

Thank you for downloading and using my Python Advanced summary! I hope you find it useful. If you find any errors or have feedback, please write to me at eric.lindegger@stud.hslu.ch

Inhaltsverzeichnis

1	Fundamentals – Basic Review	2
1.1	Special Characters	2
1.2	Number Formats	2
1.3	Data Types	2
1.4	Strings	2
1.4.1	Basic Strings	2
1.4.2	F-Strings	2
1.4.3	String Functions	2
1.5	Sets	2
1.6	Dictionaries	3
1.6.1	Dict Basics	3
1.6.2	Nested Dicts	3
1.6.3	Dict Functions	3
1.6.4	JSON Export of Dict	3
1.6.5	Dict Comprehension	3
1.7	Enums	3
1.7.1	IntEnum	3
1.7.2	StrEnum	4
1.7.3	Iterating Enums	4
1.7.4	Comparing Enums	4
1.8	Operators	4
1.8.1	Arithmetic Operators	4
1.8.2	Comparison Operators	4
1.9	if-else Chains	4
1.9.1	Ternary Operator	4
1.10	while Loop	4
1.11	for Loop	4
1.12	Functions	4
1.12.1	Definition and Call	4
1.12.2	Default Values	5
1.12.3	Generators	5
2	Virtual Environments	5
2.1	pip	5
2.2	pipenv	5
2.2.1	Creating a Virtual Environment	5
2.2.2	Installing Modules in the Virtual Environment	5
2.2.3	Locating the venv	5
2.2.4	Activating and Deactivating	5
3	OOP	6
3.1	Basics	6
3.1.1	Classes	6
3.1.2	Objects	6
3.1.3	Attributes	6
3.1.4	Class Attributes	6
3.1.5	Methods	6
3.1.6	self Parameter	6
3.1.7	Object References	6
3.2	Definition	6
3.3	Accessing Objects	6
3.4	Private Methods/Attributes	6
3.5	Inheritance	6
3.5.1	Implementation Example	6
3.5.2	Overriding Methods	6
3.6	Polymorphism	6
3.7	Class Attributes	7
3.8	Class Methods	7
3.9	Static Methods	7
3.10	Properties	7
3.10.1	Read-Only	7
3.11	Magic Methods	7
3.12	Abstract Classes	8
4	Exceptions	8
4.1	Catching Exceptions	8
4.1.1	Multiple Exceptions	8
4.1.2	Grouping Exceptions	8
4.2	Cleaning Up	8
4.3	Raising Exceptions	8

4.4	User-Defined Exceptions	8
-----	-------------------------	---

5	REST API	9
5.1	Client–Server Model	9
5.2	HTTP Requests	9
5.2.1	Parsing JSON	9
5.2.2	Request Methods	9
5.2.3	Status Codes	9
5.2.4	Example Endpoints	9
5.3	Creating Services with Flask	9
5.4	Documenting with Swagger	9
6	SQL	9
6.1	SQLite	9
6.1.1	Basic Commands	9
6.1.2	Inserting Rows	9
6.1.3	Queries	9
6.1.4	Updating and Deleting	9
6.1.5	Joins with Foreign Keys	10

1 Fundamentals – Basic Review

1.1 Special Characters

```
1 \t # Tab
2 \n # New line
3 \ # Backslash
4 " # " character
5 ' # ' character
```

1.2 Number Formats

Type	Syntax	Output
Binary	0b1010	bin(x)
Octal	0o12	oct(x)
Hexadecimal	0xA	hex(x)

1.3 Data Types

Data Type	Example
Integer (int)	3
Floating point (float)	3.14159
String (str)	"text"
Bool	True / False

Typecasting using: int(x), float(x), str(x), bool(x)

1.4 Strings

Attention! Strings are immutable!

1.4.1 Basic Strings

```
1 s = "this is my string"
2
3 print(s[0]) # first element
4 print(s[5:10]) # elements 5 through 9
5 print(s[5:]) # from element 5 to end
6 print(s[:10]) # from start to element 9
7 print(s[::2]) # every second element
8 print(s[::-1]) # string reversed
9 print(s[::-2]) # every second element in reverse
10
11 print(s[start:stop:step])
```

```
t
is my
is my string
this is my
ti sm tig
```

Indexing starts at 0 from the front and at -1 from the back!

0	1	2	3	4
-5	-4	-3	-2	-1

Strings can be concatenated easily:

```
1 s1 = "abc"
2 s2 = "xyz"
3
4 s_total = s1 + s2
```

1.4.2 F-Strings

Advantages:

Inclusion of variables and specification of variable formatting.

```
f_string = f"Hello, {Name}. You are {age}."
```

Format Specifier

```
1 print(f"{pi:.4f}") -> 3.1416 # round to 4 decimal places
2 print(f"{12:05}") -> 00012 # zero-pad to 5 digits
3 print(f"{3:b}") -> 11 # output as binary
4 print(f"{11:x}") -> b # output as hexadecimal
5 print(f"{11:o}") -> 13 # output as octal
```

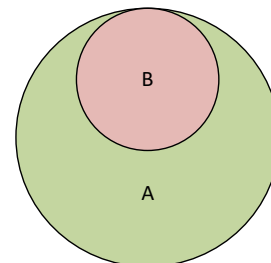
1.4.3 String Functions

```
1 len(s) # returns the number of characters in the string
2 "Juan" in s # checks if substring occurs in string
3
4 name = "Wolfgang"
5 print(name.upper()) -> WOLFGANG
6 print(name.lower()) -> wolfgang
7
8 s = "1_2_3_4_5_6_7_8"
9 s.split("_") -> ['1','2','3','4','5','6','7','8']
10 s.count("_") -> 7
11
12 word = "Hallo"
13 bytes_word = word.encode() # UTF-8 encoding to bytes
14 print(list(bytes_word)) --> [72,97,108,108,111]
```

1.5 Sets

An ordered collection of elements that are immutable! Each element appears only once in the set. Adding and removing elements is still possible.

A: is superset of B
B: is subset of A



```
1 my_set = {"a","b","c"}
2 my_set.add("x") # adds "x" and returns something?
3 my_set.remove("x") # removes "x" if present and returns something
4 A.issuperset(B) -> True
5 B.issubset(A) -> True
```

If an element is not present when attempting to remove it, a `KeyError` is raised!

1.6 Dictionaries

1.6.1 Dict Basics

A dictionary is a mutable data structure that allows storing key-value pairs. Keys are unique and cannot occur more than once! Keys can be **INT**, **FLOAT**, or **STRING**. By convention they are strings. Values do not need to be unique.

```
1 my_dict = {"key1": "value"} # definition
2 my_dict["key1"] # -> "value" # indexing
```

1.6.2 Nested Dicts

Nested dicts 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 # assignment of new key "Peter"
8 my_dict["Peter"] = {
9     "Age": 35,
10    "Gender": "m",
11    "Profession": "Electrician"
12 }
13
14 # access
15 my_dict["John"]["Age"] # -> 48
```

1.6.3 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 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 my_dict.pop("John") # returns John's values and removes John
19
20 # pop with default if key not present
21 my_dict.pop("John", "Key not found")
```

Default Values

To check if a key exists, get its value, and create it with a default if not:

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

1.6.4 JSON Export of Dict

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

1.6.5 Dict Comprehension

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

Values can also be initialized empty as . They don't have to depend on elem.

1.7 Enums

Enums are created by inheriting from Enum. 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)
```

1.7.1 IntEnum

Special enum type 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
```

1.7.2 StrEnum

Special enum type treated as strings.

```
1 from enum import StrEnum
2
3 class Color(StrEnum):
4     RED = "red"
5     GREEN = "green"
6     BLUE = "blue"
7
8 print(Color.RED) # -> red
```

1.7.3 Iterating Enums

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

1.7.4 Comparing Enums

```
1 c1 = Color.RED
2 c2 = Color.BLUE
3
4 if c1 == Color.RED:
5     print("Red")
6 if c1 is Color.RED:
7     print("Red")
```

1.8 Operators

1.8.1 Arithmetic Operators

Operator	Meaning
+, -, *, /	addition, subtraction, multiplication, division
**	exponentiation
%	modulo
//	floor division
^	bitwise XOR

1.8.2 Comparison Operators

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

Remember:

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

1.9 if-else Chains

```
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")
```

True

1.9.1 Ternary Operator

Can also be written as:

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

1.10 while Loop

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

1.11 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 "word":
2     print(char)
```

w, r, d

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

Outputs tuples until the shortest iterable ends:

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

1.12 Functions

Note:

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

1.12.1 Definition and Call

```
1 def say_hello(n: int, x: str) -> None:
2     """
3     Prints the greeting x, n times.
4     """
5     print(n * x)
6
7 say_hello(4, "Hello")
```

HelloHelloHelloHello

1.12.2 Default Values

```
1 def add_to_list(item, list_to_add: list = None):
2     """
3     Default list is None; initialize inside if needed.
4     """
```

1.12.3 Generators

A generator is a function that returns an iterator. Use `yield` to define one:

```
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)
```

return vs. yield

`return` restarts the function, whereas `yield` preserves its state.

Generator Comprehension

Analogous to list comprehensions, but with parentheses:

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

Converting Generators

If needed, convert to list (losing memory benefits):

```
1 my_gen = (i for i in range(10))
2 list(my_gen) # -> [0,1,2,3,4,5,6,7,8,9]
```

2 Virtual Environments

Python modules aren't always backwards-compatible. It's therefore advantageous to create a virtual environment with the correct module versions.

2.1 pip

Install or uninstall external modules:

```
pip install numpy
pip uninstall numpy

pip list
```

2.2 pipenv

2.2.1 Creating a Virtual Environment

```
python -m venv env # env is the name of the virtual env
```

```
pipenv install
pipenv install --ignore-pipfile
```

```
pipenv update --outdated
```

```
pipenv --rm
```

Run commands in the terminal at the desired path.

2.2.2 Installing Modules in the Virtual Environment

```
pipenv install requests
```

2.2.3 Locating the venv

```
pipenv --venv
```

2.2.4 Activating and Deactivating

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

3 OOP

Object-oriented programming helps organize functions and data types.

3.1 Basics

3.1.1 Classes

Classes are blueprints defining data and methods.

3.1.2 Objects

An object is an instance of a class.

3.1.3 Attributes

Variables inside an object.

3.1.4 Class Attributes

Shared by all instances; defined outside methods.

3.1.5 Methods

Functions defined within a class.

3.1.6 self Parameter

self refers to the instance in methods.

3.1.7 Object References

Use `copy.copy()` to duplicate, not `car2 = car1`.

```
1 import copy
2 car2 = copy.copy(car1)
```

3.2 Definition

`__init__` must be defined.

```
1 class Car:
2     """
3     Docstrings work for classes too.
4     """
5     def __init__(self, brand: str = "default", age: int = 0):
6         self.brand = brand
7         self.age = age
8
9     def make_older(self, amount: int):
10         self.age += amount
11
12 car1 = Car(brand="Mercedes", age=19)
```

3.3 Accessing Objects

```
1 car1.make_older(2)
2 print(car1.age) # -> 21
```

3.4 Private Methods/Attributes

```
1 class Account:
2     def __init__(self, pin):
3         self.pin = pin
4         self.__secretPin = pin # private
5
6 acct = Account(1234)
7 print(acct._Account__secretPin)
```

3.5 Inheritance

Subclasses inherit from base classes using parentheses.

```
1 class Shape:
2     pass
3
4 class Rectangle(Shape):
5     pass
```

3.5.1 Implementation Example

In `shape.py`:

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

In `rectangle.py`:

```
1 from shape import Shape
2
3 class Rectangle(Shape):
4     def __init__(self, x, y, width, height, color):
5         super().__init__(x, y, color)
6         self._width = width
7         self._height = height
8
9     def resize(self, width, height):
10        self._width, self._height = width, height
11
12    def draw(self):
13        print(f"{self._color} rectangle")
```

3.5.2 Overriding Methods

Use `super()` to call the base method.

3.6 Polymorphism

Same method name, different behavior in subclasses.

3.7 Class Attributes

```
1 class Point:
2     _instances = 0
3
4     def __init__(self, x, y):
5         Point._instances += 1
6         self.x, self.y = x, y
7
8 p1 = Point(3,4)
```

3.8 Class Methods

```
1 class Point:
2     _instances = 0
3
4     def __init__(self, x, y):
5         Point._instances += 1
6         self.x, self.y = x, y
7
8     @classmethod
9     def get_instances(cls):
10        return cls._instances
11
12    @classmethod
13    def create_zero_point(cls):
14        return cls(0,0)
15
16 p0 = Point.create_zero_point()
```

3.9 Static Methods

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

Pros & Cons:

Pros:

- Logical grouping of methods
- No need for instance
- No state changes

Cons:

- No access to instance or class state

3.10 Properties

Use the property decorator for getters/setters.

```
1 class Point:
2     def __init__(self, x, y):
3         self._x = x
4         self._y = y
5
6     @property
7     def x(self):
8         print("Getter called")
9         return self._x
10
11    @x.setter
12    def x(self, value):
13        if value < 1000:
14            self._x = value
15        else:
16            print("Value out of range!")
17
18 p = Point(0,0)
19 p.x += 125
```

3.10.1 Read-Only

```
1 class Point:
2     def __init__(self, x, y):
```

```
3         self._x = x
4         self._y = y
5
6     @property
7     def x(self):
8         return self._x
9
10 p = Point(0,0)
11 p.x = 5 # Error: cannot set read-only property
```

3.11 Magic Methods

Implement special behavior by defining dunder methods:

<code>__init__</code>	object initialization
<code>__str__</code>	string representation
<code>__hash__</code>	object's hash code
<code>__add__</code>	define addition (+)
<code>__len__</code>	define len()
<code>__eq__</code>	define == comparison

Example:

```
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.
```

Addition and equality example:

```
1 class Point:
2     def __init__(self, x, y):
3         self.__x = x
4         self.__y = y
5
6     def __add__(self, other):
7         return Point(self.__x + other.__x,
8                       self.__y + other.__y)
9
10    def __eq__(self, other):
11        return (self.__x, self.__y) == (other.__x, other.__y)
12
13    def __str__(self):
14        return f"Point({self.__x},{self.__y})"
15
16 p1 = Point(1,2)
17 p2 = Point(3,4)
18 p3 = p1 + p2
19 print(p3) # -> Point(4,6)
20 print(p1 == p2) # -> False
```

Hash example:

```
1 def __hash__(self):
2     return hash((self.__x, self.__y))
```

3.12 Abstract Classes

Use the abc module and @abstractmethod decorator:

```
1 from abc import ABC, abstractmethod
2
3 class Shape(ABC):
4     def __init__(self, x, y):
5         self.__x = x
6         self.__y = y
7
8     @abstractmethod
9     def draw(self):
10        pass
```

Multiple Decorators:

Place @abstractmethod immediately above the method; other decorators may be above it:

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

Interfaces:

Python does not have interfaces natively; abstract classes serve as their substitute.

4 Exceptions

At runtime many things can go wrong. Common reasons:

- Programming errors
- Invalid inputs/data
- Missing files

You can handle exceptions to keep the program running, log errors, or re-raise them.

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
try:	code that may raise an exception
except	code to run on a specific exception
else:	runs if no exception was raised

4.1.1 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...")
```

4.1.2 Grouping Exceptions

```
1 try:
2     age = int(input("Age: "))
3     factor = 10 / age
4 except (ValueError, ZeroDivisionError):
5     print("Invalid or zero age!")
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("Invalid age!")
7 else:
8     print("No exceptions were thrown.")
9 finally:
10    file.close()
```

finally: always runs, with or without an exception.

Alternative File Handling:

Use with to avoid manual 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

```
1 def calculate_factor(age):
2     if age <= 0:
3         raise ValueError("Age must be positive")
4     return 10 / age
5
6 calculate_factor(-1) # raises ValueError
```

Exceptions are Costly:

Use exceptions for unexpected errors, not for normal control flow.

4.4 User-Defined Exceptions

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

5 REST API

REST: Representational State Transfer Modern web services use REST over HTTP to exchange JSON.

5.1 Client–Server Model

Server: provides services **Client:** consumes services via HTTP

5.2 HTTP Requests

Use requests to make HTTP calls:

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

5.2.1 Parsing JSON

```
1 data = response.json() # convert JSON to dict
2 value = data.get("key")
```

5.2.2 Request Methods

GET	Read resource
POST	Create resource
PUT	Update resource
DELETE	Delete resource

5.2.3 Status Codes

1xx	Informational
2xx	Success
3xx	Redirection
4xx	Client error
5xx	Server error

5.2.4 Example Endpoints

```
1 # GET all users
2 response = requests.get(BASE_URL + "/api/users")
3 # GET user #2
4 response = requests.get(BASE_URL + "/api/users/2")
5
6 # POST create user
7 user = {'name': 'Daisy'}
8 response = requests.post(BASE_URL + "/api/users", json=user)
9
10 # PUT update user
11 update = {'name': 'DaisyUpdated'}
12 response = requests.put(BASE_URL + "/api/users/2", json=
    update)
13
14 # DELETE user
15 response = requests.delete(BASE_URL + "/api/users/3")
```

5.3 Creating Services with Flask

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

5.4 Documenting with Swagger

Use flasgger to auto-generate API docs:

```
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():
9     """
10    ---
11    responses:
12      200:
13        description: A greeting message
14    """
15    if request.method == 'GET':
16        return jsonify("Hello")
17    else:
18        return jsonify("Goodbye!")
```

6 SQL

SQL: Structured Query Language Used to create, read, update, and delete data in relational databases.

CRUD:

Create, Read, Update, Delete

6.1 SQLite

- Lightweight embedded database
- File-based
- Easy to use in Python

6.1.1 Basic Commands

```
1 import sqlite3
2 conn = sqlite3.connect("myDB.db") # open or create
3 cursor = conn.cursor()
4
5 # Create table
6 cursor.execute('''
7     CREATE TABLE users (
8         name TEXT NOT NULL,
9         id INTEGER PRIMARY KEY AUTOINCREMENT,
10        status INTEGER DEFAULT 0
11    )
12 ''')
13 conn.commit() # save changes
14 conn.close() # close connection
```

6.1.2 Inserting Rows

```
1 INSERT INTO users (name, status)
2 VALUES ("Donald Duck", 1);
```

6.1.3 Queries

```
1 SELECT name FROM users;
2 SELECT * FROM users WHERE id > 1;
3 SELECT DISTINCT status FROM users;
4 SELECT * FROM users ORDER BY id DESC;
```

6.1.4 Updating and Deleting

```
1 UPDATE users SET status = 2 WHERE id = 1;
2 DELETE FROM users WHERE id = 3;
```

6.1.5 Joins with Foreign Keys

Define foreign key relationships to enforce referential integrity and combine tables:

```
1 -- Creating tables with FK
2 CREATE TABLE Ducks (
3   id INTEGER PRIMARY KEY,
4   name TEXT NOT NULL
5 );
6 CREATE TABLE Assets (
7   id INTEGER PRIMARY KEY,
```

```
8   duck_id INTEGER,
9   name TEXT,
10  FOREIGN KEY(duck_id) REFERENCES Ducks(id) ON DELETE
    CASCADE
11 );
12
13 -- Join query
14 SELECT Ducks.name, Assets.name
15 FROM Ducks
16 JOIN Assets ON Ducks.id = Assets.duck_id;
```