PACKAGES AND IMPORTS

General tools

- `dir(module_name)` → Lists all available attributes: `module.xxx`.
- `pow(x, 2)` → Computes (x^2).
- `hypot(x, y)` → Returns the length of the hypotenuse when the legs are `x` and `y`.

Random module

- `random.seed(x: int | None)`
  - If `x` is `None`, the seed uses the current system time (seconds).
  - If `x` is an integer, the generator becomes deterministic (same seed → same sequence, except for minor floating-point variations).
- `random.choice(sequence)` → Returns a single random element from a non-empty sequence.
- `random.sample(sequence, n)` → Returns a list of `n` distinct items from the sequence (`n ≤ len(sequence)`).

Platform module

- `platform.platform(aliased=False, terse=False)`
  - `terse=True` → shorter, less detailed output. Returns a detailed OS name string.
- `platform.machine()` → Returns a generic CPU architecture name.
- `platform.processor()` → Returns a precise processor name.
- `platform.system()` → Returns the generic OS name.
- `platform.version()` → OS version string.
- `platform.python_implementation()` → Returns the Python implementation (usually `CPython`).
- `platform.python_version_tuple()` → Returns the Python version as `(major, minor, patch)`.

Modules and execution

- `__name__ = '__main__'` when the file is executed as a script.
- `__name__ = module_name` when imported as a module.

sys module

- `sys.path` → List of filesystem paths where Python looks for packages.
    - Use `.append()` to add directories.
    - Use double backslashes `\\` on Windows.

pip

- `pip install --user package_name` → installs a package for the current user only.
- `pip install -U package_name` → upgrades an installed package.
- “The Cheese Shop” → humorous reference to PyPI.

Definitions

- A *namespace* is a space in which a name exists.
- `pip --version`, `pip version`, and `pip -version` all display the pip version.

STRINGS

General concepts

- *Internationalization* is shortened to **I18N**.
- Classic ASCII uses **8 bits per character**.
- A *code point* is the numeric value representing a Unicode character.
- **UCS-4** uses 32 bits per character and stores raw Unicode code points.
- **UTF-8 encoding** (simplified explanation):
    - ASCII characters → 8 bits
    - Many non-Latin characters → 16 bits
    - Ideograms (e.g., Chinese) → 24 bits
- A **BOM** (Byte Order Mark) indicates the encoding used (UTF-8, UCS-4, etc.).

Properties

- Python strings are **immutable sequences**: `"str" object does not support item assignment`.
- In multiline strings, the `\n` newline character counts toward `len()`.
- `chr(code_point)` → converts a code point into its character.

- `ord(character)` → converts a character into its code point.

Slicing `sequence[start : stop : step]` → `stop` is excluded. To reverse a string: `text[::-1]`.

STRING METHODS

1. Modification, analysis, and transformation

- `capitalize()` → first character uppercase, others lowercase. Returns a new string.
- `center(width, fillchar)` → returns a string of length `width` padded with `fillchar`. If `width` is too small, returns the original string.
- `count(substring)` → counts occurrences.
- `join(iterable)` → joins elements using the string as a separator. Example: `",".join(["omicron","pi","rho"]) → "omicron,pi,rho"`.
- `lower()` → lowercase version.
- `lstrip(chars)` → strips characters from the start. `"www.cisco.com".lstrip("w.") → "cisco.com"`.
- `replace(a, b, max)` → replace substring `a` with `b` up to `max` times.
- `rfind(substring)` → like `find` but searches from the end.
- `rstrip(chars)` → strips characters from the end.
- `split()` → splits on whitespace, ignoring leading/trailing whitespace.
- `strip()` → strips whitespace from both ends.
- `swapcase()` → lowercase → uppercase and uppercase → lowercase.
- `title()` → first letter of each word uppercase, rest lowercase.
- `upper()` → uppercase version.
- `find(substring, start)` → returns index or `-1` if not found.
- `"a".index("a")` → returns first index; raises `ValueError` if not found.

2. Boolean string tests

All return True or False:

- `endswith(substring)`
- `isalnum()` → letters + digits only
- `isalpha()` → letters only
- `isdigit()` → digits only
- `islower()`

- `isspace()`
- `isupper()`
- `startswith(substring)`

STRING COMPARISON RULES

- `'alpha' < 'alphabet'` (shorter prefix is smaller)
- `'beta' > 'Beta'` → lowercase characters are greater than uppercase.
- `'alpha' == 'alpha'`
- `'alpha' != 'Alpha'`
- `'10' != '010'` because `'1' > '0'` in first character comparison.
- `string == number` → always `False`.
- Uppercase < lowercase in ASCII/Unicode ordering.

EXCEPTIONS

General information

- Python 3 defines **63 built-in exceptions**.
- They form an inheritance tree. For example: `ZeroDivisionError` is a subclass of `ArithmeticError`.
- **Order of exception handlers matters**: always catch more specific exceptions before more general ones.
- `except (exc1, exc2):` is allowed.
- `raise` without specifying an exception is allowed, but **only inside an `except` block**.
- `assert x` triggers `AssertionError` if `x` is evaluated as `False`, `0`, `''`, or `None`.

IMPORTANT EXCEPTIONS

ArithmeticError Location: `BaseException ← Exception ← ArithmeticError` Description: Abstract exception for arithmetic-related errors (e.g., invalid numeric domain).

AssertionError Location: `BaseException ← Exception ← AssertionError` Description: Raised by `assert` when the tested expression is False, None, 0, or empty.

BaseException Location: `BaseException` Description: Root of all Python exceptions. `except:` is equivalent to `except BaseException:`.

IndexError Location: `BaseException ← Exception ← LookupError ← IndexError` Description: Accessing an out-of-range index in a sequence.

KeyboardInterrupt Location: `BaseException ← KeyboardInterrupt` Description: Raised when a user presses interrupt keys (e.g., Ctrl+C).

LookupError Location: `BaseException ← Exception ← LookupError` Description: Abstract parent class for errors caused by invalid lookups (indexes, keys, etc.).

MemoryError Location: `BaseException ← Exception ← MemoryError` Description: Raised when no memory is available for an operation.

OverflowError Location: `BaseException ← Exception ← ArithmeticError ← OverflowError` Description: Raised when a number is too large to store.

ImportError Location: `BaseException ← Exception ← StandardError ← ImportError` Description: Raised when an import operation fails.

KeyError Location: `BaseException ← Exception ← LookupError ← KeyError` Description: Raised when trying to access a dictionary key that does not exist.

OBJECT-ORIENTED PROGRAMMING

Procedural vs Object-Oriented Paradigm

- **Procedural approach** Divides the world into two separate domains: *the world of data* and *the world of code (functions)*.

- **Object-oriented approach** Groups data and code together into **classes**. An **object** is a combination of:

    - a set of traits (**properties / attributes**)
    - a set of behaviors (**methods**)

Classes, Objects, and Hierarchy

- A **class** is a set of objects.

- An **object** is a member (an instance) of a class.

- A class can inherit from another class:

    - **superclass** (parent)

    - **subclass** (child)

- The top-level class has **no superclass**.

- Relations between classes are shown as arrows from **subclass → superclass**.

- Creating an object is called **instantiation**; the created object is an **instance**.

Object Attributes

An object conceptually contains three groups of attributes:

1. **A name** that identifies it within its namespace (some objects may be anonymous).

2. **A set of individual properties** that describe its unique state (some objects may have none).

3. **A set of abilities (methods)** capable of:

    - modifying itself
    - interacting with other objects

Encapsulation and Private Attributes

- Any class attribute starting with `__` becomes **private**.

- It cannot be accessed directly from outside the class.

- Python uses **name mangling** to store private attributes under:

  ```
  _ClassName__attribute
  ```

  Example: `example_object_1._ExampleClass__first`

- Methods inside the class can access all properties and methods of the actual object.

Inheritance and Constructor Rules

- Python **does not automatically call** the superclass constructor.
- You must explicitly invoke it inside `__init__()`.

Example of calling overridden method from superclass:

- You must specify:

1. the **superclass name**
2. the **target object** as the first argument (because Python does not insert `self` automatically here)

This process illustrates **method overriding**: A subclass defines a method with the same name but a different implementation.

`__dict__` Attribute

- Every object stores its properties in `object.__dict__` (a dictionary).
- Example: For private attribute `__first` inside class `ExampleClass`, the object stores:

  ```
  '_ExampleClass__first': <value>
  ```

Class Variables vs Instance Variables

Example:

```python
class ExampleClass:
    counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1

example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)
example_object_3 = ExampleClass(4)
```

Outputs:

```
example_object_1.__dict__, example_object_1.counter → {'_ExampleClass__first': 1} 3
example_object_2.__dict__, example_object_2.counter → {'_ExampleClass__first': 2} 3
example_object_3.__dict__, example_object_3.counter → {'_ExampleClass__first': 4} 3
```

Key points:

- **Class variables**:
  - do **not** appear in instance `__dict__`
  - have **one shared copy** for all objects
  - are stored in the **class's own `__dict__`**

- **Instance variables**:
  - exist **only in objects**
  - every object may have different ones
  - may be private using `__name` (still accessible through name-mangled form)

Checking Attributes with `hasattr()`

To check if an object/class contains an attribute:

```
hasattr(object_or_class, "attribute_name")
```

- The second argument must be a **string**.
- Useful to avoid `AttributeError`.

AttributeError

- Raised when attempting to access an attribute that does not exist.

Modifying a Private Attribute (Name-Mangling Trick)

Even private attributes are accessible if you know their mangled name:

```
version_2._Python__venomous = not version_2._Python__venomous
```

This directly negates the private property `__venomous` of `version_2`.