Unit 3
Data Types

Er. Narayan Sapkota
M.Sc. Computational Science and Engineering

Kathmandu University (Visiting Faculty)

December 12, 2024

Table of Contents

Python Data Types

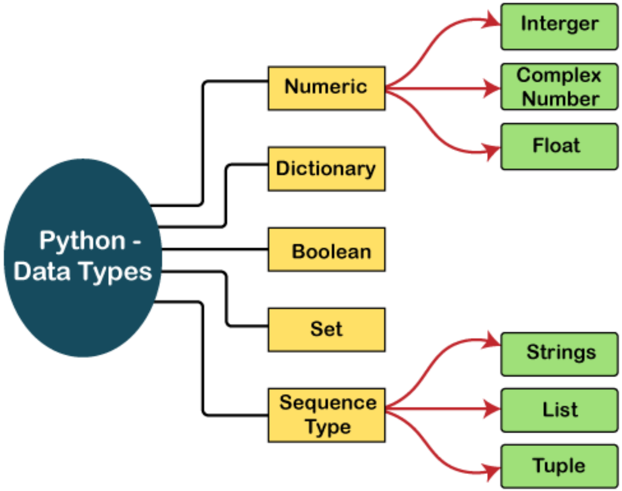

Table of Contents

Numeric Types

Python has three main numeric data types:

- Integer (`int`): Whole numbers, both positive and negative, without a decimal point. Examples: 20, 40, 60, 80
- Float (`float`): Numbers with decimal points. Examples: 20.05, 40.12, 60.39, 80.02
- Complex Number (`complex`): Numbers with a real and an imaginary part.

Integer

An `int` (integer) is a whole number without a decimal point. It can be positive or negative.

```python
x = 42  # This is an integer
print(type(x))  # Output: <class 'int'>
```

Integers are of arbitrary precision, meaning they can grow as large as the memory allows.

Float

A `float` represents a real number with a decimal point. Floats are used for precise calculations involving fractional values.

```
y = 3.14  # This is a float
print(type(y))  # Output: <class 'float'>
```

Floats can also represent scientific notation, like `1.5e2` for 150.0.

Complex Number

A `complex` number consists of two parts: the real and imaginary parts. Complex numbers are written in the form `a + bj`, where `a` is the real part and `b` is the imaginary part.

```
z = 2 + 3j  # This is a complex number
print(type(z))  # Output: <class 'complex'>
```

The real and imaginary parts can be access using the `real` and `imag` attributes:

```
print(z.real)  # Output: 2.0
print(z.imag)  # Output: 3.0
```

Table of Contents

Boolean Type (1)

- A Boolean type represents one of two possible values: True or False.
- It is often used in conditional statements and logical operations.
- Boolean values are of the bool type in Python.
- True and False are the two built-in Boolean constants. True is equivalent to 1, and False is equivalent to 0.
- Boolean Expressions: Boolean expressions evaluate to either True or False

```python
p = True
q = False
# Logical AND (and)
print("p and q:", p and q)  # False, because q is False
# Logical OR (or)
print("p or q:", p or q)  # True, because p is True
# Logical NOT (not)
print("not p:", not p)  # False, because p is True
print("not q:", not q)  # True, because q is False
```

Boolean Type (2)

Output:

```
p and q:  False
p or q:  True
not p:  False
not q:  True
```

- Truth Value Testing:
  - Any object can be tested for truth value in Python.
  - Objects considered ”false” include:
    - `None`, `False`, 0, empty sequences (e.g., `""`, `[]`, `()`)
  - All other values are considered ”true”.

Table of Contents

Sequence Types (1)

- Python has several sequence data types that allow you to store multiple items in a single variable. The main sequence data types are:
  - String: A immutable sequence of characters used to represent text data.
  - List: A mutable sequence that can store multiple items of different types.
  - Tuple: An immutable sequence, which cannot be modified after creation.
  - Range: Represents a sequence of numbers, typically used for iteration in loops.
- A mutable object is one whose state or content can be changed or modified after it is created. The object itself is updated in memory. Example: Lists, dictionaries, and sets
- An immutable object is one whose state or content cannot be changed after it is created. Any operation that appears to modify an immutable object will actually create a new object. Example: Strings, tuples, and integers

Sequence Types (2)

Table 1: Common Sequence Operations

Operation	Result
x in s	True if an item of s is equal to x, else False
x not in s	False if an item of s is equal to x, else True
s + t	The concatenation of s and t
s * n or n * s	Equivalent to adding s to itself n times
s[i]	ith item of s, origin 0
s[i:j]	Slice of s from i to j
s[i:j:k]	Slice of s from i to j with step k
len(s)	Length of s
min(s)	Smallest item of s
max(s)	Largest item of s
s.index(x[, i[, j]])	Index of the first occurrence of x in s (at or after index i and before index j)
s.count(x)	Total number of occurrences of x in s

Sequence Types (3)

Table 2: Mutable Sequence Types Operations

Operation	Result
s[i] = x	Item i of s is replaced by x
s[i:j] = t	Slice of s from i to j is replaced by the contents of the iterable t
del s[i:j]	Same as s[i:j] = []
s[i:j:k] = t	The elements of s[i:j:k] are replaced by those of t
del s[i:j:k]	Removes the elements of s[i:j:k] from the list
s.append(x)	Appends x to the end of the sequence
s.clear()	Removes all items from s (same as del s[:])
s.copy()	Creates a shallow copy of s (same as s[:])
s.extend(t) or s += t	Extends s with the contents of t
s *= n	Updates s with its contents repeated n times
s.insert(i, x)	Inserts x into s at the index given by i (same as s[i:i] = [x])
s.pop() or s.pop(i)	Retrieves the item at i and also removes it from s
s.remove(x)	Removes the first item from s where s[i] is equal to x
s.reverse()	Reverses the items of s in place

String (1)

- A string is a sequence of characters. Python treats anything inside quotes as a string. This includes letters, numbers, and symbols.
- Python has no character data type so single character is a string of length 1.

```
string = "I am String"
print(type(string))  # <class 'str'>

a = 20
print("a is", type(a))  # a is <class 'int'>

a = str(20)
print("a is", type(a))  # a is <class 'str'>
```

Strings implement all of the common sequence operations, along with the additional methods. These additional methods can be seen at python documentation or use this python code: `print(dir(str))`

String (2)

Table 3: Python String Methods and Examples

Method	Example
`capitalize()`	`"hello".capitalize()` → `"Hello"`
`upper()`	`"hello".upper()` → `"HELLO"`
`lower()`	`"HELLO".lower()` → `"hello"`
`strip()`	`" hello ".strip()` → `"hello"`
`replace()`	`"hello world".replace("world", "Python")` → `"hello Python"`
`split()`	`"apple,banana".split(",")` → `['apple', 'banana']`
`find()`	`"hello".find("e")` → `1`
`join()`	`", ".join(["apple", "banana"])` → `"apple, banana"`
`startswith()`	`"hello".startswith("he")` → `True`
`endswith()`	`"hello".endswith("lo")` → `True`
`count()`	`"hello".count("l")` → `2`
`isalpha()`	`"hello".isalpha()` → `True`
`isdigit()`	`"12345".isdigit()` → `True`
`isspace()`	`" ".isspace()` → `True`

String (3)

- **Indexing and Slicing:**
  - Access individual characters in a string using indexing: `s[i]` where `i` is the index (starting from 0).
  - Slicing allows you to extract a substring: `s[i:j]` gives the characters from index `i` to `j-1`.
  - You can also specify a step: `s[i:j:k]` for slicing with step `k`.
- **String Formatting:**
  - `f-strings` for embedding expressions: `f"Hello, name!"`.
  - Using `str.format()` for placeholders: `"Hello, !".format(name)`.
  - Old-style formatting with %: `"Hello, %s!" % name`.
- **Escape Sequences:**
  - Special characters like newline (`\n`), tab (`\t`), backslash \.
  - Example: `"Hello \n World"` will print "Hello" and then "World" on the next line.
- **Boolean:**

String (4)

- Strings can be evaluated as booleans: `bool("non-empty")` is `True`, while `bool("")` is `False`.
- Empty strings are considered `False`, while non-empty strings are considered `True`.

```python
# String Initialization
string = "Hello, Python!"

# Indexing: Accessing individual characters
char_at_index_0 = string[0]  # 'H'
char_at_index_7 = string[7]  # 'P'

# Slicing: Extracting a substring
substring = string[7:13]  # 'Python'
substring_with_step = string[0:13:2]  # 'Hlo yhn'

# String Formatting
name = "Alice"
```

String (5)

```python
greeting = f"Hello, {name}!"  # Using f-string for
    formatting

# Escape Sequences
escaped_string = "Hello\nWorld\tWelcome!"  # \n for
    newline, \t for tab

# Boolean: Checking conditions
is_python_in_string = "Python" in string  # True
is_hello_not_in_string = "Hello" not in string  # False

# Printing results
print(f"Character at index 0: {char_at_index_0}")
print(f"Character at index 7: {char_at_index_7}")
print(f"Substring (7:13): {substring}")
print(f"Substring with step (0:13:2): {
    substring_with_step}")
print(f"Greeting with formatting: {greeting}")
```

String (6)

```python
print(f"String with escape sequences: {escaped_string}")
print(f"Is 'Python' in the string? {is_python_in_string}
    ")
print(f"Is 'Hello' not in the string? {
    is_hello_not_in_string}")
```

Output:

```
Character at index 0:  H
Character at index 7:  P
Substring (7:13):  Python
Substring with step (0:13:2):  Hlo yhn
Greeting with formatting:  Hello, Alice!
String with escape sequences:  Hello
World Welcome!
Is 'Python' in the string?  True
Is 'Hello' not in the string?  False
```

List (1)

- A `list` is a mutable sequence, meaning that you can change the content of the list after it has been created. Lists can store multiple data types, including integers, floats, strings, or even other lists.

```
my_list = [1, 2, 3, 4.5, "hello"]  # This is a list
print(type(my_list))  # Output: <class 'list'>
```

- Lists are created using square brackets, and individual elements can be accessed via index:

```
print(my_list[0])  # Output: 1
print(my_list[4])  # Output: "hello"
```

- Lists can be modified, and new elements can be appended:

```
my_list.append(6)  # Adds 6 to the list
print(my_list)  # Output: [1, 2, 3, 4.5, 'hello', 6]
```

List (2)

List implement all of the common sequence operations, along with the additional methods. These additional methods can be seen at python documentation or use this python code: `print(dir(list))`

- Indexing and Slicing:
  - Access individual elements with `s[i]` where `i` is the index.
  - Extract a sublist with `s[i:j]` (from index `i` to `j-1`).
  - Use a step in slicing: `s[i:j:k]` for stepping through elements.
- Updating Items:
  - Update an item in a list with `s[i] = value`.
  - Replace a slice with new elements: `s[i:j] = new_values`.
- Adding and Removing Items:
  - Add an item with `s.append(x)` or `s.insert(i, x)`.
  - Remove an item with `s.remove(x)` or `s.pop(i)`.
  - Remove items in a range: `del s[i:j]`.
- Looping:

List (3)

- Loop through a list with a `for` loop: `for item in s:`.
- Use `enumerate(s)` to loop with index: `for i, item in enumerate(s):`

- Copying:
  - Create a shallow copy with `s.copy()` or `s[:]`.
- List Comprehension:
  - Create a new list by applying an expression to each element: `[x*2 for x in s]`.
- Sorting:
  - Sort a list with `s.sort()` (in place).
  - Get a sorted copy with `sorted(s)`.
- Joining:
  - Join elements of a list into a string with `''.join(s)` (works with strings).

List (4)

Table 4: Python List Methods and Examples

Method	Example
`append()`	`[1, 2].append(3)` $\rightarrow$ `[1, 2, 3]`
`extend()`	`[1, 2].extend([3, 4])` $\rightarrow$ `[1, 2, 3, 4]`
`insert()`	`[1, 2].insert(1, 3)` $\rightarrow$ `[1, 3, 2]`
`remove()`	`[1, 2, 3].remove(2)` $\rightarrow$ `[1, 3]`
`pop()`	`[1, 2, 3].pop()` $\rightarrow$ 3
`clear()`	`[1, 2].clear()` $\rightarrow$ `[]`
`index()`	`[1, 2, 3].index(2)` $\rightarrow$ 1
`count()`	`[1, 2, 2].count(2)` $\rightarrow$ 2
`sort()`	`[3, 1, 2].sort()` $\rightarrow$ `[1, 2, 3]`
`reverse()`	`[1, 2, 3].reverse()` $\rightarrow$ `[3, 2, 1]`
`copy()`	`[1, 2].copy()` $\rightarrow$ `[1, 2]`
`sort(reverse=True)`	`[1, 2, 3].sort(reverse=True)` $\rightarrow$ `[3, 2, 1]`
`extend()`	`[1, 2].extend([3, 4])` $\rightarrow$ `[1, 2, 3, 4]`
`reverse()`	`[1, 2, 3].reverse()` $\rightarrow$ `[3, 2, 1]`
`copy()`	`[1, 2, 3].copy()` $\rightarrow$ `[1, 2, 3]`

List (5)

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

# Indexing and Slicing
print(lst[1])        # Indexing: 2
print(lst[1:4])      # Slicing: [2, 3, 4]

# Updating Items
lst[1] = 10          # Update the second element
print(lst)           # [1, 10, 3, 4, 5]

# Adding and Removing Items
lst.append(6)        # Add to the end
lst.insert(1, 7)     # Insert at index 1
lst.remove(3)        # Remove first occurrence of 3
print(lst)           # [1, 7, 10, 4, 5, 6]

# Looping
for item in lst:
```

List (6)

```
  print(item, end=" ")  # Loop and print each item

# Copying
lst_copy = lst.copy()  # Copy list
print(lst_copy)        # [1, 7, 10, 4, 5, 6]

# List Comprehension
squared = [x**2 for x in lst]  # Square each item
print(squared)                 # [1, 49, 100, 16, 25,
    36]

# Sorting
lst.sort()                   # Sort in ascending order
print(lst)                    # [1, 4, 5, 6, 7, 10]

# Joining
joined = ', '.join(map(str, lst))  # Join list into a
    string
```

List (7)

```
print(joined)                          # "1, 4, 5, 6, 7, 10"
```

Tuple (1)

- A `tuple` is similar to a list, but it is immutable, meaning its elements cannot be changed after creation. Tuples are created using parentheses `()` instead of square brackets.

```
my_tuple = (1, 2, 3, 4.5, "hello")  # This is a tuple
print(type(my_tuple))  # Output: <class 'tuple'>
```

- Elements of a tuple can be accessed by index, similar to lists:

```
print(my_tuple[0])  # Output: 1
print(my_tuple[4])  # Output: "hello"
```

- However, since tuples are immutable, you cannot modify the contents once created:

```
# The following will raise an error:
my_tuple[0] = 10  # TypeError: 'tuple' object does not
    support item assignment
```

Tuple (2)

Tuple implement all of the common sequence operations, along with the additional methods. These additional methods can be seen using this python code: `print(dir(tuple))`

- Updating:
  - Tuples are immutable, so you cannot update or modify their elements directly.
  - To "update" a tuple, you must create a new tuple (e.g., concatenating, slicing).
- Indexing and Slicing:
  - Access individual elements using indexing: `t[i]` where `i` is the index.
  - Slice a tuple to extract a sub-tuple: `t[i:j]` (from index `i` to `j-1`).
  - You can specify a step in slicing: `t[i:j:k]`.
- Unpacking:
  - Assign tuple elements to variables using unpacking: `a, b, c = t`.
  - Can be used with both full and partial unpacking: `a, _ = t`.

Tuple (3)

- Looping:
  - Loop through a tuple with a `for` loop: `for item in t:`.
  - Use `enumerate(t)` to loop with an index: `for i, item in enumerate(t`
- Joining:
  - Tuples are immutable, but you can convert them to a string by joining their elements.
  - Use `''.join(map(str, t))` to join elements into a string (works for string elements).

Tuple (4)

Table 5: Python Tuple Methods and Examples

Method	Example
`count()`	`(1, 2, 3, 1).count(1)` → 2
`index()`	`(1, 2, 3).index(2)` → 1
`__add__()`	`(1, 2) + (3, 4)` → `(1, 2, 3, 4)`
`__contains__()`	`2 in (1, 2, 3)` → `True`
`__eq__()`	`(1, 2) == (1, 2)` → `True`
`__getitem__()`	`(1, 2, 3)[1]` → 2
`__iter__()`	`for x in (1, 2, 3):   print(x)` → prints 1, 2, 3
`__len__()`	`len((1, 2, 3))` → 3
`__ne__()`	`(1, 2) != (1, 3)` → `True`
`__mul__()`	`(1, 2) * 2` → `(1, 2, 1, 2)`
`__repr__()`	`repr((1, 2))` → `"(1, 2)"`
`__setitem__()`	`t = (1, 2, 3); t[0] = 4` → Error
`index()`	`(1, 2, 3, 1).index(1)` → 0

Tuple (5)

```
# Tuple creation
t = (1, 2, 3, 4, 5)

# Indexing and Slicing
print(t[1])          # Indexing: Output 2
print(t[1:4])        # Slicing: Output (2, 3, 4)

# Unpacking
a, b, *rest = t
print(a, b, rest)    # Output: 1 2 [3, 4, 5]

# Looping
for item in t:
  print(item, end=" ")  # Output: 1 2 3 4 5

# Joining (using str.join for tuples of strings)
str_t = ('apple', 'banana', 'cherry')
print(", ".join(str_t))  # Output: apple, banana, cherry
```

Range (1)

A `range` object represents a sequence of numbers and is commonly used in loops. It is an immutable sequence type.

```
my_range = range(5)  # Generates numbers from 0 to 4
print(list(my_range))  # Output: [0, 1, 2, 3, 4]
```

We can also specify a start, stop, and step value when creating a range:

```
my_range = range(1, 10, 2)  # Starts at 1, stops at 10, step
     size of 2
print(list(my_range))  # Output: [1, 3, 5, 7, 9]
```

A range is often used in `for` loops:

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

Table of Contents

Binary Types (1)

- Bytes:
  - Immutable sequence of bytes.
  - Created using `b"string"` syntax, e.g., `b'hello'`.
  - Useful for working with raw binary data or data in a specific encoding.
  - Can be indexed and sliced.
  - Example: `b = b"hello"`.
- Bytearray:
  - Mutable sequence of bytes.
  - Created using `bytearray()` constructor or `bytearray("string", encod`
  - Allows modification of individual elements.
  - Example: `ba = bytearray([65, 66, 67])`.
- Memoryview:
  - A view object that exposes an array's data without copying it.
  - Can be created from `bytes` or `bytearray`.
  - Allows efficient handling of large data without duplicating it in memory.
  - Example: `mv = memoryview(b"hello")`.

Binary Types (2)

- Conversion Between Types:
  - `bytes()` and `bytearray()` can be used to convert between binary types.
  - Example: `ba = bytearray(b'hello')` and `b = bytes(ba)`.
- Common Operations:
  - `len()` to get the length of a binary type.
  - `in` keyword to check membership.
  - `join()` to combine binary sequences.
  - `slice()` to extract parts of binary sequences.

Table of Contents

- Python `set` is an unordered collection of data types that is iterable, mutable, and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.
- The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.
- Accessing:
  - Sets do not support indexing or slicing since they are unordered collections.
  - To check membership, use `x in s`, which returns `True` if `x` is an element of `s`.
- Adding and Removing Items:
  - Add an item to a set with `s.add(x)`.
  - Add multiple items using `s.update(t)` where `t` is an iterable.
  - Remove an item with `s.remove(x)` (raises an error if `x` is not in `s`).
  - Remove an item safely with `s.discard(x)` (does not raise an error if `x` is not in `s`).
  - Pop an arbitrary item with `s.pop()`.

- Clear all items with `s.clear()`.

- Set Operations:
  - Union: `s | t` or `s.union(t)`.
  - Intersection: `s & t` or `s.intersection(t)`.
  - Difference: `s - t` or `s.difference(t)`.
  - Symmetric Difference: `s t̂` or `s.symmetric_difference(t)`.
- Frozenset:
  - Frozensets are immutable sets.
  - Created with `frozenset(iterable)`.
  - They support the same operations as sets but cannot be modified after creation.
- Range:
  - A range object represents an immutable sequence of numbers.
  - Can be converted to a set: `set(range(start, stop, step))`.
  - Useful for iterating over a specific range of values.

Table 6: Python Set Methods and Examples

Method	Example
add()	set1.add(4) $\rightarrow$ set1 = {1, 2, 3, 4}
clear()	set1.clear() $\rightarrow$ set1 = {}
copy()	set2 = set1.copy() $\rightarrow$ set2 = {1, 2, 3}
difference()	set1.difference(set2) $\rightarrow$ {1}
difference_update()	set1.difference_update(set2) $\rightarrow$ set1 = {1}
discard()	set1.discard(2) $\rightarrow$ set1 = {1, 3}
intersection()	set1.intersection(set2) $\rightarrow$ {3}
intersection_update()	set1.intersection_update(set2) $\rightarrow$ set1 = {3}
isdisjoint()	set1.isdisjoint(set2) $\rightarrow$ False
issubset()	set1.issubset(set2) $\rightarrow$ True
issuperset()	set1.issuperset(set2) $\rightarrow$ True
pop()	set1.pop() $\rightarrow$ set1 = {2, 3}
remove()	set1.remove(2) $\rightarrow$ set1 = {1, 3}
union()	set1.union(set2) $\rightarrow$ {1, 2, 3, 4}
update()	set1.update(set2) $\rightarrow$ set1 = {1, 2, 3, 4}

```python
# Creating a set and frozenset
s = {1, 2, 3}
frozen_s = frozenset([4, 5, 6])

# Accessing (iterating over set)
for item in s:
  print(item, end=" ")  # Output: 1 2 3

# Adding an item to the set
s.add(4)
print("\nAfter adding 4:", s)  # Output: {1, 2, 3, 4}

# Removing an item from the set
s.remove(2)
print("After removing 2:", s)  # Output: {1, 3, 4}

# Set Operations (Union, Intersection, Difference)
s2 = {3, 4, 5}
print("Union:", s.union(s2))  # Output: {1, 3, 4, 5}
print("Intersection:", s.intersection(s2))  # Output:
    {3, 4}
```

```
print("Difference:", s.difference(s2))  # Output: {1}

# Frozenset example (immutable)
print("Frozenset:", frozen_s)  # Output: frozenset({4,
    5, 6})

# Range example
r = range(1, 5)
print("Range:", list(r))  # Output: [1, 2, 3, 4]
```

Table of Contents

Dictionary (1)

- A Python dictionary is a data structure that stores the value in key: value pairs.
- Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be immutable.
- Creation:
  - Create a dictionary using curly braces: `d = {key1: value1, key2: value2}`.
  - Another way to create a dictionary: `d = dict(key1=value1, key2=valu`
  - An empty dictionary: `d = {}`.
- Accessing:
  - Access a value by key: `value = d[key]`.
  - Use the `get` method to avoid errors: `value = d.get(key)`.
  - Get all keys with `d.keys()` and all values with `d.values()`.
  - Check if a key exists with `key in d`.
- Conditionals:

Dictionary (2)

- Test if a key is in the dictionary: `if key in d:`.
- Test if a key is not in the dictionary: `if key not in d:`.
- We can perform conditional updates: `d[key] = new_value if conditio` `else old_value`.

- Looping:
    - Loop through dictionary keys: `for key in d:`.
    - Loop through dictionary values: `for value in d.values():`.
    - Loop through both keys and values: `for key, value in d.items():`.

Dictionary (3)

Table 7: Python Dictionary Methods and Examples

Method	Example
`clear()`	`d.clear()` → {}
`copy()`	`d.copy()` → {'a': 1, 'b': 2}
`get()`	`d.get('a')` → 1
`items()`	`d.items()` → {('a', 1), ('b', 2)}
`keys()`	`d.keys()` → {'a', 'b'}
`pop()`	`d.pop('a')` → 1
`popitem()`	`d.popitem()` → ('b', 2)
`setdefault()`	`d.setdefault('c', 3)` → 3
`update()`	`d.update('c': 3)` → {'a': 1, 'b': 2, 'c': 3}
`values()`	`d.values()` → {1, 2}
`fromkeys()`	`dict.fromkeys(['a', 'b'], 0)` → {'a': 0, 'b': 0}
`del`	`del d['a']` → {'b': 2}
`pop`	`d.pop('b')` → 2
`clear()`	`d.clear()` → {}

Dictionary (4)

```
# Dictionary Creation
d = {'a': 1, 'b': 2, 'c': 3}

# Accessing a value
print(d['a'])  # Output: 1

# Conditionals
if 'b' in d:
  print("Key 'b' exists")  # Output: Key 'b' exists

# Looping through the dictionary
for key, value in d.items():
  print(f"{key}: {value}")
# Output:
# a: 1
# b: 2
# c: 3
```

Table of Contents

- `None` is a special constant in Python that represents the absence of a value or a null value.
- It is often used as a placeholder or default value.
- `None` is a singleton, meaning there is only one instance of `None` in a Python program.
- Usage of None
  - Default return value of a function that does not explicitly return a value.
  - Used to represent the absence of a value in variables or parameters.
  - Often used in comparisons and conditionals to check for missing data.
- Comparison with None
  - To check if a value is `None`, use `is` (not ==).

    ```
    if x is None:
    ```

  - `None` is not equal to any other value, including `False`, `0`, or an empty string `""`.
- Examples:

```python
def fun():
  return None

x = fun() # x is assigned None
```

- None in Conditionals:

```python
if not x:
  print("x is None")
```