Python Data Structures: A Practical Revision

Functions, Lists, and Tuples for Robotics and Machine Learning Engineers

Minor in AI, IIT Ropar
21st March, 2025

Welcome, budding AI and Machine Learning Engineers! This document is your friendly guide to mastering essential Python data structures – functions, lists, and tuples – crucial building blocks for your AI journey. We'll learn through hands-on examples and practical exercises.

1 Functions - Your Code's Superpowers!

Imagine you are working in the AI industry and you are asked to develop autonomous shuttles that would identify different routes, and for each route it would have to perform calculations for the distance, traffic and time it would take to reach the other end. You might need to repeat these calculations multiple times. Instead of writing the same set of instructions again and again, wouldn't it be great if you could bundle them into a single unit, like a mini-program within your program? That's precisely what a function allows you to do!

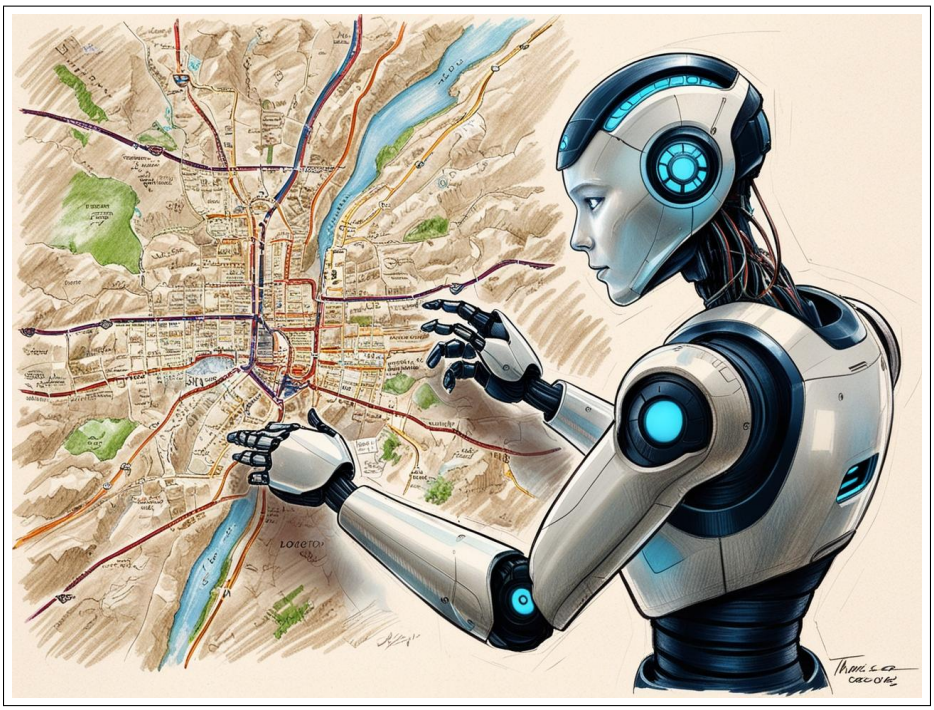

Figure 1: Building AI agents is a goal within your reach!

1.1 What is a Function?

Think of a function as a recipe. It takes ingredients (inputs), performs a series of steps (code), and delivers a dish (output).

- **A function is a block of organized, reusable code that performs a specific task.** This prevents redundancy, improves code readability, and makes your programs more manageable.

1.2 Defining a Function

In Python, we use the `def` keyword to define a function, followed by the function name, parentheses `()`, and a colon `:`. The code block within the function is indented.

```python
def greet(): #Definition of the function
    print("Hello, world!") #Main body of the function
```

1.3 Calling a Function

Defining a function only creates it. To actually execute the code inside, you need to *call* the function by simply writing its name followed by parentheses:

```python
greet()  # Calling the function
#Output: Hello, world!
```

1.4 Function Arguments

Functions can accept inputs called *arguments*. These arguments are specified within the parentheses during the function definition.

```python
def greet(name):
    print("Hello,", name)

greet("Alice") #Output: Hello, Alice
greet("Bob") #Output: Hello, Bob
```

In this example, `name` is an argument to the function `greet`. When we call the function with `greet("Alice")`, the value `"Alice"` is passed as the argument.

```python
def add_numbers(number1, number2):
    print(number1 + number2)

add_numbers(5, 6) #Output: 11
add_numbers(10, 20) #Output: 30
```

- **Parameters** are the variables listed inside the parentheses in the function definition.
- **Arguments** are the actual values passed to the function when it is called.

1.5 The `return` Statement

Functions can also *return* a value. The `return` statement exits the function and passes back a value to the caller.

```python
def square(number):
    result = number * number
    return result

output = square(5)
print(output) #Output: 25
```

Important points to remember

1. You are not able to access the variable `result` outside the function, which means that the scope of this variable is local.
2. The `return` statement has to be the last line of the function.

1.6 The pass Statement

The `pass` statement is a placeholder. It does nothing. You can use it when you need a function definition but don't have any code to put inside yet.

```
def my_function():
    pass  # Placeholder, function does nothing

my_function() # the function will not return any error
```

1.7 Built-in Functions

Python provides many built-in functions that are always available. Examples include `print()`, `len()`, `sqrt()`, `pow()`, and many more.

```
import math

print(math.sqrt(16)) #Output: 4.0
print(pow(2, 3)) #Output: 8
```

Note

Some functions require you to import the relevant module (like `math` for `sqrt`).

1.8 Default Arguments

You can specify default values for function arguments. If the caller doesn't provide a value for that argument, the default value is used.

```
def greet(name, message="Good morning!"):
    print(name + ", " + message)

greet("Alice") #Output: Alice, Good morning!
greet("Bob", "Good evening!") #Output: Bob, Good evening!
```

1.9 Recursive Functions

A *recursive function* is a function that calls itself. This can be useful for solving problems that can be broken down into smaller, self-similar subproblems.

Example: Factorial

The factorial of a number n (denoted as $n!$) is the product of all integers from 1 to n. For example, 5! = 5 * 4 * 3 * 2 * 1 = 120.

```
def factorial(number):
    if number == 1:
        return 1
    else:
        return number * factorial(number - 1)

print(factorial(6)) #Output: 720
```

Explanation:

- The function checks if the number is 1. If so, it returns 1 (base case).

- Otherwise, it returns the number multiplied by the factorial of the number minus 1. This recursive call breaks the problem down into smaller subproblems until it reaches the base case.

2 Lists - Your Versatile Data Container!

Imagine you are working in an AI project that involves collecting and analyzing data about a group of students. You need to store information like their names, ages, and courses they are enrolled in. Using separate variables for each piece of information would be cumbersome. That's where *lists* come in handy!

2.1 What is a List?

- **A list is an ordered, mutable (changeable) collection of items.** It allows you to store multiple values within a single variable.

2.2 Creating a List

Lists are created using square brackets `[]`, with items separated by commas.

```
ages = [25, 28, 42, 30]
names = ["Alice", "Bob", "Charlie"]
```

You can also use the `list()` constructor:

```
fruits = list(("apple", "banana", "cherry")) # note the double round-brackets
```

Lists can contain items of different data types:

```
student_data = ["Jack", 20, "Computer Science", [90, 95, 85]]
```

To create an empty list, simply use empty square brackets:

```
empty_list = []
```

2.3 List Characteristics

- **Ordered:** The items have a defined order, and that order is maintained.
- **Mutable:** You can change, add, or remove items after the list is created.
- **Allows Duplicates:** Lists can contain multiple items with the same value.

2.4 Accessing List Items

Each item in a list has an index, starting from 0. You can access items using their index within square brackets.

```
languages = ["Python", "Swift", "C++"]
print(languages[0]) #Output: Python
print(languages[1]) #Output: Swift
print(languages[2]) #Output: C++
```

You can also use negative indexing to access items from the end of the list:

```
print(languages[-1]) #Output: C++ (last item)
print(languages[-2]) #Output: Swift (second to last item)
```

2.5 Slicing Lists

You can extract a portion of a list using *slicing*. Slicing uses the colon operator `:` to specify a range of indices.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8]
print(my_list[2:5]) #Output: [3, 4, 5] (items from index 2 up to, but not including,
    index 5)
```

- `my_list[start:end]` extracts items from index `start` up to (but not including) index `end`.
- If `start` is omitted, it defaults to 0 (the beginning of the list).
- If `end` is omitted, it defaults to the end of the list.

2.6 Checking for Item Existence

You can use the `in` keyword to check if an item exists in a list.

```
fruits = ["apple", "banana", "orange"]
print("apple" in fruits) #Output: True
print("mango" in fruits) #Output: False
```

2.7 Changing List Items

You can change the value of a specific item by assigning a new value to its index.

```
colors = ["red", "green", "blue"]
colors[0] = "purple"
print(colors) #Output: ['purple', 'green', 'blue']
```

You can also change a range of values using slicing:

```
colors = ["red", "black", "green", "white"]
colors[2:4] = ["blue", "grey"]
print(colors) #Output: ['red', 'black', 'blue', 'grey']
```

2.8 Looping Through a List

You can iterate over the items in a list using a `for` loop:

```
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print(fruit)
```

2.9 Sorting Lists

You can sort a list using the `sorted()` function (which creates a new sorted list) or the `sort()` method (which sorts the list in place).

```
numbers = [5, 1, 4, 3]
sorted_numbers = sorted(numbers)
print(sorted_numbers) #Output: [1, 3, 4, 5]

numbers.sort() #Sorts in place
```

```
print(numbers) #Output: [1, 3, 4, 5]
```

You can also sort in descending order:

```
numbers = [5, 1, 4, 3]
numbers.sort(reverse=True)
print(numbers) #Output: [5, 4, 3, 1]
```

2.10 List Methods

Python provides several built-in methods for working with lists. Here are a few examples:

- `append(item)`: Adds an item to the end of the list.
- `extend(iterable)`: Appends elements from an iterable (e.g., another list) to the end of the list.
- `insert(index, item)`: Inserts an item at a specific index.
- `remove(item)`: Removes the first occurrence of an item.
- `pop(index)`: Removes and returns the item at a specific index (or the last item if no index is specified).
- `clear()`: Removes all items from the list.
- `index(item)`: Returns the index of the first occurrence of an item.
- `count(item)`: Returns the number of times an item appears in the list.

3 Tuples - Your Immutable Data Container!

Imagine you are working on an AI model that requires you to store a set of fixed coordinates for a particular location. These coordinates should not be accidentally changed during the program's execution. This is where *tuples* become extremely useful.

3.1 What is a Tuple?

- **A tuple is an ordered, immutable (unchangeable) collection of items.** Tuples are similar to lists, but they cannot be modified after creation.

3.2 Creating a Tuple

Tuples are created using parentheses `()`, with items separated by commas.

```
coordinates = (10, 20)
names = ("Alice", "Bob", "Charlie")
```

You can also use the `tuple()` constructor:

```
fruits = tuple(["apple", "banana", "cherry"]) # note the use of paranthesis
```

Tuples can contain items of different data types:

```
mixed_tuple = ("Jack", 20, "Computer Science")
```

Important Point to Remember: If you want to create a tuple with a single item, you need to add a comma after the item.

```
single_item_tuple = ("hello",) #Output: ('hello',)
not_a_tuple = ("hello") #Output: 'hello'
```

3.3 Tuple Characteristics

- **Ordered:** The items have a defined order, and that order is maintained.
- **Immutable:** You cannot change, add, or remove items after the tuple is created.
- **Allows Duplicates:** Tuples can contain multiple items with the same value.

3.4 Accessing Tuple Items

You can access items in a tuple using their index, just like lists.

```
coordinates = (10, 20)
print(coordinates[0]) #Output: 10
print(coordinates[1]) #Output: 20
```

You can also use negative indexing to access items from the end of the tuple:

```
print(coordinates[-1]) #Output: 20 (last item)
```

3.5 Slicing Tuples

You can extract a portion of a tuple using slicing, just like lists.

```
my_tuple = (1, 2, 3, 4, 5, 6, 7, 8)
print(my_tuple[2:5]) #Output: (3, 4, 5)
```

3.6 Deleting a Tuple

You can delete an entire tuple using the `del` keyword.

```
animals = ("dog", "cat", "bird")
del animals
# print(animals) will cause an error because 'animals' no longer exists
```

3.7 Updating a Tuple (Workaround)

Since tuples are immutable, you cannot directly modify their items. However, you can use a workaround to achieve a similar effect:

1. Convert the tuple to a list.
2. Modify the list.
3. Convert the list back to a tuple.

```
fruits = ("apple", "banana", "cherry")
fruits_list = list(fruits) #Converting the tuple into a list
fruits_list.append("orange") #Appening the list
fruits = tuple(fruits_list) #Converting the list back into a tuple

print(fruits)
```

3.8 Looping Through a Tuple

You can iterate over the items in a tuple using a `for` loop, just like lists.

```
fruits = ("apple", "banana", "cherry")
for fruit in fruits:
    print(fruit)
```

3.9 Joining Tuples

You can join two tuples together using the `+` operator.

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3) #Output: ('a', 'b', 'c', 1, 2, 3)
```

3.10 Tuple Methods

Tuples have only frequently used two built-in methods:

- `count(item)`: Returns the number of times an item appears in the tuple.

- `index(item)`: Returns the index of the first occurrence of an item.

Congratulations! You've successfully navigated the world of functions, lists, and tuples in Python. These fundamental data structures are essential tools for any aspiring robotics and machine learning engineer. Keep practicing, experimenting, and exploring, and you'll be well on your way to building amazing AI-powered applications.