

1 Basic Repetition

1.1 Dictionaries

1.1.1 Dict Basics

A dictionary is a mutable data structure that allows storing key–value pairs. Keys are unique and may not occur more than once! Keys can be **INT**, **FLOAT**, or **Strings**. However, by convention they are Strings. The values associated with keys need not be unique.

```
1 my_dict = {"key1" : "value"}           # Definition
2 my_dict["key1"]  #>>> "value"         # Indexing
```

1.1.2 Nested Dicts

Nested dictionaries allow you to store information in a logically structured way.

```
1 my_dict = {"John": {
2     "Age": 48,
3     "Gender": "m",
4     "Profession": "Carpenter"
5 }}
6
7
8 # Assign a new key "Peter"
9 my_dict["Peter"] = {
10     "Age": 35,
11     "Gender": "m",
12     "Profession": "Electrician"
13 }
14
15 # Access
16 my_dict["John"]["Age"]  #>>> 48
```

1.1.3 Dict Functions

```
1 # len() always returns the number of keys
2 len(my_dict)           #>>> 2
3 len(my_dict["John"])   #>>> 3
4
5 # Iterate through keys
6 for key in my_dict:
7     print(key)          #>>> "John", "Peter"
8
9 # Iterate through dict with keys and values
10 for key, value in my_dict.items():
11     print(f'{key}: {value}')    #>>> John: {'Age': 48, ...}
12                                #>>> Peter: {'Age': 35, ...}
13
14 # Presence check
15 "John" in my_dict.keys()      #>>> True
16 "Age" in my_dict["John"].keys() #>>> True
17
18 # pop returns the values for "John" and removes "John" from the dict
19 my_dict.pop("John")
20
21 # pop with default if key not present
22 my_dict.pop("John", "Key not found")
```

Default Values

If you want to check if a key exists, get its value, and if it doesn't exist, create it with a default value.

```
1 # Checks if "Robert" exists and if not, creates it with {}
2 my_dict.setdefault("Robert", {})
```

1.1.4 JSON Export of Dict

```
1 import json
2
3 # Save as JSON
4 with open(file="dict.json", mode="w+") as fp:
5     json.dump(obj=my_dict, fp=fp, indent=4)
6
7 # Read a dict from JSON
8 with open(file="dict.json", mode="r+") as fp:
9     my_read_dict = json.load(fp=fp)
```

1.1.5 Dict Comprehension

```
1 dict_comp = {f"Key_{elem}": f"Value_{elem}" for elem in range(10)}
2
3 {
4     'Key_0': 'Value_0',
5     .
6     .
7     'Key_9': 'Value_9'
8 }
```

The value can also be initialized empty as . It does not have to be a function of elem.

1.2 Enums

Enums are created by inheriting from the Enum class. Example:

```
1 from enum import Enum
2
3 class Color(Enum):
4     RED    = 1
5     GREEN  = 2
6     BLUE   = 3
7
8 print(Color.RED)
```

1.2.1 IntEnum

A special enum type whose members are automatically treated as ints.

```
1 from enum import IntEnum
2
3 class Status(IntEnum):
4     OK          = 200
5     NOT_FOUND   = 404
6     SERVER_ERROR = 500
7
8 print(Status.OK)    #>>> 200
```

1.2.2 StrEnum

A special enum type whose values are treated as strings.

```
1 from enum import StrEnum
2
3 class Color(StrEnum):
4     RED    = "red"
5     GREEN  = "green"
6     BLUE   = "blue"
```

1.5 while Loop

```
1 while i < 10:
2     print(i)
3     i += 1
```

1.6 for Loop

```
1 for i in range(10):
2     print(i)
```

'0,1,2,3,4,5,6,7,8,9'

```
1 for char in "wrld":
2     print(char)
```

'w', 'r', 'd'

```
1 for i, char in enumerate("wrld"):
2     print(f'{i}th letter is {char}')
```

```
1 for x1, x2 in zip([1,2,3,4,5,6], "exstring"):
2     print(x1, x2)
```

Output: Produces pairs as long as there are elements in the shorter sequence.

1.7 Functions

Note:

Functions should always have input and return values, be annotated with type hints, and include a docstring.

1.7.1 Definition and Call

```
1 def say_hello(n: int, x: str) -> str:
2     print(n * x)
3
4 say_hello(4, "Hello")
```

HelloHelloHelloHello

1.7.2 Default Values

```
1 def add_to_list(item, list_to_add: list = None):
2     """
3     Default value is specified after the colon
4     """
```

1.7.3 Generators

In Python, a generator is a function that returns an iterator, producing values only when iterated over. The yield keyword defines a generator:

```
1 def generator_name():
2     while condition: # optional
3         statements
4         yield something
```

The yield keyword saves the current state and returns the value. When the function is called again, it resumes at the same point with the same local variables. The main advantage is lower memory and CPU usage compared to normal functions.

```
7
8 print(Color.RED)    #>>> red
```

1.2.3 Iterating Over Enums

```
1 for color in Color:
2     print(color)
```

1.2.4 Comparing Enum Values

```
1 color1 = Color.RED
2 color2 = Color.BLUE
3
4 if color1 == Color.RED:
5     print("Red")
6 if color1 is Color.RED:
7     print("Red")
```

1.3 Operators

1.3.1 Arithmetic Operators

Operator	Meaning
+ / - *	Addition / Division Subtraction / Multiplication
**	Exponentiation
%	Modulo
//	Floor Division
<	Bitwise XOR

1.3.2 Comparison Operators

Operator	Meaning
==	is equal to
!=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

Remember:

Comparison operators can be combined using and, or, and not.

1.4 if–else Constructs

```
1 x = 5
2
3 if x < 10:
4     print(x < 10)
5 elif x > 10:
6     print(x > 10)
7 else:
8     print("error")
```

'x < 10'

1.4.1 Ternary Operator

The if–else can also be written as:

```
1 message = "OK" if status > 10 else "Error"
```

return vs. yield

return restarts the function each time, whereas **yield** preserves local state across calls.

Infinite Counter

Generators can be used to create infinite counters, which can crash your program if not stopped. Such counters need exit conditions to halt them before a crash!

```
1 def count_endless():
2     i = 0
3     while True:
4         i += 1
5         yield i
6
7 infinite_counter = count_endless()
8
9 for elem in infinite_counter:
10     if elem > 10:
11         break
12     else:
13         print(elem)
```

Generator Comprehension

Generator comprehensions allow concise generator creation in one line, analogous to list comprehensions.

```
1 generator = (f(elem) for elem in iterable)
2
3 count_up = (i for i in range(1000))
4 count_odd = (i * 2 + 1 for i in range(10))
```

Converting Generators

Generators can be converted to lists. You lose the aforementioned advantages, but sometimes it's necessary.

```
1 my_gen = my_generator(10)
2 list(my_gen) # Convert generator to list
```

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

2 Virtual Environments

Python modules are not always backward compatible. It is therefore advantageous to create a virtual environment with the correct module versions for your code. Virtual environments allow installing modules with different version numbers.

2.1 pip

Not all modules are in the standard library. Additional modules can be installed. Using pip (the Python package installer), you can install or uninstall modules via the command line.

```
1 pip install numpy
2 pip uninstall numpy
3
4 pip list
```

2.2 pipenv

2.2.1 Creating a Virtual Environment

```
1 python -m venv env # "env" is the name of the virtual environ
2
3 pipenv install
4 pipenv install --ignore-pipfile
5
6 pipenv update --outdated
7
8 pipenv --rm
```

Important: Run these commands in the terminal at the path where you want the environment to be installed.

2.2.2 Installing Modules in the Virtual Environment

```
1 pipenv install requests
```

2.2.3 Locating the Virtual Environment

```
1 pipenv --venv
```

2.2.4 Activating and Deactivating the Environment

```
1 path/app >> pipenv shell
2 path/app >> exit
```

3 OOP

Object-oriented programming helps organize functions and data types more effectively.

3.1 Basics

3.1.1 Classes

Classes are the “blueprints” of a data type and define what is stored in that type. You can also store methods to be used within the class.

3.1.2 Objects

An object is an instance of a class. For example, if `car1` is of class `Car`, then `car1` is an instance of `Car`.

3.1.3 Attributes

An attribute is a variable within an object.

3.1.4 Class Attributes

Class attributes are accessible by all instances of a class (e.g., the count of objects of that class). They are defined outside of any methods.

3.1.5 Methods

Methods are like functions, but defined inside a class.

3.1.6 The self Parameter

The `self` parameter is very important in OOP. It always refers to the instance itself. Every method in the class should include `self` as the first parameter.

3.1.7 Object References

An object's name is a pointer to its memory address. Be careful: objects cannot be copied by simple assignment:

```
1 # This does NOT create a copy:
2 car2 = car1
3
4 # Correct way:
5 import copy
6 car2 = copy.copy(car1)
```

3.2 Definition

It is important to always define the `__init__` method; without it the class will not function.

```
1 class Car:
2     """
3     By the way: docstrings also work for classes
4     """
5     # The init method is always executed when the class is instantiated
6     def __init__(self, brand: str = "default", age: int):
7         self.brand = brand
8         self.age = age
9
10    # Define a method
11    def make_older(self, amount: int):
12        self.age += amount
13
14    # Create an object
15    car1 = Car(brand="Mercedes", age=19)
```

3.3 Accessing Object Members

```
1 car1.make_older(2)
2 print(car1.age)
```

```
21
```

You can access an object's methods and attributes using the dot notation.

3.4 Private Methods and Attributes

```
1 class Account:
2     def __init__(self, pin) -> None:
3         self.pin = pin
4         # this attribute is intended to be private
5         self.__secret_pin = pin
6
7 # Ugly: You can still access it:
8 print(account1._Account__secret_pin)
```

3.5 Inheritance

Inheritance is a concept in OOP that allows a class to inherit properties and methods from another class. It promotes efficient coding and better structure. The derived class extends the base class by inheriting its properties and methods, then adding or overriding as needed.

3.5.1 Inheritance in Python

Base classes are specified in parentheses after the class name. Multiple inheritance is possible but should generally be avoided. If you don't define a new constructor, the parent's `__init__` is inherited.

```
1 class BaseClass:
2     ...
3     pass
4
5 class SubClass(BaseClass):
6     ...
7     pass
```

3.5.2 Example Implementation

In a file named `shape.py`, implement the base class:

```
1 # Base class
2 class Shape:
3     # Base class constructor
4     def __init__(self, x, y, color):
5         self.xPos = x
6         self.yPos = y
7         self._color = color
8
9     def get_origin(self):
10        return self._xPos, self._yPos
11
12    def set_origin(self, xPos, yPos):
13        self._xPos = xPos
14        self._yPos = yPos
```

In another file named `rectangle.py`, implement the Rectangle class:

```
1 from shape import Shape
2
3 # Derived class
4 class Rectangle(Shape):
5     # New constructor
6     def __init__(self, x, y, width, height, color):
7         # Call base constructor
8         super().__init__(x, y, color)
9         self._width = width
10        self._height = height
11
12    def resize(self, width, height):
13        self._width = width
14        self._height = height
15
16    def draw(self):
17        print(f"{{self._color}} rectangle drawn with width {{
18        {self._width}} and height {self._height}")
```

3.5.3 Overriding Methods

Overriding a method means redefining a method from the base class in the derived class. The overridden method uses the same parameter signature as the base method but alters its behavior. You can still call the base implementation using `super()`.

3.6 Polymorphism

Polymorphism allows functions with the same name to be implemented in multiple ways. Depending on the object, the method can produce different results. Polymorphism is closely tied to inheritance.

3.7 Class Attributes

Class attributes are defined at the class level and shared by all instances of the class. They have the same value across the entire class.

```
1 class Point:
2     # class attribute
3     _point_instances = 0
4
5     def __init__(self, xPos, yPos):
6         Point._point_instances += 1
7         self.xPos = xPos
8         self.yPos = yPos
```

```
9
10 p1 = Point(3, 4)
```

3.8 Class Methods

Class methods are methods that have access to the class (via `cls`) but not to instance attributes.

```
1 class Point:
2     _point_instances = 0
3
4     def __init__(self, xPos, yPos):
5         self._xPos = xPos
6         self._yPos = yPos
7         Point._point_instances += 1
8
9     @classmethod
10    def get_point_instances(cls):
11        return cls._point_instances
12
13    @classmethod # Factory method
14    def create_zero_point(cls):
15        return cls(0, 0)
16
17 class Point3D(Point):
18     def __init__(self, xPos, yPos, zPos):
19         super().__init__(xPos, yPos)
20         self._zPos = zPos
21
22 p1 = Point.create_zero_point()
23 p2 = Point3D.create_zero_point()
```

3.9 Static Methods

Static methods are bound to a class but have no access to instance or class attributes. They don't use `self` or `cls` and can be called without instantiating the class.

```
1 class Calculator:
2
3     @staticmethod
4     def multiply(a, b):
5         return a * b
6
7 result = Calculator.multiply(4, 5)
```

Advantages and Disadvantages:

Advantages:

- Logical grouping of methods
- No need to create an instance
- Clarity — no state change

Disadvantages:

- Less flexibility (no access to instance data)

3.10 Properties

Properties allow attribute access to behave like simple variables while enabling complex getter/setter logic. By using the `@property` decorator, you can control access and enforce encapsulation, preventing unwanted changes to internal state.

```
1 class Point:
2     def __init__(self, xPos, yPos):
3         self._xPos = xPos
4         self._yPos = yPos
```

```
5
6     @property
7     def xPos(self):
8         print("Property called")
9         return self._xPos
10
11    @xPos.setter
12    def xPos(self, value):
13        if value < 1000:
14            self._xPos = value
15        else:
16            print("Value out of range!")
17
18 # Access / instantiation
19 p1 = Point(0, 0)
20 p1.xPos += 125
```

3.10.1 Read-Only

```
1 class Point:
2     def __init__(self, xPos, yPos):
3         self._xPos = xPos
4         self._yPos = yPos
5
6     @property
7     def xPos(self):
8         print("Property called")
9         return self._xPos
```

```
1 # Access / instantiation
2 p1 = Point(0, 0)
3 p1.xPos += 125
4 # Error: p1.xPos cannot be assigned
```

3.11 Magic Methods (Dunder Methods)

Magic methods are special methods that give Python objects additional functionality. They begin and end with a double underscore, allowing you to customize built-in operations like comparisons, addition, and more.

3.11.1 Common Magic Methods

Method	Description
<code>__init__</code>	Object initialization
<code>__str__</code>	String representation of the object
<code>__hash__</code>	Returns the object's hash code
<code>__add__</code>	Defines the addition operation (+)
<code>__len__</code>	Returns the object's length
<code>__eq__</code>	Defines behavior for equality (==)

Example implementation (`__init__` and `__str__`):

```
1 class Person:
2     def __init__(self, name, age):
3         self._name = name
4         self._age = age
5
6     def __str__(self):
7         return f"{self._name} is {self._age} years old."
8
9 p = Person("Alice", 30)
10 print(p)
>>> Alice is 30 years old.
```

Example implementation (`__add__` and `__eq__`):

```
1 class Point:
2     def __init__(self, xPos, yPos):
3         self._xPos = xPos
4         self._yPos = yPos
5
6     def __add__(self, other):
7         return Point(self._xPos + other._xPos,
8                       self._yPos + other._yPos)
9
10    def __eq__(self, other):
11        return (self._xPos == other._xPos and
12              self._yPos == other._yPos)
13
14    def __str__(self):
15        return f"Point({self._xPos},{self._yPos})"
16
17 p1 = Point(1, 2)
18 p2 = Point(3, 4)
19 p3 = p1 + p2
20 print(p3)
21 print(p1 == p2)
```

```
>>> Point(4,6)
>>> False
```

Example implementation of `__hash__`:

```
1 def __hash__(self):
2     return hash((self._xPos, self._yPos))
```

3.12 Abstract Classes

Abstract classes are a common OOP concept. An abstract class cannot be instantiated directly. It serves as a template for subclasses, defining core structure and methods. Use the `abc` module and decorate methods with `@abstractmethod`.

3.12.1 Example:

```
1 from abc import ABC, abstractmethod
2
3 class Shape(ABC):
4     def __init__(self, xPos, yPos):
5         self._xPos = xPos
6         self._yPos = yPos
7
8     @abstractmethod
9     def draw(self):
10        pass
```

Multiple Decorators:

If using multiple decorators, place `@abstractmethod` immediately above the method; other decorators may go above it:

```
1 @property
2 @abstractmethod
3 def color(self):
4     pass
```

Multiple Decorators:

Interfaces are not native in Python. Abstract classes often serve as a substitute for interfaces.

4 Exceptions

Many things can go wrong at runtime. Errors can occur and terminate the program. Common causes include:

- Programming bugs
- Invalid input/data
- Missing files

It is the programmer's responsibility to handle these issues. In case of an error, an exception can be raised and propagated to the caller, which can then:

- Catch the exception and execute fallback code (program continues).
- Catch the exception, execute fallback code, and re-raise the exception.
- Do nothing (exception propagates up and may terminate the program).

4.1 Catching Exceptions

```
1 try:
2     age = int(input("Age: "))
3 except ValueError:
4     print("You didn't enter a valid age!")
5 else:
6     print("No exceptions were thrown.")
7 print("Program continues...")
```

Block	Description
<code>try:</code>	Block that may raise an exception.
<code>except</code>	Execute this code for a specific exception.
<code>else:</code>	Executes if no exception occurs.

Multiple exceptions:

```
1 try:
2     age = int(input("Age: "))
3     factor = 10 / age
4 except ValueError:
5     print("You didn't enter a valid age!")
6 except ZeroDivisionError:
7     print("Age cannot be 0!")
8 else:
9     print("No exceptions were thrown.")
10 print("Program continues...")
```

Grouping exceptions with the same handler:

```
1 try:
2     age = int(input("Age: "))
3     factor = 10 / age
4 except (ValueError, ZeroDivisionError):
5     print("You didn't enter a valid age or age was zero!")
6 else:
7     print("No exceptions were thrown.")
8 print("Program continues...")
```

4.2 Cleaning Up

```
1 try:
2     file = open("text.txt", "a")
3     age = int(input("Age: "))
4     factor = 10 / age
5 except (ValueError, ZeroDivisionError):
6     print("You didn't enter a valid age")
7 else:
8     print("No exceptions were thrown")
9 finally:
10    file.close()
```

The `finally`: block always runs, regardless of exceptions.

Alternative File Handling:

You can use a `with` statement instead of manually closing in `finally`:

```
1 try:
2     with open("text.txt", "a") as file:
3         print("File opened")
4     age = int(input("Age: "))
5     factor = 10 / age
6 except ...
```

4.3 Raising Exceptions

You can raise exceptions yourself. Example:

```
1 def calculate_xfactor(age):
2     if age <= 0:
3         raise ValueError("Age must be greater than zero")
4     return 10 / age
5
6 calculate_xfactor(-1)
```

Exceptions Are Costly:

Exceptions should not be used for normal control flow but only for rare error conditions, as they are resource-intensive.

4.4 User-Defined Exceptions

You can easily create custom exceptions by subclassing `Exception`:

```
1 class NegativeNumberError(Exception):
2     def __init__(self, message="Negative number!"):
3         super().__init__(message)
```

You can then raise this exception as usual.

5 REST API

REST: Representational State Transfer
Web services consumed over the web are typically implemented with REST today. REST is an architectural style for distributed applications using existing web technologies, especially HTTP.

5.1 Client-Server Model

Server: Provides services to clients.

Client: Consumes services over the internet by contacting the server. The client sends a request, and the server returns a response.

5.2 Application Protocols

To interoperate, distributed parts must agree on rules and a common language—this is called the application protocol.

5.3 HTTP Requests

In Python, we use the `requests` library to make HTTP requests.

```
1 import requests
2
3 URL = "https://example.com"
4 response = requests.get(URL)
5
6 status = response.status_code
7 content = response.text
```

REST is built directly on HTTP.

5.4 REST Structure

Services are treated as resources addressed by URLs. Calling a service corresponds to making an HTTP request. The server responds with a JSON object and an HTTP status code. The JSON is analogous to a Python dict.

5.4.1 Parsing JSON

```
1 response = requests.get(URL)
2 response_dict = response.json() # Convert JSON to dict
3 value = response_dict.get("key")
```

5.4.2 Generating JSON (Server Side)

```
1 from flask import jsonify
2
3 return jsonify({"users": user_list})
```

5.4.3 HTTP Methods

HTTP can fetch, create, update, and delete resources:

GET	Read
POST	Create
PUT	Update
DELETE	Delete

5.4.4 Status Codes

1xx	Informational response
2xx	Success
3xx	Redirection (further action needed)
4xx	Client error
5xx	Server error

5.4.5 GET Requests

```
1 response = requests.get(BASE_URL + "/api/users") # all users
2 response_2 = requests.get(BASE_URL + "/api/users/2") # user with ID 2
```

GET is defined as safe and idempotent: it does not modify server state, and repeated calls return the same result.

5.4.6 PUT Requests

Updates an existing resource, sending new data in the request body as JSON. PUT is idempotent but not safe.

5.4.7 POST Requests

```
1 user = {'name': 'Daisy'}
2 response = requests.post(BASE_URL + "/api/users", json=user)
```

5.4.8 DELETE Requests

```
1 user_id = 3
2 response = requests.delete(f"{BASE_URL}/api/users/{user_id}")
```

5.5 Creating REST Services

Use an integrated web server—in Python, typically Flask—to handle REST services.

```
1 from flask import Flask, jsonify, request
2
3 app = Flask(__name__)
4
5 # GET: hello world; DELETE: bye bye
6 @app.route('/api/hello', methods=['GET', 'DELETE'])
7 def hello_world():
8     print("Method:", request.method)
9     if request.method == 'GET':
10         return jsonify("Hello World!"), 200
11     else:
12         return jsonify("Bye bye!"), 200
13
14 if __name__ == "__main__":
15     print("Starting REST server")
16     app.run(debug=True)
```

5.6 Documenting a REST API

Swagger is a framework for documenting REST APIs in a textual format. Documentation is available at <http://<hostname>apidocs/>. For Flask use “Flasgger”.

```
1 from flask import Flask, jsonify, request
2 from flasgger import Swagger
3
4 app = Flask(__name__)
5 swagger = Swagger(app)
6
7 @app.route('/api/hello', methods=['GET', 'DELETE'])
8 def hello_world():
9     """
10     API documentation goes here
11     """
12     print("Method:", request.method)
13     if request.method == 'GET':
14         return jsonify("Hello"), 200
15     else:
16         return jsonify("Bye bye!"), 200
17
18 if __name__ == "__main__":
19     print("Starting REST server")
20     app.run(debug=True)
```

The API documentation is written in the function's docstring.

6 SQL

SQL: Structured Query Language
SQL provides access to and manipulation of relational databases. A database is a collection of related tables storing a structured set of data.

CRUD:

SQL is based on CRUD: Create, Read, Update, Delete

6.1 SQLite

- The most widely used SQL database engine
- All-in-one library (serverless)
- Simple to set up and use

6.2 SQL Commands

6.2.1 Important Commands

SELECT	Extracts data from a database
DELETE	Deletes data from a database
INSERT INTO	Inserts new data into a database
CREATE DATABASE	Creates a new database
CREATE TABLE	Creates a new table
DROP TABLE	Deletes a table

6.2.2 Create a Table

The following schema is used:

```
1 CREATE TABLE table_name (
2     column_name DATATYPE,
3     column_name2 DATATYPE ADDITIONAL_SETTINGS,
4     column_name3 DATATYPE ADDITIONAL_SETTINGS
5 );
```

column_name	column_name_2	column_name_3

6.2.3 SQL Data Types

Data types can be INTEGER, REAL, TEXT, or BLOB (they can also represent Boolean values).

6.2.4 Important ADDITIONAL_SETTINGS

```
1 CREATE TABLE test_table (
2     name TEXT NOT NULL,
3     id INTEGER PRIMARY KEY AUTOINCREMENT,
4     status INTEGER DEFAULT 0
5 );
```

name	id	status
Donald Duck	1	2
Daisy Duck	2	0
Daniel Düsentrieb	3	5
Tick Duck	4	11

PRIMARY KEY	Sets a column as a Primary Key
AUTOINCREMENT	Automatically increments the value with new entries
NOT NULL	Values of new entries can not be 0
DEFAULT 0	If not set, value of new entry is 0
CHECK (column_name<10)	Checks if new entries of column_name follow certain conditions

6.2.5 Adding Rows

```
1 INSERT INTO table_name (name,
2     id,
3     status)
4 VALUES ('Donald Duck', 1, 2); -- TEXT, INTEGER, INTEGER
```

name	id	status
Donald Duck	1	2

AUTOINCREMENT and DEFAULT-defined columns can be omitted; they will be filled automatically.

6.2.6 SQL Cursor

name	id	status
Donald Duck	1	2
Daisy Duck	2	0
Daniel Düsentrieb	3	0

SQL operates on a row-by-row processing model, similar to iteration. In Python, SQLite can abstract this as an iterator.

6.2.7 SQLite Commands

```
1 import sqlite3
2 conn = sqlite3.connect("myDB.db") # open or create
3 cursor = conn.cursor()           # get cursor
4
5 conn.close()                     # close the database connection
```

Creating a simple table:

```
1 cursor.execute('''
2     CREATE TABLE users (
3         name TEXT NOT NULL,
4         id INTEGER PRIMARY KEY AUTOINCREMENT,
5         status INTEGER DEFAULT 0
6     )
7 ''')
```

IMPORTANT: Changes are not permanent until committed:

```
1 conn.commit() # commit and save the changes
```

Changes are written to a .db-journal file. After committing, that file is deleted.

testDB.db
testDB.db-journal

6.2.8 SELECT

```
1 SELECT name FROM table;
2 SELECT * FROM table; -- selects every column
```

name	id	status
Fabian	1	2
Christian	2	0
Oliver	3	0

6.2.9 SELECT WHERE

```
1 SELECT name, id FROM test_table WHERE id > 1;
```

name	id	status
Fabian	1	2
Christian	2	0
Oliver	3	0

Possible comparison operators:

=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

You can combine comparisons with logical operators:

ALL	True if all values in the subquery meet the condition
AND	True if all conditions separated by AND are true
ANY	True if any value in the subquery meets the condition
BETWEEN	True if the operand is within the specified range
EXISTS	True if the subquery returns one or more records
IN	True if the operand is equal to one of a list of expressions
LIKE	True if the operand matches a pattern
NOT	True if the condition is not true (negation)
OR	True if any of the conditions separated by OR is true
SOME	Same as ANY: true if any value in the subquery meets the condition

6.2.10 SELECT DISTINCT

```
1 SELECT DISTINCT status FROM test_table;
```

name	id	status
Fabian	1	2
Christian	2	0
Oliver	3	0

>>>

status
2
0

6.2.11 ORDER BY

The ORDER BY keyword sorts the resulting set in ascending or descending order.

```
1 SELECT * FROM test_table ORDER BY id;
2 SELECT * FROM test_table ORDER BY id DESC;
```

name ↑	id	status
Fabian	1	2
Christian	2	0
Oliver	3	0

6.2.12 DELETE

```
1 DELETE FROM table WHERE id = 1; % deletes the row with id = 1
```

name	id	status
Christian	2	0
Oliver	3	0

6.2.13 INSERT

```
1 INSERT INTO test_table (name, id, status) VALUES ('Fabian', 1, 2);
```

6.2.14 CURSOR Iterator

```
import sqlite3
connection = sqlite3.connect(r"C:\path\file.db")
cursor = connection.cursor()

using cursor-iterator {
    cursor.execute("SELECT * FROM test")
    for row in cursor: # Implicitly fetches one row at a time
        print(row)

using fetchone() {
    cursor.execute("SELECT * FROM test")
    while True:
        row = cursor.fetchone() # Explicitly fetches one row at a time
        if row is None: break    # No more rows
        print(row)
```

6.2.15 UPDATE

UPDATE allows you to modify rows within a table. Syntax:

```
1 UPDATE <table> SET <assignment> WHERE <condition>;
2 UPDATE table SET status = 100 WHERE id = 2;
```

6.2.16 DELETE FROM

DELETE allows you to remove rows based on specific conditions:

```
1 DELETE FROM <table> WHERE <condition>;
2 DELETE FROM table WHERE id = 4;
```

6.3 SQL Injection

A common issue is SQL injection. When f-strings (or string concatenation) are used, there is a risk that malicious SQL code can be injected into the query, leading to unauthorized actions. Example:

```
1 name = "; DELETE FROM Inventory;" # SQL injection payload
2 sqlQuery = f"SELECT * FROM Inventory WHERE name = '{name}'"
```

6.4 Foreign Keys (Linking Tables)

A foreign key (FK) in a table is a column that references the primary key (PK) of another table. This allows tables to reference other tables:

name	id (PK)	status	asset_id (FK)	duck_id (FK)	name
Donald Duck	1	2	3	3	Geldspeicher
Daisy Duck	2	100	2	2	Schönes Haus
Dagobert Duck	3	5	1	1	Leere Geldbörse

Ducks (Einwohner) Assets (Besitztümer)

The main characteristics or advantages are:

- Modeling relationships: Links data across different tables
- Prevents inconsistent or orphaned records

6.4.1 Linking Tables with SELECT

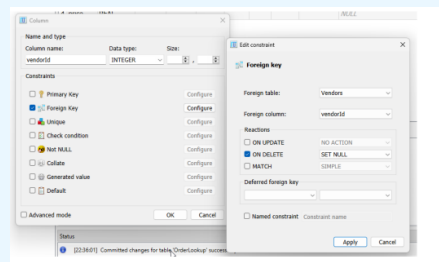
```
1 SELECT Ducks.name, Assets.name
2 FROM Ducks, Assets
3 WHERE Ducks.id == Assets.duck_id;
```

name	id (PK)	status	asset_id (FK)	duck_id (FK)	name
Donald Duck	1	2	3	3	Geldspeicher
Daisy Duck	2	100	2	2	Schönes Haus
Dagobert Duck	3	5	1	1	Leere Geldbörse

Ducks (Einwohner) Assets (Besitztümer)

Referential Integrity:

Deleting a table entry that references another table is not possible, because the related ownership would become orphaned. Error: FOREIGN KEY constraint failed! To allow this, you must first permit NULL for the FK (ON DELETE: SET NULL):



However, this allows orphaned records! If you choose CASCADE instead of SET NULL, the referenced entries will also be deleted.

name	id	status
Donald Duck	1	2
Daisy Duck	2	0
Daniel Düsentrieb	3	0

Current cursor position →

SQL operates on a row-by-row processing model, similar to iteration. In Python, SQLite can present this as an iterator.

6.4.2 SQLite Commands

```
1 import sqlite3
2 conn = sqlite3.connect("myDB.db") # open or create database
3 cursor = conn.cursor()           # get cursor
4
5 conn.close()                      # close the database connection
```

Creating a simple table:

```
1 cursor.execute('''
2     CREATE TABLE users (
3         name TEXT NOT NULL,
4         id INTEGER PRIMARY KEY AUTOINCREMENT,
5         status INTEGER DEFAULT 0
6     )
7 ''')
```

IMPORTANT: Changes are not permanent until committed:

```
1 conn.commit() # commit and save the changes
```

Changes are written to a .db-journal file. After committing, that file is deleted.

testDB.db
testDB.db-journal

6.4.3 SELECT

```
1 SELECT name FROM table;
2 SELECT * FROM table; -- selects every column
```

name	id	status
Fabian	1	2
Christian	2	0
Oliver	3	0

6.4.4 SELECT WHERE

```
1 SELECT name, id
2 FROM test_table
3 WHERE id > 1;
```

name	id	status
Fabian	1	2
Christian	2	0
Oliver	3	0

Possible comparison operators:

=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<>	not equal to

You can combine comparisons with logical operators:

ALL	True if all values in the subquery meet the condition
AND	True if all conditions separated by AND are true
ANY	True if any value in the subquery meets the condition
BETWEEN	True if the operand is within a specified range
EXISTS	True if the subquery returns one or more records
IN	True if the operand equals one of a list of values
LIKE	True if the operand matches a pattern
NOT	True if the condition is not met
OR	True if any condition separated by OR is true
SOME	True if any value in the subquery meets the condition

6.4.5 SELECT DISTINCT

```
1 SELECT DISTINCT status
2 FROM test_table;
```

name	id	status
Fabian	1	2
Christian	2	0
Oliver	3	0

>>>

status
2
0

6.4.6 ORDER BY

The ORDER BY keyword sorts the result set in ascending or descending order.

```
1 SELECT *
2 FROM test_table
3 ORDER BY id;
4
5 SELECT *
6 FROM test_table
7 ORDER BY id DESC;
```

name	id	status
Fabian	1	2
Christian	2	0
Oliver	3	0

6.4.7 DELETE

```
1 DELETE FROM table
2 WHERE id = 1; -- deletes the row with id = 1
```

name	id	status
Christian	2	0
Oliver	3	0

6.4.8 INSERT

```
1 INSERT INTO test_table (name, id, status)
2 VALUES ("Fabian", 1, 2);
```

6.4.9 CURSOR Iterator

```
import sqlite3
connection = sqlite3.connect(r"C:\path\File.db")
cursor = connection.cursor()

using cursor-iterator {
    cursor.execute("SELECT * FROM test")
    for row in cursor: # Implicitly fetches one row at a time
        print(row)
}

using fetchone() {
    cursor.execute("SELECT * FROM test")
    while True:
        row = cursor.fetchone() # Explicitly fetches one row at a time
        if row is None: break    # No more rows
        print(row)
}
```

6.4.10 UPDATE

UPDATE allows you to modify existing rows in a table. Syntax:

```
1 UPDATE <table>
2 SET <column>=<value>
3 WHERE <condition>;
4
5 UPDATE table
6 SET status = 100
7 WHERE id = 2;
```

6.4.11 DELETE FROM

DELETE removes rows from a table based on a condition:

```
1 DELETE FROM <table>
2 WHERE <condition>;
3
4 DELETE FROM table
5 WHERE id = 4;
```

6.5 SQL Injection

A common issue is SQL injection: if you build queries with f-strings, malicious SQL can be injected into the string, leading to unauthorized actions. Example:

```
1 name = " "; DELETE FROM Inventory;" # SQL INJECTION
2 sqlQuery = f"SELECT * FROM Inventory WHERE name = '{name}'"
```

6.6 Foreign Keys (Linking Tables)

A foreign key (FK) in one table refers to the primary key (PK) of another table. This allows tables to reference each other:



name	id (PK)	status
Donald Duck	1	2
Daisy Duck	2	100
Dagobert Duck	3	5

Ducks (Einwohner)

asset_id (PK)	duck_id (FK)	name
3	3	Geldspeicher
2	2	Schönes Haus
1	1	Leere Geldbörse

Assets (Besitztümer)

Key benefits:

- Models relationships: links data across tables
- Prevents inconsistent or orphaned entries

6.6.1 Joining Tables with SELECT

```

1 SELECT Ducks.name, Assets.name
2 FROM   Ducks, Assets
3 WHERE  Ducks.id = Assets.duck_id;
```



name	id (PK)	status
Donald Duck	1	2
Daisy Duck	2	100
Dagobert Duck	3	5

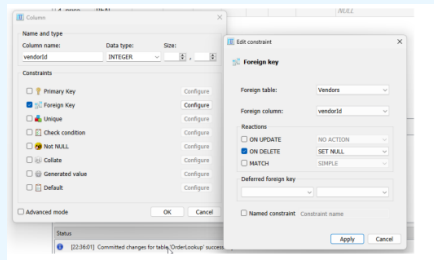
Ducks (Einwohner)

asset_id (PK)	duck_id (FK)	name
3	3	Geldspeicher
2	2	Schönes Haus
1	1	Leere Geldbörse

Assets (Besitztümer)

Referential Integrity:

Deleting a row that is referenced by another table is not allowed, as it would leave orphaned records. Error: **FOREIGN KEY constraint failed!**
To allow deletion, you must permit NULL for the FK (e.g., **ON DELETE SET NULL**):



However, that can still leave orphaned entries. Alternatively, use **ON DELETE CASCADE** to delete all dependent records automatically.