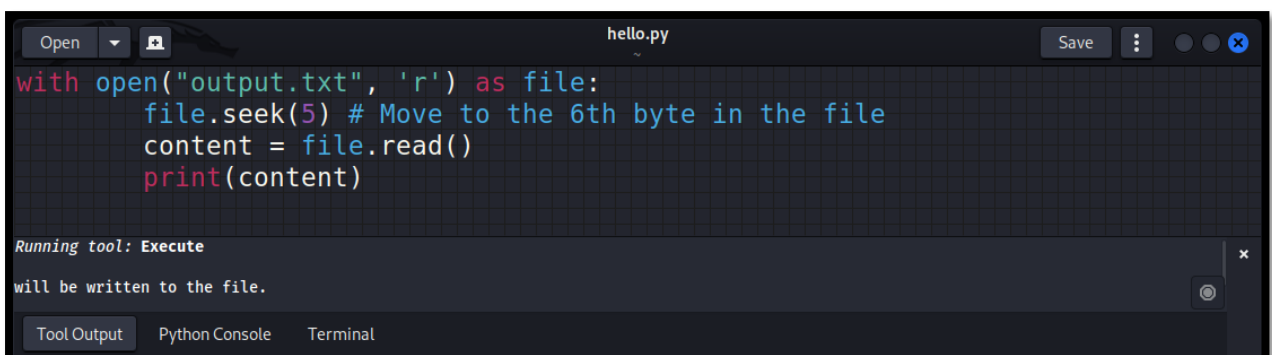


```
hello.py
with open("image.jpg", 'rb') as file:
    data = file.read()
```

File Positions

The `tell()` method returns the current file position, and the `seek(offset)` method changes the file position.

Example:



```
hello.py
with open("output.txt", 'r') as file:
    file.seek(5) # Move to the 6th byte in the file
    content = file.read()
    print(content)

Running tool: Execute
will be written to the file.
```

Handling File Exceptions

It's crucial to handle potential file exceptions, such as **FileNotFoundError**.

Example:



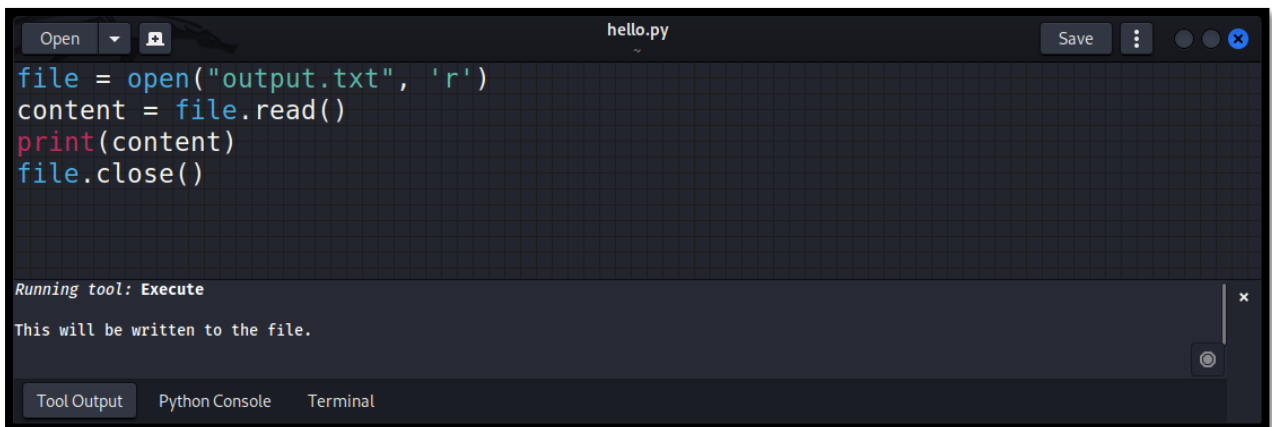
```
Open | hello.py | Save | [Icons]
try:
    with open("nonexistent.txt", 'r') as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("The file was not found.")

Running tool: Execute
The file was not found.
Tool Output | Python Console | Terminal
```

Closing Files

While the **with** statement automatically closes files, if you open a file without it, you should close it using the **close()** method.

Example:



```
Open | hello.py | Save | [Icons]
file = open("output.txt", 'r')
content = file.read()
print(content)
file.close()

Running tool: Execute
This will be written to the file.
Tool Output | Python Console | Terminal
```

Working with File Modes

In Python, when working with files, the mode in which you open a file is crucial. It determines whether you can read from, write to, or both read and write to the file. It also affects how the file pointer behaves and how the file's content is treated (e.g., text vs. binary).

Overview of File Modes

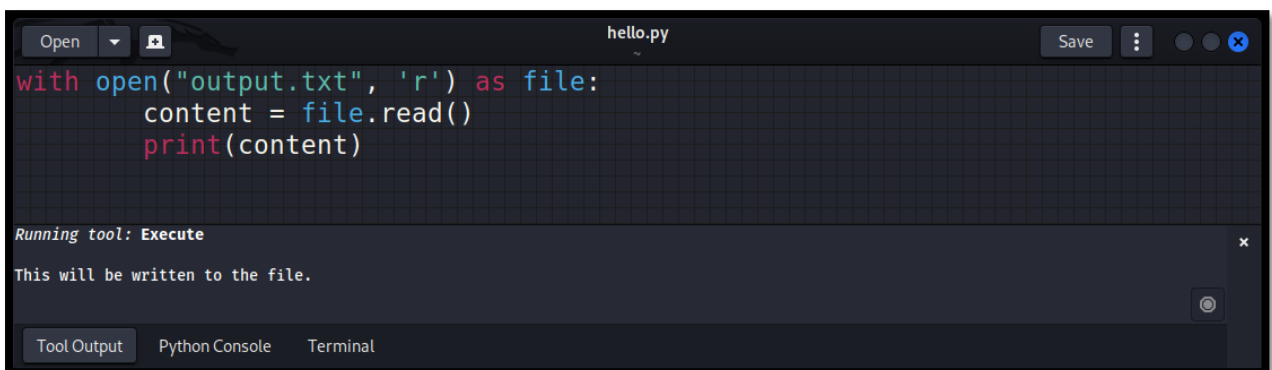
The `open()` function in Python accepts a **mode** parameter, which determines the mode in which the file is opened. Here are the primary file modes:

- `'r'`: Read mode (default)
- `'w'`: Write mode
- `'a'`: Append mode
- `'x'`: Exclusive creation mode
- `'b'`: Binary mode
- `'t'`: Text mode (default)

Read Mode ('r')

Opens the file for reading (default). If the file does not exist, it raises a **FileNotFoundError**.

Example:



```
Open | hello.py | Save | [Icons]
with open("output.txt", 'r') as file:
    content = file.read()
    print(content)

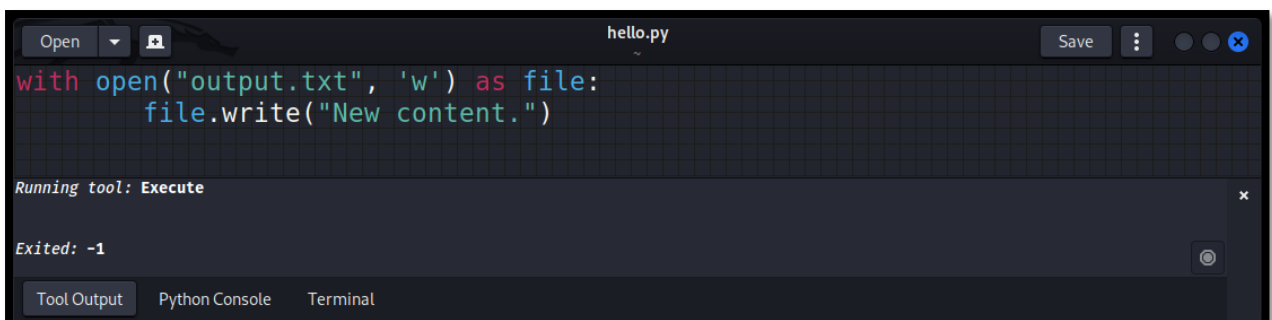
Running tool: Execute
This will be written to the file.

Tool Output | Python Console | Terminal
```

Write Mode ('w')

Opens the file for writing. If the file already exists, it truncates (deletes) its content. If the file does not exist, it creates a new file.

Example:



```
Open | hello.py | Save | [Icons]
with open("output.txt", 'w') as file:
    file.write("New content.")

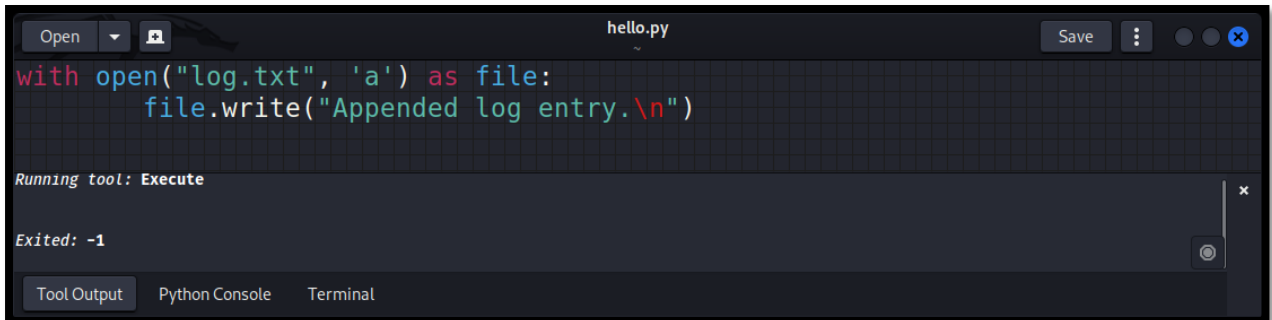
Running tool: Execute
Exited: -1

Tool Output | Python Console | Terminal
```


Append Mode ('a')

Opens the file for writing, but it appends to the end of the file instead of truncating it. If the file does not exist, it creates a new file.

Example:



```
hello.py
with open("log.txt", 'a') as file:
    file.write("Appended log entry.\n")


Running tool: Execute

Exited: -1
Tool Output Python Console Terminal
```

Exclusive Creation Mode ('x')

Opens the file for exclusive creation. If the file already exists, the operation will fail with a **FileExistsError**. This mode is useful when you want to ensure that you're not overwriting an existing file.

Example:



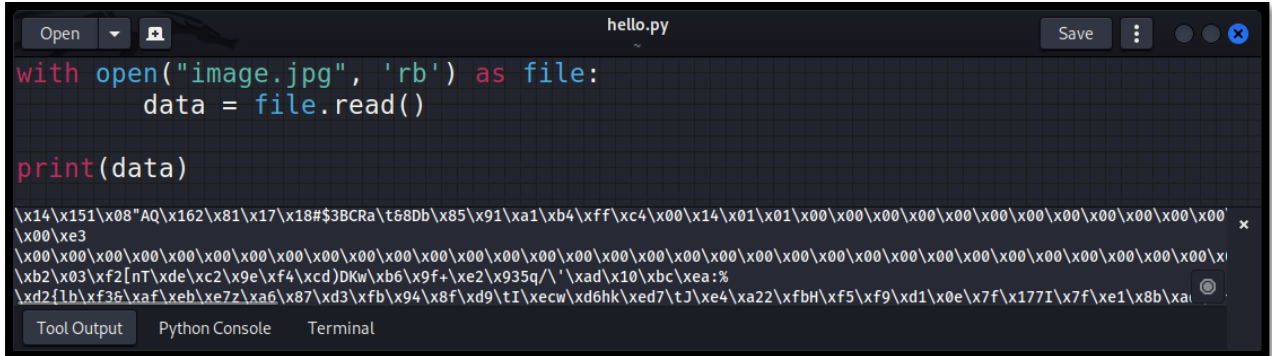
```
*hello.py
try:
    with open("unique.txt", 'x') as file:
        file.write("Unique content.")
except FileExistsError:
    print("File already exists.")

Running tool: Execute

Exited: -1
Tool Output Python Console Terminal
```

Binary Mode ('b')

Opens the file in binary mode, rather than text mode. This mode is used for non-text files like images, audio files, etc.

Example:


The screenshot shows a Python IDE window titled 'hello.py'. The code in the editor is:

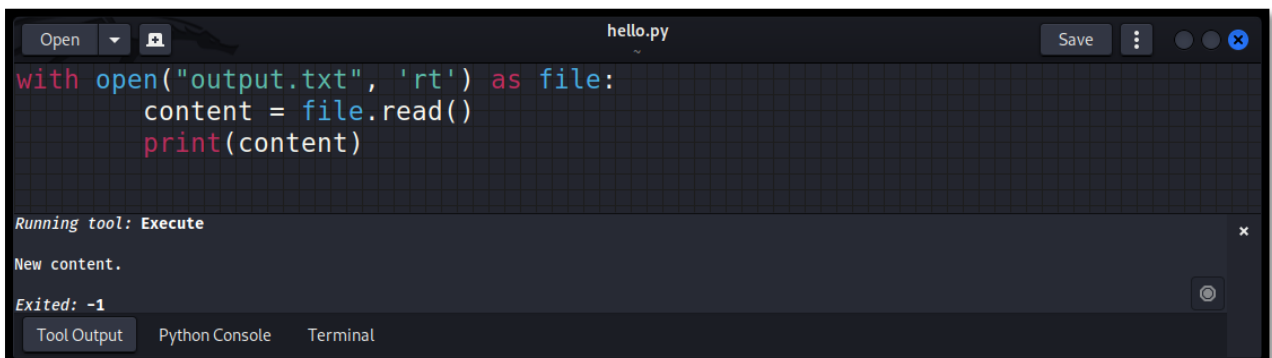
```
with open("image.jpg", 'rb') as file:
    data = file.read()

print(data)
```

The output of the program is displayed in the Python Console, showing a long string of hexadecimal characters representing the binary data of the image file.

Text Mode ('t')

Opens the file in text mode (default). In this mode, Python handles encoding and decoding of data and translates line endings (`\n`).

Example:


The screenshot shows a Python IDE window titled 'hello.py'. The code in the editor is:

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

The output of the program is displayed in the Python Console, showing the text content of the file: "New content."

Combining Multiple Modes

You can combine multiple modes by stringing them together. For instance, `'rb'` means to open the file in both read and binary modes.

Example:


The screenshot shows a Python IDE window titled 'hello.py'. The code in the editor is:

```
with open("data.bin", 'wb') as file:
    file.write(b'Binary data.')
```

Error Handling


Common Python Errors

Errors are inevitable in the programming world. Whether you're a beginner or an experienced developer, you'll encounter errors in your code. Understanding common Python errors and their causes can help you debug your code more efficiently.

SyntaxError

A **SyntaxError** is raised when the Python parser encounters incorrect syntax.

Example:



```
hello.py
print("Hello, World!")

Running tool: Execute
File "/home/kali/hello.py", line 1
print("Hello, World!")
      ^
SyntaxError: unterminated string literal (detected at line 1)
```

Explanation: The string is missing a closing quotation mark, leading to a syntax error.

NameError

A **NameError** is raised when a variable or function name is used but not defined.

Example:



```
hello.py
print(variable_does_not_exist)

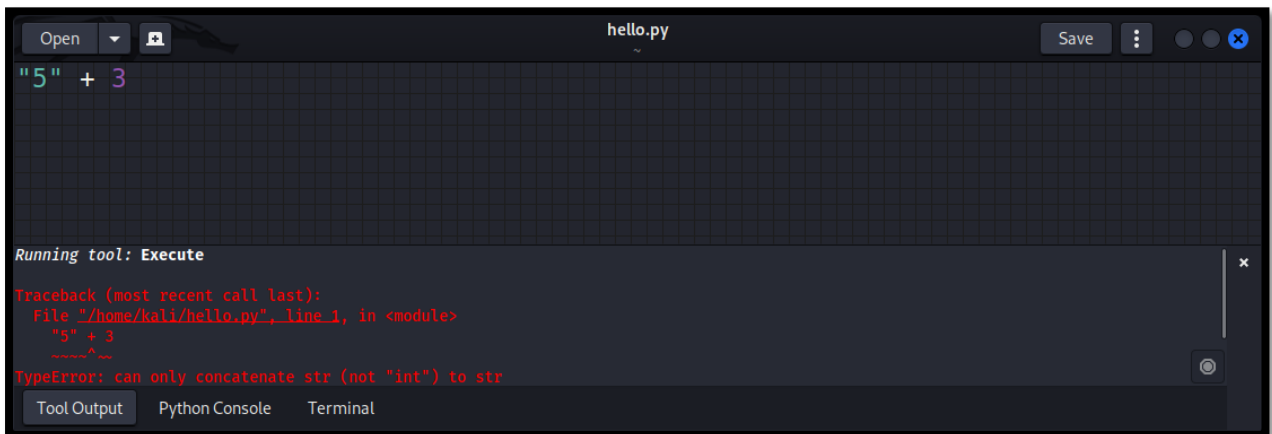
Running tool: Execute
Traceback (most recent call last):
  File "/home/kali/hello.py", line 1, in <module>
    print(variable_does_not_exist)
          ^^^^^^^^^^^^^^^^^^^^^^^
NameError: name 'variable_does_not_exist' is not defined
```

Explanation: The variable `variable_does_not_exist` has not been defined before its use.

TypeError

A **TypeError** is raised when an operation or function is applied to an object of an inappropriate type.

Example:



The screenshot shows a code editor window titled 'hello.py' with a 'Save' button and window controls. The code in the editor is `"5" + 3`. Below the code, a terminal window displays the following error message:

```
Running tool: Execute
Traceback (most recent call last):
  File "/home/kali/hello.py", line 1, in <module>
    "5" + 3
    ~~~~^
TypeError: can only concatenate str (not "int") to str
```

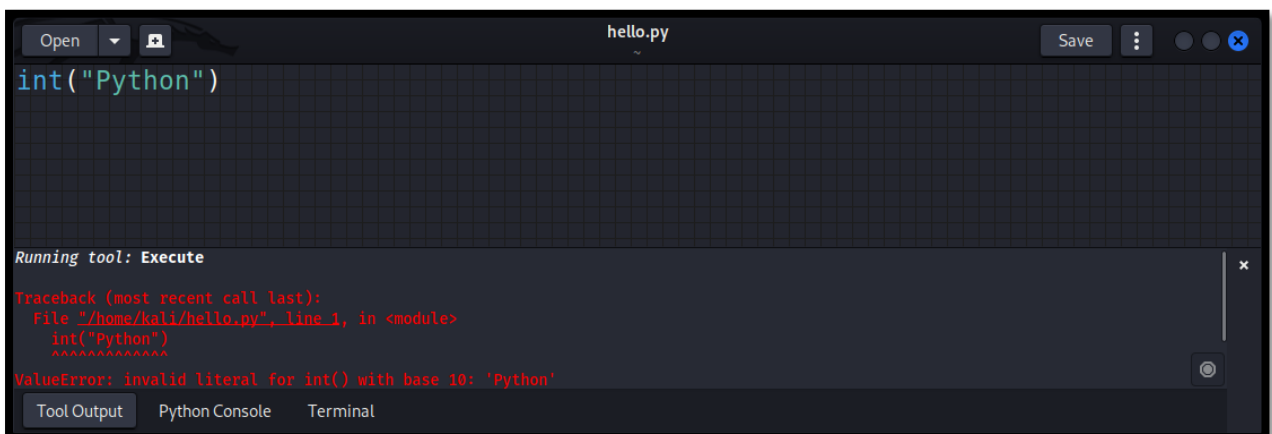
At the bottom of the terminal, there are three tabs: 'Tool Output', 'Python Console', and 'Terminal'.

Explanation: You cannot add a string and an integer directly.

ValueError

A **ValueError** is raised when a function receives an argument of the correct type but an inappropriate value.

Example:



The screenshot shows a code editor window titled 'hello.py' with a 'Save' button and window controls. The code in the editor is `int("Python")`. Below the code, a terminal window displays the following error message:

```
Running tool: Execute
Traceback (most recent call last):
  File "/home/kali/hello.py", line 1, in <module>
    int("Python")
    ~~~~~
ValueError: invalid literal for int() with base 10: 'Python'
```

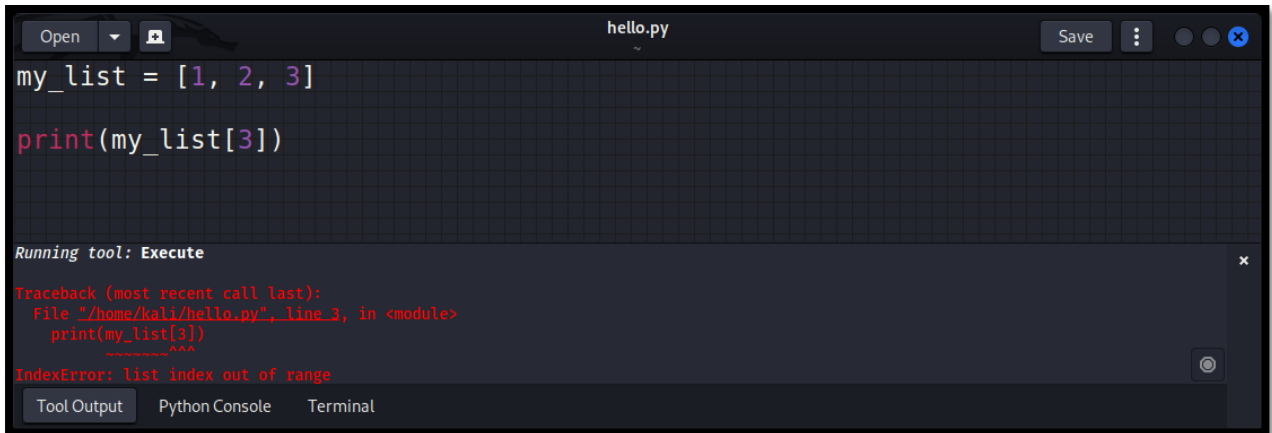
At the bottom of the terminal, there are three tabs: 'Tool Output', 'Python Console', and 'Terminal'.

Explanation: The `int()` function expects a string that can be converted to an integer. The string "Python" cannot be converted to an integer.

IndexError

An **IndexError** is raised when you try to access an index that does not exist in a sequence, like a list or tuple.

Example:



```
hello.py
Open Save
my_list = [1, 2, 3]
print(my_list[3])

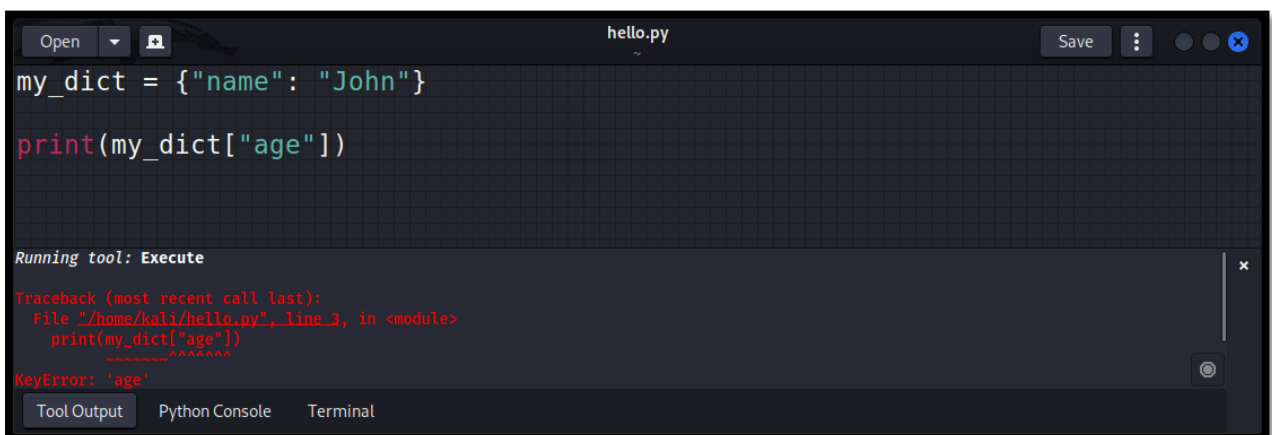
Running tool: Execute
Traceback (most recent call last):
  File "/home/kali/hello.py", line 3, in <module>
    print(my_list[3])
          ~~~~~^^^
IndexError: list index out of range
Tool Output Python Console Terminal
```

Explanation: The list has indices 0, 1, and 2. Index 3 does not exist.

KeyError

A **KeyError** is raised when you try to access a dictionary key that does not exist.

Example:



```
hello.py
Open Save
my_dict = {"name": "John"}
print(my_dict["age"])

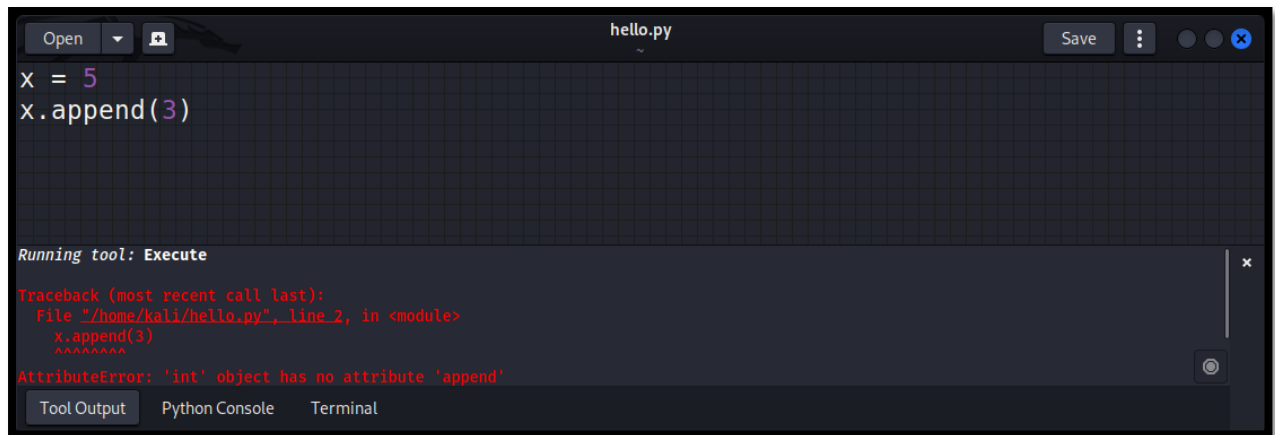
Running tool: Execute
Traceback (most recent call last):
  File "/home/kali/hello.py", line 3, in <module>
    print(my_dict["age"])
          ~~~~~^^^^^^
KeyError: 'age'
Tool Output Python Console Terminal
```

Explanation: The key "age" does not exist in the dictionary.

AttributeError

An **AttributeError** is raised when you try to access an attribute or method that does not exist for a particular object.

Example:



```
Open hello.py Save
```

```
x = 5
x.append(3)
```

Running tool: Execute

```
Traceback (most recent call last):
  File "/home/kali/hello.py", line 2, in <module>
    x.append(3)
    ~~~~~^
AttributeError: 'int' object has no attribute 'append'
```

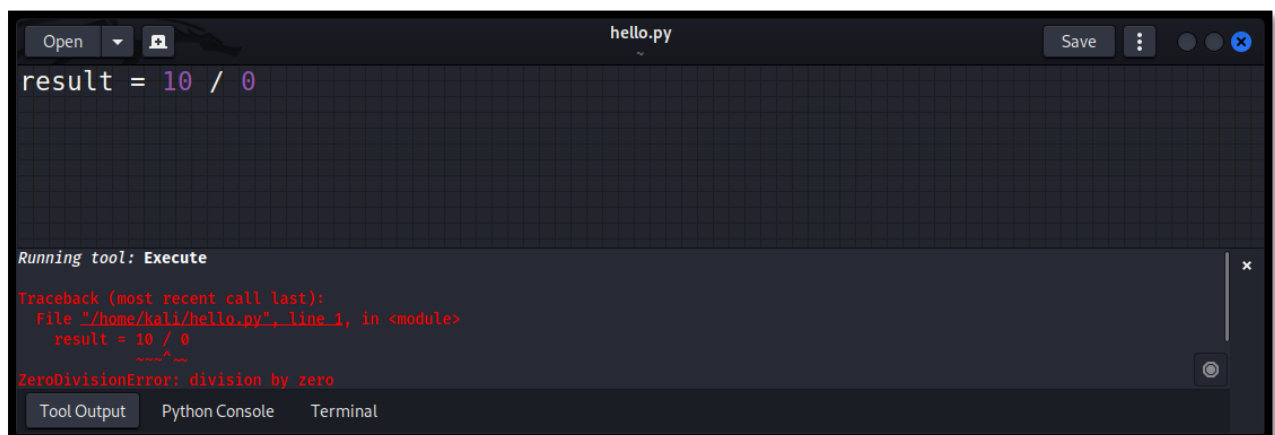
Tool Output Python Console Terminal

Explanation: Integers do not have an **append** method.

ZeroDivisionError

A **ZeroDivisionError** is raised when you try to divide a number by zero.

Example:



```
Open hello.py Save
```

```
result = 10 / 0
```

Running tool: Execute

```
Traceback (most recent call last):
  File "/home/kali/hello.py", line 1, in <module>
    result = 10 / 0
    ~~~~~^
ZeroDivisionError: division by zero
```

Tool Output Python Console Terminal

Explanation: Division by zero is mathematically undefined.

ImportError

An **ImportError** is raised when you try to import a module or function that does not exist.

Example:



```
Open hello.py Save
import non_existent_module

Running tool: Execute
Traceback (most recent call last):
  File "/home/kali/hello.py", line 1, in <module>
    import non_existent_module
ModuleNotFoundError: No module named 'non_existent_module'
```

Explanation: The module `non_existent_module` does not exist.

The try, except Blocks

In programming, exceptions are events that can disrupt the normal flow of a program. Python provides a mechanism to handle exceptions gracefully using the **try** and **except** blocks. This allows developers to respond to errors in a controlled manner, ensuring that the program can continue running or terminate gracefully.

The try Block

The **try** block is used to enclose a section of code that might raise an exception. It allows developers to "try" to execute a block of code and catch any exceptions that arise.

Syntax:

```
try: # Code that might raise an exception
```

The except Block

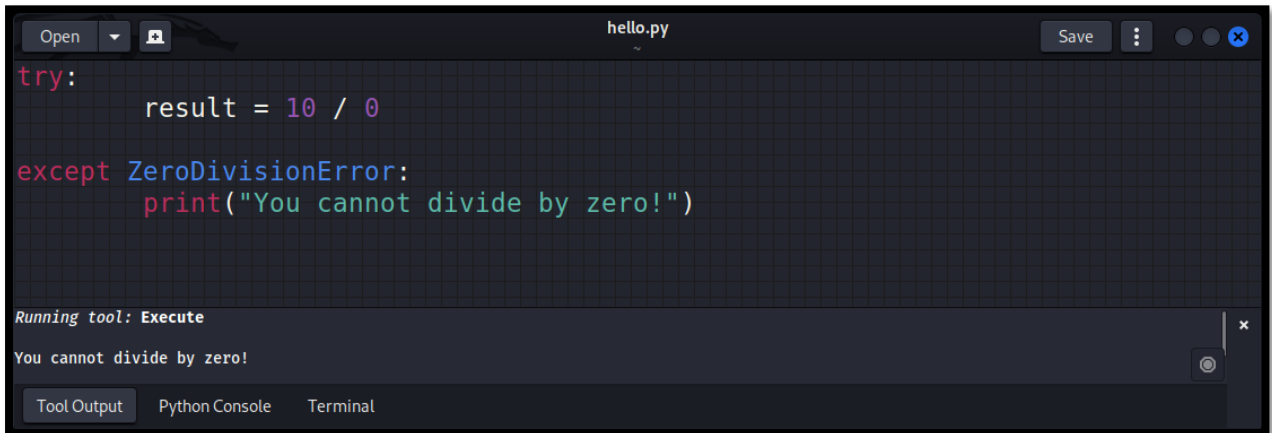
The **except** block follows the **try** block and is used to catch and handle exceptions. If an exception occurs in the **try** block, the code in the **except** block is executed.

Syntax:

```
try: # Code that might raise an exception
except ExceptionType: # Code to handle the exception
```

Basic try, except Example

Example:



```
Open hello.py Save
```

```
try:
    result = 10 / 0

except ZeroDivisionError:
    print("You cannot divide by zero!")
```

Running tool: Execute

You cannot divide by zero!

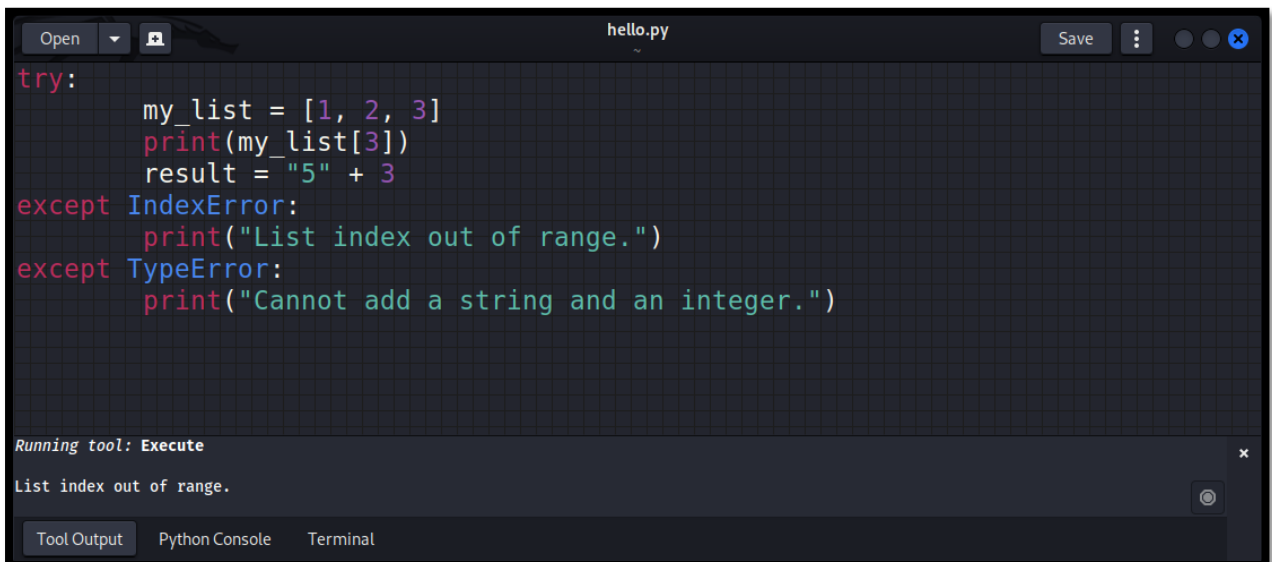
Tool Output Python Console Terminal

In the example above, a **ZeroDivisionError** is raised in the **try** block, and the corresponding **except** block handles it by printing an error message.

Handling Multiple Exceptions

You can have multiple **except** blocks to handle different types of exceptions.

Example: This is just an example, the list and the string can be replaced by any other objects



```
Open hello.py Save
```

```
try:
    my_list = [1, 2, 3]
    print(my_list[3])
    result = "5" + 3

except IndexError:
    print("List index out of range.")

except TypeError:
    print("Cannot add a string and an integer.")
```

Running tool: Execute

List index out of range.

Tool Output Python Console Terminal

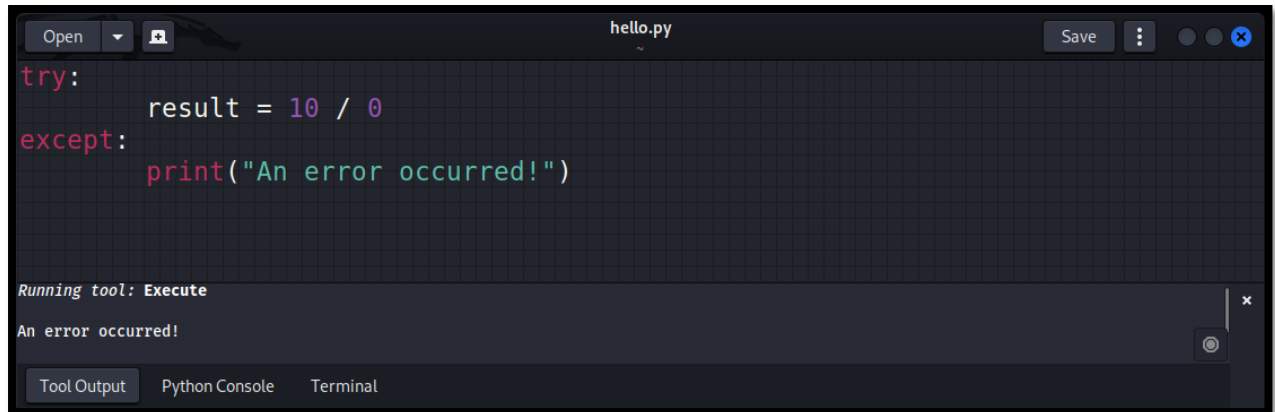
Output:

List index out of range.

Catching All Exceptions

If you want to catch all exceptions, regardless of their type, you can use a general **except** block without specifying an exception type.

Example:



```
Open | hello.py | Save | [Icons]
try:
    result = 10 / 0
except:
    print("An error occurred!")

Running tool: Execute
An error occurred!
Tool Output | Python Console | Terminal
```

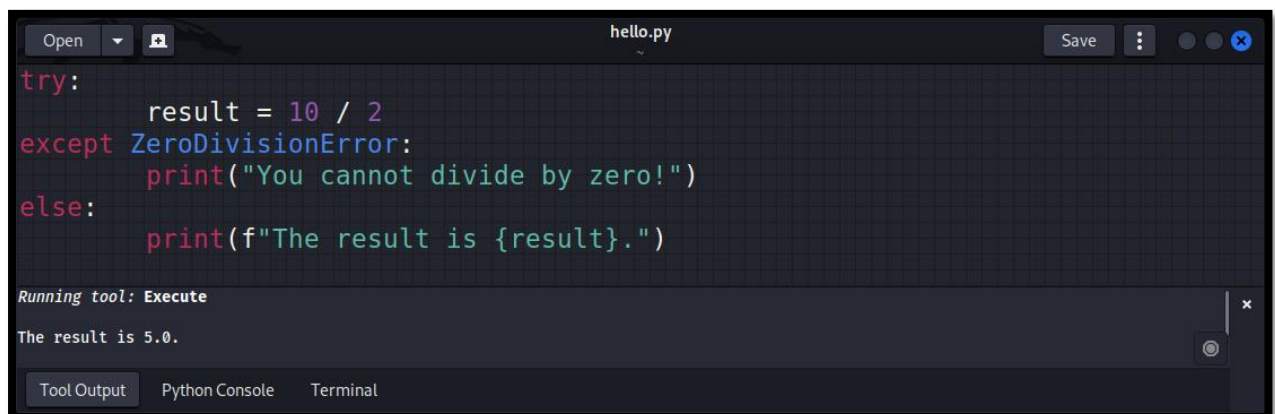
Output: An error occurred!

However, it's generally a good practice to specify the exact exceptions you're expecting to handle to avoid masking unexpected issues.

The else Block in Exception Handling

The **else** block can be used in conjunction with **try** and **except**. It is executed if no exceptions occur in the **try** block.

Example:



```
Open | hello.py | Save | [Icons]
try:
    result = 10 / 2
except ZeroDivisionError:
    print("You cannot divide by zero!")
else:
    print(f"The result is {result}.")

Running tool: Execute
The result is 5.0.
Tool Output | Python Console | Terminal
```

Output: The result is 5.0.

The finally Block

In the realm of exception handling, the **finally** block holds a special place. It is designed to house code that must be executed regardless of whether an exception was raised or not.

Purpose of the finally Block

The primary purpose of the **finally** block is to ensure that specific code runs no matter what — whether an exception occurs or not. This is particularly useful for cleanup actions, such as closing files or releasing resources.

Basic Structure of the finally Block

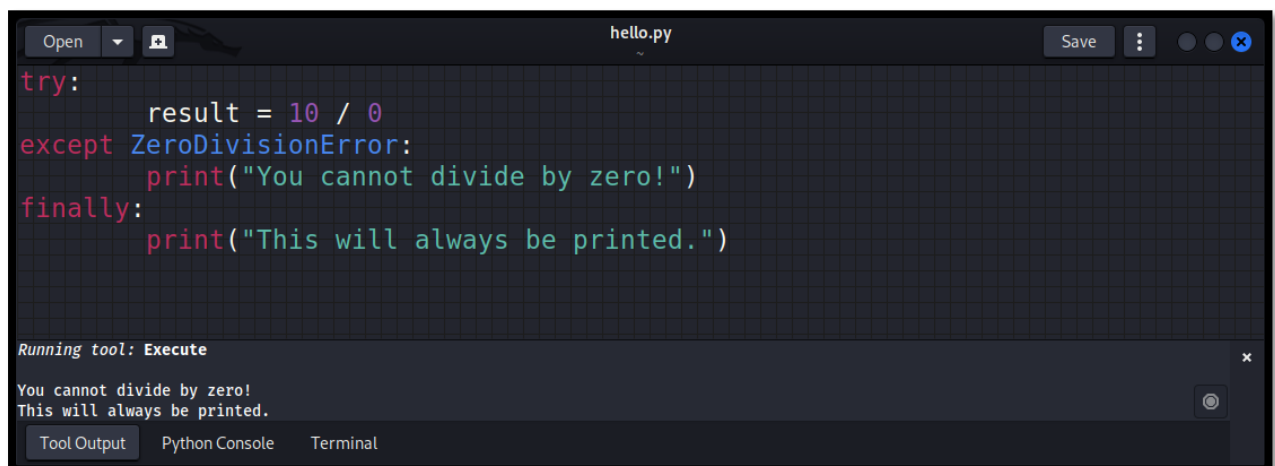
The **finally** block follows the **try** and **except** blocks and is always executed.

Syntax:

```
try: # Code that might raise an exception
except ExceptionType: # Code to handle the exception
finally: # Code to be executed regardless of exceptions
```

Basic finally Block Example

Example:



The screenshot shows a code editor window titled 'hello.py' with the following Python code:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You cannot divide by zero!")
finally:
    print("This will always be printed.")
```

Below the code editor, a 'Running tool: Execute' panel shows the output of the program:

```
You cannot divide by zero!
This will always be printed.
```

The IDE interface includes buttons for 'Tool Output', 'Python Console', and 'Terminal' at the bottom.

Output:

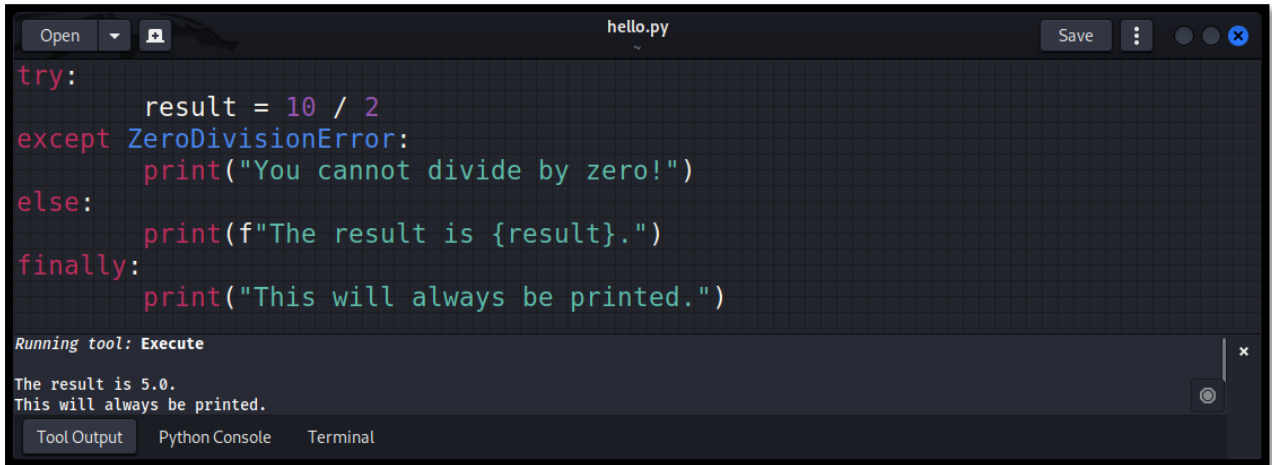
```
You cannot divide by zero!
This will always be printed.
```

In the example above, even though a **ZeroDivisionError** is raised, the **finally** block ensures that its code is executed.

Using finally with else

The **finally** block can also be used in conjunction with the **else** block.

Example:



```
hello.py
Open Save
try:
    result = 10 / 2
except ZeroDivisionError:
    print("You cannot divide by zero!")
else:
    print(f"The result is {result}.")
finally:
    print("This will always be printed.")

Running tool: Execute
The result is 5.0.
This will always be printed.
Tool Output Python Console Terminal
```


Output:

The result is 5.0.
This will always be printed.

Practical Applications of the finally Block

1. **Resource Cleanup:** Ensuring that resources like files or network connections are closed properly.

Example:



```
hello.py
Open Save
file = None
try:
    file = open("data.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found.")
finally:
    if file: file.close()
    print("File closed.")

Running tool: Execute
File not found.
File closed.
Tool Output Python Console Terminal
```

2. **Resetting States:** If a piece of code changes a global state or configuration, the **finally** block can be used to reset or revert those changes.

3. **Logging:** Logging information about code execution, errors, or other events for debugging or audit purposes.

Raising Exceptions

While handling exceptions is crucial, there are times when developers need to raise exceptions intentionally, signaling that an error condition has occurred.

Why Raise Exceptions?

Raising exceptions intentionally can be beneficial for several reasons:

1. **Error Reporting:** To notify other parts of the program that something unexpected has occurred.
2. **Input Validation:** To ensure that functions and methods are used correctly.
3. **Fail Fast:** To stop the program immediately when an unrecoverable error is detected, making it easier to debug.

The raise Statement

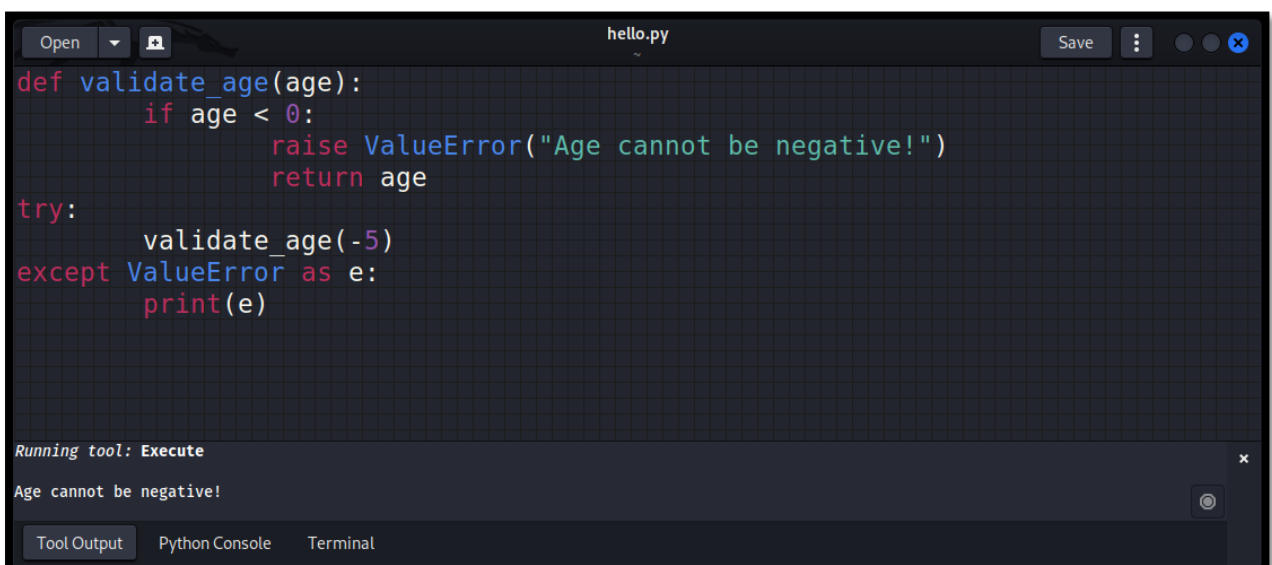
The **raise** statement is used to raise an exception in Python. You can raise a specific exception and provide a custom error message.

Syntax:

```
raise ExceptionType("Error Message")
```

Basic Example of Raising an Exception

Example:



```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    return age

try:
    validate_age(-5)
except ValueError as e:
    print(e)
```

Running tool: Execute

Age cannot be negative!

Tool Output Python Console Terminal

In the example above, the **validate_age** function raises a **ValueError** if the provided age is negative.

Re-raising Exceptions

In some scenarios, you might want to catch an exception, perform some operations, and then re-raise the same exception. You can do this using the **raise** statement without an argument inside an **except** block.

Example:

```
hello.py
Open Save
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Logging the error...")
    raise

Running tool: Execute
Logging the error...
Traceback (most recent call last):
  File "/home/kali/hello.py", line 2, in <module>
    result = 10 / 0
            ~~~^~
ZeroDivisionError: division by zero
Tool Output Python Console Terminal
```

Output:

```
Logging the error..
ZeroDivisionError: division by zero
```

Modules

What is a Module?

In the vast world of programming, organization and code reusability are paramount. Python addresses this by introducing the concept of modules.

Definition of a Module

A module in Python is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` added. Modules allow for logical organization of code, promoting code reusability and maintainability.

Why Use Modules?

Modules offer several benefits:

1. **Code Reusability:** Write once, use everywhere. Functions, classes, or variables defined in a module can be reused across multiple programs.
2. **Logical Structuring:** Modules help in organizing related code into separate files, making the codebase more manageable.
3. **Namespace Avoidance:** Modules help avoid naming conflicts by providing a separate namespace for identifiers.

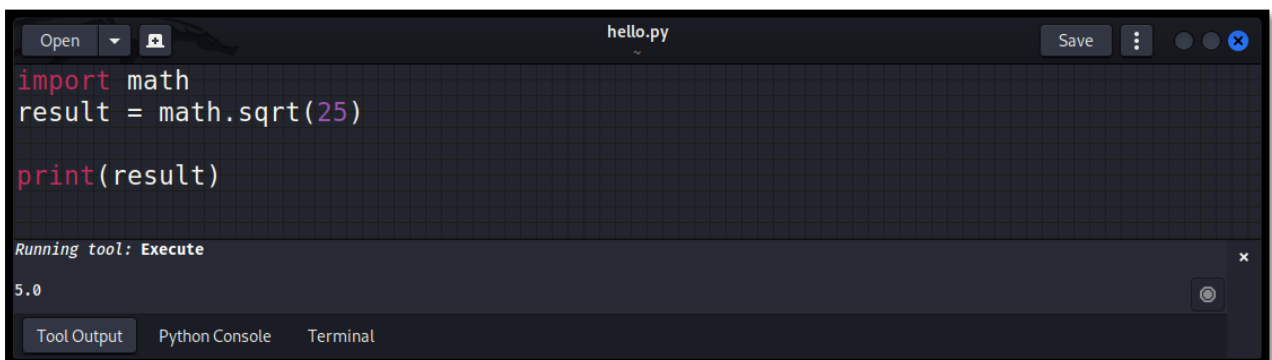
Importing Modules

Modules enhance the functionality of Python by allowing developers to use pre-defined functions, classes, and variables. However, to leverage these functionalities, one must first understand how to import them into their programs.

Basic Module Import

The simplest way to import a module is using the **import** statement followed by the module name.

Example:



```
Open  hello.py  Save  [Menu]  [Close]
```

```
import math
result = math.sqrt(25)

print(result)
```

Running tool: Execute

```
5.0
```

Tool Output Python Console Terminal

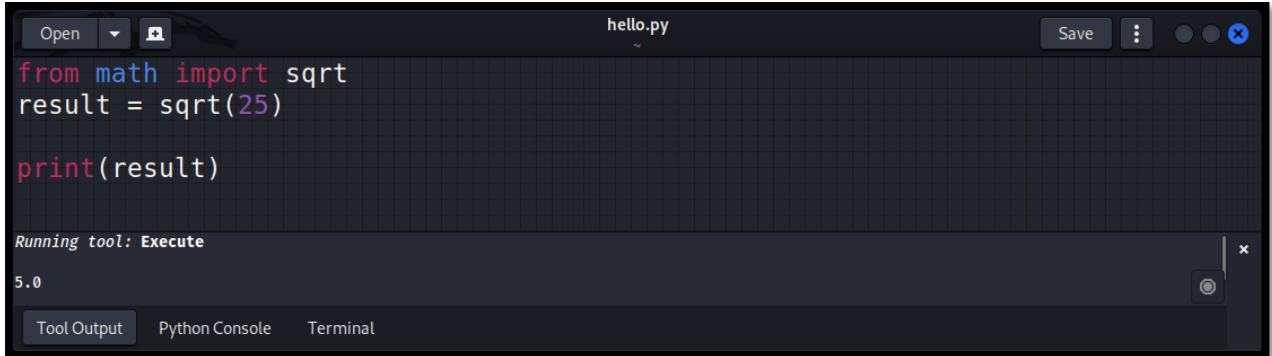
Output: 5.0

Here, the **math** module is imported, and its **sqrt** function is used.

Importing Specific Attributes

Instead of importing an entire module, you can choose to import only specific attributes (functions, classes, variables) using the **from ... import ...** statement.

Example:



```
Open hello.py Save
```

```
from math import sqrt
result = sqrt(25)

print(result)
```

Running tool: Execute

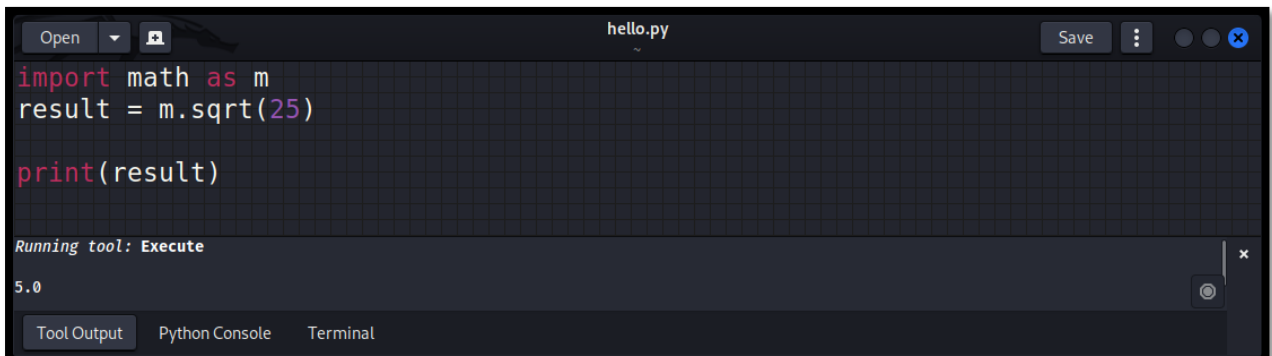
```
5.0
```

Tool Output Python Console Terminal

Renaming Modules or Attributes on Import

For convenience or to avoid naming conflicts, you can rename a module or its attributes when importing.

Example:



```
Open hello.py Save
```

```
import math as m
result = m.sqrt(25)

print(result)
```

Running tool: Execute

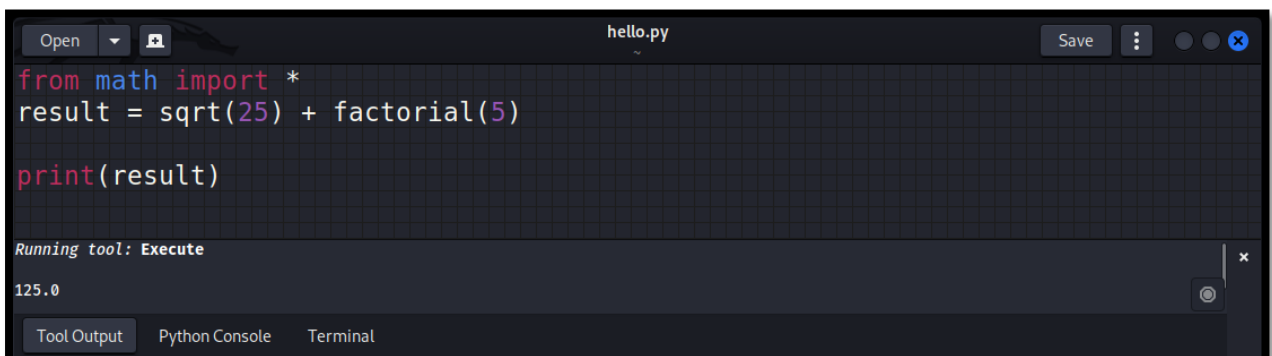
```
5.0
```

Tool Output Python Console Terminal

Importing All Attributes from a Module

While not recommended due to potential naming conflicts, you can import all attributes from a module using the ***** wildcard.

Example:



```
Open hello.py Save
```

```
from math import *
result = sqrt(25) + factorial(5)

print(result)
```

Running tool: Execute

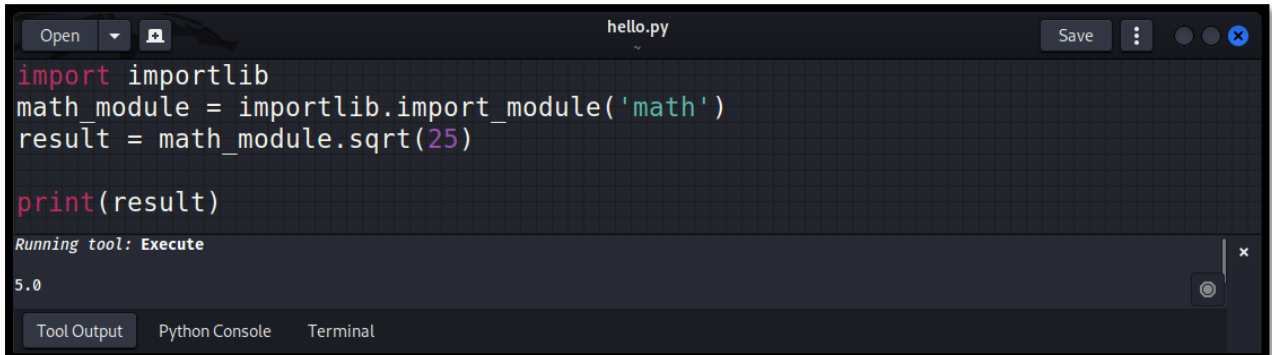
```
125.0
```

Tool Output Python Console Terminal

The importlib Module

Python provides the **importlib** module, which contains functions to import modules programmatically.

Example:



```
Open | hello.py | Save | [Window Controls]
import importlib
math_module = importlib.import_module('math')
result = math_module.sqrt(25)

print(result)

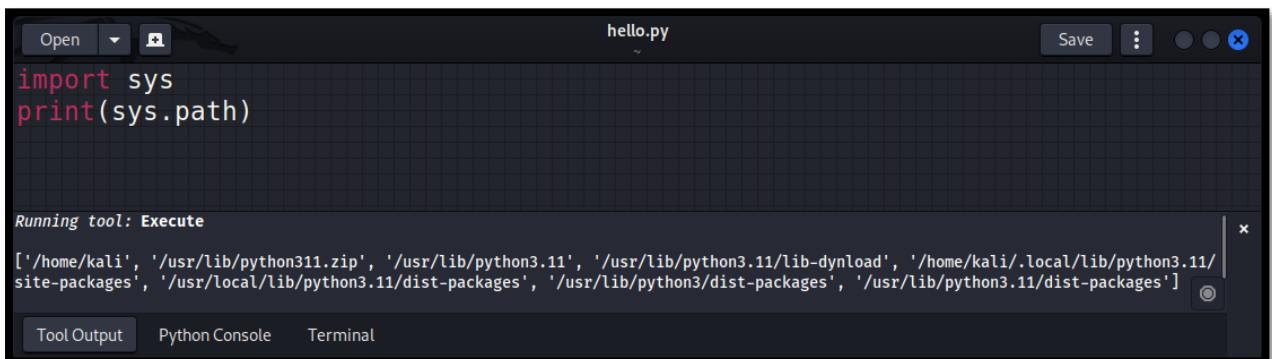
Running tool: Execute
5.0
Tool Output | Python Console | Terminal
```

Output: 5.0

Module Search Path

When a module is imported, Python searches for it in a list of directories defined in **sys.path**. This list typically includes the current directory, directories listed in the PYTHONPATH environment variable, and standard library directories.


Example:



```
Open | hello.py | Save | [Window Controls]
import sys
print(sys.path)

Running tool: Execute
['/home/kali', '/usr/lib/python311.zip', '/usr/lib/python3.11', '/usr/lib/python3.11/lib-dynload', '/home/kali/.local/lib/python3.11/site-packages', '/usr/local/lib/python3.11/dist-packages', '/usr/lib/python3/dist-packages', '/usr/lib/python3.11/dist-packages']
Tool Output | Python Console | Terminal
```

To add a new directory to the search path:



```
Open | hello.py | Save | [Window Controls]
import sys
sys.path.append('/home/kali/Desktop')

print(sys.path)

Running tool: Execute
['/home/kali', '/usr/lib/python311.zip', '/usr/lib/python3.11', '/usr/lib/python3.11/lib-dynload', '/home/kali/.local/lib/python3.11/site-packages', '/usr/local/lib/python3.11/dist-packages', '/usr/lib/python3/dist-packages', '/usr/lib/python3.11/dist-packages', '/home/kali/Desktop']
Tool Output | Python Console | Terminal
```

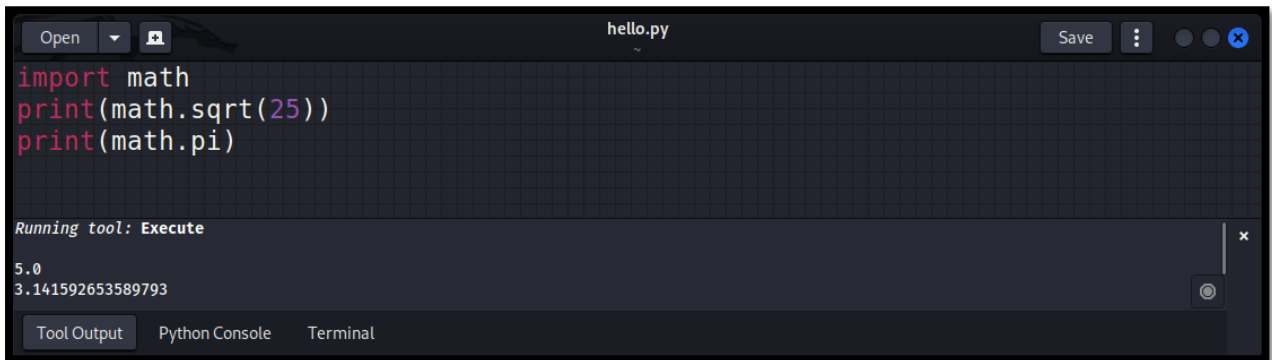

Common built-in Modules

Python's standard library is vast, offering a wide range of modules that provide utilities, data structures, and tools for various tasks.

The math Module

The **math** module provides mathematical functions and constants.

Example:



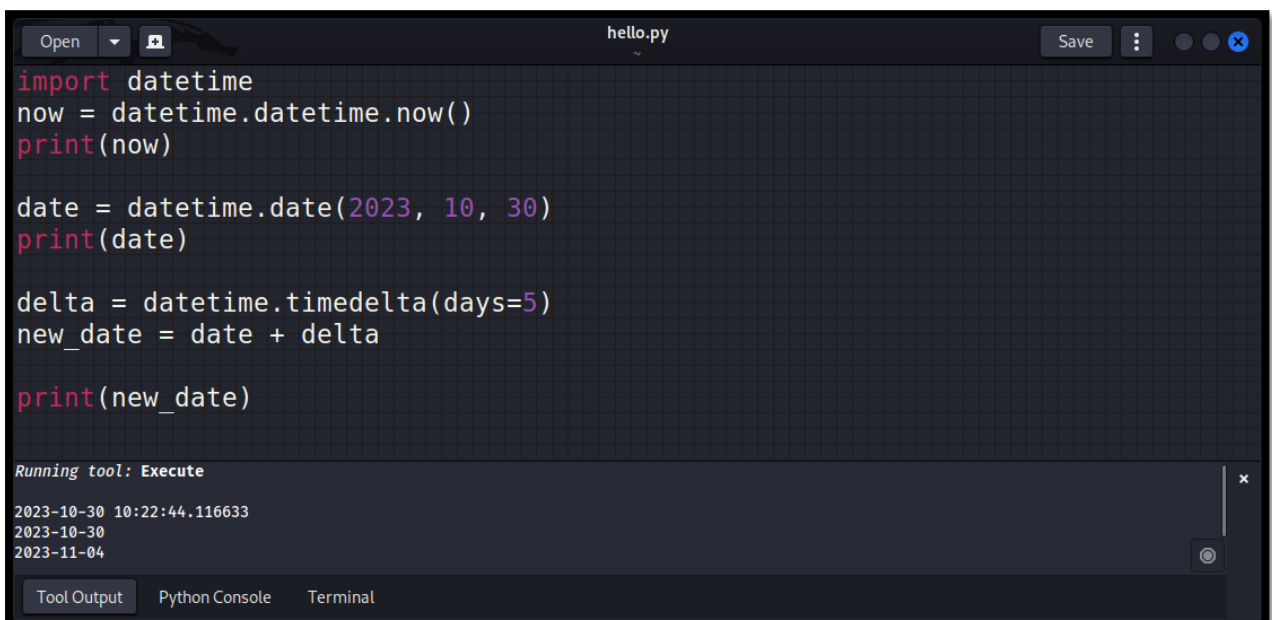
```
hello.py
import math
print(math.sqrt(25))
print(math.pi)

Running tool: Execute
5.0
3.141592653589793
```

The datetime Module

The **datetime** module supplies classes to work with date and time.

Example:



```
hello.py
import datetime
now = datetime.datetime.now()
print(now)

date = datetime.date(2023, 10, 30)
print(date)

delta = datetime.timedelta(days=5)
new_date = date + delta

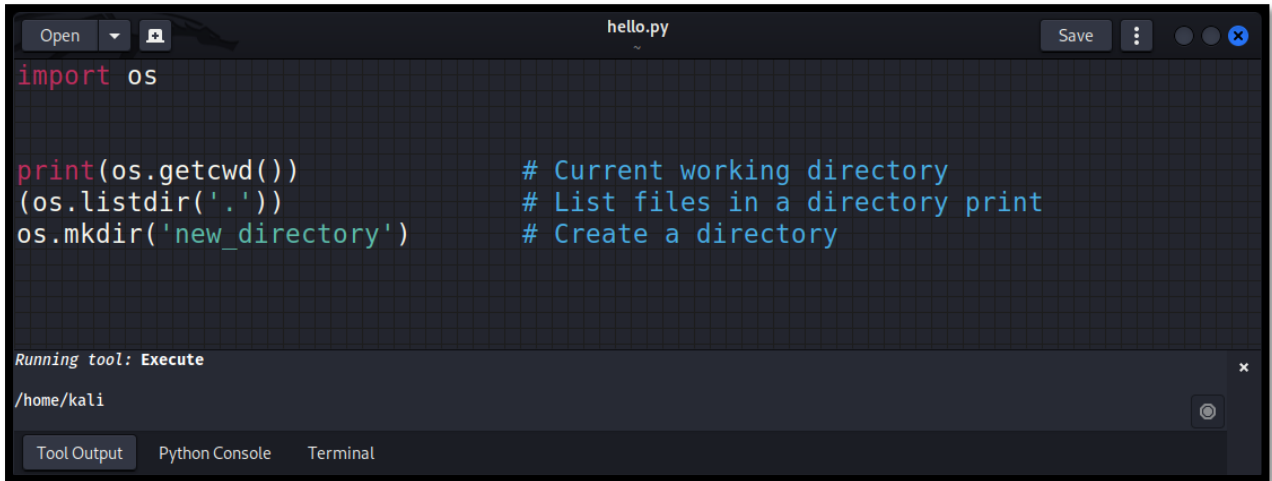
print(new_date)

Running tool: Execute
2023-10-30 10:22:44.116633
2023-10-30
2023-11-04
```

The os Module

The **os** module provides a way to use operating system-dependent functionalities.

Example:



```
hello.py
import os

print(os.getcwd())           # Current working directory
(os.listdir('.'))           # List files in a directory print
os.mkdir('new_directory')   # Create a directory

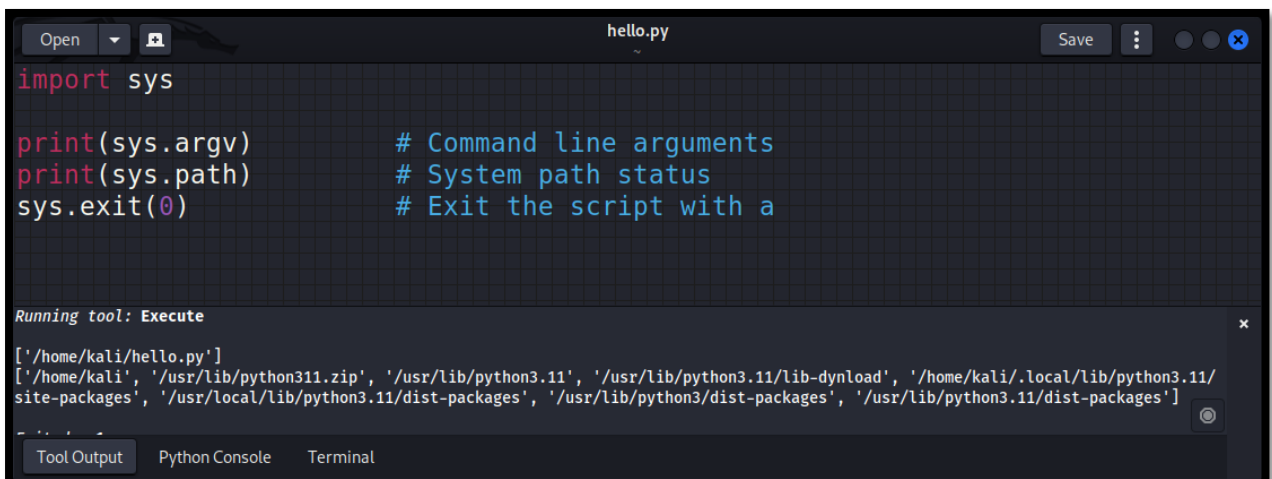
Running tool: Execute
/home/kali

Tool Output Python Console Terminal
```

The sys Module

The **sys** module provides access to Python interpreter variables and functions.

Example:



```
hello.py
import sys

print(sys.argv)             # Command line arguments
print(sys.path)             # System path status
sys.exit(0)                 # Exit the script with a

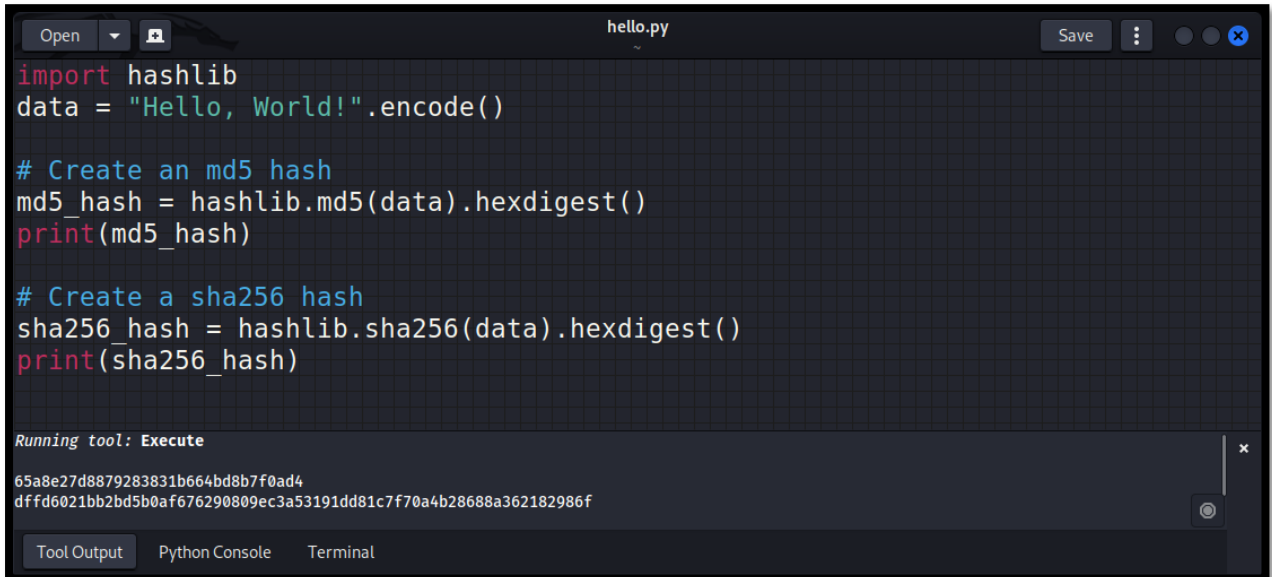
Running tool: Execute
['/home/kali/hello.py']
['/home/kali', '/usr/lib/python3.11.zip', '/usr/lib/python3.11', '/usr/lib/python3.11/lib-dynload', '/home/kali/.local/lib/python3.11/site-packages', '/usr/local/lib/python3.11/dist-packages', '/usr/lib/python3/dist-packages', '/usr/lib/python3.11/dist-packages']

Tool Output Python Console Terminal
```

The hashlib Module

The **hashlib** module provides algorithms for message digests (hashing).

Example:



```
Open hello.py Save
import hashlib
data = "Hello, World!".encode()

# Create an md5 hash
md5_hash = hashlib.md5(data).hexdigest()
print(md5_hash)

# Create a sha256 hash
sha256_hash = hashlib.sha256(data).hexdigest()
print(sha256_hash)

Running tool: Execute
65a8e27d8879283831b664bd8b7f0ad4
dfffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688a362182986f
Tool Output Python Console Terminal
```

The random Module

The **random** module provides functions to generate random numbers.

Example:



```
Open hello.py Save
import random
# Random float between 0 and 1
print(random.random())

# Random integer between 1 and 10
print(random.randint(1, 10))

# Random choice from a list
choices = ['apple', 'banana', 'cherry']
print(random.choice(choices))

Running tool: Execute
0.13195155437284567
10
cherry
Tool Output Python Console Terminal
```

Working with JSON, CSV

While plain text files are common and straightforward, many applications require more structured data formats like JSON and CSV. Python provides built-in libraries to handle these formats, making it easy to read from and write to such files.

Working with JSON in Python

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate.

The json Module

Python includes the `json` module in its standard library, which provides methods to encode and decode JSON data.

Reading JSON Data

To parse JSON data in Python, you can use the `json.load()` method for files or `json.loads()` for strings.

Example:



```
hello.py
import json
with open("data.json", "r") as file:
    data = json.load(file)
    print(data)

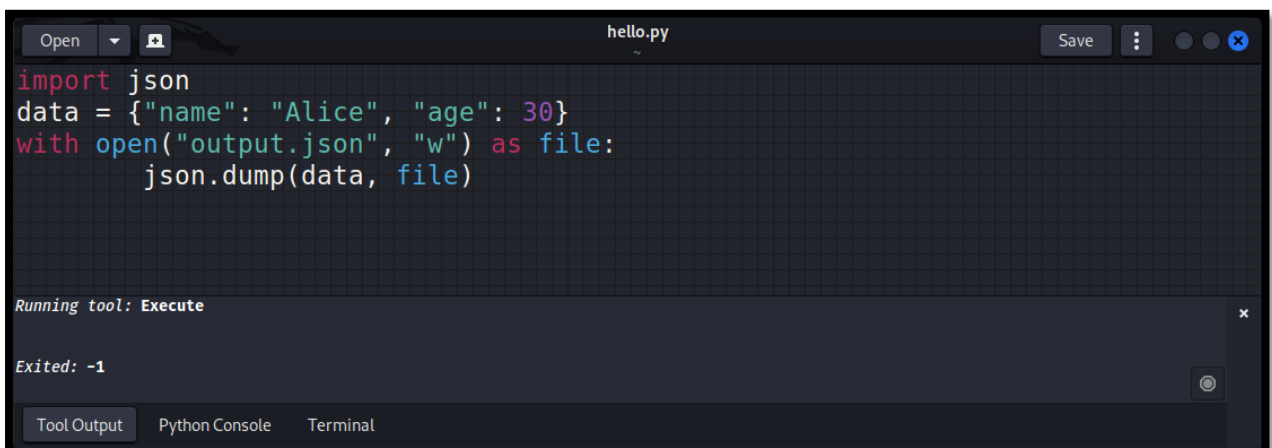
Running tool: Execute
{'description': 'Variables', 'questions': [{'question': '1. If you set the variable 'age' to 14, how would you command it to show its value?', 'answer': 'print(age)', 'accurate': True}, {'question': '2. t=15, t=t+30; what would be the output of 't'?', 'answer': '45', 'accurate': True}, {'question': '3. t=1 and x=1.0; what will be the output when executing print(t=x)?', 'answer': 'True', 'accurate': True}]}

Tool Output Python Console Terminal
```

Writing JSON Data

To write JSON data, you can use the `json.dump()` method for files or `json.dumps()` for strings.

Example:



```
hello.py
import json
data = {"name": "Alice", "age": 30}
with open("output.json", "w") as file:
    json.dump(data, file)

Running tool: Execute
Exited: -1

Tool Output Python Console Terminal
```

Working with CSV in Python

CSV (Comma-Separated Values) is a simple file format used to store tabular data, such as spreadsheets and databases.

The csv Module

Python's standard library includes the `csv` module, which provides functionality to read from and write to CSV files.

Reading CSV Data

The `csv.reader()` function allows you to read CSV files.

Example:



```
import csv with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Writing CSV Data

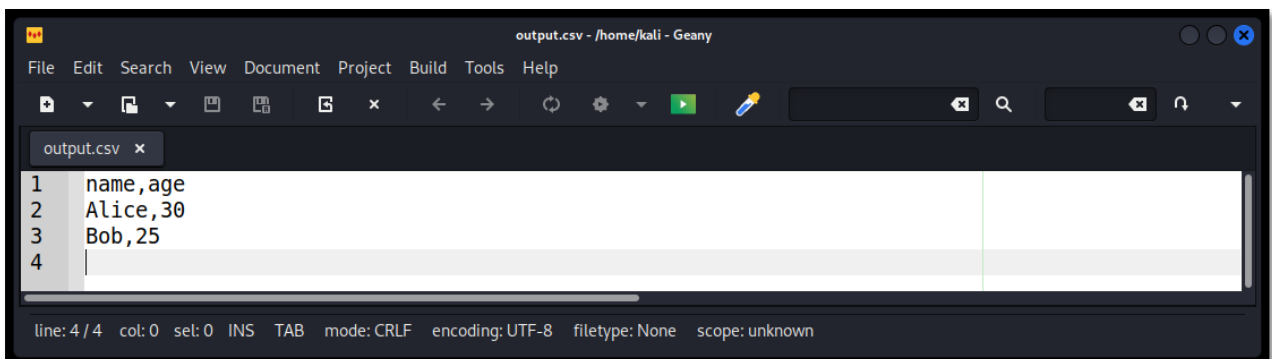
To write data to a CSV file, you can use the `csv.writer()` function.

Example:



```
import csv
data = [{"name", "age"}, {"Alice", 30}, {"Bob", 25}]
with open("output.csv", "w", newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)
```

File Output:



```
output.csv x
1 name,age
2 Alice,30
3 Bob,25
4
```

line: 4 / 4 col: 0 sel: 0 INS TAB mode: CRLF encoding: UTF-8 filetype: None scope: unknown

Functions

Defining a Function

Functions are the building blocks of a program, allowing for code modularity, reusability, and organization. In Python, functions are first-class citizens, meaning they can be passed around and used as arguments just like any other object.

What is a Function?

A function is a block of organized, reusable code that performs a specific task. Functions provide better modularity for your application and allow for high levels of code reuse.

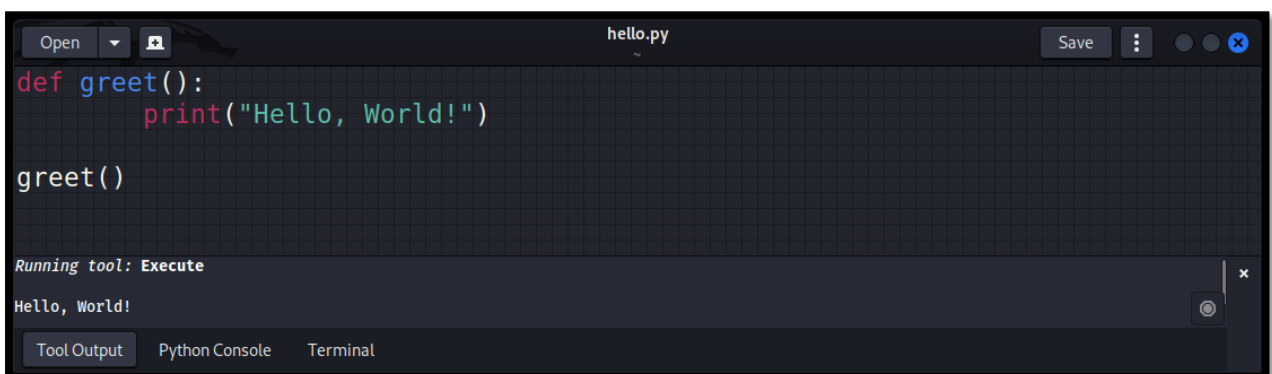
Basic Function Definition

In Python, a function is defined using the **def** keyword, followed by the function name, a pair of parentheses, and a colon. The function body starts after the colon and is indented.

Syntax:

```
def function_name():  
    # function body pass
```

Example:



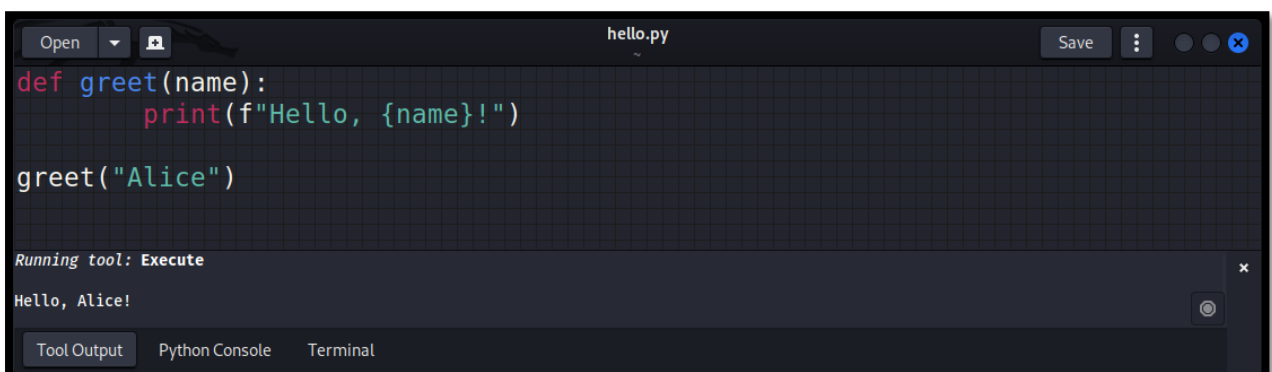
```
hello.py  
def greet():  
    print("Hello, World!")  
  
greet()  
  
Running tool: Execute  
Hello, World!
```

The screenshot shows a code editor window titled 'hello.py'. The code defines a function 'greet()' that prints 'Hello, World!'. Below the code, the function is called 'greet()'. A 'Running tool: Execute' button is visible, and the output 'Hello, World!' is displayed in the console area at the bottom.

Function Parameters

Functions can take arguments, which are values you supply to the function so it can perform an action based on those values. These arguments are called parameters.

Example:



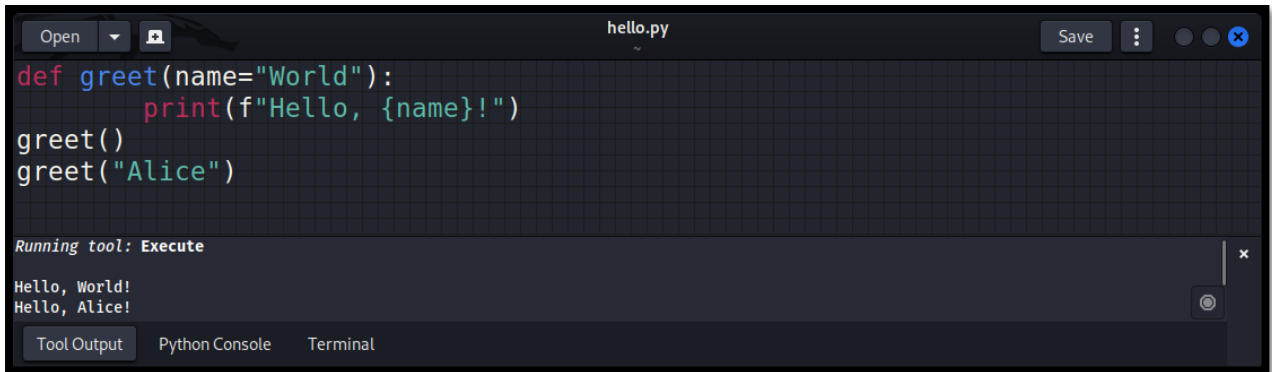
```
hello.py  
def greet(name):  
    print(f"Hello, {name}!")  
  
greet("Alice")  
  
Running tool: Execute  
Hello, Alice!
```

The screenshot shows a code editor window titled 'hello.py'. The code defines a function 'greet(name)' that takes a parameter 'name' and prints 'Hello, {name}!'. Below the code, the function is called 'greet("Alice")'. A 'Running tool: Execute' button is visible, and the output 'Hello, Alice!' is displayed in the console area at the bottom.

Default Parameter Values

You can provide default values for function parameters. If a value for that parameter is not provided when the function is called, the default value is used.

Example:



```
def greet(name="World"):
    print(f"Hello, {name}!")
greet()
greet("Alice")
```

Running tool: Execute

```
Hello, World!
Hello, Alice!
```

Tool Output Python Console Terminal

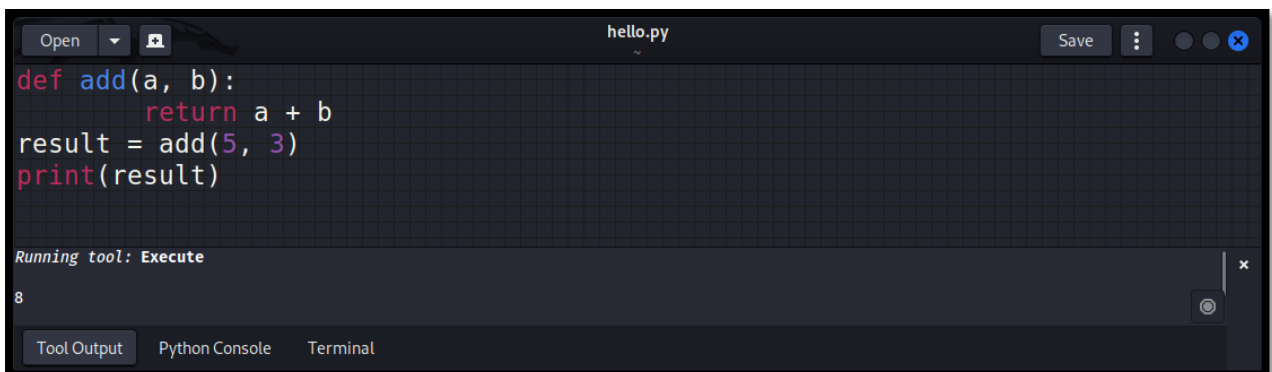
Output:

```
Hello, World!
Hello, Alice!
```

Return Values

Functions can return values using the **return** statement. Once a function returns a value, it immediately exits and does not execute any code that follows.

Example:



```
def add(a, b):
    return a + b
result = add(5, 3)
print(result)
```

Running tool: Execute

```
8
```

Tool Output Python Console Terminal

Function Parameters and Arguments

Function parameters and arguments are fundamental concepts in Python, allowing developers to create flexible and reusable functions.

Basic Parameters and Arguments

Parameters are the names listed in the function definition, while arguments are the values passed into the function when it is called.

Example:



```
def greet(name):          # 'name' is a parameter
    print(f"Hello, {name}!")

greet("Alice")           # "Alice" is an argument
```

Running tool: Execute

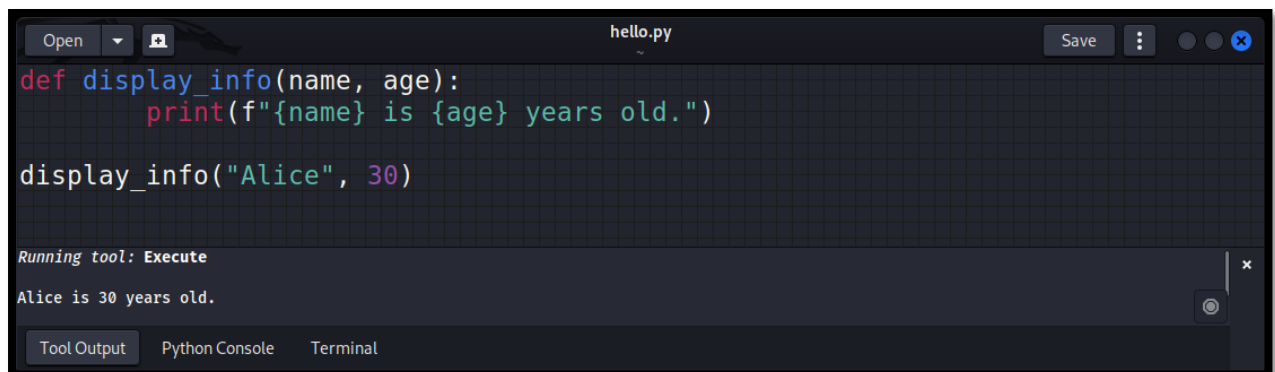
Hello, Alice!

Tool Output Python Console Terminal

Positional Arguments

Positional arguments are arguments that need to be passed in the same order as the parameters in the function definition.

Example:



```
def display_info(name, age):
    print(f"{name} is {age} years old.")

display_info("Alice", 30)
```

Running tool: Execute

Alice is 30 years old.

Tool Output Python Console Terminal

Default Parameter Values

You can assign default values to parameters, making them optional when calling the function.

Example:



```
def greet(name="World"):
    print(f"Hello, {name}!")

greet()
greet("Alice")
```

Running tool: Execute

Hello, World!
Hello, Alice!

Tool Output Python Console Terminal

Output:

```
Hello, World!
Hello, Alice!
```

Keyword Arguments

Keyword arguments allow you to pass arguments out of order by specifying their names.

Example:



```
def display_info(name, age):
    print(f"{name} is {age} years old.")

display_info(age=30, name="Alice")
```

Running tool: Execute

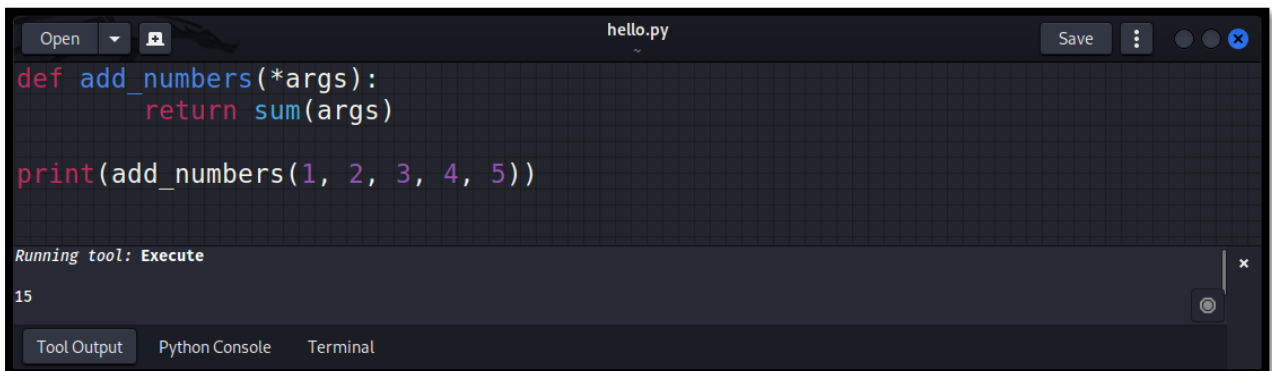
Alice is 30 years old.

Tool Output Python Console Terminal

Order doesn't matter when using keyword arguments.

Arbitrary Positional Arguments (*args)

If you're unsure about the number of positional arguments that will be passed to a function, you can use ***args** to capture them as a tuple.

Example:

```
def add_numbers(*args):  
    return sum(args)  
  
print(add_numbers(1, 2, 3, 4, 5))
```

Running tool: Execute

```
15
```

Tool Output Python Console Terminal

Arbitrary Keyword Arguments (kwargs)**

For an unknown number of keyword arguments, use ****kwargs** to capture them as a dictionary.

Example:

```
def display_data(**kwargs):  
    for key, value in kwargs.items():  
        print(f"{key}: {value}")  
  
display_data(name="Alice", age=30, profession="Engineer")
```

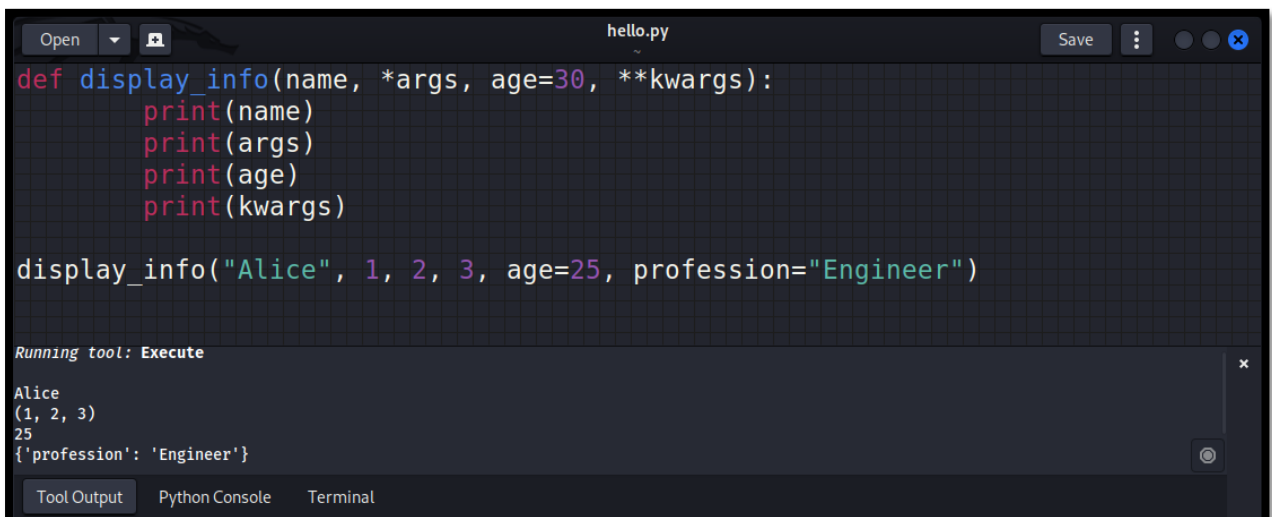
Running tool: Execute

```
name: Alice  
age: 30  
profession: Engineer
```

Tool Output Python Console Terminal

Mixing Argument Types

You can mix positional, ***args**, keyword, and ****kwargs** arguments in a function definition. However, the order should be: standard positional arguments, ***args**, standard keyword arguments, ****kwargs**.

Example:

```
def display_info(name, *args, age=30, **kwargs):  
    print(name)  
    print(args)  
    print(age)  
    print(kwargs)  
  
display_info("Alice", 1, 2, 3, age=25, profession="Engineer")
```

Running tool: Execute

```
Alice  
(1, 2, 3)  
25  
{'profession': 'Engineer'}
```

Tool Output Python Console Terminal

Passing Lists and Dictionaries as Arguments

You can unpack and pass lists and dictionaries as function arguments using `*` and `**`, respectively.

Example:



```
def display_info(name, age):  
    print(f"{name} is {age} years old.")  
  
data_list = ["Alice", 30]  
data_dict = {"name": "Bob", "age": 25}  
  
display_info(*data_list)  
display_info(**data_dict)
```

Running tool: Execute

```
Alice is 30 years old.  
Bob is 25 years old.
```

Tool Output Python Console Terminal

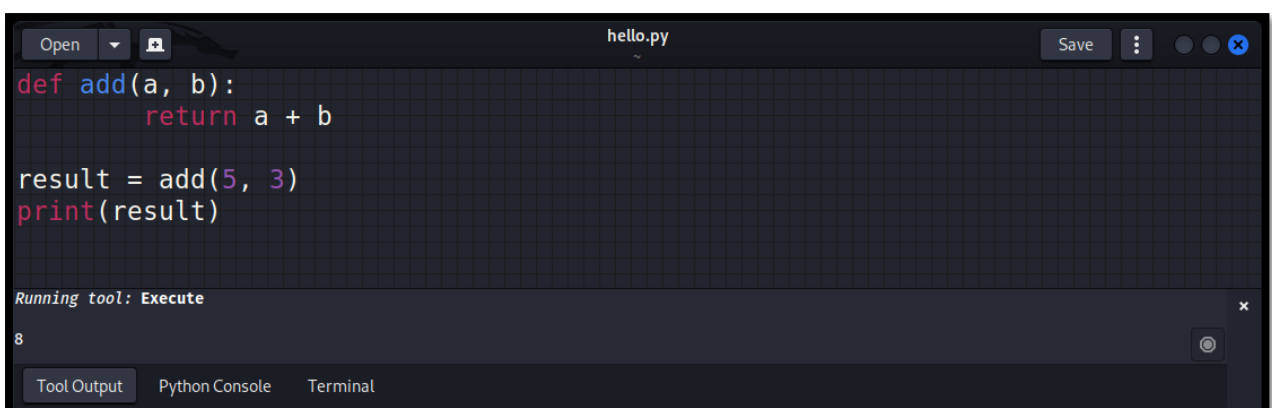
Return Statement

The **return** statement is a fundamental aspect of functions in Python, allowing developers to send results back from a function to the point where the function was called.

Basics of the Return Statement

The **return** statement is used to exit a function and send a result back to the caller. Once a **return** statement is executed, no other code within the function is run.

Example:



```
def add(a, b):  
    return a + b  
  
result = add(5, 3)  
print(result)
```

Running tool: Execute

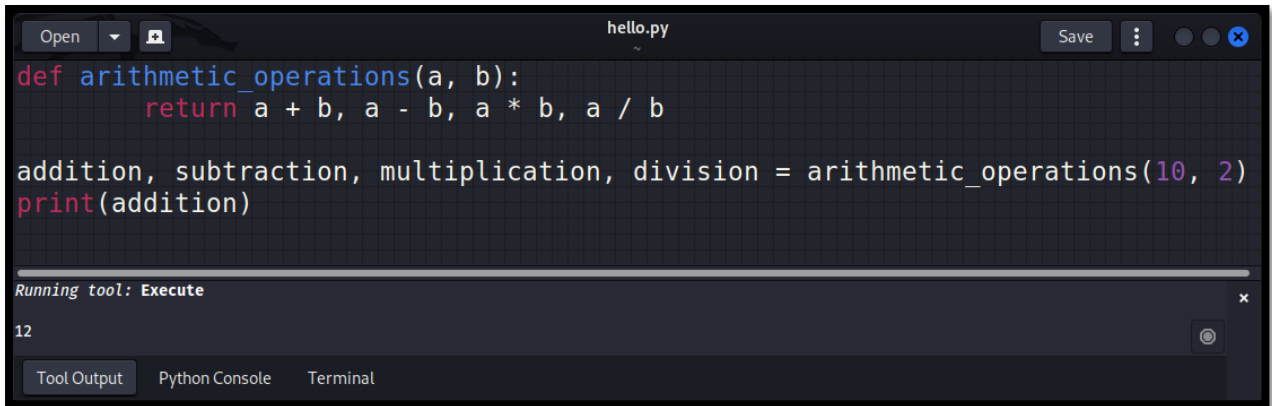
```
8
```

Tool Output Python Console Terminal

Returning Multiple Values

A function in Python can return multiple values in the form of a tuple, list, dictionary, or any other collection type.

Example:



```
def arithmetic_operations(a, b):  
    return a + b, a - b, a * b, a / b  
  
addition, subtraction, multiplication, division = arithmetic_operations(10, 2)  
print(addition)
```

Running tool: Execute

12

Tool Output Python Console Terminal

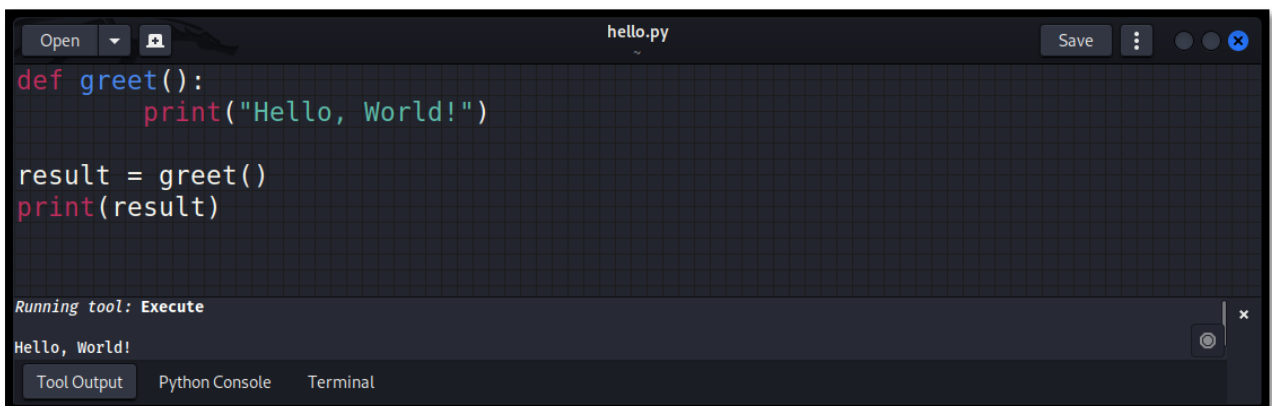
Output:

12

The Implicit Return

If a function doesn't have a **return** statement, it implicitly returns **None**.

Example:



```
def greet():  
    print("Hello, World!")  
  
result = greet()  
print(result)
```

Running tool: Execute

Hello, World!

Tool Output Python Console Terminal

Output of print(result): None

Conditional Return

You can have multiple **return** statements in a function, typically used in conjunction with conditional statements.

Example:



```
def is_even(num):  
    if num % 2 == 0:  
        return True  
    else:  
        return False  
  
print(is_even(4))  
print(is_even(7))
```

Running tool: Execute

True
False

Tool Output Python Console Terminal

Return Statement with Recursive Functions

Recursive functions are functions that call themselves. The **return** statement plays a crucial role in ensuring that recursive functions produce the desired output and eventually terminate.

Example:



```
def factorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n-1)  
  
print(factorial(5))
```

Running tool: Execute

120

Tool Output Python Console Terminal

Output: 120

Variable Scope (local vs global)

In Python, as in many programming languages, variables have a "scope" that determines their visibility and lifespan within a program.

What is Variable Scope?

Variable scope refers to the region of the code where a variable can be accessed or modified. The two primary types of variable scopes in Python are "local" and "global."

Local Scope

A variable declared within a function has a local scope. It's accessible only inside that function and not outside it.

Example:



```
def greet():
    message = "Hello, World!"
    print(message)

greet()
print(message) # This would raise an error since 'message' is not accessible
                # outside the function.
```

Running tool: Execute

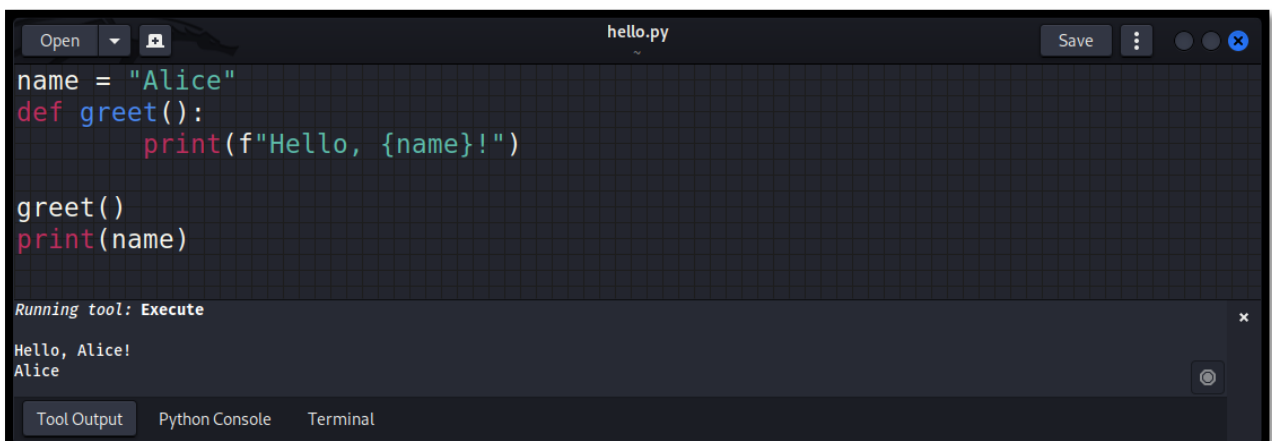
```
Hello, World!
Traceback (most recent call last):
  File "/home/kali/hello.py", line 5, in <module>
    print(message)
    ^^^^^^^
NameError: name 'message' is not defined
```

Tool Output Python Console Terminal

Global Scope

A variable declared outside all functions has a global scope. It's accessible throughout the file, including inside functions (unless shadowed by a local variable with the same name).

Example:



```
name = "Alice"
def greet():
    print(f"Hello, {name}!")

greet()
print(name)
```

Running tool: Execute

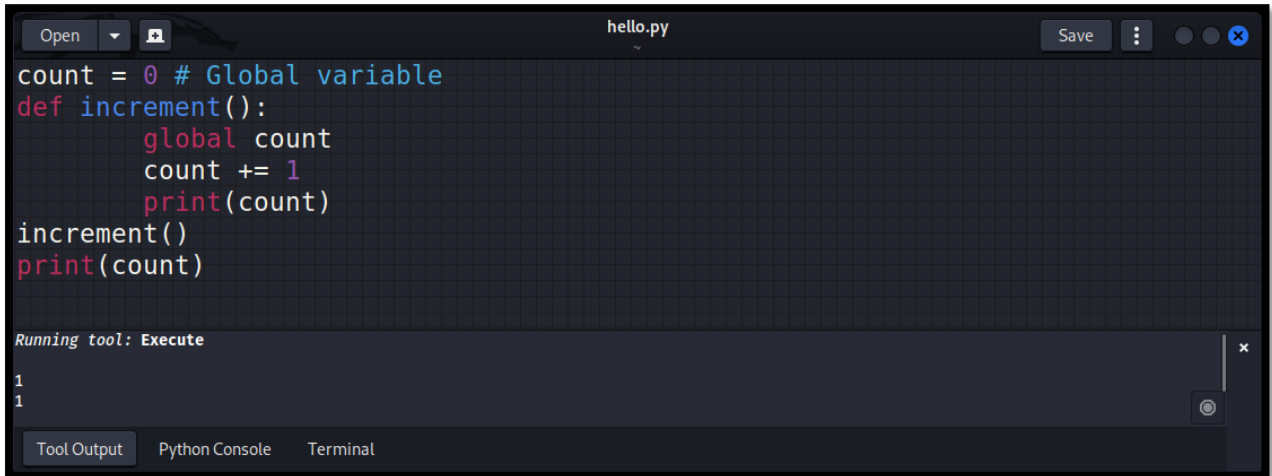
```
Hello, Alice!
Alice
```

Tool Output Python Console Terminal

The global Keyword

If you need to modify a global variable from within a function, you can use the **global** keyword.

Example:



```
count = 0 # Global variable
def increment():
    global count
    count += 1
    print(count)
increment()
print(count)
```

Running tool: Execute

1
1

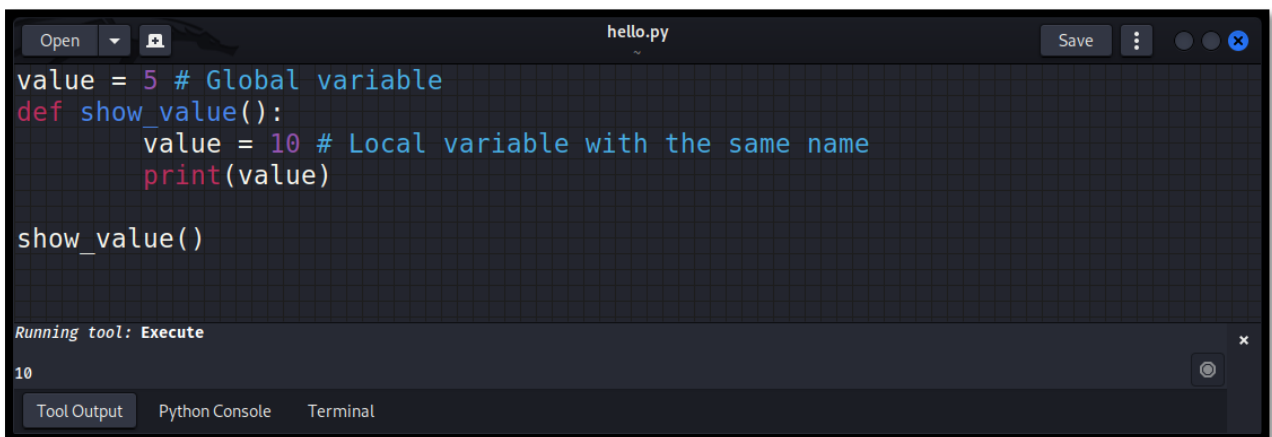
Tool Output Python Console Terminal

Without the **global** keyword, the function would create a local variable named **count**, and the global **count** would remain unchanged.

Shadowing

If a local variable has the same name as a global variable, the local variable will "shadow" the global one within its scope.

Example:



```
value = 5 # Global variable
def show_value():
    value = 10 # Local variable with the same name
    print(value)
show_value()
```

Running tool: Execute

10

Tool Output Python Console Terminal

Best Practices

1. **Avoid Global Variables:** While global variables can be convenient, they can make code harder to understand and debug. It's generally better to pass variables as function parameters instead.
2. **Be Careful with Shadowing:** Shadowing can make code confusing. It's often better to use unique variable names to avoid this issue.
3. **Limit Use of global and nonlocal:** While these keywords can be useful, overusing them can make code harder to follow.

Lambda Functions

Lambda functions, often referred to as "anonymous functions," are a unique and powerful feature of Python. They allow for the creation of simple functions in a concise manner.

What is a Lambda Function?

A lambda function is a small, unnamed function defined using the **lambda** keyword. Unlike regular functions defined using the **def** keyword, lambda functions can have any number of arguments but only one expression.

Basic Syntax of Lambda Functions

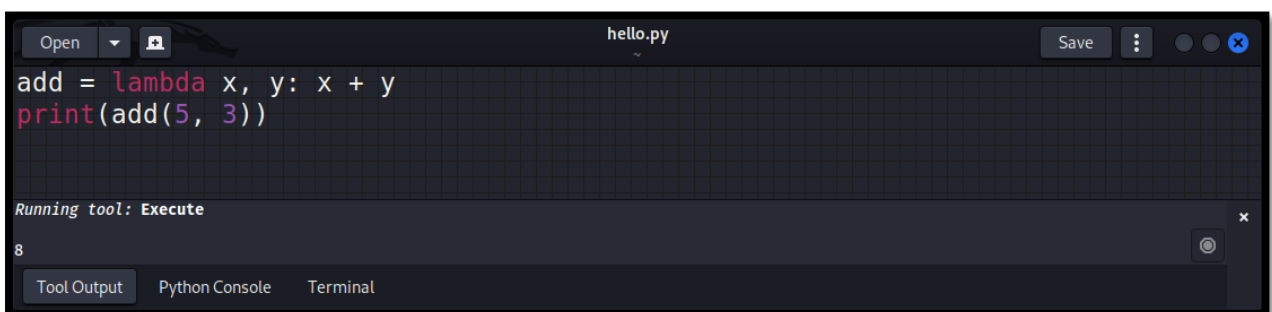
The general syntax of a lambda function is:

lambda arguments: expression

The expression is executed and returned when the lambda function is called.

Basic Example

Here's a simple example of a lambda function that adds two numbers:

A screenshot of a code editor window titled 'hello.py'. The editor contains the following Python code:

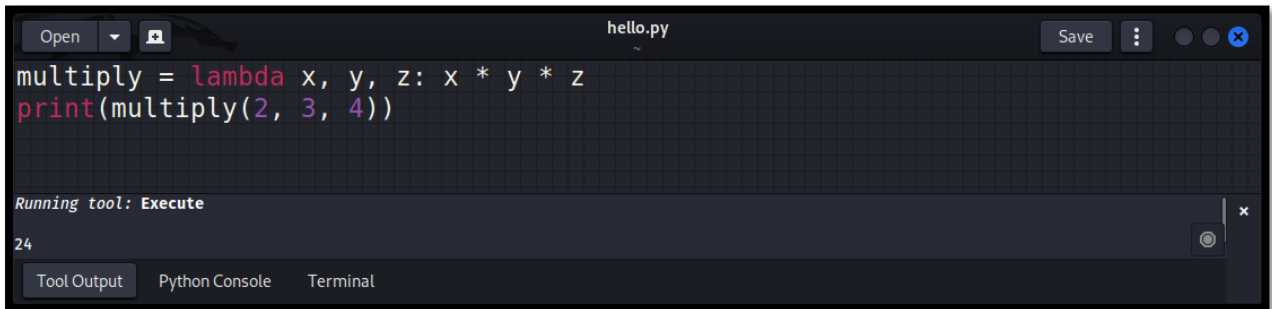
```
add = lambda x, y: x + y
print(add(5, 3))
```

Below the code editor, there is a 'Running tool: Execute' section. The output of the execution is displayed as the number '8'. At the bottom of the window, there are tabs for 'Tool Output', 'Python Console', and 'Terminal'.

Output: 8

Lambda Functions with Multiple Arguments

Lambda functions can accept any number of arguments:



```
hello.py
multiply = lambda x, y, z: x * y * z
print(multiply(2, 3, 4))

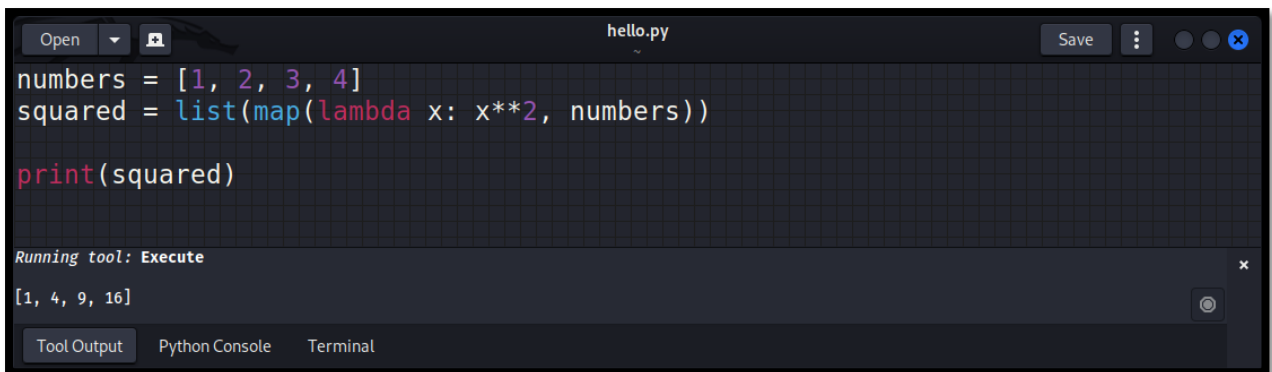
Running tool: Execute
24
Tool Output Python Console Terminal
```

Output: 24

Using Lambda Functions with Built-in Functions

Lambda functions are commonly used with built-in functions like **map()**, **filter()**, and **sorted()**.

Example with map():

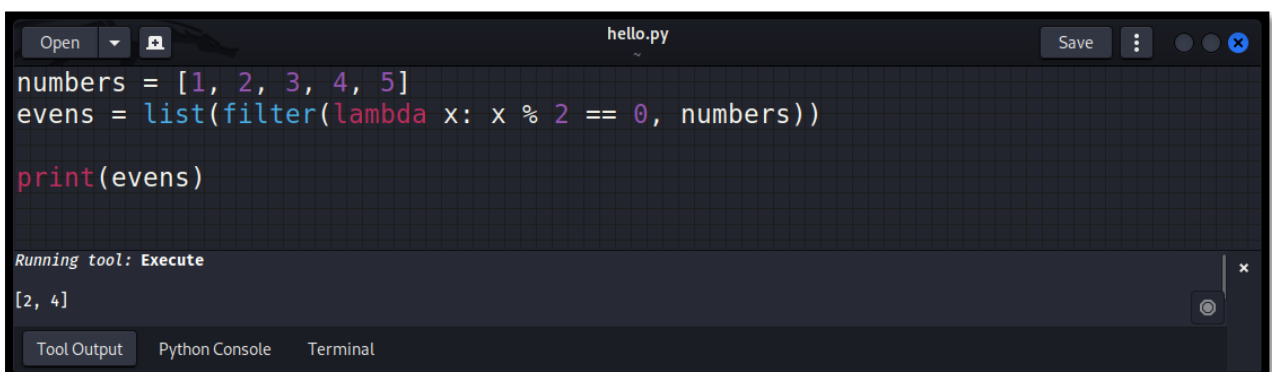


```
hello.py
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))

print(squared)

Running tool: Execute
[1, 4, 9, 16]
Tool Output Python Console Terminal
```

Example with filter():



```
hello.py
numbers = [1, 2, 3, 4, 5]
evens = list(filter(lambda x: x % 2 == 0, numbers))

print(evens)

Running tool: Execute
[2, 4]
Tool Output Python Console Terminal
```

Lambda Functions for Sorting

Lambda functions are particularly useful for custom sorting.

Example:



```
hello.py
data = [('apple', 3), ('banana', 1), ('cherry', 2)]
sorted_data = sorted(data, key=lambda x: x[1])

print(sorted_data)

Running tool: Execute
[('banana', 1), ('cherry', 2), ('apple', 3)]
Tool Output Python Console Terminal
```

Output: [('banana', 1), ('cherry', 2), ('apple', 3)]

Limitations of Lambda Functions

While lambda functions are powerful, they have some limitations:

1. **Single Expression:** Lambda functions can only have one expression, which means they can't contain multiple statements or assignments.
2. **Less Readable:** For complex operations, lambda functions can be less readable than regular functions.
3. **Limited Functionality:** Without the ability to include statements, certain operations can't be performed within a lambda.

When to Use Lambda Functions

Lambda functions are best suited for simple operations that can be expressed in a single expression. They're especially useful when you need a short-lived function for a specific task, like with **map()**, **filter()**, or **sorted()**.