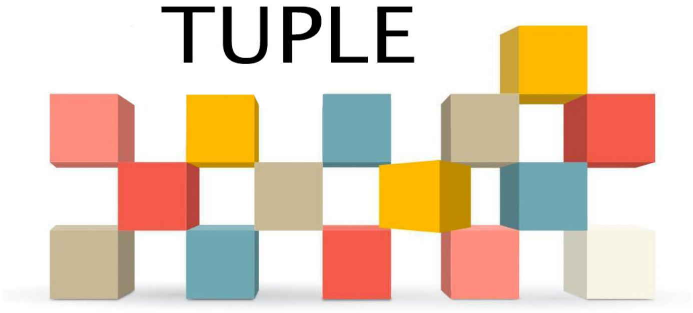
COMP 1023 Introduction to Python Programming

Collections - Container Data Types (Part II)

Dr. Cecia Chan, Prof. SC Cheung, Dr. Alex Lam, Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Tuples



Introduction

- Tuples are similar to lists, but their elements are fixed. Once a tuple is created, you cannot add, delete, replace, or reorder the elements.
- If the contents of a list in your application shouldn't change, you can use a tuple to prevent accidental modifications.
- Tuples are generally more efficient than lists due to Python's internal optimizations.
- You create a tuple by enclosing its elements in parentheses (i.e., `()`), with the elements separated by commas.

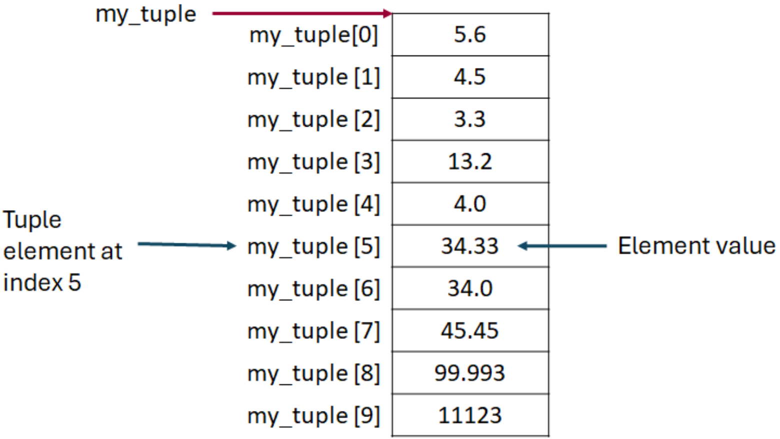
Tuple Basics

- To create a tuple, you can use the following syntax:

```
tuple1 = ()                                  # Create an empty tuple
tuple2 = (1, 3, 5)                           # Create a tuple with elements 1, 3, 5
tuple3 = ("red", "green", "blue")            # Create a tuple with strings
tuple4 = tuple([1, 2, 3])                    # Create a tuple from a list
tuple5 = tuple([2 * x for x in range(1, 5)]) # Create a tuple from a list comprehension
tuple6 = tuple("abcd")                       # Create a tuple from a string
                                             # tuple6 is ('a', 'b', 'c', 'd')
tuple7 = (1, "two", 3)                       # Create a tuple with mixed types elements
```

Tuple Demonstration

```
my_tuple = (5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123)
```



my_tuple

my_tuple[0]	5.6
my_tuple [1]	4.5
my_tuple [2]	3.3
my_tuple [3]	13.2
my_tuple [4]	4.0
my_tuple [5]	34.33
my_tuple [6]	34.0
my_tuple [7]	45.45
my_tuple [8]	99.993
my_tuple [9]	11123

Tuple element at index 5

Element value

- The tuple `my_tuple` has 10 elements with indexes ranging from 0 to 9.

Common Error

- Accessing a tuple out of bounds is a common programming error that results in a runtime 'IndexError'.
- To avoid this error, ensure that you do not use an index beyond len(my_tuple) - 1.
- Here is an example of an out-of-bounds error:

```python
# Filename: tuple_out_of_bounds_error.py

def main():
    my_tuple = (5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123)
    i = 0
    while i <= len(my_tuple):  # This condition causes the error
       print(my_tuple[i])
       i += 1

if __name__ == "__main__":
    main()
```

How can we fix it?

Functions for Tuples

```python
def main():  # Filename: functions_for_tuples.py
    my_tuple1 = (1, 2, 3, 4, 5)
    my_tuple2 = tuple([4, 5, 6, 7, 8])

    print("4 in my_tuple1:", 4 in my_tuple1)
    print("4 not in my_tuple1:", 4 not in my_tuple1)
    print("my_tuple1 + my_tuple2:\n",
           my_tuple1 + my_tuple2)
    print("2 * my_tuple1:", 2 * my_tuple1)
    print("my_tuple1[3]:", my_tuple1[3])
    print("my_tuple1[3:5]:", my_tuple1[3:5])
    print("my_tuple1[-1]:", my_tuple1[-1])
    print("len(my_tuple1):", len(my_tuple1))
    print("min(my_tuple1):", min(my_tuple1))
    print("max(my_tuple1):", max(my_tuple1))
    print("sum(my_tuple1):", sum(my_tuple1))

    for i in my_tuple1:
        print(i, end=" ")
    print()

    print("my_tuple1 < my_tuple2:", my_tuple1 < my_tuple2)

    del my_tuple1  # Delete the whole tuple, so my_tuple1 no longer exists

if __name__ == "__main__": main()
```

Output:

```
4 in my_tuple1: True
4 not in my_tuple1: False
my_tuple1 + my_tuple2:
 (1, 2, 3, 4, 5, 4, 5, 6, 7, 8)
2 * my_tuple1: (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
my_tuple1[3]: 4
my_tuple1[3:5]: (4, 5)
my_tuple1[-1]: 5
len(my_tuple1): 5
min(my_tuple1): 1
max(my_tuple1): 5
sum(my_tuple1): 15
1 2 3 4 5
my_tuple1 < my_tuple2: True
```

Comparison Operators for Tuples

- Equality (==):
  - Checks if two tuples have the same elements in the same order.

```
(1, 2, 3) == (1, 2, 3)  # True
(1, 2, 3) == (3, 2, 1)  # False
```

- Inequality (!=):
  - Checks if two tuples are not equal.

```
(1, 2) != (1, 2, 3)     # True
```

- Less Than (<) and Greater Than (>):
  - Compares tuples lexicographically (like dictionary order).
  - Compares element by element until a difference is found.

```
(1, 2, 3) < (1, 2, 4)   # True
(1, 2) < (1, 2, 0)      # True
```

- Less Than or Equal To (<=) and Greater Than or Equal To (>=):
  - Similar to < and >, but include equality.

```
(1, 2, 3) <= (1, 2, 3)  # True
(1, 2) >= (1, 2, 0)     # False
```

Index Operator [ ]

- An element in a tuple can be accessed using the index operator, with the following syntax:

```
my_tuple[index]
```

- Tuple indexes are 0-based, meaning they range from 0 to len(my_tuple) - 1.
- my_tuple[index] can be used like a variable, so it is also referred to as an indexed variable.
- For example, the following code prints the value in my_tuple[1]:

```
print(my_tuple[1])
```

- The following loop prints 0 to my_tuple[0], 1 to my_tuple[1], ..., 9 to my_tuple[9]:

```
for i in range(len(my_tuple)):
    print(my_tuple[i])
```

The following is an error!

```
my_tuple[1] = 10  # Error!
```

Since tuples in Python are immutable, meaning their elements cannot be directly changed, added, or removed after the tuple is created.

Elements in a Tuple May Be Mutable

```python
# Filename: tuple_element_mutable.py

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def setRadius(self, radius):
        self.radius = radius

    def getRadius(self):
        return self.radius

def main():
    circles = (Circle(2), Circle(4), Circle(7))
    circles[0].setRadius(30)
    print(circles[0].getRadius())  # Print 30

if __name__ == "__main__":
    main()
```

- Each element in the tuple is a `Circle` object. While you cannot add, delete, or replace circle objects in the tuple, you can change a circle's radius since a circle object is mutable.
- Tuple elements are immutable, but they can contain mutable objects, such as lists.

More details about objects will be discussed later!

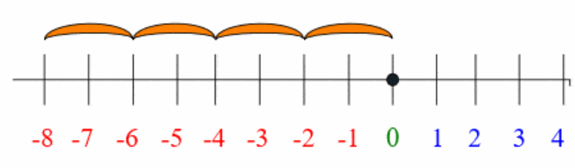
Negative Numbers as Indexes

- Python allows the use of negative numbers as indexes to reference positions relative to the end of the tuple.
- The actual position is obtained by adding the length of the tuple to the negative index.
- For example:

```
my_tuple = (2, 3, 5, 2, 33, 21)
print(my_tuple[-1])  # Print 21
print(my_tuple[-3])  # Print 2
```

  - `my_tuple[-1]` is the same as `my_tuple[-1 + len(my_tuple)]` (i.e., `my_tuple[-1 + 6]`).
  - `my_tuple[-3]` is the same as `my_tuple[-3 + len(my_tuple)]` (i.e., `my_tuple[-3 + 6]`).

Exactly the same as lists!!!



Tuple Slicing

- The slicing operator returns a slice of the tuple using the syntax:

  my_tuple[start : end : step]

- The slice is a sub-tuple from index start to index end - 1 with the specified step.
  - By default, step is 1.

```
my_tuple = (2, 3, 5, 7, 9, 1)
print(my_tuple[2 : 4])      # Print (5, 7)
print(my_tuple[0 : 5 : 2]) # Print (2, 5, 9)
```

Tuple Slicing

- You can use a negative index in slicing.

```
my_tuple = (2, 3, 5, 7, 9, 1)
print(my_tuple[1 : -3])   # Print (3, 5)
print(my_tuple[-4 : -2])  # Print (5, 7)
```

  - `my_tuple[1 : -3]` is the same as `my_tuple[1 : -3 + len(my_tuple)]`.
  - `my_tuple[-4 : -2]` is the same as `my_tuple[-4 + len(my_tuple) : -2 + len(my_tuple)]`.

- You cannot assign values to a slice of a tuple.

```
my_tuple = (2, 3, 5, 7, 9, 1)
my_tuple[1 : 3] = (91, 92, 93, 94) # Error!
```

Exactly the same as lists, except slices cannot be assigned new values!!!

Tuple Slicing: Default Values and Edge Cases

- The starting index or ending index may be omitted. Then, default values will be used.
    - Positive step (i.e., step > 0):
        - If you omit the start index: Default is 0 (start from the beginning).
        - If you omit the end index: Default is the length of the tuple (i.e., len(my_tuple)).
        - If start index $\geq$ end index, the result will be an empty tuple.
    - Negative step (i.e., step < 0):
        - If you omit the start index: Default is the last index (i.e., len(my_tuple)-1).
        - If you omit the end index: Default is None (will go until the start of the tuple).
        - If end index $\leq$ start index, the result will be an empty tuple.
    - If you omit the step: Default is 1.
- If `start` or `end` specifies a position beyond the end of the tuple, Python will use the length of the tuple for `start` or `end` instead.

Exactly the same as lists!!!

Tuple Slicing Examples

- Positive steps (i.e., step $>$ 0):

```
my_tuple = (2, 3, 5, 7, 9, 1)
print(my_tuple[  : 2 : 1])  # Equivalent to print(my_tuple[0 : 2 : 1]), Print (2, 3)
print(my_tuple[3 :   : 1])  # Equivalent to print(my_tuple[3 : 6 : 1]), Print (7, 9, 1)
print(my_tuple[3 : 1 : 1])  # Empty tuple
```

- Negative steps (i.e., step $<$ 0):

```
my_tuple = (2, 3, 5, 7, 9, 1)
print(my_tuple[  : 2 : -1])  # Equivalent to print(my_tuple[5 : 2 : -1]), Print (1, 9, 7)
print(my_tuple[3 :   : -1])  # Equivalent to print(my_tuple[3 : None : -1]), Print (7, 5, 3, 2)
print(my_tuple[1 : 3 : -1])  # Empty tuple
```

- start or end specifies a position beyond the end of the tuple:

```
my_tuple = (2, 3, 5, 7, 9, 1)
print(my_tuple[3 : 8]) # Equivalent to print(my_tuple[3 : 6]), Print (7, 9, 1)
print(my_tuple[7 : 5]) # Equivalent to print(my_tuple[6 : 5]), Print ()
print(my_tuple[7 : 8]) # Equivalent to print(my_tuple[6 : 6]), Print ()
```

Slicing handles out-of-range indices gracefully!

Exactly the same as lists!!!

Traversing Elements in a Tuple

- The elements in a Python tuple are iterable.
- Python supports a convenient `for` loop, which enables you to traverse the tuple sequentially without using an index variable.
- For example, the following code displays all the elements in the tuple `my_tuple`:

```
my_tuple = (5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123)
for u in my_tuple:
    print(u, end=' ')
# Print 5.6 4.5 3.3 13.2 4.0 34.33 34.0 45.45 99.993 11123
```

- You still have to use an index variable if you wish to traverse the tuple in a different order. For example, the following code displays the elements at even-numbered indices:

```
my_tuple = (5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123)
for i in range(0, len(my_tuple), 2):
    print(my_tuple[i], end=' ')
# Print 5.6 3.3 4.0 34.0 99.993
```

Exactly the same as lists!!!

No Python Tuple Comprehension

- Python does not support tuple comprehensions directly.
- Instead, you can use:
  - tuple() function with a generator expression.

```
my_tuple = tuple(x for x in range(5))
```

Tuple Methods

Method	Description
`count(element)`: value	Returns the number of times the given element appears in the tuple.
`index(element, start, end)`: value	Returns the first occurrence of the given element from the tuple starting from `start` and stopping at `end`.

```python
my_tuple1 = (0, 1, 2, 3, 2, 3, 1, 2, 3)
my_tuple2 = ("COMP", "1023", "is", "the", "best", "COMP", "course")

c1 = my_tuple1.count(3)       # Count the number of times 3 appears in the tuple
print(c1)    # Print 3
c2 = my_tuple2.count("COMP")  # Count the number of times "COMP" appears in the tuple
print(c2)    # Print 2

pos1 = my_tuple1.index(3)     # Find the first occurrence of 3
print(pos1)  # Print 3
pos2 = my_tuple1.index(3, 4)  # Find the first occurrence of 3 starting at index 4
print(pos2)  # Print 5
# pos3 = my_tuple1.index(4)   # Error: 4 is not in the tuple
```

Splitting a String into a Tuple

- To split the characters in a string `s` into a tuple, use `tuple(s)`:

```
my_tuple = tuple("abc")
print(my_tuple)  # Print ('a', 'b', 'c')
```

- The `str` class contains the `split` method, which is useful for splitting items in a string into a list and then explicitly converting it to a tuple:

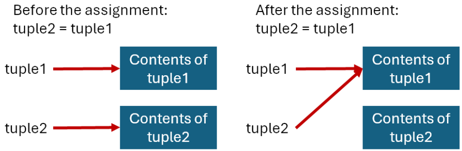
```
items1 = tuple("COMP1023 is the best COMP course".split())  # Delimited by spaces
print(items1)     # Print ('COMP1023', 'is', 'the', 'best', 'COMP', 'course')

items2 = tuple("12/25/2025".split("/"))  # Delimited by /
print(items2)     # Print ('12', '25', '2025')
```

Copying Tuples

Question

Does `tuple2 = tuple1` duplicate a tuple?

- The above statement does not copy the contents of the tuple referenced by `tuple1` to `tuple2`.
- It copies the reference from `tuple1` to `tuple2`.
- After this statement, `tuple1` and `tuple2` refer to the same tuple.


Before the assignment:
tuple2 = tuple1

After the assignment:
tuple2 = tuple1

```
tuple1 = (1, 2)
tuple2 = (3, 4, 5)
print(id(tuple1))  # Example ID: 132576440962624
print(id(tuple2))  # Example ID: 132575771145280
tuple2 = tuple1
print(id(tuple2))  # Example ID: 132576440962624
```

- The tuple previously referenced by `tuple2` is no longer referenced. The memory space occupied by `tuple2` will be automatically collected and reused by the Python interpreter.

How to duplicate a tuple?

Copying Tuples

- In Python, tuples are immutable, so you typically don't need to create a deep copy because their contents can't be changed. However, if your tuple contains mutable objects (e.g., lists), and you want to create a new tuple with deep copies of those objects, you can use the copy.deepcopy() function from the copy module.

```python
# Filename: copy_tuples.py

import copy

tuple1 = (1, 2, [3, 4])     # Contains a mutable object (list)
tuple2 = copy.deepcopy(tuple1)

# Modify the mutable object in the original to verify the deep copy
tuple1[2].append(99)

print(id(tuple1), tuple1)  # Print 132575769680704 (1, 2, [3, 4, 99])
print(id(tuple2), tuple2)  # Print 132575769135168 (1, 2, [3, 4])
```

Two-Dimensional Tuples

- A two-dimensional tuple is a tuple that consists of rows.
- Each row is a tuple that contains the values.
- The rows can be accessed using an index, called a row index.
- The values in each row can be accessed through another index, called a column index.

```
matrix = (
    (1, 2, 3, 4, 5),
    (6, 7, 0, 0, 0),
    (0, 1, 0, 0, 0),
    (1, 0, 0, 0, 8),
    (0, 0, 9, 0, 3)
)
```

	[0]	[1]	[2]	[3]	[4]
[0]	1	2	3	4	5
[1]	6	7	0	0	0
[2]	0	1	0	0	0
[3]	1	0	0	0	8
[4]	0	0	9	0	3

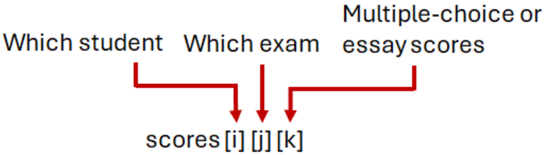
```
matrix[0] is (1, 2, 3, 4, 5)
matrix[1] is (6, 7, 0, 0, 0)
matrix[2] is (0, 1, 0, 0, 0)
matrix[3] is (1, 0, 0, 0, 8)
matrix[4] is (0, 0, 9, 0, 3)

matrix[0][0] is 1
matrix[4][4] is 3
```

- Each value can be accessed using `matrix[i][j]`, where `i` and `j` are the row and column indexes.

Multidimensional Tuples

- Occasionally, you need to represent n-dimensional data, for any integer *n*.
- For example, you can use a three-dimensional tuple to store exam scores for a class of 6 students with 5 exams, where each exam has 2 parts (multiple-choice and essay):

```
scores = ( ((11.5, 20.5), (11.0, 22.5), (15, 33.5), (13, 21.5), (15, 2.5)),
           ((4.5, 21.5),  (11.0, 22.5), (15, 34.5), (12, 20.5), (14, 11.5)),
           ((6.5, 30.5),  (11.4, 11.5), (11, 33.5), (11, 23.5), (10, 2.5)),
           ((6.5, 23.5),  (11.4, 32.5), (13, 34.5), (11, 20.5), (16, 11.5)),
           ((8.5, 26.5),  (11.4, 52.5), (13, 36.5), (13, 24.5), (16, 2.5)),
           ((11.5, 20.5), (11.4, 42.5), (13, 31.5), (12, 20.5), (16, 6.5)) )
```



`scores[0][1][0]` refers to the multiple-choice score for the first student's second exam.

Automatic Packing and Unpacking

- You can create a tuple from comma-separated values.
- This is called automatic packing of a tuple:

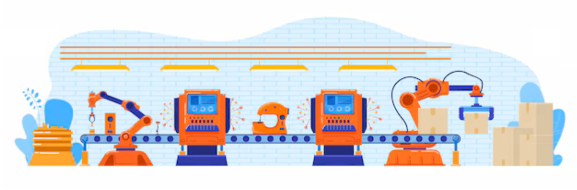
```
t = 4, 5, 1
```

```
return v1, v2
```

- This actually returns a tuple with values v1 and v2.

```
v1, v2 = range(2, 4)
```

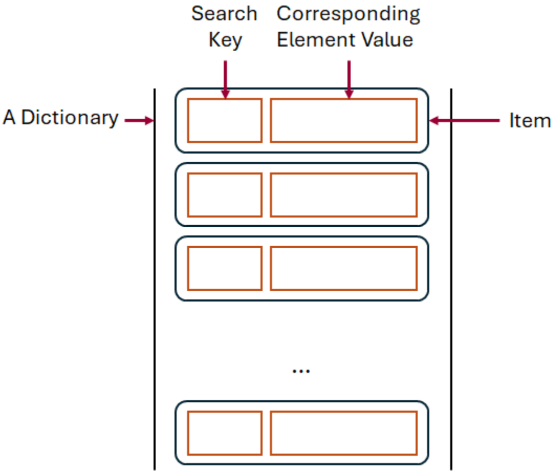
- This unpacks a sequence. The above statement assigns 2 and 3 to v1 and v2.



Dictionaries

Introduction

- A dictionary is a container object that stores a collection of key/value pairs.
- It enables fast retrieval, deletion, and updating of values using keys.
- A dictionary cannot contain duplicate keys; each key maps to one value, and its corresponding value forms an item (or entry) stored in the dictionary.
- The data structure is called a "dictionary" because it resembles a word dictionary, where the words are the keys and the definitions are the values.
- A dictionary is also known as a map, which maps each key to a value.



Dictionary Basics

```
my_dict1 = {}                                # Create an empty dictionary
my_dict2 = dict()                            # Create an empty dictionary

my_dict3 = { "21053124": "Tammy",            # Create a dictionary with two items
             "21543257": "Elvis" }           # The item is in the form key:value.
# The key in the first item is 21053124, and its corresponding value is Tammy.
# Note: The key must be of a hashable type such as numbers and strings.
#       The value can be of any type.

my_dict4 = dict(name="Tammy", id="Elvis")    # Create a dictionary with two items
# The key in the first item is name, and its corresponding value is Tammy.

my_dict5 = dict([ ("21053124", "Tammy"),     # Create a dictionary using a list of tuples
                  ("21543257", "Elvis") ])
my_dict6 = { x: x ** 2 for x in range(5) }   # Create a dictionary using
                                             # dictionary comprehension
```

Note

Keys of dictionary are not limited to strings.

Functions for Dictionaries

```python
def main():  # Filename: functions_for_dictionary.py
    students1 = { "21053124": "Tammy", "21543257": "Elvis" }
    students2 = { "22356267": "Peter", "25141321": "John" }

    print("21053124 in students1:", "21053124" in students1)
    print("21053124 not in students1:",
          "21053124" not in students1)
    # Error
    # print("students1 + students2:\n", students1 + students2)
    # print("2 * students1:", 2 * students1) # Error
    print("len(students1):", len(students1))
    print("min student ID:", min(students1))
    print("max student ID:", max(students1))
    for key in students1:
        print(key + ": " + str(students1[key]))

    # Error
    # print("students1 < students2:", students1 < students2)
    print("students1 == students2:", students1 == students2)
    print("students1 != students2:", students1 != students2)

if __name__ == "__main__":
   main()
```

Output:

```
21053124 in students1: True
21053124 not in students1: False
len(students1): 2
min student ID: 21053124
max student ID: 21543257
21053124: Tammy
21543257: Elvis
students1 == students2: False
students1 != students2: True
```

Adding, Modifying, and Retrieving Values

- To add an item to a dictionary, use the syntax:

  ```
  dictionaryName[key] = value
  ```

  If the key is already in the dictionary, this statement replaces the value for that key.
- To retrieve a value, simply write an expression using:

  ```
  dictionaryName[key]
  ```

  If the key is in the dictionary, the value for that key is returned. Otherwise, an error occurs.
- To delete an item from a dictionary, use the syntax:

  ```
  del dictionaryName[key]
  ```

  This statement deletes the item with the specified key from the dictionary. If the key is not in the dictionary, an error occurs.

Example

```
def main():
    students = { "21053124": "Tammy", "21543257": "Elvis" }
    students["27272312"] = "Desmond"     # Add a new item
    print(students["27272312"])          # Print Desmond
    students["21053124"] = "Tammy Wong"  # Replace the value for the key "21053124"
    print(students["21053124"])          # Print Tammy Wong
    del students["27272312"]             # Delete the item with the key "27272312"
    # print(students["22222222"])        # Uncommenting this will raise a KeyError

if __name__ == "__main__":
   main()
```

No Subscript Indices and Slicing for Dictionaries

- You cannot use subscript indices (e.g., [0], [1], etc.) to access dictionary elements in Python by default because dictionaries are meant to be accessed by keys, not positions.
- Also, dictionaries do not support slicing (e.g., dict[1:3]).

Traversing Elements in a Dictionary

```
def main():
    students = { "21053124": "Tammy", "21543257": "Elvis" }

    # Accessing Values by Key during Iteration
    for key in students:
        print(key + ": " + str(students[key]))

    # Iterating through Keys
    for key in students.keys():
        print(key)

    # Iterating through Values
    for value in students.values():
        print(value)

    # Iterating through Key-Value Pairs
    for key, value in students.items():
        print(str(key) + ": " + str(value))

if __name__ == "__main__":
    main()
```

Dictionary Comprehensions

- Dictionary comprehension is a concise syntax that creates a dictionary by processing another sequence of data.
- A dictionary comprehension consists of {} containing an expression followed by a for clause and then zero or more for or if clauses.
- The dictionary comprehension produces a dictionary with the results from evaluating the expression.

```
dict1 = {x: x**2 for x in range(5)}
print(dict1)  # Print {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

dict2 = {key: value for key, value in dict1.items() if key > 2}
print(dict2)  # Print {3: 9, 4: 16}

dict3 = {key: value for key, value in dict2.items() if value > 10}
print(dict3)  # Print {4: 16}
```

The Dictionary Methods

Method	Description
clear(): None	Removes all the items from the dictionary.
get(key, default)	Returns the value for the specified key. If the key is not found, it returns default (defaults to None if not specified).
items(): tuple	Returns a view object containing a list of key-value pairs as tuples.
keys(): tuple	Returns a view object containing a list of all keys in the dictionary.
pop(key, default): value	Removes the item with the specified key and returns its value. If the key is not found and default is provided, it returns default; otherwise, it raises a KeyError.
popitem(): tuple	Removes and returns the last inserted key-value pair as a tuple.
update(other_dict): None	Updates the dictionary with key-value pairs from other_dict. If a key from other_dict already exists in the original dictionary, its value is updated. Otherwise, the new key-value pair is added.
values(): tuple	Returns a view object containing a list of values in the dictionary.

Examples

```
students = { "21053124":"Tammy", "21543257":"Elvis" }
print(tuple(students.keys()))
print(tuple(students.values()))
print(students.get("21053124"))
print(students.get("22222222"))
print(students.pop("21053124"))
print(students)
print(students.items())
students.clear()
print(students)
```

Output:

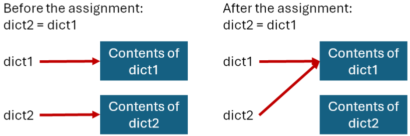
```
('21053124', '21543257')
('Tammy', 'Elvis')
Tammy
None
Tammy
{'21543257': 'Elvis'}
dict_items([('21543257', 'Elvis')])
{}
```

Copying Dictionaries

Question

Does `dict2 = dict1` duplicate a dictionary?

- The above statement does not copy the contents of the dictionary referenced by dict1 to dict2.
- It copies the reference from dict1 to dict2.
- After this statement, dict1 and dict2 refer to the same dictionary.



Before the assignment:
dict2 = dict1

dict1

Contents of dict1

dict2

Contents of dict2

After the assignment:
dict2 = dict1

dict1

Contents of dict1

dict2

Contents of dict2

```
dict1 = { "21053124": "Tammy", "21543257": "Elvis" }
dict2 = { "22312315": "John", "2251525": "Peter" }
print(id(dict1))  # Print 132575574499392
print(id(dict2))  # Print 132575574492736
dict2 = dict1
print(id(dict2))  # Print 132575574499392
```

- The dictionary previously referenced by dict2 is no longer referenced. The memory space occupied by that dictionary will be automatically collected and reused by the Python interpreter.

How to duplicate a dictionary?

Copying Dictionaries

- Shallow Copy
  - Dictionary comprehension

```
dict1 = { "21053124": "Tammy", "21543257": "Elvis" }
dict2 = { key: value for key, value in dict1.items() }
```

  - Dictionary constructor

```
dict1 = { "21053124": "Tammy", "21543257": "Elvis" }
dict2 = dict(dict1)
```

- Deep Copy
  - Using `copy.deepcopy()`

```
import copy
dict1 = { "21053124": "Tammy", "21543257": "Elvis" }
dict2 = copy.deepcopy(dict1)
```

Shallow copies share references to nested objects, while deep copies create independent copies.

Note

- Tuples can be used as keys in dictionaries if all elements are immutable, and as elements of sets (a self-study topic), while lists cannot.
- Here is an example:

```
my_dict1 = {(x, x+1): x for x in range(10)}
my_tuple = (5,6)
print(my_dict1[my_tuple])  # Print 5
print(my_dict1[(1,2)])     # Print 1

# my_dict2 = { [x, x+1]: x for x in range(10) } # Error
# my_set = { [1,2,3], [4] } # Error
```

When to Use Each Container Data Type?

- Lists
  - Use when you need an ordered sequence of items that can be modified (add, remove, change).
  - Suitable for scenarios where the order of elements matters.
- Tuples
  - Use when you need an ordered sequence of items that should **not** be modified after creation.
  - Tuples are immutable and are often used to represent fixed collections of items.
  - Tuples can be used as keys in dictionaries when you need to associate multiple values with a single key.
  - Iterating through a tuple is faster than iterating through a list.
- Dictionaries
  - Use when you need to store key-value pairs and perform fast lookups based on keys.
  - Dictionaries are ideal for representing mappings between items.
  - Useful for associating data with specific identifiers or labels.

Key Terms

- Dictionary
- Dictionary comprehension
- Immutable tuple
- Key/Value pair
- Tuple

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. You can use a ________________ loop to traverse all elements in a ________________, ________________, ________________, and ________________.
2. A ________________ is an immutable list. You cannot add, delete, or replace elements in a ________________.

Answer: 1. for; list; tuple; set; dictionary, 2. tuple; tuple.

Further Reading

- Read Chapters 7 & 14 of the textbook “Introduction to Python Programming and Data Structures”.

That's all!

Any question?