

Inside Python

Modules, Lambdas, Decorators, Iterators and Regular Expressions

1. Modules

Each file in python is referred to as a module which can be imported in a different file.

Import fibo imports all functions from the [fibo.py](#) file and we can use functions from the file by prefixing with the fibo. file name such as fibo.fib1(1)

```
>>>fibo.__name__  
'Fibo'
```

which indicates fibo is not the main file but it is the imported file.

from fibo import * imports all functions from [fibo.py](#) and we can use functions from the file without prefixing with any prefix

For example here:-

We can import as

```
> from . import echo  
> from .. import formats  
> from ..filters import equalizer
```

sound/	Top-level package
__init__.py	Initialize the sound package
formats/	Subpackage for file format conversions
__init__.py	
wavread.py	
wavwrite.py	
aiffread.py	
aiffwrite.py	
auread.py	
auwrite.py	
...	
effects/	Subpackage for sound effects
__init__.py	
echo.py	
surround.py	
reverse.py	

```

...
filters/                               Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

The built-in function [`dir\(\)`](#) is used to find out which names a module defines. It returns a sorted list of strings:

```

import fibo, sys
dir(fibo)

['__name__', 'fib', 'fib2']

```

The `dir()` function in Python lists all the attributes and methods available for an object. It's a helpful tool for exploration and debugging.

Basic usage:

```

python
# List attributes of a string
dir("hello")

# List attributes of a list
dir([1, 2, 3])

# List attributes of a custom object
class MyClass:
    def __init__(self):
        self.value = 42

    def my_method(self):
        pass

obj = MyClass()
dir(obj)

```

What it returns:

`dir()` returns a list of strings representing attribute and method names. This includes built-in methods (like `__init__`, `__str__`), user-defined methods, and properties.

Common use cases:

Exploring objects. When you're working with an unfamiliar library or object, `dir()` shows you what you can do with it.

```
python
import math

dir(math) # See all available functions in the math module
```

Finding available methods. Quickly check what methods are available on a string, list, or other object without looking at documentation.

```
python
s = "hello"

print(dir(s)) # Shows methods like upper(), lower(), split(), etc.
```

Debugging. When something isn't working as expected, `dir()` helps you verify that an attribute or method actually exists.

WHAT ARE @staticmethod and @classmethod in python?

These are decorators that define how methods behave in relation to the class and its instances.

@staticmethod

A static method doesn't receive the class or instance as an argument. It's basically a regular function that happens to live inside a class namespace.

```
class MyClass:
    @staticmethod
    def static_method(x, y):
        return x + y

# Call it on the class or instance - doesn't matter
MyClass.static_method(3, 5) # Returns 8
obj = MyClass()
obj.static_method(3, 5)      # Also returns 8
```

Use static methods when you have utility functions related to the class but that don't need access to instance or class data.

@classmethod

A class method receives the class itself as the first argument (conventionally named `cls`). It can access and modify class-level data.

```
class MyClass:
    count = 0

    def __init__(self):
        MyClass.count += 1

    @classmethod
    def get_count(cls):
        return cls.count

obj1 = MyClass()
obj2 = MyClass()
MyClass.get_count() # Returns 2
```

Class methods are commonly used for alternative constructors or operations that affect the entire class.

Quick Comparison

Feature	Instance Method	@staticmethod	@classmethod
First argument	<code>self</code>	None	<code>cls</code>
Access instance data	Yes	No	No
Access class data	Yes	No	Yes
Can modify class state	Yes	No	Yes

Instance methods are the default and most common. Use **@staticmethod** for utility functions, and **@classmethod** when you need to work with class-level data or create alternative constructors.

What will happen if i pass self as an argument in static method?

If you pass `self` as an argument to a static method, it will just be treated as a regular parameter—nothing special happens. The decorator doesn't enforce anything; it just changes how the method is called.

python

```
class MyClass:
    @staticmethod
    def static_method(self, x):
        print(f"self: {self}, x: {x}")

obj = MyClass()

# You have to manually pass self
obj.static_method(obj, 5) # Prints: self: <__main__.MyClass object at ...>, x: 5

# Or pass anything else
MyClass.static_method("hello", 5) # Prints: self: hello, x: 5
```

Notice the difference from a regular instance method:

```
class MyClass:
    def instance_method(self, x):
        print(f"self: {self}, x: {x}")

obj = MyClass()

# self is passed automatically
obj.instance_method(5) # Prints: self: <__main__.MyClass object at ...>, x: 5

# This would fail - you can't call it this way on the class
# MyClass.instance_method(5) # TypeError
```

In short: with `@staticmethod`, `self` becomes just another regular parameter you have to pass manually. **It's not automatically bound like it is with instance methods.** This is why static methods are generally pointless if you're planning to pass `self`—you'd just want a regular instance method instead.

SOME OTHER COMMON DECORATORS IN PYTHON

@property

Lets you access a method like an attribute without calling it.

python

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

p = Person("Alice")
print(p.name)      # Calls the getter, prints "Alice"
p.name = "Bob"     # Calls the setter
```

@functools.wraps

Used when creating wrapper functions or decorators. Preserves the original function's metadata.

```
import functools

def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Before")
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def greet(name):
    """Greets someone"""
```

```
    return f"Hello {name}"

print(greet.__name__) # Prints "greet" (not "wrapper")
```

@abstractmethod

Used in abstract base classes to define methods subclasses must implement.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof"
```

@deprecated

Marks a function as deprecated to warn users.

```
import warnings

def deprecated(func):
    def wrapper(*args, **kwargs):
        warnings.warn(f"{func.__name__} is deprecated", DeprecationWarning)
        return func(*args, **kwargs)
    return wrapper

@deprecated
def old_function():
    pass
```

@timing

A custom decorator example for measuring execution time.

```
import time
import functools

def timing(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.time() - start:.4f}s")
        return result
    return wrapper

@timing
def slow_function():
    time.sleep(1)
```

VIRTUAL FUNCTIONS IN C++

In Python, the equivalent of virtual functions in C++ are simply methods that can be overridden in subclasses — because all methods in Python are virtual by default.

```
class Base:
    def speak(self):
        print("Base speaking")
class Derived(Base):
    def speak(self):
        print("Derived speaking")
obj = Derived()
obj.speak() # Derived speaking
```

- ◆ To enforce overriding (optional)

If you want to make a method *must be overridden* (like a pure virtual function in C++), you can use Python's `abc` (Abstract Base Class) module.

What is python's abstract base class module (ABC)?

How to import?

```
from abc import ABC, abstractmethod
class Base(ABC):
    @abstractmethod
    def speak(self):
        pass
class Derived(Base):
    def speak(self):
        print("Derived speaking")
obj=Derived()
obj.speak() -> Prints Derived speaking
```

What is an ABC?

An abstract base class is a class that cannot be instantiated directly. It exists to be inherited from, and it defines methods that subclasses must implement. Think of it as a contract—"if you inherit from me, **you must implement these methods.**"

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

    @abstractmethod
    def move(self):
        pass
```

```
    def describe(self):
        return "This is an animal"

# This raises TypeError - can't instantiate abstract class
# animal = Animal()

class Dog(Animal):
    def make_sound(self):
        return "Woof"

    def move(self):
        return "Running on four legs"

# This works - all abstract methods implemented
dog = Dog()
print(dog.make_sound()) # Woof
```

If you try to create a subclass without implementing all abstract methods, you get an error:

```
class Incomplete(Animal):
    def make_sound(self):
        return "Some sound"
    # Forgot to implement move()

# TypeError: Can't instantiate abstract class Incomplete
# with abstract method move
incomplete = Incomplete()
```

UNDERSTANDING SAMPLE CODE

EXAMPLE 1 -

```
class TimestampExtension:
    """An implementation of the Timestamp extension.

    This extension implementation illustrates several ways for an extension
    to provide functionality to agent developers. In general, the support
    methods range from totally hands off, where all responsibility for using the
    extension correctly is left to the developer, to totally hands-on, where
    the developer sets up strategic decorators for core classes which then
    manage implementing the extension logic. Each of the methods have
    comments indicating the level of support they provide.
    """

    def __init__(self, now_fn: Callable[[], float] | None = None):
        self._now_fn = now_fn or time.time
```

What does the above code mean?

EXPLANATION:-

Line 1: `def __init__(self, now_fn: Callable[[], float] | None = None):`

This is the constructor method with one parameter:

`now_fn: Callable[[], float] | None = None` - This parameter has several parts:

- `Callable[[], float]` - Type annotation indicating `now_fn` should be a callable (function) that:
 - Takes no arguments (the empty `[]`)
 - Returns a float value
 - Example: `time.time` is a perfect match - it takes no args and returns a float timestamp
- `| None` - Union type operator (Python 3.10+) meaning the parameter can be either `Callable[[], float]` or `None`
- `= None` - Default value, so this parameter is optional. If not provided, it defaults to `None`

Line 2: `self._now_fn = now_fn or time.time`

This uses Python's "or" operator for default value pattern:

- If `now_fn` is provided (truthy): `self._now_fn` gets the custom function
- If `now_fn` is `None` (falsy): `self._now_fn` gets `time.time` as the default

Note that `time.time` here is the function object itself, not a call to it (no parentheses).

Why This Pattern?

This is dependency injection for testability:

- In production: use default `time.time` for real timestamps
- In tests: inject a mock function to control what "now" returns (e.g., `lambda: 1234567890.0`)

This makes the code easier to test because you can control time behavior without monkey-patching.

EXAMPLE 2

what does the leading underscore mean here -

`class _TimestampingAgentExecutor(AgentExecutor):`

🟡 The leading underscore (`_`) in `class _TimestampingAgentExecutor` is a Python naming convention that indicates this class is intended to be private or internal to the module.

What it means:

1. Private/Internal API: The underscore signals "this is an implementation detail, not part of the public API"

2. Not imported with `*`: If someone does `from timestamp_ext import *`, classes/functions starting with `_` are not imported (they're excluded by

default)

3. Convention, not enforcement: Python doesn't truly enforce privacy - you can still access `_TimestampingAgentExecutor` from outside the module if

you really want to. It's a "gentleman's agreement" that says "don't use this directly"

In this context:

Looking at this extension, the public API is likely just the main extension class or factory function, while `_TimestampingAgentExecutor` is an

internal implementation detail that users of the extension shouldn't need to know about or instantiate directly.

Contrast:

- `_TimestampingAgentExecutor` - Internal class, subject to change without notice
- `TimestampingAgentExecutor` (no underscore) - Public class, part of the stable API

This helps maintain a clean separation between what's public interface (stable, documented) vs internal implementation (may change between

versions).

> can we call such a class from outside the module

🟡 Yes, absolutely! You can call/access `_TimestampingAgentExecutor` from outside the module.

Python's Privacy is Just Convention. The leading underscore is a convention, not an enforcement mechanism. Python doesn't have true private members like Java or C++.

How to access it:

From another module/file:

```
from timestamp_ext import _TimestampingAgentExecutor # This works!
```

Or:

```
import timestamp_ext
```

```
executor = timestamp_ext._TimestampingAgentExecutor(...) # Also works!
```

The only "restriction":

The only technical difference is with wildcard imports:

```
from timestamp_ext import *
```

```
# _TimestampingAgentExecutor won't be imported here
```

```
# But you can still do:
```

```
from timestamp_ext import _TimestampingAgentExecutor # Explicit import works
```

Why use the underscore then?

It's a social contract that communicates:

- "This is internal implementation - I might change or remove it without warning"
- "Don't depend on this - use the public API instead"
- Helps IDEs and linters warn you: "Hey, you're using something marked as private"

Bottom line: The underscore is like a "Caution" sign, not a locked door. You can access it, but the author is saying "you probably shouldn't, and

if you do, don't complain if it breaks later."

EXAMPLE 3

```
__all__ = [  
    'TIMESTAMP_FIELD',  
    'URI',  
    'MessageTimestamp',  
    'TimestampExtension',  
]
```

What does the above code mean?

Looking at the file, here's what's defined:

What EXISTS in the module:

Constants:

- _CORE_PATH (private)
- URI
- TIMESTAMP_FIELD
- _MESSAGING_METHODS (private)

Classes:

- TimestampExtension
- MessageTimestamp

- _TimestampingAgentExecutor (private)
- _TimestampingEventQueue (private)
- _TimestampClientFactory (private)
- _TimestampingClient (private)
- _TimestampingClientInterceptor (private)

What WILDCARD IMPORT would give you:

```
from timestamp_ext import *
```

Only these 4 items (defined in __all__):

TIMESTAMP_FIELD	#	✓
URI	#	✓
MessageTimestampper	#	✓
TimestampExtension	#	✓

NOT imported (not in __all__):

_TimestampingAgentExecutor	#	✗
_TimestampingEventQueue	#	✗
_TimestampClientFactory	#	✗
_TimestampingClient	#	✗
_TimestampingClientInterceptor	#	✗
_CORE_PATH	#	✗
_MESSAGING_METHODS	#	✗

So you'd get exactly 4 items - the two constants and two public classes. All the private classes (starting with _) are hidden from wildcard imports because they're not in __all__.

EXAMPLE 4

What are the dunder methods `__enter__` and `__exit__` used for? `__init__` is for constructor definition and `__all__` for defining all constants and methods which can be imported from the current module through wildcard imports?

```
def __enter__(self) -> TraceRecord:
    """Context manager entry point that returns the trace step.

    Returns:
        TraceRecord: The trace record instance to be used in the context.
    """
    return self.step

def __exit__(
    self,
    exc_type: type[BaseException] | None,
    exc_val: BaseException | None,
    exc_traceback: types.TracebackType | None,
) -> bool:
    """Context manager exit point that finalizes the trace step.

    Args:
        exc_type: The exception type if an exception was raised in the context.
        exc_val: The exception value if an exception was raised in the context.
        traceback: The traceback if an exception was raised in the context.

    Returns:
        bool: False to indicate that exceptions should not be suppressed.
    """
    error_msg = None
    if exc_type:
        error_msg = ''.join(
            exc_traceback.format_exception(exc_type, exc_val, exc_traceback)
        )
    self.step.end_step(error=error_msg)
    if self.response_trace:
```



```
self.response_trace.add_step(self.step)
# Do not suppress exceptions
return False
```

The `__enter__` and `__exit__` dunder methods implement Python's context manager protocol, enabling this class to be used with the `with` statement.

Here's how they work in this code:

`__enter__` (line 230-236)

- Called when entering the `with` block
- Returns `self.step` (a `TraceRecord` object) which becomes available via the `as` clause
- This allows tracing to begin when the context is entered

`__exit__` (line 238-263)

- Called when exiting the `with` block (whether normally or due to exception)
- Handles exceptions: If an exception occurred, it captures the full traceback and passes it to `step.end_step(error=msg)`
- Finalizes the trace: Calls `step.end_step()` to mark when the traced operation completed
- Records the step: Adds the completed step to `response_trace` if it exists
- Returns `False`: This means exceptions are not suppressed and will propagate normally

Usage Example

```
with SomeClass(...) as trace_record:
    # __enter__() is called, returns self.step
    # You can use trace_record here
    do_something()
# __exit__() is called here, recording end time and any errors
```

This pattern ensures that trace steps are properly started and ended with accurate timing and error information, even if exceptions occur during execution.

Where does python search for its modules? - please write a section on this from claude account

TOMORROW'S TASKS

- > ABOUT THE MODULE SEARCH PATH - Complete notes**
- > UNDERSTAND EACH LINE OF A2A-SAMPLES DIRECTORY AND A2A DIRECTORY AND WRITE ABOUT THEM HERE**

UNDERSTANDING A2A-SAMPLES DIRECTORY LINE BY LINE