Lecture3

September 6, 2024

1 Lecture 3: Iterable Objects or Containers

1.0.1 Objective:

This lecture aims to present `iterable objects` and `containers`, focusing on `lists`, `tuples`, `dictionaries`, and `sets`. We will review some of the most commonly encountered operations with these data structures, particularly in data organization and retrieval tasks. Additionally, we will explore how to use `lists`, `tuples`, `dictionaries`, and `sets` to efficiently store and manage collections of data. ****

As mentioned in previous sections, a Python program always operates based on the manipulation of objects. The instructions (whether defined in a free code, within an if clause, or in a while or for loop) are always executed on objects. Variables, as previously discussed, represent a particular type of Python object. In general, instructions in a Python program are defined from a collection of objects that come in various forms of value sequences. This sequence of values can, for example, consist of a set of variables with a single value. This ultimately means that a variable defined by a single value is generally not sufficient to form the architecture of a program. In the Python language, we have:

- A **list** object is a sequence of values (numerical and/or characters) that are **indexed** and specified within **square brackets**, separated by commas. Example: "'python x = [1, 5, 8, 10, 4, 50, 8] # List consisting only of numbers y = ["Olivier", "ENGEL", "Strasbourg"] # List consisting only of characters z = [1, "Olivier", 5, 8, "ENGEL", 10, 4, 50, "Strasbourg"] # List consisting of both numbers and characters.

  ~~~
  A list object can be generated manually by specifying the values within square brackets or
  ```python
    x = list(range(1, 10))
    print(x) # returns [1, 2, 3, 4, 5, 6, 7, 8, 9]
  ~~~

- A **tuple** object is a sequence of values (numerical and/or characters) that are **indexed** and specified within **parentheses**, separated by commas. Example: "'python x = (1, 5, 8, 10, 4, 50, 8) # Tuple consisting only of numbers y = ("Olivier", "ENGEL", "Strasbourg") # Tuple consisting only of characters z = (1, "Olivier", 5, 8, "ENGEL", 10, 4, 50, "Strasbourg") # Tuple consisting of both numbers and characters.

  ~~~
  A tuple object can be generated manually by specifying the values within parentheses or au
  ```python
    x = tuple(range(1, 10))
    print(x) # returns (1, 2, 3, 4, 5, 6, 7, 8, 9)
  ~~~

- A **set** object is a sequence of values (numerical and/or characters) that are **non-duplicated** and **non-indexed**, and specified within **curly braces**, separated by commas. Example: “‘python x = {1, 5, 8, 10, 4, 50, 8} # Set consisting only of numbers y = {“Olivier”, “ENGEL”, “Strasbourg”} # Set consisting only of characters z = {1, “Olivier”, 5, 8, “ENGEL”, 10, 4, 50, “Strasbourg”} # Set consisting of both numbers and characters.

  ~~~
  A set object can be generated manually by specifying the values within curly braces or aut
  ```python
     v = [2, 4, "orange", "meat", 4, "orange"]
     V = set(v)
     print(V) # returns {2, 'orange', 4, 'meat'}
  ~~~

- A **dictionary** object is a sequence of values (numerical and/or characters) that are **indexed** by keys and specified within **curly braces**, separated by commas. Each `key` corresponds to one or more `values`. A dictionary consists of a set of `key-value` pairs. Example:
  “‘python x = {‘name’: ‘Jean’, ‘weight’: 70, ‘height’: 1.75} # Keys with unique values. y = {‘Jean’: [25, 70, 1.75], ‘Paul’: [30, 65, 1.80], ‘Pierre’: [35, 75, 1.65]} # Key with multiple values formed by a list

  `“In the dictionaryx, the keys are name, weight, and height. The values corresponding to each of these keys are Jean, 70, and 1.75, respectively. In the dictionaryy, the keys are Jean, Paul, and Pierre, while the values are lists consisting of three elements (age, weight, and height). A dictionary object can be generated manually by specifying the values within curly braces or automatically using thedict()‘` function to create an empty dictionary. We will delve more into the definition of dictionaries later.

The four objects we have just mentioned are also what we call `containers` simply because they `can contain` something. Thus, strings, lists, tuples, and dictionaries are the basic iterable objects in Python. Lists and dictionaries are mutable – their elements can be changed on the fly – while strings and tuples are immutable.

The following three objects we mention are not iterable, but we will study them in more detail later on.

- A **function** object is a program designed to perform a specific operation. For example, the `print()` function is used to display the result of the instruction provided to it on the screen;
- A **class** object is a collection of functions, that is, an association of related functions;
- A **module** object is a collection of classes, that is, a collection of related classes.”

1.1 0. Lists

1.1.1 0.0. Definition of a list object

A list object can be declared and defined manually or by using the `list()` function. There are two categories of lists: one-dimensional lists (or simple lists) and multi-dimensional lists.

- A simple list is a list whose elements consist of unique values separated by commas.

[36]:
```
voltage = [-2.0, -1.0, 0.0, 1.0, 2.0]

courant = [-1.0, -0.5, 0.0, 0.5, 1.0]
```

```
print(voltage)
print("============================= \n")
print(courant)
```

```
[-2.0, -1.0, 0.0, 1.0, 2.0]
=============================

[-1.0, -0.5, 0.0, 0.5, 1.0]
```

[37]:
```
type(voltage) # Returns the type of the variable 'courant'
```

[37]:
```
list
```

[38]:
```
type(courant)
```

[38]:
```
list
```

[39]:
```
help(voltage.sort())
```

```
Help on NoneType object:

class NoneType(object)
 |  Methods defined here:
 |
 |  __bool__(self, /)
 |      True if self else False
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
```

Moreover, in some situations, you may first create an empty list which will later be filled with values using the `append()` function. To create an empty list, you can use: `x = list()` or `x = []`.

- Unlike a simple list, a multi-dimensional list is a list where the individual elements consist of multiple values. In general, a multi-dimensional list is presented as a list of lists.

[40]:
```
x = [[1,2,3],[2,3,4],[3,4,5]] # liste à deux dimensions (liste de listes)

y = [[[1,2],[2,3]],[[4,5],[5,6]]] # liste à trois dimensions (liste de listes␣
 ↪de listes)
```

```
x,y
```

[40]:
```
([[1, 2, 3], [2, 3, 4], [3, 4, 5]], [[[1, 2], [2, 3]], [[4, 5], [5, 6]]])
```

Since a list is a sequence of indexed values, you can access each value or groups of values by specifying their index:

[41]:
```
x = list(['Monday', 'Tuesday', 'Wednesday', 1800, 20.357, 'Thursday',␣
 ↪'Friday']) # Definition of a list
print(x) # Displays all the elements of the list x
```

```
['Monday', 'Tuesday', 'Wednesday', 1800, 20.357, 'Thursday', 'Friday']
```

[42]:
```
print(x[0]) # Returns the first element of x: 'Monday' (Note: indexing starts␣
 ↪at 0)
print("=================================================")
print('\t')

print(x[3]) # Returns the element at index 3 (fourth element of x): 1800
print("=================================================")
print('\t')

print(x[1:3]) # Returns all elements between index 1 and index 3 (Note: element␣
 ↪at index 3 is excluded)
print("=================================================")
print('\t')

print(x[1:6:2]) # Returns all elements between index 1 and index 6 with a step␣
 ↪of 2 elements each time ['Tuesday', 1800, 'Thursday'] (element at index 6 is␣
 ↪excluded).
print("=================================================")
print('\t')

print(x[2:]) # Returns all elements starting from index 2 (inclusive).
print("=================================================")
print('\t')

print(x[:3]) # Returns all elements up to but not including index 3
print("=================================================")
print('\t')

print(x[-1]) # Negative indexing, returns the last element of the list␣
 ↪(equivalent to x[6])
print("=================================================")
print('\t')

print(x[-2]) # Negative indexing, returns the second to last element of the␣
 ↪list (equivalent to x[5])
```

```
print("=================================================")
print('\t')

print(x[::2]) # Iterates through all elements between index 0 and the last␣
 ↪index, returning every second element ['Monday', 'Wednesday', 20.357,␣
 ↪'Friday'].
print("=================================================")
print('\t')

print(x[::-1]) # Returns a list containing all elements of x, rearranged from␣
 ↪the last element to the first. This is a reverse of x. Returns ['Friday',␣
 ↪'Thursday', 20.357, 1800, 'Wednesday', 'Tuesday', 'Monday'].
# The same result can be obtained by using x.reverse()
x_rev = x.reverse()
x_rev
```

```
Monday
================================================

1800
================================================

['Tuesday', 'Wednesday']
================================================

['Tuesday', 1800, 'Thursday']
================================================

['Wednesday', 1800, 20.357, 'Thursday', 'Friday']
================================================

['Monday', 'Tuesday', 'Wednesday']
================================================

Friday
================================================

Thursday
================================================

['Monday', 'Wednesday', 20.357, 'Friday']
================================================

['Friday', 'Thursday', 20.357, 1800, 'Wednesday', 'Tuesday', 'Monday']
```

[43]:
```
With a multi-dimensional list, indexing is done at multiple levels. For example
```

```
  Cell In [43], line 1
    With a multi-dimensional list, indexing is done at multiple levels. For␣
 ↪example
         ^
SyntaxError: invalid syntax
```

[44]:
```python
x = [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
print(x)
print("==================================================")
print('\t')

print(x[0]) # Returns the first sublist [1, 2, 3]
print("==================================================")
print('\t')

print(x[0][0]) # Returns the first element of the first sublist, which is 1
print("==================================================")
print('\t')

print(x[2]) # Returns the third sublist [3, 4, 5]
print("==================================================")
print('\t')

print(x[2][1]) # Returns the element at index 1 of the third sublist, which is 4
print("==================================================")
print('\t')

print(x[1:]) # Returns all sublists starting from index 1: [[2, 3, 4], [3, 4,␣
 ↪5]]
print("==================================================")
print('\t')

print(x[1:][0]) # Returns the first sublist from the sliced list: [2, 3, 4]
print("==================================================")
print('\t')

print(x[-1]) # Returns the last sublist [3, 4, 5]
print("==================================================")
print('\t')

print(x[1][:2]) # Returns the first two elements of the second sublist: [2, 3]
print("==================================================")
print('\t')
print(x[1][1:]) # Returns elements from index 1 to the end of the second␣
  ↪sublist: [3, 4]
```

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

[1, 2, 3]
==================================================

1
==================================================

[3, 4, 5]
==================================================

4
==================================================

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

[2, 3, 4]
==================================================

[3, 4, 5]
==================================================

[2, 3]
==================================================

[3, 4]
```

1.1.2 0.1. The `range()` Function

Value sequences are variables that are frequently encountered in Python programs. They represent a set of successive and ordered values that can be extracted like a list. Value sequences are generated using the `range()` function.

The function has the following syntax:

`range(start, stop, step)`

- start (optional): The value of the first number in the sequence. If omitted, the sequence starts from 0.
- stop: The end of the sequence. This value is not included in the sequence.
- step (optional): The difference between each pair of consecutive values in the sequence. If omitted, the default step is 1.

Example:

[45]:
```python
x = range(10) # Creates a sequence of integer values from 0 to 9
print(x)
print("================================================")
print('\t')

x = range(2, 10) # Creates a sequence of integer values from 2 to 9
print(x)
print("================================================")
print('\t')

x = range(1, 10, 2) # Creates a sequence of integer values from 1 to 9 with a␣
 ↪step of 2. It returns 1, 3, 5, 7, and 9
print(x)
```

```
range(0, 10)
================================================

range(2, 10)
================================================

range(1, 10, 2)
```

[46]:
```
To display the generated values, you can use the `list()` function, as shown␣
 ↪below:
```

```
  Cell In [46], line 1
    To display the generated values, you can use the `list()` function, as show␣
 ↪below:
       ^
SyntaxError: invalid syntax
```

[47]:
```python
x = range(10)

print(list(x)) # Returns a list: values in square brackets [0, 1, 2, 3, 4, 5,␣
 ↪6, 7, 8, 9]
```

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

1.1.3 0.2. Operations on Lists

Once a list is defined, several operations can be performed to modify its structure or its elements.

- To find the number of elements in a list, we use the `len()` function. Example: consider the list `x` defined as follows:

```python
x = ['Monday', 'Tuesday', 'Wednesday', 1800, 20.357, 'Thursday', 'Friday']
```

To find the length of `x` (the number of elements in `x`), we do:

```
    print(len(x)) # returns 7
```

[48]:
```
x = ['Monday', 'Tuesday', 'Wednesday', 1800, 20.357, 'Thursday', 'Friday']
print(len(x)) # returns 7
```

```
7
```

- We can concatenate two lists to form a single list using the + operator, which allows for list concatenation:

```
  x = ['giraffe', 'tiger']
  y = ['monkey', 'mouse']
  z = x + y
  print(z) # returns ['giraffe', 'tiger', 'monkey', 'mouse']
```

[49]:
```
x = ['giraffe', 'tiger']
y = ['monkey', 'mouse']
z = x + y
print(z) # returns ['giraffe', 'tiger', 'monkey', 'mouse']
```

```
['giraffe', 'tiger', 'monkey', 'mouse']
```

- We can repeat the elements of a list using the multiplication operator *:

```
    x = ['giraffe', 24, 18, 'tiger', 2400, 150]
    y = x * 3
    print(y) # returns ['giraffe', 24, 18, 'tiger', 2400, 150, 'giraffe', 24, 18, 'tiger', 240
```

[50]:
```
x = ['giraffe', 24, 18, 'tiger', 2400, 150]
y = x * 3
print(y) # returns ['giraffe', 24, 18, 'tiger', 2400, 150, 'giraffe', 24, 18,
         # 'tiger', 2400, 150, 'giraffe', 24, 18, 'tiger', 2400, 150]
```

```
['giraffe', 24, 18, 'tiger', 2400, 150, 'giraffe', 24, 18, 'tiger', 2400, 150,
'giraffe', 24, 18, 'tiger', 2400, 150]
```

- It is possible to modify a particular element in a list by using its index:

```
    x = ['Monday', 'Tuesday', 'Wednesday', 1800, 20.357, 'Thursday', 'Friday']
    x[3] = x[3] + 100
    print(x) # returns ['Monday', 'Tuesday', 'Wednesday', 1900, 20.357, 'Thursday', 'Friday']

    x[6] = x[6] + ' Saint' # Note the space in ' Saint', otherwise you'll get 'FridaySaint'.
    print(x) # returns ['Monday', 'Tuesday', 'Wednesday', 1900, 20.357, 'Thursday', 'Friday Sa
```

[51]:
```
x = ['Monday', 'Tuesday', 'Wednesday', 1800, 20.357, 'Thursday', 'Friday']
x[3] = x[3] + 100
print(x) # returns ['Monday', 'Tuesday', 'Wednesday', 1900, 20.357, 'Thursday',␣
 ↪'Friday']

print("=================================================================\n")
```

```
x[6] = x[6] + ' Saint' # Note the space in ' Saint', otherwise you'll get␣
  ↪'FridaySaint'.
print(x) # returns ['Monday', 'Tuesday', 'Wednesday', 1900, 20.357, 'Thursday',␣
  ↪'Friday Saint']
```

```
['Monday', 'Tuesday', 'Wednesday', 1900, 20.357, 'Thursday', 'Friday']
==================================================================

['Monday', 'Tuesday', 'Wednesday', 1900, 20.357, 'Thursday', 'Friday Saint']
```

- New elements can be added in addition to the initial elements. For this, we use the `append()` function:

  ```
  x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
  x.append('Saturday')
  x.append('Sunday')
  print(x)
  ```

[52]:
```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
x.append('Saturday')
x.append('Sunday')
print(x)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

As we can see, the `append()` function can only add one element to a list at a time. Therefore, we can use the `extend()` function when we want to add multiple elements at once. Example:

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
x.extend(['Saturday', 'Sunday'])
print(x)
```

[53]:
```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
x.extend(['Saturday', 'Sunday'])
print(x)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

For further study, explore the following methods and how they operate on lists: `insert()`, `remove()`, `delete()`, `index()`, `count()`, `join()`, `zip()`. ***

Given that their usage is quite uncommon, we will not cover `tuple` and `set` objects in this course. The brief presentation given above is relatively sufficient, and readers who explicitly need more information can consult the extensive documentation available online.

However, we can mention that a `set` is an unordered iterable collection of distinct hashable elements and that classic mathematical operations on sets can be performed in Python. For example:

<img src="images/set.png" width="20%>

```
X, Y = set('abcd'), set('sbds')
print("X =", X) # X = {'a', 'c', 'b', 'd'}
```

```
print("Y =", Y) # Y = {'s', 'b', 'd'} : only one element 's'

print('c' in X)  # True
print('a' in Y)  # False
print(X - Y)     # {'a', 'c'}
print(Y - X)     # {'s'}
print(X | Y)     # {'a', 'c', 'b', 'd', 's'}
print(X & Y)     # {'b', 'd'}
```

[54]:
```
X, Y = set('abcd'), set('sbds')
print("X =", X) # X = {'a', 'c', 'b', 'd'}
print("==========================================\n")

print("Y =", Y) # Y = {'s', 'b', 'd'} : only one element 's'
print("==========================================\n")

print('c' in X)  # True
print("==========================================\n")

print('a' in Y)  # False
print("==========================================\n")

print(X - Y)     # {'a', 'c'}
print("==========================================\n")

print(Y - X)     # {'s'}
print("==========================================\n")

print(X | Y)     # {'a', 'c', 'b', 'd', 's'}
print("==========================================\n")

print(X & Y)     # {'b', 'd'}
```

```
X = {'a', 'b', 'c', 'd'}
==========================================

Y = {'s', 'b', 'd'}
==========================================

True
==========================================

False
==========================================

{'a', 'c'}
==========================================
{'s'}
==========================================

{'b', 'c', 's', 'd', 'a'}
==========================================

{'b', 'd'}
```

1.2 1. Dictionaries

1.2.1 1.0. About dictionaries

In Pythonic design, a dictionary (also called a map) is a versatile collection of objects that adheres to the `key-value` principle. Unlike lists, where elements are accessed by their position or index, dictionaries use unique keys to identify and retrieve values. These keys are typically strings but can also be other immutable types, such as numbers or tuples. This key-value structure allows for efficient lookups, additions, and deletions, making dictionaries a powerful tool for organizing and managing data in a more intuitive and flexible way compared to lists.

[55]:
```
x = {'name': 'Jean', 'age': 25, 'weight': 70, 'height': 1.75}
y = {'Jean': [25, 70, 1.75], 'Paul': [30, 65, 1.80], 'Pierre': [35, 75, 1.65]}
z = {'Jean': (25, 70, 1.75), 'Paul': (30, 65, 1.80), 'Pierre': (35, 75, 1.65)}
k = {'name': 'Jean', 'biometrics': [25, 70, 1.75], 'score': (12, 17, 15),␣
 ↪'rank': 25}
```

The four variables above represent **four** typical ways to define a dictionary. Variable `x` is a dictionary where the keys are **name, age, weight, and height**. The corresponding values are **Jean, 25, 70**, and **1.75**. In the definition of `x`, it is noted that each key corresponds to a unique value. However, it is very common to associate multiple values with a single key. This is the case with dictionary `y`.

1.2.2 1.1. Operations on dictionaries

- To access the elements of a dictionary, you use the keys. The method `keys()` returns the list of keys in the dictionary, and the method `values()` returns the values.

```
X = {'name': 'Jean', 'age': 25, 'weight': 70, 'height': 1.75}
print(X.keys())  # returns ['name', 'age', 'weight', 'height']
print(X.values()) # returns ['Jean', 25, 70, 1.75]
print(X['name'])  # returns 'Jean'
x = {'Jean': [25, 70, 1.75], 'Paul': [30, 65, 1.80], 'Pierre': [35, 75, 1.65]}
print(x['Jean']) # returns [25, 70, 1.75]
print(x['Jean'][0]) # returns 25
print(x['Jean'][0:2]) # returns [25, 70]
```

[56]:
```
X = {'name': 'Jean', 'age': 25, 'weight': 70, 'height': 1.75}
print("==========================================\n")

print(X.keys())  # returns ['name', 'age', 'weight', 'height']
print("==========================================\n")
```

```python
print(X.values()) # returns ['Jean', 25, 70, 1.75]
print("==========================================\n")

print(X['name'])  # returns 'Jean'
print("==========================================\n")

x = {'Jean': [25, 70, 1.75], 'Paul': [30, 65, 1.80], 'Pierre': [35, 75, 1.65]}
print("==========================================\n")

print(x['Jean']) # returns [25, 70, 1.75]
print("==========================================\n")

print(x['Jean'][0]) # returns 25
print("==========================================\n")

print(x['Jean'][0:2]) # returns [25, 70]
```

```
==========================================

dict_keys(['name', 'age', 'weight', 'height'])
==========================================

dict_values(['Jean', 25, 70, 1.75])
==========================================

Jean
==========================================

==========================================

[25, 70, 1.75]
==========================================

25
==========================================

[25, 70]
```

- Adding or modifying keys or values: You can modify a dictionary by either changing existing keys and values or by adding new ones. You can also remove values as well as keys.

```python
x = {} # Creates an empty dictionary. You could also use x = dict()
x['name'] = 'Jean' # Adds the key-value pair 'name' and 'Jean' to the initial dictionary x
x['biometrics'] = [25, 70, 1.75] # Adds the key-value pair 'biometrics' and [25, 70, 1.75] t
x['biometrics'] = [30, 70, 1.80] # Modifies the values of the 'biometrics' key by redefining
x['biometrics'][0] = 2 # Modifies the element at index 0 in the list of values corresponding

Y = {'Jean': [25, 70, 1.75], 'Paul': [30, 65, 1.80], 'Pierre': [35, 75, 1.65]}
del Y['Jean'] # Deletes the key 'Jean' and all its corresponding values
```

```
  del Y['Paul'][0] # Deletes the element at index 0 in the value sequence corresponding to the
```

[57]:
```
x = {} # Creates an empty dictionary. You could also use x = dict()
print(x)
print("==========================================\n")

x['name'] = 'Jean' # Adds the key-value pair 'name' and 'Jean' to the initial␣
 ↪dictionary x
print(x)
print("==========================================\n")

x['biometrics'] = [25, 70, 1.75] # Adds the key-value pair 'biometrics' and␣
 ↪[25, 70, 1.75] to dictionary x
print(x)
print("==========================================\n")

x['biometrics'] = [30, 70, 1.80] # Modifies the values of the 'biometrics' key␣
 ↪by redefining it
print(x)
print("==========================================\n")

x['biometrics'][0] = 2 # Modifies the element at index 0 in the list of values␣
 ↪corresponding to the 'biometrics' key (previously defined)
print(x)
print("==========================================\n")



Y = {'Jean': [25, 70, 1.75], 'Paul': [30, 65, 1.80], 'Pierre': [35, 75, 1.65]}
print(Y)
print("==========================================\n")

del Y['Jean'] # Deletes the key 'Jean' and all its corresponding values

del Y['Paul'][0] # Deletes the element at index 0 in the value sequence␣
 ↪corresponding to the key 'Paul'. For a key with a single value, use del␣
 ↪x['keyName'] where 'keyName' is the name of the key with the single value.


print(Y)
```

```
{}
==========================================

{'name': 'Jean'}
==========================================

{'name': 'Jean', 'biometrics': [25, 70, 1.75]}
```

```
=========================================

{'name': 'Jean', 'biometrics': [30, 70, 1.8]}
=========================================

{'name': 'Jean', 'biometrics': [2, 70, 1.8]}
=========================================

{'Jean': [25, 70, 1.75], 'Paul': [30, 65, 1.8], 'Pierre': [35, 75, 1.65]}
=========================================

{'Paul': [65, 1.8], 'Pierre': [35, 75, 1.65]}
```

- To rename a key in a dictionary, you use the `pop()` function as defined in the following example:

```
x = {'Jean': [25, 70, 1.75], 'Paul': [30, 65, 1.80], 'Pierre': [35, 75, 1.65]}
x['John'] = x.pop('Jean') # Renames the key 'Jean' to 'John'
print(x)
```

[58]:
```
x = {'Jean': [25, 70, 1.75], 'Paul': [30, 65, 1.80], 'Pierre': [35, 75, 1.65]}
print(x)

x['John'] = x.pop('Jean') # Renames the key 'Jean' to 'John'
print(x)
```

```
{'Jean': [25, 70, 1.75], 'Paul': [30, 65, 1.8], 'Pierre': [35, 75, 1.65]}
{'Paul': [30, 65, 1.8], 'Pierre': [35, 75, 1.65], 'John': [25, 70, 1.75]}
```

1.3 Further reading

0. Programming with Python; Section 20
1. Programming with Python; Section 21