Lecture notes on Python Programming

By Jason Pandian

Assistant Professor, Department of Information Technology

LISTS, TUPLES DICTIONARIES AND FUNCTIONS

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters- Tuples: tuple assignment, tuple as return value- Dictionaries: operations and methods, advanced list processing – list comprehension. Functions and User Defined Functions: Simple and Mathematical Built–in Functions, Recursion -Illustrative Problems

What is List?

- In Python, a list is a versatile data structure used to store a collection of items.
- Lists are ordered, mutable (modifiable), and can contain elements of different data types, including other lists. - They are denoted by square brackets [], with elements separated by commas.

List Basic Examples

In [28]:
```
# Define a list containing integers
my_list = [1, 2, 3, 4, 5]

# Accessing elements of a list using index
print(my_list[0])  # Output: 1

# Define a list containing strings
```

```
fruits = ['apple', 'banana', 'orange', 'grape']

# Modifying elements of a list
fruits[0] = 'pear'
print(fruits)      # Output: ['pear', 'banana', 'orange', 'grape']

# Define a list containing mixed data types
mixed_list = [1, 'hello', 3.14, True]

# List concatenation
new_list = my_list + fruits
print(new_list)    # Output: [1, 2, 3, 4, 5, 'pear', 'banana', 'orange', 'gr

# List slicing
print(my_list[1:3])  # Output: [2, 3]

# Length of a list
print(len(my_list))  # Output: 5

# Nested lists (list containing lists)
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
1
['pear', 'banana', 'orange', 'grape']
[1, 2, 3, 4, 5, 'pear', 'banana', 'orange', 'grape']
[2, 3]
5
```

List Operations

Appending Elements append(): Adds an element to the end of the list.

In [29]:
```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

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

Extending Lists:

extend(): Appends elements from another list to the end of the list.

In [30]:
```
my_list = [1, 2, 3]
another_list = [4, 5, 6]
my_list.extend(another_list)
print(my_list)  # Output: [1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Inserting Elements:

insert(): Inserts an element at a specified position.

In [31]:
```
my_list = [1, 2, 3]
my_list.insert(1, 5)
print(my_list)  # Output: [1, 5, 2, 3]
```

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

Removing Elements:

remove(): Removes the first occurrence of a specified value.

In [32]:
```
my_list = [1, 2, 3, 4, 3]
my_list.remove(3)
print(my_list)  # Output: [1, 2, 4, 3]
```

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

Popping Elements:

pop(): Removes and returns the element at a specified index. If no index is specified, it removes and returns the last element.

In [33]:
```
my_list = [1, 2, 3]
popped_element = my_list.pop(1)
print(my_list)       # Output: [1, 3]
print(popped_element)  # Output: 2
```

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

Indexing:

index(): Returns the index of the first occurrence of a specified value.

In [34]:
```
my_list = [1, 2, 3, 4, 3]
index = my_list.index(3)
print(index)  # Output: 2
```

```
2
```

Counting:

count(): Returns the number of occurrences of a specified value.

In [35]:
```
my_list = [1, 2, 3, 4, 3]
count = my_list.count(3)
print(count)  # Output: 2
```

```
2
```

Sorting :

sort(): Sorts the list in ascending order.

In [36]:
```
my_list = [3, 1, 4, 2]
my_list.sort()
print(my_list)  # Output: [1, 2, 3, 4]
```

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

Reversing:

reverse(): Reverses the order of the elements in the list.

my_list = [1, 2, 3, 4] my_list.reverse() print(my_list) # Output: [4, 3, 2, 1]

Copying Lists:

copy(): Returns a shallow copy of the list.

In [37]:
```
my_list = [1, 2, 3]
copied_list = my_list.copy()
print(copied_list)  # Output: [1, 2, 3]
```

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

In [38]:
```
# Define a list
my_list = [1, 2, 3, 4, 5]

# Append method: adds an element to the end of the list
my_list.append(6)
print("After appending 6:", my_list)

# Extend method: appends elements from another list to the end of the list
another_list = [7, 8, 9]
my_list.extend(another_list)
print("After extending with [7, 8, 9]:", my_list)

# Insert method: inserts an element at a specified position
my_list.insert(2, 10)
print("After inserting 10 at index 2:", my_list)

# Remove method: removes the first occurrence of a specified value
my_list.remove(3)
print("After removing the first occurrence of 3:", my_list)

# Pop method: removes and returns the element at a specified index, or the l
popped_element = my_list.pop(4)
print("Popped element:", popped_element)
print("After popping the element at index 4:", my_list)
```

```
# Index method: returns the index of the first occurrence of a specified val
index_of_2 = my_list.index(2)
print("Index of 2:", index_of_2)

# Count method: returns the number of occurrences of a specified value
count_of_5 = my_list.count(5)
print("Count of 5:", count_of_5)

# Sort method: sorts the list in ascending order
my_list.sort()
print("After sorting:", my_list)

# Reverse method: reverses the order of the elements in the list
my_list.reverse()
print("After reversing:", my_list)

# Copy method: returns a shallow copy of the list
copied_list = my_list.copy()
print("Copied list:", copied_list)
```

```
After appending 6: [1, 2, 3, 4, 5, 6]
After extending with [7, 8, 9]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
After inserting 10 at index 2: [1, 2, 10, 3, 4, 5, 6, 7, 8, 9]
After removing the first occurrence of 3: [1, 2, 10, 4, 5, 6, 7, 8, 9]
Popped element: 5
After popping the element at index 4: [1, 2, 10, 4, 6, 7, 8, 9]
Index of 2: 1
Count of 5: 0
After sorting: [1, 2, 4, 6, 7, 8, 9, 10]
After reversing: [10, 9, 8, 7, 6, 4, 2, 1]
Copied list: [10, 9, 8, 7, 6, 4, 2, 1]
```

In [39]:
```
name = 'hello'
name[0:2]
```

Out[39]:
```
'he'
```

List slices

In Python, list slices allow you to access a subset of elements from a list. They provide a convenient way to work with a portion of a list without modifying the original list. Here's how you can use list slices in Python:

In [40]:
```
# Create a sample list
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Basic slicing: [start:stop:step]
# Access elements from index 2 to index 5 (exclusive)
slice_1 = my_list[0:2]
print("Slice 1:", slice_1)
```

```
Slice 1: [1, 2]
```

In [41]:
```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Access elements from the beginning to index 6 (exclusive)
slice_2 = my_list[:6]
print("Slice 2:", slice_2)
```

```
Slice 2: [1, 2, 3, 4, 5, 6]
```

In [42]:
```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Access elements from index 3 to the end
slice_3 = my_list[3:]
print("Slice 3:", slice_3)
```

```
Slice 3: [4, 5, 6, 7, 8, 9, 10]
```

In [43]:
```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Access every second element
slice_4 = my_list[::2]
print("Slice 4:", slice_4)
```

```
Slice 4: [1, 3, 5, 7, 9]
```

In [44]:
```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Access elements in reverse order
slice_5 = my_list[::-1]
print("Slice 5:", slice_5)
```

```
Slice 5: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

In Python, slicing syntax follows the pattern [start:stop:step], where: start: The starting index of the slice (inclusive). stop: The ending index of the slice (exclusive). step: The step size for selecting elements (optional, defaults to 1). You can omit any of these parameters, and Python will use default values:

If start is omitted, it defaults to 0 (beginning of the list). If stop is omitted, it defaults to the end of the list. If step is omitted, it defaults to 1 (select every element).

List Loop

In [45]:
```
# Create a sample list
my_list = [1, 2, 3, 4, 5]

# Loop through the list and print each element
for element in my_list:
    print(element)
```

```
1
2
3
4
5
```

Aliasing

- In Python, aliasing refers to the situation where two or more variables refer to the same object in memory.
- This concept is relevant when working with mutable objects like lists.
- When you create a new variable and assign it the value of another variable containing a list, both variables reference the same list object in memory.
- Therefore, modifications made to one variable will affect the other.

In [46]:
```
# Creating a list
original_list = [1, 2, 3, 4, 5]

# Creating an alias by assigning the list to a new variable
alias_list = original_list

# Modifying the alias list
alias_list.append(6)

# Printing both lists
print("Original List:", original_list)
print("Alias List:", alias_list)
```

```
Original List: [1, 2, 3, 4, 5, 6]
Alias List: [1, 2, 3, 4, 5, 6]
```

As you can see, when we modify the alias_list by appending an element, the change is reflected in the original_list as well because both variables reference the same list object in memory.

To avoid aliasing and create a copy of the list instead, you can use slicing or the `copy()` method:

Copy Method

In [47]:
```
# Using slicing to create a copy of the original list
copied_list = original_list[:]

# Modifying the copied list
copied_list.append(7)

# Printing both lists
print("Original List:", original_list)
print("Copied List:", copied_list)
```

```
Original List: [1, 2, 3, 4, 5, 6]
Copied List: [1, 2, 3, 4, 5, 6, 7]
```

List Parameters

In [48]:
```
# Define a function that takes a list as a parameter
def process_list(my_list):
    # Iterate through the elements of the list and print each element
```

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

# Create a list
my_list = [1, 2, 3, 4, 5]

# Call the function with the list as an argument
process_list(my_list)
```

```
1
2
3
4
5
```

In [49]:
```
# Define a function that takes two lists as parameters
def merge_lists(list1, list2):
    # Concatenate the two lists and return the result
    return list1 + list2

# Create two lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Call the function with the two lists as arguments
result = merge_lists(list1, list2)
print(result)  # Output: [1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Tuples

- Tuples in Python are immutable sequences, similar to lists, but with the key difference that tuples cannot be modified once created.
- They are typically used to store collections of heterogeneous data, and they support indexing, slicing, and other sequence operations.
- Tuples are defined using parentheses ().

In [50]:
```
# Creating a tuple
my_tuple = (1, 2, 3, 'a', 'b', 'c')
print(my_tuple)
```

```
(1, 2, 3, 'a', 'b', 'c')
```

In [51]:
```
# Accessing elements of the tuple
my_tuple = (1, 2, 3, 'a', 'b', 'c')
print(my_tuple[0])  # Output: 1
print(my_tuple[3])  # Output: 'a'
```

```
1
a
```

In [52]:
```
# Slicing a tuple
my_tuple = (1, 2, 3, 'a', 'b', 'c')
```

```
print(my_tuple[2:5])  # Output: (3, 'a', 'b')
```

```
(3, 'a', 'b')
```

In [53]:
```
# Tuple unpacking
a, b, c = (1, 2, 3)
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

```
1
2
3
```

In [54]:
```
# Trying to modify a tuple will result in an error
my_tuple[0] = 10  # Raises TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[54], line 2
      1 # Trying to modify a tuple will result in an error
----> 2 my_tuple[0] = 10  # Raises TypeError

TypeError: 'tuple' object does not support item assignment
```

In [55]:
```
# Length of a tuple
print(len(my_tuple))  # Output: 6

# Membership test
print('a' in my_tuple)  # Output: True


# Increment tuple/ reassign and change tuple
my_tuple = (1, 2, 3, 'a', 'b', 'c')
print(my_tuple)
my_tuple = (0, 2, 3, 'a', 'b', 'c', 'd')
print(my_tuple)
```

```
6
True
(1, 2, 3, 'a', 'b', 'c')
(0, 2, 3, 'a', 'b', 'c', 'd')
```

In [56]:
```
# Alaising
my_tuple = (1, 2, 3, 'a', 'b', 'c')
new_tuple = my_tuple
print(new_tuple)
```

```
(1, 2, 3, 'a', 'b', 'c')
```

Basic Tuple Assignment

In [57]:
```
# Define a tuple
person = ("Mathy", 25, "Chennai")

# Unpack the tuple into individual variables
```

```
name, age, city = person

# Print the variables
print("Name:", name)
print("Age:", age)
print("City:", city)
```

```
Name: Mathy
Age: 25
City: Chennai
```

Swap in Tuple

In [58]:
```
# Define two variables
a = 5
b = 10

# Swap the values of the variables using tuple assignment
a, b = b, a

# Print the variables after swapping
print("a:", a)
print("b:", b)
```

```
a: 10
b: 5
```

In [59]:
```
# Define a tuple
numbers = (1, 2, 3, 4, 5)

# Assign the first three values of the tuple to a single variable
first, second, third, *remaining = numbers

# Print the variables
print("first:", first)
print("second:", second)
print("third:", third)
print("Remaining:", remaining)
```

```
first: 1
second: 2
third: 3
Remaining: [4, 5]
```

In [60]:
```
# Define a tuple
point = (10, 20, 30)

# Unpack the tuple, but ignore the third value using an underscore
x, y, _ = point

# Print the variables
print("x:", x)
print("y:", y)
```

```
x: 10
y: 20
```

Tuple as return value

In [61]:
```python
def get_coordinates():
    # Simulate fetching coordinates
    x = int(input("value of x"))
    y = int(input("value of y"))
    return x, y

# Call the function and receive the tuple
coordinates = get_coordinates()

# Unpack the tuple
x, y = coordinates

# Print the coordinates
print("X Coordinate:", x)
print("Y Coordinate:", y)
```

```
X Coordinate: 1
Y Coordinate: 1
```

Dictionaries

- Dictionaries in Python are data structures that store key-value pairs.
- They are mutable, unordered collections, allowing you to store and retrieve data based on keys rather than numerical indices.

Creating Dictionaries

Dictionaries are defined using curly braces `{}` with key-value pairs separated by colons `:`.

In [62]:
```python
# Creating a dictionary
person = {
    "name": "Mathy",
    "age": 25,
    "city": "Chennai"
}
print(person)
```

```
{'name': 'Mathy', 'age': 25, 'city': 'Chennai'}
```

Accessing Values

Values in a dictionary can be accessed by specifying the corresponding key.

In [63]:
```python
# Accessing values in the dictionary
print("Name:", person["name"])  # Output: Mathy
```

```
print("Age:", person["age"])    # Output: 30
```

```
Name: Mathy
Age: 25
```

Adding and Modifying Values

You can add new key-value pairs or modify existing ones in a dictionary.

In [64]:
```
# Adding a new key-value pair
person["gender"] = "Male"

# Modifying an existing value
person["age"] = 26

print("Name:", person["gender"])
print("Age:", person["age"])
print(person)
```

```
Name: Male
Age: 26
{'name': 'Mathy', 'age': 26, 'city': 'Chennai', 'gender': 'Male'}
```

Dictionary Methods

Python dictionaries come with several built-in methods for manipulation:

In [65]:
```
print(person)

# Getting all keys
keys = person.keys()
print("Keys:", keys)  # Output: dict_keys(['name', 'age', 'city', 'gender'])
```

```
{'name': 'Mathy', 'age': 26, 'city': 'Chennai', 'gender': 'Male'}
Keys: dict_keys(['name', 'age', 'city', 'gender'])
```

In [66]:
```
# Getting all values
values = person.values()
print("Values:", values)  # Output: dict_values(['Mathy', 31, 'Chennai', 'Ma
```

```
Values: dict_values(['Mathy', 26, 'Chennai', 'Male'])
```

In [67]:
```
# Checking if a key exists
print("city" in person)  # Output: True
print("Female" in person['gender'])
```

```
True
False
```

In [68]:
```
# Removing a key-value pair
del person["age"]

print(person)
```

```
{'name': 'Mathy', 'city': 'Chennai', 'gender': 'Male'}
```

Iteration over key and values

In [69]:
```
# Iterating over keys
for key in person:
    print(key, ":", person[key])
```

```
name : Mathy
city : Chennai
gender : Male
```

In [70]:
```
# Iterating over keys
for key in person.keys():
    print(key, ":", person[key])
```

```
name : Mathy
city : Chennai
gender : Male
```

In [71]:
```
# Iterating over values
for value in person.values():
    print(value)
```

```
Mathy
Chennai
Male
```

In [72]:
```
# Iterating over key-value pairs
for key, value in person.items():
    print(key, ":", value)
```

```
name : Mathy
city : Chennai
gender : Male
```

Any Questions?

- 1. Python Programming for Beginners: Skyrocket Your Code and Master Python in Less than a Week. Discover the Foolproof, Practical Route to Uncover Insider Hacks, Unlock New Opportunities, and Revolution Kindle Edition by Kit Jackson (Author), 31 May 2023
- 2. Python Programming for Beginners,ISBN-13-979-8870875248, Narry Prince, 2023

For More

Refer the Lectures/Tutorials GitHub Page