OOP - Advanced

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

Polymorphism

Polymorphism is the provision of a single interface to entities of different types[1]. Polymorphism enables a programming language to dynamically determine which method to use at runtime, depending on the parameters sent to the method.

```
# Example: the implementation of `f()` depends on the object class.
obj = A()  # B()
obj.f()
```

---

[1] https://www.stroustrup.com/glossary.html#Gpolymorphism

Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to invoke at run time.
The decision on which version of a method to call can be based either on a single object/parameter (single dispatch), or on a combination of objects (multiple dispatch).

Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to invoke at run time.
The decision on which version of a method to call can be based either on a single object/parameter (single dispatch), or on a combination of objects (multiple dispatch).

- **Single dispatch** is a type of polymorphism where only one parameter is used to determine the call. Typically, this parameter is the object itself (**self**, or **this**).

Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to invoke at run time.
The decision on which version of a method to call can be based either on a single object/parameter (single dispatch), or on a combination of objects (multiple dispatch).

- **Single dispatch** is a type of polymorphism where only one parameter is used to determine the call. Typically, this parameter is the object itself (**self**, or **this**).
- **Multiple dispatch** is a form of polymorphism where multiple parameters are employed in determining which method to call.

Example

```
a.divide(b)  # `a/b`
```

Single dispatch: the implementation will be chosen based only on `a` type (rational, floating point, matrix,..), disregarding the type or value of divisor `b`.

Multiple dispatch: the implementation will be chosen based on the combination of operands. The `dividend` and `divisor` types determine which kind of `divide` operation will be performed.

Exercise

Write the classic `rock-paper-scissor` [2] game.

Rock, paper and scissor are instances of the classes `Rock`, `Paper`, `Scissor`, respectively (but this is not the only possibility).
The three classes are subclasses of the abstract class `Weapon`.

---

[2] https://en.wikipedia.org/wiki/Rock_paper_scissors

Define the classes in a way that objects respond to the `fight_against` method as follows:

```
list_of_weapons = [Scissor(), Paper(), Rock()]
for w1 in list_of_weapons:
  for w2 in list_of_weapons:
    print(w1, "vs", w2, "->", w1.fight_against(w2))
```

```
Output
--------------------
Scissor vs Scissor -> TIE
Scissor vs Paper -> Scissor
Scissor vs Rock -> Rock
Paper vs Scissor -> Scissor
Paper vs Paper -> TIE
Paper vs Rock -> Paper
Rock vs Scissor -> Rock
Rock vs Paper -> Paper
Rock vs Rock -> TIE
```

We can start from the classic implementation using a conditional structure in the `fight_against` method

```
from abc import ABC, abstractmethod


class Weapon(ABC):

  @abstractmethod
  def fight_against(self, other_weapon):
    pass

  def __str__(self):
    return self.__class__.__name__


class Scissor(Weapon):
  def fight_against(self, other_weapon):
    if isinstance(other_weapon, Paper):
      return "Scissor"
    elif isinstance(other_weapon, Rock):
      return "Rock"
    else:
      return "TIE"
```

Consider extending the game to incorporate additional weapons, such as in "Rock Paper Scissors Spock Lizard"[3].

---

[3] https://en.wikipedia.org/wiki/Rock_paper_scissors#Additional_weapons

Consider extending the game to incorporate additional weapons, such as in "Rock Paper Scissors Spock Lizard"[3].

The 'IF' based solution has several limitations:

- **Conditional Structure Extension**: If new classes are added, the conditional structure must be extended by adding new branches. This leads to code that is tightly coupled and requires modifications whenever new weapons are introduced.

---

[3] https://en.wikipedia.org/wiki/Rock_paper_scissors#Additional_weapons

Consider extending the game to incorporate additional weapons, such as in "Rock Paper Scissors Spock Lizard"[3].

The 'IF' based solution has several limitations:

- **Conditional Structure Extension**: If new classes are added, the conditional structure must be extended by adding new branches. This leads to code that is tightly coupled and requires modifications whenever new weapons are introduced.
- **Code Duplication**: The addition of new branches in the conditional structure may result in portions of code that are similar but not exactly identical. This redundancy complicates code maintenance and can lead to inconsistencies.

---

[3] https://en.wikipedia.org/wiki/Rock_paper_scissors#Additional_weapons

Consider extending the game to incorporate additional weapons, such as in "Rock Paper Scissors Spock Lizard"[3].

The 'IF' based solution has several limitations:

- **Conditional Structure Extension**: If new classes are added, the conditional structure must be extended by adding new branches. This leads to code that is tightly coupled and requires modifications whenever new weapons are introduced.
- **Code Duplication**: The addition of new branches in the conditional structure may result in portions of code that are similar but not exactly identical. This redundancy complicates code maintenance and can lead to inconsistencies.
- **Dependency on Implementation Order**: The order in which conditions are checked in the 'IF' structure can be crucial. Changing the order may alter the behavior of the code.

---

[3] https://en.wikipedia.org/wiki/Rock_paper_scissors#Additional_weapons

To avoid the limitations and adhere to the **Open/Closed Principle (OCP)**[4], it is possible to use the 'double dispatch' approach.

The behavior of the method `fight_against` is determined dynamically at runtime based on the weapon types, allowing for more flexibility and scalability.

---

[4] Code should be **open for extension**, but **closed for modification**

To avoid the limitations and adhere to the **Open/Closed Principle (OCP)**[4], it is possible to use the 'double dispatch' approach.

The behavior of the method `fight_against` is determined dynamically at runtime based on the weapon types, allowing for more flexibility and scalability.

When invoking `w1.fight_against(w2)`, it triggers a method in `w2` to carry out the task. The specific `w2` method is dynamically determined at runtime based on the weapon classes, eliminating the need for a conditional structure. This is possible because both `w1` and `w2` are aware of their own classes.

---

[4] Code should be **open for extension**, but **closed for modification**

To avoid the limitations and adhere to the **Open/Closed Principle (OCP)**[4], it is possible to use the 'double dispatch' approach.

The behavior of the method `fight_against` is determined dynamically at runtime based on the weapon types, allowing for more flexibility and scalability.

When invoking `w1.fight_against(w2)`, it triggers a method in `w2` to carry out the task. The specific `w2` method is dynamically determined at runtime based on the weapon classes, eliminating the need for a conditional structure.
This is possible because both `w1` and `w2` are aware of their own classes.

This way, the behavior of the game can be extended without modifying the existing code.
New weapons can be introduced by adding new classes, with a more modular and maintainable codebase.

---

[4] Code should be **open for extension**, but **closed for modification**

A possible solution

```python
from abc import ABC, abstractmethod


class Weapon(ABC):

  def __str__(self):
    return self.__class__.__name__

  @abstractmethod
    def fight_against(self, other_weapon):
      pass

  # second dispatch
  @abstractmethod
  def _fight_against_scissor(self):
    pass

  @abstractmethod
    def _fight_against_rock(self):
      pass

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

```python
class Scissor(Weapon):

  def fight_against(self, other_weapon):
    # dear other_weapon, I am a Scissor object!
    # I don't know who are you,
    # BUT YOU KNOW WHO ARE YOU and WHO I AM
    return other_weapon._fight_against_scissor()

  # second dispatch
  def _fight_against_scissor(self):
    return "TIE"

  def _fight_against_rock(self):
    return "Rock"

  def _fight_against_paper(self):
    return "Scissor"
```

An alternative solution using methods overloading

```python
from abc import ABC, abstractmethod
from plum import dispatch


class Weapon(ABC):

  def __str__(self):
    return self.__class__.__name__

  @abstractmethod
  @dispatch
  def fight_against(self, other_weapon: "Scissor"):
    pass

  @abstractmethod
  @dispatch
  def fight_against(self, other_weapon: "Rock"):
    pass

  @abstractmethod
  @dispatch
  def fight_against(self, other_weapon: "Paper"):
    pass
```

```python
class Scissor(Weapon):

  @dispatch
  def fight_against(self, other_weapon: "Scissor"):
    return "TIE"

  @dispatch
  def fight_against(self, other_weapon: "Rock"):
    return "Rock"

  @dispatch
  def fight_against(self, other_weapon: "Paper"):
    return "Scissor"
```

Comment

This problem is very simple, and double dispatch was adopted only to illustrate how it works. Given the simplicity of the problem, the conditional structure is appropriate.

One alternative is to use a single class, employing a string to identify the type of weapon and a look-up table to determine the winner. This approach could simplify the code structure and eliminate the need for multiple classes.

Exercise - Double dispatch

R is the class of real number, and **C** is the class of the complex number[5].

```
class MathEntity(ABC):
  ...

class R(MathEntity):

  def __init__(self, re):
    self.re = re

class C(MathEntity):
  """represents re + i img """

  def __init__(self, re, img=0):
    self.re = re
    self.img = img
```

---

[5] This is a toy problem - real and complex are already implemented in python.

Write methods to sum instances of `R` and `C` using the 'double dispatch' approach.
(Try even other solutions, like 'lookup-table' and conditional structure with explicit type checking. Discuss the differences.)

```
values = [R(2), R(-3), C(2,3), C(3,-3)]
for op1 in values:
  for op2 in values:
    print(op1 , "+" , op2 ,"=", op1 + op2)
```

```
Output
--------------------
2 + 2 = 4
2 + -3 = -1
2 + 2+i3 = 4+i3
2 + 3-i3 = 5-i3
-3 + 2 = -1
-3 + -3 = -6
-3 + 2+i3 = -1+i3
-3 + 3-i3 = 0-i3
2+i3 + 2 = 4+i3
2+i3 + -3 = -1+i3
2+i3 + 2+i3 = 4+i6
2+i3 + 3-i3 = 5
3-i3 + 2 = 5-i3
3-i3 + -3 = 0-i3
3-i3 + 2+i3 = 5
3-i3 + 3-i3 = 6-i6
```

Solution (naive double dispatch)

```
class MathEntity(ABC):
  # define abstractmethods __init__, __repr__,
  #   __add__, _add_real, _add_complex


class R(MathEntity):

  def __init__(self, re):
    self.re = re

  def __repr__(self):
    return str(self.re)

  def __add__(self, op2):
    return op2._add_real(self)

  def _add_real(self, op1):
    return R (self.re + op1.re)

  def _add_complex(self, op1):
    return C (self.re + op1.re, op1.img)
```

```
class C(MathEntity):
  """represents re + i img """

  def __init__(self, re, img=0):
    self.re = re
    self.img = img

  def __repr__(self):
    s_sign= ["-", "+"][self.img>0]
    s_img = "" if  self.img == 0 else s_sign + "i" + str(abs(self.img))
    return str(self.re) + s_img

  def __add__(self, op2):
    return op2._add_complex(self)

  def _add_real(self, op1):
    return C (self.re + op1.re, self.img)

  def _add_complex(self, op1):
    return C (self.re + op1.re, self.img + op1.img)
```

Solution (method overloading)

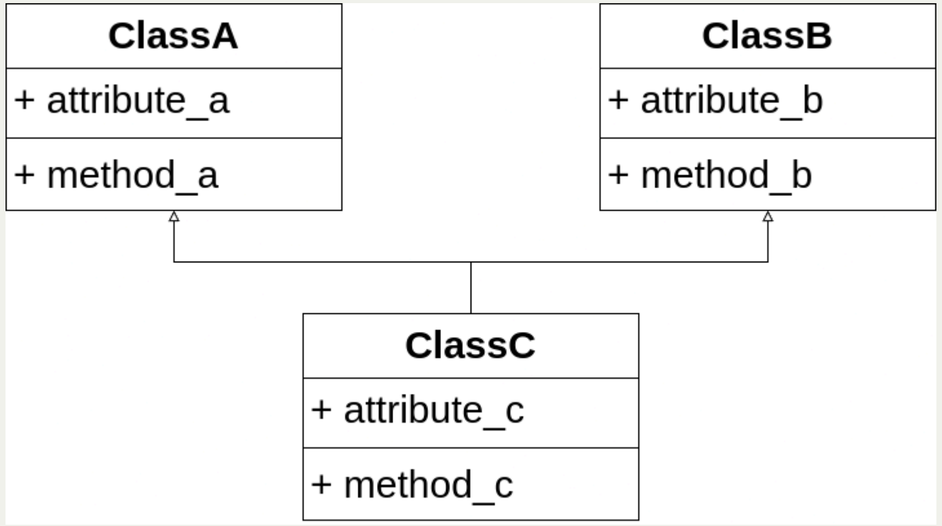
```python
from abc import ABC, abstractmethod
from plum import dispatch


class MathEntity(ABC):
  # define abstractmethods __init__, __repr__, __add__


class R(MathEntity):

  def __init__(self, re):
    self.re = re

  def __repr__(self):
    return str(self.re)

  @dispatch
  def __add__(self, op2: "R"):
    return R(self.re + op2.re)

  @dispatch
  def __add__(self, op2: "C"):
    return C(self.re + op2.re, op2.img)
```

```python
class C(MathEntity):
  """represents re + i img """

  def __init__(self, re, img=0):
    self.re = re
    self.img = img

  def __repr__(self):
    s_sign = ["-", "+"][self.img > 0]
    s_img = "" if self.img == 0 else s_sign + "i" + str(abs(self.img))
    return str(self.re) + s_img

  @dispatch
  def __add__(self, op2: "R"):
    return C(self.re + op2.re, self.img)

  @dispatch
  def __add__(self, op2: "C"):
    return C(self.re + op2.re, self.img + op2.img)
```

We can use the double dispatch approach when the behavior depends on the pair of objects involved in the relationship, and we plan to extend the number of classes involved.

Multiple Inheritance

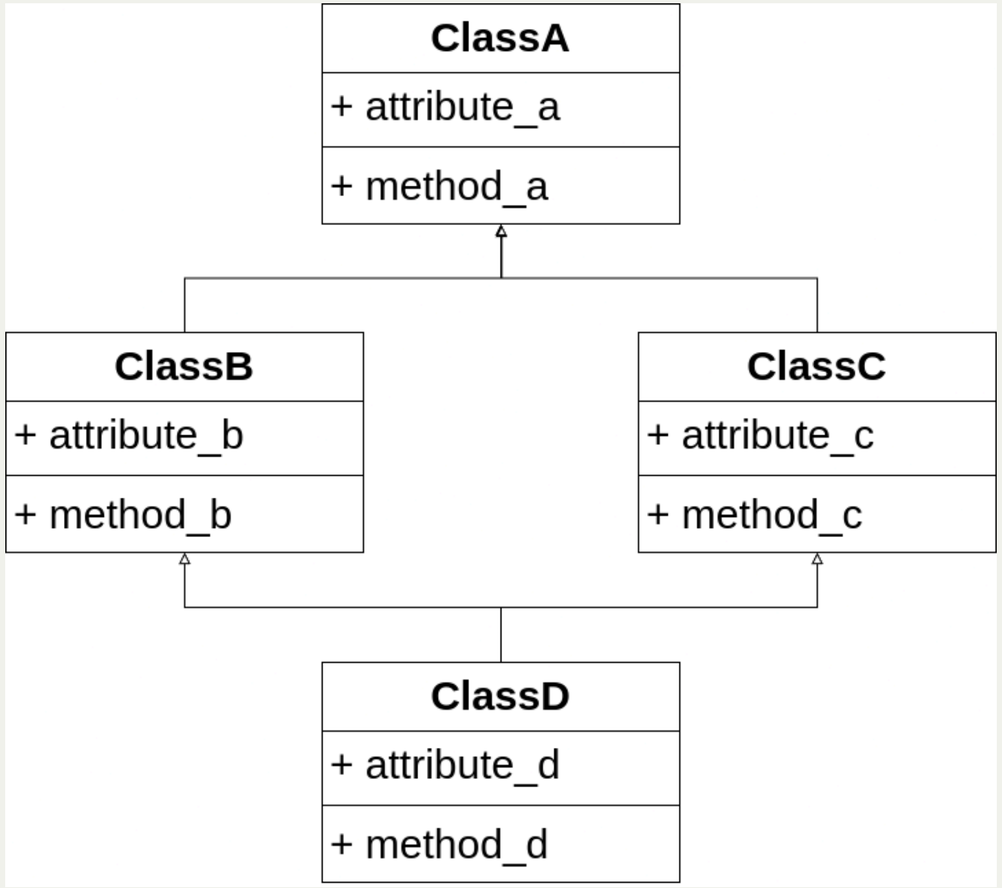
```
class A:
  ...


class B:
  ...


class C(A, B):
  ...
```



What happens if the parent classes have attributes/methods with the same names (e.g., the `__init__` method)?

The diamond problem

What happens in this case, if we want to call the `__init__` methods of all the classes when instantiating a ClassD object?



Mixin Classes

Mixin Classes are classes that does not belong to any inheritance hierarchy, and are not intended to be instantiated.

Their purpose is to contain functionalities that can be imported from other classes with inheritance, providing reuse of code without the needing to construct complex hierarchies.

They can also be used to build classes by composition.

Decorators

They are a functionality provided by Python, which allows to add some behaviour to functions, methods or classes, by simply *decorating* them.

```
@decorator
def function_to_be_decorated():
  ...
```

Decorators

They are a functionality provided by Python, which allows to add some behaviour to functions, methods or classes, by simply *decorating* them.

```
@decorator
def function_to_be_decorated():
  ...
```

Decorators are very useful as they can easily extend the software with additional operations, providing code reusability. We will introduce decorators for functions and methods.

A decorator is a function (or a class), which takes as input the function to be decorated, and returns another function that will be called in place of it.

```
def decorator(function_to_be_decorated):

  # defines a new wrapper function
  def wrapper():
    # performs some additional action
    function_to_be_decorated()  # calls the original function
    # performs some additional action

  return wrapper  # returns the wrapper reference
```

Example

```
def decorator(function_to_be_decorated):

  def wrapper():
    print("additional operations before calling the decorated function")
    function_to_be_decorated()
    print("additional operations after calling the decorated function")

  return wrapper


@decorator
def function_to_be_decorated():
  print("operations of the function to be decorated")
```

Example - preserving the function's metadata

```
from functools import wraps


def decorator(function_to_be_decorated):

  @wraps(function_to_be_decorated)
  def wrapper():
    """this is the wrapper"""
    print("additional operations before calling the decorated function")
    function_to_be_decorated()
    print("additional operations after calling the decorated function")

  return wrapper


@decorator
def function_to_be_decorated():
  """this is the function to be decorated"""
  print("operations of the function to be decorated")


if __name__ == "__main__":
  print("function name:", function_to_be_decorated.__name__)
  print()
  help(function_to_be_decorated)
```

Example - receiving and returning arguments

```
from functools import wraps


def decorator(function_to_be_decorated):

  @wraps(function_to_be_decorated)
  def wrapper(*args, **kwargs):
    """this is the wrapper"""
    print("before the decorated function I'd like to say Hi!")
    output = function_to_be_decorated(*args, **kwargs)
    output += " ...and the decorator says: Goodbye!"
    return output

  return wrapper


@decorator
def function_to_be_decorated(x):
  """this is the function to be decorated"""
  return f"the function to be decorated says: {x}"


if __name__ == "__main__":
  print(function_to_be_decorated("Hello!"))
```

Example - passing arguments to the decorator

```
from functools import wraps


def decorator(decorator_parameter):

    def inner_decorator(function_to_be_decorated):

        @wraps(function_to_be_decorated)
        def wrapper(*args, **kwargs):
            """this is the wrapper"""
            print("before the decorated function I'd like to say Hi!")
            output = function_to_be_decorated(*args, **kwargs)
            output += f" ...and the decorator says: {decorator_parameter}"
            return output

        return wrapper

    return inner_decorator


@decorator("Goodbye!")
def function_to_be_decorated(x):
    """this is the function to be decorated"""
    return f"the function to be decorated says: {x}"


if __name__ == "__main__":
    print(function_to_be_decorated("Hello!"))
```

Example - decorator class

```
class decorator:

    def __init__(self, function_to_be_decorated):
        self.function_to_be_decorated = function_to_be_decorated

    def __call__(self, *args, **kwargs):
        print("before the decorated function I'd like to say Hi!")
        output = self.function_to_be_decorated(*args, **kwargs)
        output += f" ...and the decorator says: Goodbye!"
        return output


@decorator
def function_to_be_decorated(x):
    """this is the function to be decorated"""
    return f"the function to be decorated says: {x}"


if __name__ == "__main__":
    print(function_to_be_decorated("Hello!"))
```

Metaprogramming

It refers to the ability of computer programs to read, manipulate and/or write other programs (which are treated as their data).

Metaprogramming

It refers to the ability of computer programs to read, manipulate and/or write other programs (which are treated as their data).

In reflective programming (also known as reflection), a program performs such operations on itself.

Metaprogramming

It refers to the ability of computer programs to read, manipulate and/or write other programs (which are treated as their data).

In reflective programming (also known as reflection), a program performs such operations on itself.

Metaprogramming is often useful to reduce the length of the code required for some operation, and enables to dynamically modify a software.

Introspection

It allows programs to inspect the data types at runtime.

```
dir(obj)
type(obj)
hasattr(obj, "attribute")
getattr(obj, "attribute")
callable(obj)
issubclass(class, class)
isinstance(obj, type)


from inspect import *


getmembers(obj)
isclass(obj)
ismethod(obj)
isfunction(obj)
isbuiltin(obj)
...
```

Metaclasses (6)

- Objects have a `type`.
- Everything is an object.
- Classes are objects as well.
- A class also has a type.

---

(6) The following is true only in python 3.x, which uses the so-called New-Style classes. For the difference between new and old style: https://wiki.python.org/moin/NewClassVsClassicClass)

Metaclasses [6]

- Objects have a `type`.
- Everything is an object.
- Classes are objects as well.
- A class also has a type.

What is the type of a class? → **`type`**

---

[6] The following is true only in python 3.x, which uses the so-called New-Style classes. For the difference between new and old style: https://wiki.python.org/moin/NewClassVsClassicClass)

Metaclasses [6]

- Objects have a `type`.
- Everything is an object.
- Classes are objects as well.
- A class also has a type.

What is the type of a class? → **`type`**

What is the type of `type`? → **`type`**

---

[6] The following is true only in python 3.x, which uses the so-called New-Style classes. For the difference between new and old style: https://wiki.python.org/moin/NewClassVsClassicClass)

Metaclasses [6]

- Objects have a `type`.
- Everything is an object.
- Classes are objects as well.
- A class also has a type.

What is the type of a class? → **`type`**

What is the type of `type`? → **`type`**

type is a metaclass. Its instances are classes.

---

[6] The following is true only in python 3.x, which uses the so-called New-Style classes. For the difference between new and old style: https://wiki.python.org/moin/NewClassVsClassicClass)

You can use `type()` with three arguments to dynamically create a new class, effectively creating a new instance of the `type` metaclass.

```
type(<name>, <bases>, <dct>)
```

You can use `type()` with three arguments to dynamically create a new class, effectively creating a new instance of the `type` metaclass.

`type(<name>, <bases>, <dct>)`

- <**name**> is the class name. This becomes the `__name__` attribute of the class.

You can use `type()` with three arguments to dynamically create a new class, effectively creating a new instance of the `type` metaclass.

`type(<name>, <bases>, <dct>)`

- <**name**> is the class name. This becomes the `__name__` attribute of the class.
- <**bases**> is a tuple of the base classes from which the class inherits. This becomes the `__bases__` attribute of the class.

You can use `type()` with three arguments to dynamically create a new class, effectively creating a new instance of the `type` metaclass.

`type(<name>, <bases>, <dct>)`

- <**name**> is the class name. This becomes the `__name__` attribute of the class.
- <**bases**> is a tuple of the base classes from which the class inherits. This becomes the `__bases__` attribute of the class.
- <**dct**> specifies a namespace dictionary containing definitions for the class body. This becomes the `__dict__` attribute of the class.

Two different ways to create a class (explore `__class__` and `__dict__` attributes).

In the usual way, with the **class** statement

```
class Foo():
  pass


class Bar(Foo):

  attr = 100
```

Dynamically with **type()**

```
Foo1 = type('Foo1', (), {})
Bar1 = type('Bar1', (Foo1,), dict(attr=100))
```

Two different ways to create a class

Usual way:

```
class A:

  def mymethod(self):
    print("I am the method mymethod")
```

With `type`:

```
def f(obj):
  print("I am the method mymethod")

A1 = type("A1", (), {"mymethod": f })
```

Results:

```
print(A1.__dict__)
print(A.__dict__)

a = A()
a1 = A1()
a.mymethod()
a1.mymethod()
```

You can define your own metaclass, which derives from `type`

```
class Meta(type):

  def __new__(cls, name, bases, dct):
    new_class = super().__new__(cls, name, bases, dct)
    new_class.class_attr = 100
    print("I am your custom __new__ method to create a CLASS!")
    return new_class
```

Now you can define a new class `Foo` and specify that its metaclass is the custom metaclass `Meta`, rather than the standard metaclass `type`. This is done using the `metaclass` keyword:

```
class Foo(metaclass=Meta):
  pass


print(Foo.class_attr)
```

In the same way that a class works as a template for the creation of objects, a metaclass works as a template for the creation of classes (see SINGLETON for a practical use).

Dynamic execution

The `exec( )` function can be used to dynamically execute programs by passing them as strings or code objects. Note that it does not return anything.

Example: class definition with `exec()`

```
exec("class A:\n\tdef mymethod(self):\n\t\tprint(\"I am the method mymethod\")")

class_A = """
class A:
  def mymethod(self):
    print(\"I am the method mymethod\")
"""

exec(class_A)
```