# Lecture 11

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

**Advanced Programming - SISSA, UniTS, 2025-2026**

**Pasquale Claudio Africa**

**02 Dec 2025**

# Outline

1. OOP in Python
2. Inheritance and polymorphism
3. Decorators, getters, setters, and deleters
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:

```
advprog1 = {'first': 'Pasquale', 'last': 'Africa', 'email': 'pafrica@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. Furthermore, we have no validation, no methods and it's easy to make mistakes (e.g., typing `frist` instead of `first` ).

# Creating a class for efficiency

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

```
class AdvProgMember:
    pass
```

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

```
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)`, with `myobject` becoming the `self`.

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:

```
advprog1 = AdvProgMember('Pasquale', 'Africa')
print(advprog1.first)
print(advprog1.last)
print(advprog1.email)
```

# Methods

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

```
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}"

advprog1 = AdvProgMember('Pasquale', 'Africa')
print(advprog1.full_name())
```

# Class vs. object (or instance) attributes

Attributes can be instance-specific (`advprog1.first`) or shared among all instances (`AdvProgMember.campus`). Class attributes are defined outside the `__init__` method.

```
class AdvProgMember:
    role = "Advanced Programming member"
    campus = "SISSA" # Shared by all instances!

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

AdvProgMember.campus = "UniTS" # WARNING: This affects all instances!
advprog1 = AdvProgMember('Pasquale', 'Africa')
print(f"{advprog1.first} is at campus {advprog1.campus}.")
print(f"{advprog1.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` (which is a *decorator*, more on decorators later), act on the class itself, often serving as alternative constructors.

```
class AdvProgMember:
    @classmethod
    def from_csv(cls, csv_name):
        first, last = csv_name.split(',')
        return cls(first, last) # Create a new instance of the class and return it.

advprog1 = AdvProgMember.from_csv('Pasquale,Africa')
advprog1.full_name()
```

# Static methods

Static methods, marked with `@staticmethod` (another decorator), 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 16th", "Feb 13th"] else False

print(f"Is Dec 2nd an exam date? {AdvProgMember.is_exam_date('Dec 2nd')}")
print(f"Is Feb 13th an exam date? {AdvProgMember.is_exam_date('Feb 13th')}")
```

# Magic methods (1/4)

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.

# Magic methods (2/4)

**Example: `__add__`, `__str__`, and `__repr__`**

```
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)
    def __str__(self):
        return f"({self.x}, {self.y})"
    def __repr__(self):
        return f"Point({self.x}, {self.y})"

point1 = Point(1, 2)
point2 = Point(3, 4)
result = point1 + point2 # Call point1.__add__(point2).
print(point1)            # Call point1.__str__(): (1, 2).
print(repr(point2))      # Call point1.__repr__(): Point(3, 4).
```

# Magic methods (3/4)

**Example: `__eq__` and `__call__`**

```
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 (4/4)

**Example: `__getitem__` and `__setitem__`**

```
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:

```
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 internally replaced with `_MyClass__myvar`, where `MyClass` is the current class name with leading underscores stripped. After all, private members can still be accessed... You can check using the built-in `dir()` function.

# 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`.

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

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

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

# Inheritance and subclasses (2/4)

Creating instances of `AdvProgStudent` and accessing inherited methods:

```
student1 = AdvProgStudent('Craig', 'Smith')
student2 = AdvProgStudent('Megan', 'Scott')
print(student1.full_name())
print(student2.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:

```
student1 = AdvProgStudent('John', 'Smith')
print(student1.role)
```

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

```
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

student1 = AdvProgStudent('John', 'Smith', 28)
```

# Inheritance and subclasses (4/4)

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

```
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_student(self, student):
        self.students.remove(student)

instructor1 = AdvProgInstructor('Pasquale', 'Africa')
instructor1.add_student(student1)
instructor1.add_student(student2)
instructor1.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 do not define an `__init__` method in the subclass, Python calls the constructor of the base class automatically.

# Method Resolution Order (MRO)

**MRO** determines the order in which Python searches for methods and attributes in a class hierarchy, critical for multiple inheritance, such as in the *diamond problem*:

```
class A:
    def method(self):
        return "A"

class B(A):
    def method(self):
        return "B"
class C(A):
    def method(self):
        return "C"

class D(B, C):
    pass

d = D()
print(d.method()) # Multiple inheritance: which method() is called?
```

# Viewing the MRO

```
print(D.mro())
# [<class 'D'>, <class 'B'>, <class 'C'>, <class 'A'>, <class 'object'>]
# D searches: itself → B → C → A → object

print(D.__mro__). # Alternative syntax.
```

Python (since v2.3) uses the **C3 linearization** algorithm to compute MRO, ensuring:

1. **Children before parents**: A subclass is always checked before its parent classes.
2. **Left-to-right order preserved**: `class D(B, C)` checks B before C.
3. **Monotonicity**: If A comes before B in a parent's MRO, A comes before B in all subclasses.
4. **Consistency**: Python raises `TypeError` if no valid ordering exists.

# Polymorphism

Python doesn't require explicit inheritance for polymorphism. If an object has the required method, it can be used:

```
def make_it_speak(animal):
    print(animal.speak())

class Dog:
    def speak(self):
        return "Woof!"
class Robot:
    def speak(self):
        return "Beep boop!"

# Both work, even though they're unrelated classes.
make_it_speak(Dog())
make_it_speak(Robot())
```

Python checks for method existence at runtime, not class hierarchy at compile time.

**Duck typing philosophy:** *"If it walks like a duck and quacks like a duck, it's a duck."*

# Abstract Base Classes (ABC)

For more formal polymorphism, Python provides Abstract Base Classes:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        """Must be implemented by subclasses."""
        pass

class Rectangle(Shape):
    # ...

    def area(self):
        return self.width * self.height

shape = Shape() # TypeError: Can't instantiate abstract class

# In order to be instantiable, a class must implement all abstract methods.
rect = Rectangle(5, 3)
```

# OOP: C++ vs. Python

| Aspect | C++ | Python |
|---|---|---|
| **Access control** | `public`, `private`, `protected` (enforced by compiler) | Convention-based: `__private`, (not enforced) |
| **Constructors** | Multiple constructors via overloading | Single `__init__` with default args + `@classmethod` for alternatives |
| **Destructors** | `~ClassName()` called deterministically (RAII) | `__del__()` called by Garbage Collector (unreliable timing) |
| **Operator overloading** | `operator+`, `operator<<`, etc. | `__add__`, `__str__`, `__repr__`, etc. (magic methods) |
| | Explicit access specifier required | |

**Key philosophical difference:** C++ enforces encapsulation and type safety at compile time; Python trusts developers to follow conventions (*"we're all consenting adults here"*).

# Decorators, getters, setters, and deleters

# 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)
```

# Getters, setters, deleters

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

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

    # ...
```

Instances of `AdvProgMember` can be created and accessed:

```
advprog1 = AdvProgMember('Pasqulae', 'Africa') # Typo!
                         ^^^^^^^^
print(advprog1.first)
print(advprog1.last)
print(advprog1.email)
print(advprog1.full_name())
```

# Getters: the `@property` decorator (1/2)

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

```
advprog1.first = 'Pasquale'
print(advprog1.first)
print(advprog1.last)
print(advprog1.email) # Still prints pasqulae.africa@sissa.it!
print(advprog1.full_name())
```

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

```
class AdvProgMember:
    # ...

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

# Getters: the `@property` decorator (2/2)

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

```
advprog1 = AdvProgMember('Pasqulae', 'Africa')
advprog1.first = 'Pasquale'
print(advprog1.first)
print(advprog1.last)
print(advprog1.email) # Now the correct value is printed. Also, no parentheses needed!
print(advprog1.full_name())
```

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

```
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?

```
advprog1.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`:

```
class AdvProgMember:
    # ...

    @full_name.setter
    def full_name(self, name):
        # We could add validation here!
        first, last = name.split(' ')
        self.first = first
        self.last = last
```

Setting the `full_name` now updates the attributes:

```
advprog1 = AdvProgMember('X', 'Y')
advprog1.full_name = 'Pasquale Africa'
```

# Deleters

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

```
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:

```
advprog1 = AdvProgMember('Pasquale', 'Africa')
delattr(advprog1, "full_name")
```

# Defining decorators

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

```
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()
```

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

# Improved syntax using `@`

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

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

```
@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:

```
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` will follow).

# Practical example: timer decorator (2/2)

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

```
@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:

```
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()
obj.existing_method() # Output: 'Hello from the existing method!'
obj.new_method()      # Output: 'Hello from the new method of MyClass!'
```

# Decorators and classes (2/2)

... or be a class itself:

```
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.

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

# 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 also other 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.cpython-ver.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:

```
# 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]`: `'module_using_sys.py'`
- `sys.argv[1]`: `'we'`
- `sys.argv[2]`: `'are'`
- `sys.argv[3]`: `'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.

```
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.

```
# mymodule.py
def helper_function():
    return "I'm a helper"

def main():
    print("Running as main program")
    print(helper_function())

if __name__ == '__main__':
    # This runs only when executed directly, not when imported.
    main()
```

# Creating your own modules

Creating modules is straightforward: every Python program is a module! Just save it with a `.py` extension. For example:

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

__version__ = '1.0'
```

Now, you can use this module in another program:

```
# Example: mymodule_demo.py
import mymodule

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

# A good module structure

```
# mymodule.py

"""
Module docstring: what this module does.
"""

# Imports.
import sys
from typing import List

# Constants.
DEFAULT_VALUE = 42

# Private helpers (convention: leading underscore).
def _internal_helper():
    """Function docstring."""
    pass

# Public interface.
def public_function():
    """Function docstring."""
    pass

# Module-level code (if needed).
if __name__ == '__main__':
    # Testing code.
    pass
```

# 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 all names in the current module.

For instance:

```
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
```

# Installing and managing packages with pip

`pip` is Python's package installer, used to download and manage packages from the Python Package Index (PyPI).

### Basic pip commands

```
pip install numpy              # Install a package.
pip install pandas==2.1.0      # Install a specific version.
pip install matplotlib>=3.7.0 # Install minimum version.

pip install --upgrade scipy # Upgrade a package.

pip uninstall requests # Uninstall a package.

pip show numpy # Show package information.

pip list # List installed packages.
```

# Virtual environments

```
python -m venv myenv # Create a virtual environment.
source myenv/bin/activate # Activate it.

pip install [...]

# Save packages from current environment to requirements.txt.
pip freeze > requirements.txt

deactivate # Deactivate the environment.
```

```
# The requirements.txt file contains a list formatted like:
numpy==1.24.3
pandas>=2.0.0,<3.0.0
scikit-learn~=1.3.0 # Compatible release (>= 1.3.0, < 1.4.0).

# Now other user can install all packages from requirements.txt.
pip install -r requirements.txt
```

**Best practice:** Always use virtual environments to isolate dependencies and avoid conflicts!

# The `__init__.py` files (1/4)

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 (2/4)

## 1. Initialization code

```
# __init__.py in a package.

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

# Define package-level variables and configurations.
package_variable = 42

# Import specific modules when the package is imported.
from . import module1 # '.' means from the same folder as __init__.py.
from . import module2
```

In this example, the `__init__.py` file initializes the package, sets a package-level variables and configurations, and imports other modules from the package.

# The `__init__.py` files (3/4)

## 2. Controlling `from package import *`

```
# __init__.py in a package.

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

# 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 `__init__.py` files (4/4)

## 3. Lazy loading

```
# __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.

# How Python loads modules

When running `import mymodule`, Python searches in this order:

1. Built-in modules.
2. Current directory.
3. Directories in the `PYTHONPATH` environment variable.
4. System directories.
5. Site-packages (`pip` installed packages).

# Modules as scripts vs. pre-compiled libraries (1/2)

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.

# Modules as scripts vs. pre-compiled libraries (2/2)

2. **Python modules as dynamic libraries:**
    - **Compilation:** Modules can be pre-compiled into shared libraries for performance.
    - **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.

The use of pre-compiled modules is about optimizing performance and protecting source code, rather than altering the fundamental nature of Python as an interpreted language. You can use tools like `Cython`, `PyInstaller`, or `cx_Freeze` to generate pre-compiled modules or standalone executables, respectively, depending on specific use cases and requirements.

# Python modules: comparison

| Type | Extension | Description | Use Case |
|---|---|---|---|
| **Python script** | `.py` | Human-readable source code | Development, teaching, open-source distribution |
| **Python bytecode** | `.pyc` | Compiled bytecode (in `__pycache__/`) | Automatic optimization, faster loading |
| **C/C++ extension** | `.so` (Linux/Mac) `.pyd` (Windows) | Compiled binary from C/C++/Cython | Performance-critical code, system interfaces |
| **Wheel package** | `.whl` | Distribution format (zip archive) | Package distribution via PyPI |

# Example: NumPy's hybrid structure

```
numpy/
├── __init__.py # Python code.
├── core/
│   ├── _multiarray_umath.cpython-311-x86_64-linux-gnu.so # C extension.
│   └── multiarray.py # Python wrapper.
└── linalg/
    ├── linalg.py # Python interface.
    └── _umath_linalg.cpython-311-x86_64-linux-gnu.so # C extension.
```

**Key point:** Most scientific Python packages (NumPy, SciPy, pandas) combine Python scripts for ease of use with C extensions for performance. The `.so` / `.pyd` files contain optimized C/C++ code that Python can call directly.

# Summary

- Classes organize data and behavior; use `__init__` for initialization, `self` to reference instances.
- Python favors convention over enforcement: `__private` is a guideline, not a rule.
- Magic methods (`__add__`, `__str__`, `__eq__`) enable operator overloading.
- Use `@classmethod` for alternative constructors, `@staticmethod` for utilities, `@property` for computed attributes.
- With inheritance, Python uses MRO to resolve method calls in complex hierarchies.
- Modules (`.py` files) and packages (directories with `__init__.py`) organize code hierarchically.
- Use virtual environments (`venv` or `conda`) and `requirements.txt` to manage project dependencies.
- High-performance libraries combine Python interfaces with compiled extensions.

# Integrating C++ and Python codes.