Lecture 11

Object-oriented programming. Classes, inheritance and polymorphism. Modules and packages.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

05 Dec 2023

Outline

1. OOP in Python
2. Decorators
3. Inheritance and polymorphism
4. Modules and packages

Object-oriented programming in Python

Object-oriented programming in Python

We've encountered built-in data types like `dict` and `list`. However, Python allows us to define our own data types using classes. A class serves as a blueprint for creating objects, following the principles of object-oriented programming.

```
d = dict()
```

In this example, `d` is an object, while `dict` is a type.

```
type(d)
```

> dict

```
type(dict)
```

> type

We refer to `d` as an **instance** of the **type** `dict`.

The need for custom classes

Custom classes become invaluable when we need to organize and manage complex data structures efficiently. Let's illustrate this with an example involving the Advanced Programming course (`AdvProg`) members. Initially, we store information in a dictionary:

```
advprog_1 = {'first': Pasquale', 'last': 'Africa', 'email': 'pasquale.africa@sissa.it'}
```

To extract a member's full name, we define a function:

```
def full_name(first, last):
    return f"{first} {last}"
```

This approach requires repetitive code for each member:

```
advprog_2 = {'first': 'Marco', 'last': 'Feder', 'email': marco.feder@sissa.it'}
full_name(advprog_2['first'], advprog_2['last'])
```

Creating a class for efficiency

To address the inefficiency, we can create a class as a blueprint for AdvProg members:

```python
class AdvProgMember:
    pass
```

We enhance this blueprint by adding an `__init__` method to initialize instances with specific data:

```python
class AdvProgMember:
    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@sissa.it"
```

The `self`

Class methods have **only one** specific difference from ordinary functions: they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object *itself*, and by convention, it is given the name `self`.

You must be wondering how Python gives the value for `self` and why you don't need to give a value for it. An example will make this clear. Say you have a class called `MyClass` and an instance of this class called `myobject`. When you call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` - this is all the special `self` is about.

This also means that if you have a method which takes no arguments, then you still have to have one argument, i.e. the `self`.

The `__init__` method

There are many method names which have special significance in Python classes. We will see the significance of the `__init__` method now.

The `__init__` method is run as soon as an object of a class is instantiated (i.e. created). The method is useful to do any *initialization* (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores both at the beginning and at the end of the name.

We do not explicitly call the `__init__` method, but it is automatically invoked when creating an instance of a class:

```python
advprog_1 = AdvProgMember('Pasquale', 'Africa')
print(advprog_1.first)
print(advprog_1.last)
print(advprog_1.email)
```

Methods

To simplify accessing a member's full name, we integrate it as a class method:

```python
class AdvProgMember:
    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@sissa.it"

    def full_name(self): # Notice 'self' as an input argument.
        return f"{self.first} {self.last}"

advprog_1 = AdvProgMember('Pasquale', 'Africa')
print(advprog_1.full_name())
```

Class vs. object (or instance) attributes

Attributes can be instance-specific (`advprog_1.first`) or shared among all instances (`AdvProgMember.campus`). Class attributes are defined outside the `__init__` method.

```python
class AdvProgMember:
    role = "Advanced Programming member"
    campus = "SISSA"

    def __init__(self, first, last, email):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@sissa.it"

advprog_1 = AdvProgMember('Pasquale', 'Africa')
print(f"{advprog_1.first} is at campus {advprog_1.campus}.")
print(f"{advprog_1.first} is at campus {AdvProgMember.campus}.")
```

⚠️ **Changing class attributes affects all instances!**

Class methods

Besides regular methods, classes offer class methods and static methods. Class methods, identified by `@classmethod`, act on the class itself, often serving as alternative constructors.

```python
class AdvProgMember:
    @classmethod
    def from_csv(cls, csv_name):
        first, last = csv_name.split(',')
        return cls(first, last)

advprog_1 = AdvProgMember.from_csv('Pasquale,Africa')
advprog_1.full_name()
```

Static methods

Static methods, marked with `@staticmethod`, operate independently of instances and classes but are relevant to the class.

```
class AdvProgMember:
    @staticmethod
    def is_exam_date(date):
        return True if date in ["Jan 17th", "Feb 13th"] else False

print(f"Is Dec 5th an exam date? {AdvProgMember.is_exam_date("Dec 5th")}")
print(f"Is Feb 13th an exam date? {AdvProgMember.is_exam_date("Feb 13th")}")
```

Magic methods (1/3)

In Python, magic methods, also known as dunder (double underscore) methods, are special methods that start and end with double underscores. Magic methods are automatically invoked by the Python interpreter in response to certain events or operations.

Example: `__add__`

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other_point):
        return Point(self.x + other_point.x, self.y + other_point.y)

point1 = Point(1, 2)
point2 = Point(3, 4)
result = point1 + point2
```

Magic methods (2/3)

Example: `__eq__` and `__call__`

```python
class CustomObject:
    def __init__(self, value):
        self.value = value

    def __eq__(self, other):
        return self.value == other.value

    def __call__(self, *args, **kwargs):
        return f"Called with args: {args}, kwargs: {kwargs}"

obj1 = CustomObject(42)
obj2 = CustomObject(42)
print(obj1 == obj2)  # Output: True

result = obj1(1, 2, key="value")
print(result)  # Output: Called with args=(1, 2), kwargs={'key': 'value'}
```

Magic methods (3/3)

Example: `__getitem__` and `__setitem__`

```python
class MyContainer:
    def __init__(self, data):
        self.data = data

    def __getitem__(self, index):
        return self.data[index]

    def __setitem__(self, index, value):
        self.data[index] = value

container = MyContainer([1, 2, 3, 4, 5])
print(container[2])      # Output: 3
container[2] = 10
print(container[2])      # Output: 10
```

Summary of common magic methods (1/5)

Object initialization and cleanup

- `__init__(self[, ...])`: Constructor method, initializes a new instance.
- `__del__(self)`: Destructor method, called when the object is about to be destroyed.

Object representation

- `__str__(self)`: Used by `str()` and `print()` to get a human-readable string representation.
- `__repr__(self)`: Used by `repr()` and the interactive interpreter for a developer-friendly representation.
- `__format__(self, format_spec)`: Customizes the formatting when using the `format()` function.

Summary of common magic methods (2/5)

Attribute access

- `__getattr__(self, name)`: Called when an attribute lookup fails.
- `__setattr__(self, name, value)`: Called when an attribute is set.
- `__delattr__(self, name)`: Called when an attribute is deleted.

Container and iteration

- `__len__(self)`: Returns the length of the object; used by `len()`.
- `__getitem__(self, key)`: Enables indexing and slicing; used by `obj[key]`.
- `__setitem__(self, key, value)`: Enables index assignment `obj[key] = value`.
- `__delitem__(self, key)`: Enables deletion of an index; used by `del obj[key]`.
- `__iter__(self)`: Returns an iterator object; used by `iter()`.
- `__next__(self)`: Retrieves the next item from the iterator; used by `next()`.

Summary of common magic methods (3/5)

Comparison

- `__eq__(self, other)`: Defines equality; used by `==`.
- `__ne__(self, other)`: Defines non-equality; used by `!=`.
- `__lt__(self, other)`: Defines less than; used by `<`.
- `__le__(self, other)`: Defines less than or equal to; used by `<=`.
- `__gt__(self, other)`: Defines greater than; used by `>`.
- `__ge__(self, other)`: Defines greater than or equal to; used by `>=`.
- `__bool__(self)`: Defines truthiness; used by `bool()`.

Summary of common magic methods (4/5)

Mathematical operations

- `__add__(self, other)`: Defines addition; used by `+`.
- `__sub__(self, other)`: Defines subtraction; used by `-`.
- `__mul__(self, other)`: Defines multiplication; used by `*`.
- `__truediv__(self, other)`: Defines true division; used by `/`.
- `__floordiv__(self, other)`: Defines floor division; used by `//`.
- `__mod__(self, other)`: Defines modulo; used by `%`.
- `__pow__(self, other[, modulo])`: Defines exponentiation; used by `**`.

Summary of common magic methods (5/5)

Callable objects

- `__call__(self[, args[, kwargs]])`: Allows an instance to be called as a function.

Context management

- `__enter__(self)`
- `__exit__(self, exc_type, exc_value, traceback)`

  Used for resource acquisition and release in a `with` statement:

```python
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
    # File is automatically closed outside the 'with' block.
```

Notes for C++ programmers

- In Python, even integers are treated as objects (of the `int` class). This is unlike C++ where integers are primitive native type.
- The `self` in Python is equivalent to the `this` pointer in C++.
- All class members (including the data members) are *public* and all the methods are *virtual* in Python.
- If you use data members with names using the *double underscore prefix* such as `__myvar`, Python uses name-mangling to effectively make it (almost) a private variable. Any identifier of the form `__myvar` (at least two leading underscores or at most one trailing underscore) is replaced with `_MyClass__myvar`, where `MyClass` is the current class name with a leading underscore(s) stripped. After all, private members can still be accessed... You can check using the built-in `dir()` function.

Decorators

Decorators

Decorators in Python offer a powerful way to enhance the functionality of functions or methods. They act as wrappers, allowing you to extend or modify the behavior of the original function. Let's delve deeper into decorators with examples and explore their practical applications.

Decorators can be imagined to be a shortcut to calling a wrapper function (i.e. a function that "wraps" around another function so that it can do something before or after the inner function), so applying the `@classmethod` decorator is the same as calling:

```
from_csv = classmethod(from_csv)
```

Defining decorators

Decorators are essentially functions that take another function as input, enhance its capabilities, and return a modified version of the original function.

```python
def my_decorator(original_func):
    def wrapper():
        print(f"A decoration before {original_func.__name__}.")
        result = original_func()
        print(f"A decoration after {original_func.__name__}.")
        return result
    return wrapper

my_decorator(original_func)()

# Or, re-assigning the original symbol:
original_func = my_decorator(original_func)
original_func()
```

NB: `__name__` is a special attribute that returns the name of a function, class or module as a string.

Improved syntax using `@`

While the previous example works, Python provides a more readable syntax using the `@` symbol. The equivalent of the previous example using this syntax is:

This decorator, when applied to a function, surrounds the function call with additional actions. For instance:

```python
@my_decorator
def original_func():
    print("I'm the original function!")

original_func()
```

> A decoration before original_func.
>
> I'm the original function!
>
> A decoration after original_func.

Practical example: timer decorator (1/2)

Now, let's create a more practical decorator that measures the execution time of a function. This example utilizes the `time` module:

```python
import time

def timer(my_function):
    def wrapper():
        t1 = time.time()
        result = my_function()
        t2 = time.time()
        print(f"{my_function.__name__} ran in {t2 - t1:.3f} sec")
        return result
    return wrapper
```

(More details about `import` wil follow).

Practical example: timer decorator (2/2)

Applying this decorator to a function allows us to measure its execution time:

```python
@timer
def silly_function():
    for i in range(1e7):
        if (i % 1e6) == 0:
            print(i)

silly_function()
```

> 0
>
> 1000000
>
> 2000000
>
> ...
>
> 9000000
>
> silly_function ran in 0.601 sec

Decorators and classes (1/2)

A decorator can be applied to classes as well:

```python
def add_method(cls):
    def new_method(self):
        return f"Hello from the new method of {cls.__name__}!"

    cls.new_method = new_method

    return cls

@add_method
class MyClass:
    def existing_method(self):
        return "Hello from the existing method!"

obj = MyClass()
result_new = obj.new_method()
```

Decorators and classes (2/2)

... or be a class itself:

```python
class CustomDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f"Decorating function {self.func.__name__}")
        result = self.func(*args, **kwargs)
        print(f"Function {self.func.__name__} finished execution")
        return result

@CustomDecorator
def my_function():
    print("Executing my_function")

my_function()
```

Built-in decorators

Python comes with built-in decorators like `classmethod` and `staticmethod`, which are implemented in C for efficiency. Although we won't dive into their implementation, they are widely used in practice.

Other than logging and timing/profiling, decorators can be used to **add validation checks for input parameters** and **output cleanup** to functions or methods, or to implement **caching** mechanisms, where the result of a function is stored for a specific set of inputs, and subsequent calls with the same inputs can return the cached result.

In conclusion, decorators provide a flexible and elegant way to enhance the behavior of functions in Python. While creating custom decorators may not be a daily necessity, understanding them is crucial for leveraging Python's full potential.

Inheritance and polymorphism

Inheritance and subclasses (1/4)

Inheritance in Python enables classes to inherit methods and attributes from other classes. Previously, we worked with the `AdvProgMember` class, but now let's delve into creating more specialized classes like `AdvProgStudent` and `AdvProgInstructor`.

```python
class AdvProgMember:
    # ...
```

Now, to create an `AdvProgStudent` class inheriting from `AdvProgMember`:

```python
class AdvProgStudent(AdvProgMember):
    pass
```

Inheritance and subclasses (2/4)

Creating instances of `AdvProgStudent` and accessing inherited methods:

```
student_1 = AdvProgStudent('Craig', 'Smith')
student_2 = AdvProgStudent('Megan', 'Scott')
print(student_1.full_name())
print(student_2.full_name())
```

Here, `AdvProgStudent` inherits methods like `full_name()` from `AdvProgMember`.

To fine-tune the `AdvProgStudent` class, we adjust attributes:

```
class AdvProgStudent(AdvProgMember):
    role = "AdvProg student"
```

Inheritance and subclasses (3/4)

Now, creating a student instance reflects the updated role:

```python
student_1 = AdvProgStudent('John', 'Smith')
print(student_1.role)
```

Adding an *instance attribute* like `grade` using `super()`, or the base class name:

```python
class AdvProgStudent(AdvProgMember):
    role = "AdvProg student"

    def __init__(self, first, last, grade):
        # super().__init__(first, last) # Or the following:
        AdvProgMember.__init__(self, first, last)
        self.grade = grade

student_1 = AdvProgStudent('John', 'Smith', 28)
```

Inheritance and subclasses (4/4)

Creating another subclass, `AdvProgInstructor`, with additional methods:

```python
class AdvProgInstructor(AdvProgMember):
    role = "AdvProg instructor"

    def __init__(self, first, last, students=None):
        super().__init__(first, last)
        self.students = ([] if students is None else students)

    def add_student(self, student):
        self.students.append(student)

    def remove_course(self, student):
        self.students.remove(student)

instructor_1 = AdvProgInstructor('Pasquale', 'Africa')
instructor_1.add_student(student1)
instructor_1.add_student(student2)
instructor_1.remove_student(student1)
```

How inheritance works

To use inheritance, we specify the base class names in a tuple following the class name in the class definition (for example, `class Teacher(SchoolMember)`).

Next, we observe that the `__init__` method of the base class is explicitly called using the `self` variable so that we can initialize the base class part of an instance in the subclass. **This is very important to remember**.

Since we are defining a `__init__` method in `AdvProgStudent` and `AdvProgInstructor` subclasses, Python does not automatically call the constructor of the base class `AdvProgMember`, you have to explicitly call it yourself.

In contrast, if we have not defined an `__init__` method in a subclass, Python will call the constructor of the base class automatically.

Getters, setters, deleters

For effective class management, Python provides getters, setters, and deleters. Consider the former `AdvProgInstructor` class:

```python
class AdvProgInstructor(AdvProgMember):
    role = "AdvProg instructor"

    # ...
```

Instances of `AdvProgMember` can be created and accessed:

```python
advprog_1 = AdvProgMember('Pasqulae', 'Africa') # Typo!
                           ^^^^^^^^
print(advprog_1.first)
print(advprog_1.last)
print(advprog_1.email)
print(advprog_1.full_name())
```

The `@property` decorator (1/2)

Imagine that I mis-spelled the name of the first name and wanted to correct it. Watch what happens.

```python
advprog_1.first = 'Pasquale'
print(advprog_1.first)
print(advprog_1.last)
print(advprog_1.email) # Still prints pasqulae.africa@sissa.it!
print(advprog_1.full_name())
```

Utilizing a `@property` decorator defines `email` like a method, but keeps it as an **attribute**:

```python
class AdvProgMember:
    # ...

    @property
    def email(self):
        return self.first.lower() + "." + self.last.lower() + "@sissa.it"
```

The `@property` decorator (2/2)

Now, changes to the `first` name reflect in the `email`:

```python
advprog_1 = AdvProgMember('Pasqulae', 'Africa')
advprog_1.first = 'Pasquale'
print(advprog_1.first)
print(advprog_1.last)
print(advprog_1.email) # Now the correct value is printed.
print(advprog_1.full_name())
```

We could do the same with the `full_name()` method:

```python
class AdvProgMember:
    # ...

    @property
    def full_name(self):
        return f"{self.first} {self.last}"
```

Setter methods (1/2)

Introducing a `full_name` setter to update `first` and `last`:

But what happens if we instead want to make a change to the full name now?

```
advprog_1.full_name = 'Pasquale Africa'
```

> AttributeError: can't set attribute

We get an error: class instance doesn't know what to do with the value it was passed. Ideally, we'd like our class instance to use this full name information to update `self.first` and `self.last`.

Setter methods (2/2)

To handle this action, we need a `setter`, defined using the decorator `@<attribute>.setter`:

```python
class AdvProgMember:
    # ...

    @full_name.setter
    def full_name(self, name):
        first, last = name.split(' ')
        self.first = first
        self.last = last
```

Setting the `full_name` now updates the attributes:

```python
advprog_1 = AdvProgMember('X', 'Y')
advprog_1.full_name = 'Pasquale Africa'
```

Deleters

We've talked about getting information and setting information, but what about deletting information? This is typically used to do some clean up and is defined with the `@<attribute>.deleter` decorator.

```python
class AdvProgMember:
    # ...

    @full_name.deleter
    def full_name(self):
        print('Name deleted!')
        self.first = None
        self.last = None
```

Deleting the `full_name` attribute results in a cleanup:

```python
advprog_1 = AdvProgMember('Pasquale', 'Africa')
delattr(advprog_1, "full_name")
```

Modules and packages

Modules

Modules: reusable code in Python

In Python, the ability to reuse code is facilitated by modules. A module is a file with a `.py` extension that contains functions and variables. There are various methods to write modules, including using languages like C to create compiled modules.

When importing a module, to enhance import performance, Python creates byte-compiled files (`__pycache__/filename.pyc`). These files, platform-independent and located in the same directory as the corresponding `.py` files, speed up subsequent imports by storing preprocessed code.

Using Standard Library Modules

You can import modules in your program to leverage their functionality. For instance, consider the `sys` module in the Python standard library. Below is an example:

```python
# Example: module_using_sys.py
import sys

print("Command line arguments:", sys.argv)
```

When executed, this program prints the command line arguments provided to it. The `sys.argv` variable holds these arguments as a list. For instance, running `python module_using_sys.py we are arguments` results in `sys.argv[0]` being `'module_using_sys.py'`, `sys.argv[1]` being `'we'`, `sys.argv[2]` being `'are'`, and `sys.argv[3]` being `'arguments'`.

The `from... import...` Statement

You can selectively import variables from a module using the `from... import...` statement. However, it's generally advised to use the `import` statement to avoid potential name clashes and enhance readability.

```python
from math import sqrt
print("Square root of 16 is", sqrt(16))
```

A special case is `from math import *`, where all symbols exported by the `math` module are imported.

A module's `__name__`

Every module has a `__name__` attribute that indicates whether the module is being run standalone or imported. If `__name__` is `'__main__'`, the module is being run independently.

```python
# Example: module_using_name.py
if __name__ == '__main__':
    print("This module is being run independently.")
```

Creating your own modules

Creating modules is straightforward: every Python program is a module!

Save it with a `.py` extension. For example:

```python
# Example: mymodule.py
def say_hi():
    print("Hello, this is mymodule speaking.")

__version__ = '1.0'
```

Now, you can use this module in another program:

```python
# Example: mymodule_demo.py
import mymodule

mymodule.say_hi()
print("Version:", mymodule.__version__)
```

The `dir` function

The built-in `dir()` function lists all symbols defined in an object. For a module, it includes functions, classes, and variables. It can also be used without arguments to list names in the current module.

```python
# Example: Using the dir function.
import sys

# Names in sys module.
print("Attributes in sys module:", dir(sys))

# Names in the current module.
print("Attributes in current module:", dir())
```

Packages

Packages: organizing modules hierarchically

Packages are folders of modules with a special `__init__.py` file, indicating that the folder contains Python modules. They provide a hierarchical organization for modules.

```
<some folder in sys.path>/
└── datascience/
    ├── __init__.py
    ├── preprocessing/
    │   ├── __init__.py
    │   ├── cleaning.py
    │   └── scaling.py
    └── analysis/
        ├── __init__.py
        ├── statistics.py
        └── visualization.py
```

The `__init__.py` files (1/5)

The `__init__.py` file in a Python package serves multiple purposes. It's executed when the package or module is imported, and it can contain initialization code, set package-level variables, or define what should be accessible when the package is imported using `from package import *`.

Here are some common examples of using `__init__.py` files.

The `__init__.py` files (2/5)

1. Initialization code

```python
# __init__.py in a package.

# Initialization code to be executed when the package is imported.
print("Initializing my_package...")

# Define package-level variables.
package_variable = 42

# Import specific modules when the package is imported.
from . import module1
from . import module2
```

In this example, the `__init__.py` file initializes the package, sets a package-level variable (`package_variable`), and imports specific modules from the package.

The `__init__.py` files (3/5)

2. Controlling `from package import *`

```python
# __init__.py in a package.

# Define what should be accessible when a user writes 'from package import *'.
__all__ = ['module1', 'module2']

# Import modules within the package.
from . import module1
from . import module2
```

By specifying `__all__`, you explicitly control what is imported when using `from package import *`. It's considered good practice to avoid using `*` imports, but if you need to, this can help manage what gets imported.

The `.` symbol means that `module1.py` and `module2.py` are to be located in the same folder as the `__init.py__` file.

The `__init__.py` files (4/5)

3. Lazy loading

```python
# __init__.py in a package.

# Initialization code.
print("Initializing my_lazy_package...")

# Import modules only when they are explicitly used.
def lazy_function():
    from . import lazy_module
    lazy_module.do_something()
```

In this example, the module is initialized only when the `lazy_function` is called. This can be useful for performance optimization, especially if some modules are rarely used.

The `__init__.py` files (5/5)

4. Setting package-level configuration

```python
# __init__.py in a package.

# Configuration settings for the package.
config_setting1 = 'value1'
config_setting2 = 'value2'
```

You can use the `__init__.py` file to set package-level configuration settings that can be accessed by modules within the package.

Python modules as 1. scripts vs. 2. pre-compiled libraries

In Python, modules and packages can be implemented either as Python scripts or as pre-compiled dynamic libraries. Let's explore both concepts:

1. **Python modules as scripts:**
    - **Extension:** Modules implemented as scripts usually have a `.py` extension.
    - **Interpretation:** The Python interpreter reads and executes the script line by line.
    - **Readability:** Scripts are human-readable and editable using a text editor.
    - **Flexibility:** This is the most common form of Python modules. You can write and modify the code easily.
    - **Portability:** Python scripts can be easily shared and run on any system with a compatible Python interpreter.

2. Python modules as dynamic libraries

- **Compilation:** Modules can be pre-compiled into shared libraries for performance optimization.
- **Execution:** The compiled code is loaded into memory and executed by Python.
- **Protection of intellectual property:** Pre-compiled modules can be used to distribute proprietary code without exposing the source.
- **Performance:** Pre-compiled modules may offer better performance as they are already in machine code.

It's essential to note that Python itself is an interpreted language, and even when using pre-compiled modules, the Python interpreter is still involved in executing the code. The use of pre-compiled modules is more about optimizing performance and protecting source code than altering the fundamental nature of Python as an interpreted language.

You can use tools like Cython or PyInstaller to generate pre-compiled modules or standalone executables, respectively, depending on your specific use case and requirements.

The Python Standard Library

Python’s standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed on the website. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

In addition to the standard library, there is an active collection of hundreds of thousands of components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.

Summary

Packages are a convenient way to organize modules hierarchically, often seen in the Python standard library.

In summary, modules and packages enhance code reusability in Python. The standard library showcases the power of these concepts, and creating your own modules and packages can significantly improve code organization and maintainability.

Next, we will delve into common Python packages for scientific computing, namely NumPy, SciPy, matplotlib/seaborn and pandas.

Zen of Python:
"Explicit is better than implicit."
Run `import this` in Python to learn more.

➡️ Introduction to NumPy and SciPy for scientific computing. Plotting. Introduction to pandas for data analysis.