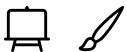


# Python Summer Course

## Course 2: Functions, Lists, Dictionaries & Classes

Théophile Gentilhomme

July 29, 2025



# List

A **list** is an ordered, changeable collection of items, written with square brackets.

```
1 fruits = ["apple", "banana", "cherry"]  
2 empty = []  
3 mixed = [1, "yes", True]
```

## Key Features:

- Ordered: items have positions (indexes)
- Mutable: you can change, add, or remove items
- Can contain any type, even mixed types



# Accessing Elements & Slicing

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 fruits = ["apple", "banana", "cherry", "orange", "melon"]
2 print(fruits[0])
3 print(fruits[-1])
```

Python Code

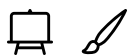
[↺ Start Over](#)[▶ Run Code](#)

```
1 print(fruits[1:3])    # From index 1 to 2 → ['banana', 'cherry']
2 print(fruits[:2])     # From start to index 1 → ['apple', 'banana']
3 print(fruits[2:])     # From index 2 to end → ['cherry', 'orange']
```

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 print(fruits[::2])    # Every 2nd item → ['apple', 'cherry']
2 print(fruits[::-1])   # Reversed list → ['orange', 'cherry', 'banana', 'apple']
```



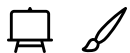
# List Manipulation

## Add & Remove

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 fruits = ["apple", "banana", "cherry"]
2
3 print(len(fruits))
4
5 fruits.append("orange")      # Add at the end
6 fruits.insert(1, "kiwi")    # Insert at position
7 fruits.remove("banana")     # Remove by value
8
9 print(fruits)
```



# Looping Through Lists

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1  for fruit in fruits:  
2      print(fruit)
```



# Useful Functions

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 print(len(fruits))      # Number of items
2 print(sorted(fruits))   # New sorted list
3 fruits.sort()           # Sort in place
4 print(fruits)
5 fruits.reverse()        # Reverse order
6 print(fruits)
```



# Indexing Reminder

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 fruits[2] = "grape"      # Modify item at index 2
2 print(fruits)
```



# List Comprehension

List comprehensions are a concise way to **create new lists** by transforming or filtering items. it is also more efficient than doing a for loop.

```
1 new_list = [expression for item in iterable]
```



# Example: Squares of numbers

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 squares = [x**2 for x in range(5)]  
2 print(squares) # [0, 1, 4, 9, 16]
```



# With Condition: Even numbers

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 evens = [x for x in range(10) if x % 2 == 0]
2 print(evens) # [0, 2, 4, 6, 8]
```



# With Transformation

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 fruits = ["apple", "banana", "cherry"]
2 uppercased = [fruit.upper() for fruit in fruits]
3 print(uppercased) # ['APPLE', 'BANANA', 'CHERRY']
```



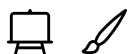
# Tuple

A **tuple** is an ordered, immutable collection of values, like a list, but you **can't change it**.

```
1 my_tuple = (1, 2, 3)
2 empty = ()
3 single = (5,) # Comma is required for single-item tuples
```

## Key Features:

- Ordered: elements have a position
- Immutable: you can't add, remove, or change items
- Can contain mixed types: `("Alice", 30, True)`
- Supports indexing and slicing like lists



# Example

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 person = ("Alice", 30)
2
3 print(person[0])
4 print(len(person))
```

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 person = ("Alice", 30)
2 # Immutable
3 person[0] = "Bob"
```



# Set

A **set** is an unordered collection of **unique** items. It is great for removing duplicates and testing membership.

Python Code

↺ Start Over

▶ Run Code

```
1 my_set = {1, 2, 3, 3, 2}
2 print(my_set) # {1, 2, 3}
3
4 # Empty set
5
6 my_empty_set = set() # not = {} !!
```

## ⚠ Properties:

- Sets are **unordered**: no indexing
- Only **immutable** items allowed (no lists/dicts inside sets)

# Set Operations

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 a = {1, 2, 3}
2 b = {3, 4, 5}
3
4 print(a | b)    # Union
5 print(a & b)    # Intersection
6 print(a - b)    # Difference
```

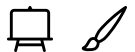


# Methods

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 a.add(6)
2 a.remove(1)
3 print(a)
```





# What I can't put in a set

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 a.add(["a", "b", "c"])
```

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 # A tuple is ok!  
2 a.add(("a", "b", "c"))  
3 print(a)  
4 a.add(tuple(["a", "b", "c"]))  
5 print(a)
```



# Dictionaries

A **dictionary** is a collection of key-value pairs, like a real-life lookup table.

Python Code

↺ Start Over

▶ Run Code

```
1 ✓ person = {  
2     "name": "Alice",  
3     "age": 30,  
4     "is_student": False  
5 }  
6  
7 print(person["name"])  
8  
9 # Empty dictionary  
10 my_dict = {} # or dict()  
11  
12 # Add an element  
13 my_dict["Counts"] = 0  
14 print(my_dict)
```



# Important:

- Keys must be **unique** and **immutable** (e.g. strings, numbers)
- Values can be any type
- Dictionaries are **unordered** before Python 3.7

Python Code

 Start Over

 Run Code

```
1 # this is ok
2 my_dict[("a", 5)] = [1, 2, 3]
```

Python Code

 Start Over

 Run Code

```
1 # this is NOT ok
2 my_dict[[1, 2, 3]] = 8
```

# Dictionary Comprehension

Python Code

[↺ Start Over](#)[▶ Run Code](#)

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

You can also add a condition:

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 evens = {x: x for x in range(10) if x % 2 == 0}  
2 print(evens)
```

# Common Operations

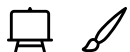
Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 person = {"name": "Alice", "age": 30}
2
3 print(person.keys())
4 print(person.values())
5 print(person.get("email"))
6
7 person["email"] = "alice@example.com"
8 del person["age"]
9 print(person)
```

## ⚠ Important:

- dictionary logic is the base of JSON format (see later)
- dictionary are used a lot to provide data samples (e.g. ML/DL)



# Functions

Functions are reusable blocks of code that perform a specific task.

```
1 def function_name(parameters):  
2     """  
3     Optional docstring describing what the function does.  
4     """  
5     # code block  
6     # many complicated things using parameters  
7     return result  
8     # or does not return anything
```

# Example

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 ✓ def greet (name) :  
2     return f"Hello, {name}!" #  
3  
4 print(greet ("Alice"))
```



# Function Parameters & Return Values

Functions can take inputs (zero, one or more) and return outputs.

## Important:

- **Parameters** are local names inside functions
- **Return** ends the function and gives back a value
- **Docstring** helps explain the function's purpose, but is optional



# Example:

Python Code

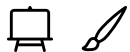
[↺ Start Over](#)[▶ Run Code](#)

```
1 ✓ def add(a, b):  
2     """Return the sum of a and b."""  
3     return a + b  
4  
5 result = add(3, 5)  
6 print(result)
```



# Mutable vs Immutable Parameters

When passing values to functions, behavior depends on **whether the object is mutable or immutable**.



# Immutable (e.g. `int`, `str`, `tuple`)

- A **copy of the value** is passed
- Changes inside the function **don't affect** the original

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 def update_number(x):  
2     x = x + 1  
3     print("Inside:", x)  
4  
5 n = 5  
6 update_number(n)  
7 print("Outside:", n)  # Still 5
```



# Mutable (e.g. `list`, `dict`, `set`)

- A **reference to the object** is passed
- Changes inside the function **do affect** the original

Python Code

↺ Start Over

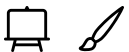
▶ Run Code

```
1 def add_item(my_list):  
2     my_list.append("new")  
3  
4 items = ["apple", "banana"]  
5 add_item(items)  
6 print(items)
```



# Functions Call Functions

Functions can be combined, one function can call another to build **modular and reusable** logic.



# Example: Grading system

Python Code

↺ Start Over

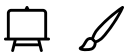
▶ Run Code

```

1  def calculate_average(scores):
2      return sum(scores) / len(scores)
3
4  def determine_grade(average):
5      if average >= 90:
6          return "A"
7      elif average >= 75:
8          return "B"
9      elif average >= 60:
10         return "C"
11     else:
12         return "F"
13
14 def grade_student(scores):
15     avg = calculate_average(scores)
16     grade = determine_grade(avg)
17     return f"Average: {avg:.1f}, Grade: {grade}"
18
19 # Example usage
20 print(grade_student([88, 92, 79]))

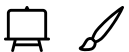
```

Functions, Lists, Dictionaries &amp; Classes



# Key Concepts

- Each function has a **clear responsibility**
- Complex logic is broken into **smaller, reusable pieces**
- You can use `if`, `for`, and `return` together



# Other features (introduced later or in provided references):

- Default values
- Function as argument
- Lambda functions



# Classes and Objects

A **class** defines a blueprint for objects: it groups **data** and **behavior** together.

We will not go into much details here, but just give an overview for you to understand objects and how to use them.

# Create a Simple Class

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 class Dog:
2     def bark(self):
3         print("Woof!")
```



# Create and Use an Object (Instance)

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 my_dog = Dog()    # Create an object
2 my_dog.bark()     # Call a method
```

The object `my_dog` is an **instance** of the class `Dog`.



# Class Initialization with `__init__`

The `__init__` method runs **when the object is created** and sets initial values of the attributes.

# Example with Constructor

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 class Dog:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6     def speak(self):
7         print(f"{self.name} says: Woof!")
8
9 my_dog = Dog("Rex", 4)
10 my_dog.speak()      # "Rex says: Woof!"
11 print(my_dog.age)   # 4
```

⚠ Important: `self` refers to the **current object**: you must include it in all methods!

# Adding Behavior with Methods

You can define your own methods to give objects useful **behaviors**.

If a class represents a **coffee machine**, the methods represent different programmes to make different coffee.

But all the programmes use the same internal attributes (e.g. coffee, water), or even internal functions (e.g. grind coffee, heat water)

# Example with State Change

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 class Counter:
2     def __init__(self):
3         self.value = 0
4
5     # We use default value if not supplied
6     def increment(self, inc = 1):
7         self.value += 1
8
9     def reset(self):
10        self.value = 0
11
12 counter = Counter()
13 counter.increment()
14 counter.increment()
15 print(counter.value)    # 2
16 counter.reset()
17 print(counter.value)    # 0
```

Objects **remember their state**, and methods can **modify** it.

Functions, Lists, Dictionaries & Classes



# Your turn!

Define student data: Create a list of students. Each student is a dictionary containing:

- "name" (string),
- "age" (int),
- "grades" (a dictionary of course:grade pairs)



# Solution

Show Solution

Enter code



Write functions, taking your list of students, to:

- Display all student names
- Calculate average grade for a given student (using building `sum()` function)

# Solution

Show Solution

Enter code



# Create a Student class

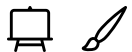
Define a class with attributes: name, age, grades, and methods to:

- Add a grade for a given course
- Compute the average for the student
- Display infos (whatever you want to print)

# Solution

Show Solution

Enter code



# More references

[Python course for data analysis](#)

[The Python tutorial](#)

