Python Summer Course

Course 2: Functions, Lists, Dictionaries & Classes

Théophile Gentilhomme

July 29, 2025

Downloading Pyodide

List

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

```
fruits = ["apple", "banana", "cherry"]
empty = []
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

Downloading package: micropip

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

```python
fruits = ["apple", "banana", "cherry"]

print(len(fruits))

fruits.append("orange")     # Add at the end
fruits.insert(1, "kiwi")    # Insert at position
fruits.remove("banana")     # Remove by value

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

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

Tuple

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

```
my_tuple = (1, 2, 3)
empty = ()
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

```python
person = ("Alice", 30)

print(person[0])
print(len(person))
```

Python Code Start Over Run Code

```python
person = ("Alice", 30)
# Immutable
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

```python
my_set = {1, 2, 3, 3, 2}
print(my_set)  # {1, 2, 3}

# Empty set

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

```python
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b)    # Union
print(a & b)    # Intersection
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

```python
person = {
    "name": "Alice",
    "age": 30,
    "is_student": False
}

print(person["name"])

# Empy dictionary
my_dict = {} # or dict()

# Add an element
my_dict["Counts"] = 0
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

```python
person = {"name": "Alice", "age": 30}

print(person.keys())
print(person.values())
print(person.get("email"))

person["email"] = "alice@example.com"
del person["age"]
print(person)
```

⚠️ Important:

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

Functions

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

```
def function_name(parameters):
    """
    Optional docstring describing what the function does.
    """
    # code block
    # many compicated things using parameters
    return result
    # or does not return anything
```

Example

Python Code | Start Over | Run Code

```python
def greet(name):
    return f"Hello, {name}!" #

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

```python
def add(a, b):
    """Return the sum of a and b."""
    return a + b

result = add(3, 5)
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

```python
def calculate_average(scores):
    return sum(scores) / len(scores)

def determine_grade(average):
    if average >= 90:
        return "A"
    elif average >= 75:
        return "B"
    elif average >= 60:
        return "C"
    else:
        return "F"

def grade_student(scores):
    avg = calculate_average(scores)
    grade = determine_grade(avg)
    return f"Average: {avg:.1f}, Grade: {grade}"

# Example usage
print(grade_student([88, 92, 79]))
```

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 arguement
- Lambda functions

Classes and Objects

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

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

Create a Simple Class

Python Code | Start Over | Run Code

```
class Dog:
    def bark(self):
        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

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def speak(self):
        print(f"{self.name} says: Woof!")

my_dog = Dog("Rex", 4)
my_dog.speak()      # "Rex says: Woof!"
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 programms to make different coffee.

But all the programms 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

```python
class Counter:
    def __init__(self):
        self.value = 0

    # We use default value if not supplied
    def increment(self, inc = 1):
        self.value += 1

    def reset(self):
        self.value = 0

counter = Counter()
counter.increment()
counter.increment()
print(counter.value)  # 2
counter.reset()
print(counter.value)  # 0
```

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

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

Python Code | Start Over | Run Code

```python
students = [
    {
        "name": "Alice",
        "age": 20,
        "grades": {"Math": 85, "Bio": 92}
    },
    {
        "name": "Bob",
        "age": 22,
        "grades": {"Math": 73, "CS": 88}
    }
]
```

Write functions, taking your list of students, to:

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

Solution

Python Code | Start Over | Run Code

```python
def get_all_names(students):
    return [student["name"] for student in students]

def calculate_average(grades):
    return sum(grades.values()) / len(grades)


print(get_all_names(students=students))

print(calculate_average(students[0]["grades"]))
```

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)

More references

Python course for data analysis

The Python tutorial

Solution

Python Code | Start Over | Run Code

```python
class Student:
    def __init__(self, name, age, grades=None):
        self.name = name
        self.age = age
        self.grades = grades or {}

    def add_grade(self, course, grade):
        self.grades[course] = grade

    def average(self):
        return sum(self.grades.values()) / len(self.grades)

    def infos(self):
        return f"{self.name} ({self.age}): {self.average()}"

student = Student("Theophile", 39)
student.add_grade("math", 60)
student.add_grade("sport", 85)

```