Basic Syntax

Comments

```
# This is a single-line comment
"""
This is a
multi-line comment
"""
```

- Single-line comments use the **#** symbol.
- Multi-line comments (technically multi-line strings) are often used as documentation or block comments.

Variables and Data Types

```
# Variable declaration
name = "Alice"        # String
age = 30              # Integer
height = 5.7          # Float
is_student = True     # Boolean
```

- Python uses **dynamic typing**—no need to declare variable types explicitly.
- Common data types: `str`, `int`, `float`, `bool`.

Basic Input/Output

```
# Input
user_input = input("Enter your name: ")

# Output
print("Hello, " + user_input + "!")
```

- `input()` reads a line of text from the user as a string.
- `print()` displays output to the console.

Data Structures

```
# List
fruits = ["apple", "banana", "cherry"]

# Tuple
dimensions = (1920, 1080)

# Dictionary
student = {"name": "Alice", "age": 30}
```

- **List:** Ordered, mutable collection.
- **Tuple:** Ordered, immutable collection.
- **Dictionary:** Unordered, key-value pairs (like objects in JavaScript or maps in Rust).

Lists

Lists are one of Python's built-in data types that allow you to store **multiple items** in a **single variable**. Lists are **ordered, mutable**, and can contain elements of different types.

Creating a List

```
my_list = [1, 2, 3, 4, 5]
```

- Creates a list of integers.

Accessing Elements

```
first_item = my_list[0] # 1
last_item = my_list[-1] # 5
```

- Use **indexing** to access specific elements. Negative indices count from the end.

Adding Elements

```
my_list.append(6)    # Adds 6 to the end
my_list.insert(0, 0) # Inserts 0 at the beginning
```

- **`append()`** adds to the end of the list.
- **`insert(index, value)`** places an item at a specific position.

Removing Elements

```
my_list.remove(3) # Removes the first occurrence of 3
popped_item = my_list.pop() # Removes and returns the last item
```

- **`remove(value)`** deletes the **first match** of the value.
- **`pop()`** removes the **last item** (or a specific index if provided).

Slicing

```
length = len(my_list)
```

- **`len()`** returns the number of items in the list.

Iterating Through a List

```
for item in my_list:
    print(item)
```

- Loop through each item in the list using a `for` loop.

List Comprehension

```
squared = [x**2 for x in my_list]
# [0, 1, 4, 9, 16, 25, 36]
```

- Creates a new list by applying an expression to each element.

Checking for Existence

```
exists = 3 in my_list  # True if 3 is in the list
```

- Use the `in` keyword to check if a value exists.

Sorting

```
my_list.sort() # Sorts the list in place
sorted_list = sorted(my_list) # Returns a new sorted list
```

- `sort()` modifies the original list.
- `sorted()` returns a **new list** without changing the original.

Dictionaries

Dictionaries are a built-in data type in Python that store **key-value pairs**. They are **unordered (prior to Python 3.7), mutable**, and optimized for **fast lookups, updates, and deletions** using keys.

Creating a Dictionary

```
my_dict = {'name': 'Alice', 'age': 25}
```

- A dictionary is defined using curly braces `{}` with key-value pairs separated by colons.

Accessing Values

```
name = my_dict['name']       # 'Alice'
age = my_dict.get('age')     # 25
```

- Access values using `dict[key]` or `dict.get(key)` (which avoids errors if the key doesn't exist).

Adding / Updating Items

```
del my_dict['age']              # Removes the key 'age'
value = my_dict.pop('city')     # Removes 'city' and returns its value
```

- `del` deletes a specific key.
- `pop(key)` removes and returns the value of the specified key.

Iterating Through a Dictionary

```
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

- Use `.items()` to loop through key-value pairs.

Checking for Existence

```
exists = 'name' in my_dict   # True if 'name' is a key in the dictionary
```

- The `in` keyword checks whether a key exists.

Dictionary Length

```
length = len(my_dict)
```

- `len()` returns the number of key-value pairs in the dictionary.

Copying a Dictionary

```
copy_dict = my_dict.copy()
```

- `.copy()` creates a shallow copy of the dictionary.

Dictionary Comprehension

```
squared_dict = {x: x**2 for x in range(5)}
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- Similar to list comprehensions, but builds a dictionary using an expression.

Merging Dictionaries

```
another_dict = {'city': 'Paris'}
merged_dict = {**my_dict, **another_dict}
```

- Use ** unpacking to merge multiple dictionaries (Python 3.5+). Later keys overwrite earlier ones.

Sets

Sets are a built-in data type in Python that store **unordered**, **unique elements**. They are useful for **membership testing**, removing duplicates, and performing **mathematical set operations**.

Creating a Set

```
my_set = {1, 2, 3, 4, 5}
```

- Sets are defined using curly braces { } and store unique items only.

Clearing a Set

```
my_set.clear()
```

- Removes **all elements** from the set.

Set Length

```
length = len(my_set)
```

- Returns the number of elements in the set.

Checking Existence

```
exists = 2 in my_set   # True if 2 is in the set
```

- Use the `in` keyword to check if a value exists in the set.

Set Operations

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}

union = set_a | set_b                # {1, 2, 3, 4, 5}
intersection = set_a & set_b         # {3}
difference = set_a - set_b           # {1, 2}
symmetric_difference = set_a ^ set_b # {1, 2, 4, 5}
```

- **Union (`|`)**: Combines all elements from both sets.
- **Intersection (`&`)**: Common elements only.
- **Difference (`-`)**: Elements in `set_a` but not in `set_b`.
- **Symmetric Difference (`^`)**: Elements in either set, but not both.

Iterating Through a Set

```
for item in my_set:
```

```
print(item)
```

- Use a `for` loop to access each element. Order is not guaranteed.

Set Comprehension

```
squared_set = {x**2 for x in range(5)}
# {0, 1, 4, 9, 16}
```

- Builds a set dynamically using an expression.

Converting a List to a Set

```
my_list = [1, 2, 2, 3, 4]
unique_set = set(my_list)  # {1, 2, 3, 4}
```

- Builds a set dynamically using an expression.

Adding Elements

```
my_set.add(6)
```

- Adds an element to the set (if it's not already present).

Removing Elements

```
my_set.remove(3) # Removes 3; raises KeyError if not found
my_set.discard(4) # Removes 4; does NOT raise an error if missing
popped_item = my_set.pop() # Removes and returns an arbitrary element
```

- `remove()` throws an error if the element doesn't exist.
- `discard()` fails silently if the item is missing.
- `pop()` removes a **random element** (due to unordered nature).

Tuples

Tuples are a built-in data type in Python used to store **ordered**, **immutable** collections. Once a tuple is created, **its elements cannot be changed**, making them useful for fixed data.

Creating a Tuple

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

- Tuples are defined using parentheses `()` and can hold elements of any type.

Accessing Elements

```
first_item = my_tuple[0] # 1
last_item = my_tuple[-1] # 5
```

- Access elements by index. Indexing starts at 0.

Single-Element Tuple

```
single_element = (1,)  # Must include a trailing comma
```

- A trailing comma is required to differentiate a tuple from a regular value.

Slicing

```
sub_tuple = my_tuple[1:4]  # (2, 3, 4)
```

- You can extract parts of a tuple using slice notation.

Length of a Tuple

```
length = len(my_tuple)
```

- Returns the number of elements in the tuple.

Iterating Through a Tuple

```
for item in my_tuple:
    print(item)
```

- Use a loop to access each item in a tuple.

Tuples are Immutable

```
# my_tuple[0] = 10  → Raises TypeError
```

- You cannot change the values of a tuple after creation.

Concatenating Tuples

```
new_tuple = my_tuple + (6, 7)
# Result: (1, 2, 3, 4, 5, 6, 7)
```

- You can create a new tuple by adding two tuples together.

Repeating Tuples

```
repeated_tuple = (1, 2) * 3
# Result: (1, 2, 1, 2, 1, 2)
```

- Repeats the tuple contents the specified number of times.

Packing and Unpacking

```
# Packing
packed = 1, 2, 3

# Unpacking
a, b, c = packed
# a = 1, b = 2, c = 3
```

- **Packing**: Storing multiple values into a tuple.
- **Unpacking**: Extracting values from a tuple into individual variables.

Range

The `range()` function generates an **immutable sequence** of numbers. It's commonly used for **iterating in loops**, especially `for` loops, and is **memory efficient** even with large ranges.

Creating a Range

```
r = range(5)
# Generates: 0, 1, 2, 3, 4
```

- Creates a sequence from 0 up to (but not including) 5.

Specifying Start and Stop

```
r = range(2, 8)
# Generates: 2, 3, 4, 5, 6, 7
```

- First argument is the start, second is the stop (exclusive).

Specifying Step

```
r = range(0, 10, 2)
# Generates: 0, 2, 4, 6, 8
```

- Third argument is the step (interval between numbers).

Converting to a List

```
list_range = list(range(5))
# [0, 1, 2, 3, 4]
```

- Converts the range object into a list to view or manipulate.

Iterating Over a Range

```
for i in range(5):
    print(i)
# Output: 0, 1, 2, 3, 4
```

- Used in loops to iterate a fixed number of times.

Reverse a Range

```
r = range(5, 0, -1)
# Generates: 5, 4, 3, 2, 1
```

- By using a **negative step**, you can create a descending sequence.

Length of a Range

```
length = len(range(5))
# Result: 5
```

- Returns the number of items in the range.

Checking Membership

```
exists = 3 in range(5)
# True if 3 is part of the range
```

- Use the `in` keyword to test if a number exists in the range.

Using with List Comprehension

```
squares = [x**2 for x in range(5)]
# Output: [0, 1, 4, 9, 16]
```

- Often used in list comprehensions to generate sequences.

Enumerate

The `enumerate()` function adds a counter to an iterable (like a list, string, or tuple) and returns an `enumerate` object. It is most useful when you need both the **index** and the **value** while iterating.

Basic Usage

```
my_list = ['a', 'b', 'c']
for index, value in enumerate(my_list):
    print(index, value)
# 0 a
# 1 b
# 2 c
```

- Tracks index automatically while iterating.

Specifying a Start Index

```
for index, value in enumerate(my_list, start=1):
    print(index, value)
# 1 a
# 2 b
# 3 c
```

- Starts indexing from 1 instead of 0.

Converting to a List of Tuples

```
enumerated_list = list(enumerate(my_list))
# [(0, 'a'), (1, 'b'), (2, 'c')]
```

- Useful for viewing or manipulating index-value pairs.

Using with Dictionary Comprehension

```
enumerated_dict = {index: value for index, value in enumerate(my_list)}
# {0: 'a', 1: 'b', 2: 'c'}
```

- Converts enumerated output into a dictionary.

Filtering Enumerated Items

```
for index, value in enumerate(my_list):
    if index % 2 == 0:
        print(value)
# a
# c
```

- Filters based on index (e.g., even positions).

Enumerate with Nested Loops

```
matrix = [[1, 2, 3], [4, 5, 6]]
for i, row in enumerate(matrix):
    for j, value in enumerate(row):
        print(f"matrix[{i}][{j}] = {value}")
# matrix[0][0] = 1
# matrix[0][1] = 2
# matrix[0][2] = 3
# matrix[1][0] = 4
# matrix[1][1] = 5
# matrix[1][2] = 6
```

- Great for working with 2D lists or matrices.

Enumerating Over a String

```
for index, char in enumerate("hello"):
    print(index, char)
# 0 h
# 1 e
# 2 l
# 3 l
# 4 o
```

- Works with any iterable, including strings.

Iterators

Iterators are objects that implement the **iterator protocol**, which includes the methods:

- `__iter__()` → returns the iterator object itself
- `__next__()` → returns the next value and raises `StopIteration` when exhausted

Creating an Iterator from a List

```
my_list = [1, 2, 3]
my_iterator = iter(my_list)
```

- `iter()` turns an iterable into an iterator.

Using next() to Retrieve Elements

```
first_item = next(my_iterator)  # 1
second_item = next(my_iterator) # 2
```

- Each call to `next()` gets the next item.
- Raises `StopIteration` if no items remain.

Iterating Using a Loop (Internally Uses an Iterator)

```
for item in my_list:
    print(item)
# 1
# 2
# 3
```

- The `for` loop uses `iter()` and `next()` under the hood.

Converting to a List

```
list_from_iterator = list(iter(range(5)))
# [0, 1, 2, 3, 4]
```

- Turns an iterable or iterator into a list.

Checking if an Object is an Iterator

```
is_iterator = hasattr(my_iterator, '__next__')

# True
```

- `iter()` returns an object with `__next__()` and `__iter__()`.

Creating a Custom Iterator

```
class MyRange:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.end:
            result = self.current
            self.current += 1
            return result
        else:
            raise StopIteration

my_range = MyRange(1, 4)
for number in my_range:
    print(number)

# 1
# 2
# 3
```

- Demonstrates a class-based iterator using custom logic.

Control Flow

Control flow determines the execution path of a program based on conditions, loops, and exception handling.

Conditional Statements

```
age = 20

if age < 18:
    print("Minor")
elif age < 65:
    print("Adult")
else:
    print("Senior")
```

- `if`, `elif`, and `else` control decision-making.

Nested Conditional Statements

```
number = 10

if number > 0:
    print("Positive")
    if number % 2 == 0:
        print("Even")
    else:
        print("Odd")
```

- Conditions can be nested inside one another.

Switch-like Structure using Dictionaries

```
def switch_case(option):
    return {
        1: "Option 1 selected",
        2: "Option 2 selected",
        3: "Option 3 selected"
    }.get(option, "Invalid option")


print(switch_case(2))
# Output: Option 2 selected
```

- Python doesn't have a `switch` statement, but dictionaries can mimic this behavior.

For Loop

```
for i in range(5):
    print(i)
# Output: 0, 1, 2, 3, 4
```

- Iterates over a sequence or range.

While Loop

```
count = 0
while count < 5:
    print(count)
    count += 1
# Output: 0, 1, 2, 3, 4
```

- Executes while the condition is true.

Break and Continue

```
for i in range(5):
    if i == 2:
        continue  # Skip 2
    print(i)
# Output: 0, 1, 3, 4

for i in range(5):
    if i == 3:
        break  # Exit loop when i is 3
    print(i)
# Output: 0, 1, 2
```

List Comprehension with Condition

```
squares = [x**2 for x in range(10) if x % 2 == 0]
print(squares)
# Output: [0, 4, 16, 36, 64]
```

- One-liner for creating filtered and transformed lists.

Try and Except for Exception Handling

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

- Catches and handles runtime errors gracefully.

Generators

Generators are a type of iterable, like lists or tuples. Unlike lists, they **generate values on the fly**, saving memory and enabling infinite sequences. They use the `yield` keyword.

Creating a Simple Generator

```
def my_generator():
    yield 1
    yield 2
    yield 3


gen = my_generator()
```

- Each `yield` pauses the function, saving its state.

Iterating Over a Generator

```
for value in gen:
    print(value)
# Output: 1, 2, 3
```

Once consumed, a generator is exhausted.

Using next()

```
gen = my_generator()
print(next(gen))  # 1
print(next(gen))  # 2
```

Generator with a Loop

```
def count_up_to(limit):
    count = 0
    while count < limit:
        yield count
        count += 1

for num in count_up_to(3):
    print(num)
# Output: 0, 1, 2
```

Using yield from (Delegating)

```
def sub_generator():
    yield from range(3)

for value in sub_generator():
    print(value)
# Output: 0, 1, 2
```

Sending Values to a Generator

```
def echo():
    value = yield
    yield value

gen = echo()
next(gen)  # Advance to first yield
print(gen.send('Hello'))  # Output: Hello
```

Generator Expression (One-liner)

```
gen_exp = (x**2 for x in range(5))
print(list(gen_exp))
# Output: [0, 1, 4, 9, 16]
```

Infinite Generator

```
def infinite_counter():
    count = 0
    while True:
        yield count
        count += 1

counter = infinite_counter()
print(next(counter))  # 0
print(next(counter))  # 1
```

Closing a Generator

```
gen = my_generator()
gen.close()
```

- Gracefully terminates the generator execution.