1. Definición

El cuarto y último de los pilares de la programación orientada a objetos, junto con la *abstracción*, el *encapsulamiento* y la *herencia*, es el **polimorfismo**. En griego, πολύς μορφή significa "muchas formas". Es por ello que, en programación, se denomina así a la capacidad de contener valores de distintos tipos (en el caso de las variables polimórficas) o a la capacidad de recibir argumentos de diferentes tipos (en el caso de los métodos polimórficos).

Según cómo sea el conjunto de los tipos posibles, Cardelli y Wegner[1] clasifican el polimorfismo en dos categorías: **polimorfismo ad-hoc**[2] (funciona con un número limitado y conocido de tipos, no necesariamente relacionados entre sí) y **polimorfismo universal** (funciona con un número prácticamente infinito de tipos relacionados entre sí). En cada categoría, a su vez, es posible encontrar dos variedades de polimorfismo: como variantes del polimorfismo ad-hoc pueden clasificarse el *polimorfismo por sobrecarga* y el *polimorfismo por coerción*, mientras que el *polimorfismo paramétrico* y el *polimorfismo por inclusión* son variantes del polimorfismo universal.

En `Python` no todas estas variantes están disponibles.

2. Polimorfismo ad-hoc

2.1. Polimorfismo por sobrecarga

El polimorfismo por sobrecarga no existe en `Python`. Este tipo de polimorfismo admite escribir dos métodos o funciones con el mismo nombre pero distintos tipos de parámetros. Un ejemplo de esto en `C++` sería el siguiente:

Programa 1: sobrecarga de funciones en `C++`

```
1 #include <iostream>
2 #include <string>
3 #include <cmath>
4 using namespace std;
5
```

[1] Cardelli, L. & Wegner, P. "On Understanding Types, Data Abstraction, and Polymorphism". En: *Computing Surveys* (Diciembre, 1985). Vol. 17, n. 4, p. 471

[2] Ad hoc es una locución latina que significa literalmente "para esto". Se usa para referirse a algo específico que es adecuado sólo para un determinado fin o en una determinada situación.

```cpp
int unir(int a, int b) {
  return (a * pow(10, ceil(log10(b + 1))) + b);
}
string unir(string a, string b) {
  return a + b;
}

int main() {
  cout << unir(123, 45) << endl;
  cout << unir("123", "45") << endl;
}
```

La salida de este programa es:

```
12345
12345
```

En C++ esto es posible, porque al ser fuertemente tipado, el compilador no confunde la función que debe ser invocada. Este es un *polimorfismo aparente*. Es decir, no es la misma función porque reciben tipos de datos distintos. En cambio en Python, al no existir el tipado (recuerde que lo que se especifica son *"typehints"*), no sería posible diferenciarlas. De hecho, lo que ocurriría es que la última declarada reemplaza a la anterior, una *sobrescritura* dentro de la misma clase.

Programa 2: Sobrescritura de un método dentro de la misma clase.

```python
from math import ceil, log10

class Unidor:
  # Nunca será llamada...
  def unir(self, a: int, b: int) -> int:
    return a * (10**ceil(log10(b + 1))) + b

  # reemplaza a la anterior...
  def unir(self, a: str, b: str) -> str:
    return a + b

if __name__ == "__main__":
  u = Unidor()
  print(u.unir(123, 45))     # muestra 168 (123+45)
  print(u.unir("123", "45")) # muesrta 12345
```

En caso de necesitar realmente una función que trabaje como *sobrecargada*, en Python tenemos diferentes aproximaciones. En particular la del ejemplo anterior podría *solucionarse* utilizando `isinstance()`.

```python
class Unidor:
```

```
    def unir(self, a: str|int, b: str|int) -> str|int:
      if isinstance(a, str) and isinstance(b, str):
        return a + b
      elif isinstance(a, int) and isinstance(b, int):
        return a * (10**ceil(log10(b + 1))) + b
```

Aquí le decimos que tanto `a` como `b` pueden ser `str` o `int`. Luego con la función `isinstance()` verificamos el tipo de dato y ejecutamos el algoritmos correspondiente. Debería haber un último `else` para manejar el error en caso de recibir tipos distintos, eso se analizará en el apunte sobre **excepciones**.

Otro caso común de sobrecarga es cuando una función recibe distintas cantidades de parámetros. un ejemplo en `C++` sería el siguiente:

```
double pedir_numero(string txt);
double pedir_numero(string txt, double min, double max);
```

En este ejemplo, la función pide un número, pero esta podría o no tener un rango. Esto en `Python` se podría solucionar utilizando **parámetros con valores por omisión** (*default*):

```
1  def pedir_numero(txt: str,
2                   minimo: float = float('-inf'),
3                   maximo: float = float('inf')) -> float:
4    n = float(input(txt))
5    while n < minimo or n > maximo:
6      n = float(input(txt))
7    return n
8
9  if __name__ == "__main__":
10   print(pedir_numero("Ingrese número: "))
11   print(pedir_numero("Ingrese número: ", -2, +2))
```

En este ejemplo, se declaran 3 parámetros, pero `minimo` tiene el valor por omisión *-infinito* y `maximo` el *infinito* positivo. En caso de no especificarlos, estos serán los valores a utilizar en el algoritmo.

2.2. Polimorfismo por coerción

Aplicar coerción sobre alguien significa forzar su voluntad o su conducta. En programación, significa forzar a que un dato de un tipo sea tratado como si fuera de otro tipo. Por ejemplo, *coerción* es la operación realizada para convertir un argumento al tipo esperado por el parámetro de un método. En este caso, también se trata de un *polimorfismo aparente*, ya que la conversión que ocurre no cambia el hecho de que el método en realidad sólo trabaja con un único tipo de dato.

```
1 class Duplicador:
2   def duplicar(self, n: float) -> float:
3     return 2*float(n)
4
5 if __name__ == "__main__":
6   d = Duplicador()
7   print(d.duplicar(2))    # int
8   print(d.duplicar(1.5))  # float
9   print(d.duplicar("3"))  # str
```

3. Polimorfismo universal

3.1. Polimorfismo paramétrico

Cuando se escribe código genérico, es decir, sin mencionar ningún tipo de dato específico, para que pueda ser usado con datos que serán tratados de manera idéntica independientemente de su tipo, se está aprovechando el *polimorfismo paramétrico*. En `Python` esto se realiza con una clase llamada `Generic` en el módulo `typing`. En el siguiente ejemplo se observa como es posible generalizar una clase para que su *typehint* sea más flexible y universal:

Programa 3: Uso de Generic

```
1 from typing import TypeVar, Generic
2
3 # Marcador de posición para un tipo de dato genérico
4 T = TypeVar('T')
5
6 class Pila(Generic[T]):
7     def __init__(self) -> None:
8         self.__pila : list[T] = [] # contenedor vacío
9
```



```
10      def apilar(self, e : T) -> None:
11          self.__pila.insert(0, e)
12
13      def desapilar(self) -> T:
14          return self.__pila.pop(0)
15
16
17  if __name__ == "__main__":
18      pila_enteros = Pila[int]()
19      pila_enteros.apilar(2)
20      pila_enteros.apilar(-7)
21      pila_enteros.apilar(20)
22      for _ in range(3):
23          print(pila_enteros.desapilar())
24      pila_strings = Pila[str]()
25      pila_strings.apilar("estás?")
26      pila_strings.apilar("como")
27      pila_strings.apilar("hola")
28      for _ in range(3):
29          print(pila_strings.desapilar())
```

Nota

A partir de `Python 3.12` la sintaxis de `Generic` se ha simplificado, sin la necesidad de importar o heredar de la clase utilizando:

```
# No es necesario importar nada...

class Pila[T]:
    ... # lo demás se mantiene igual
```

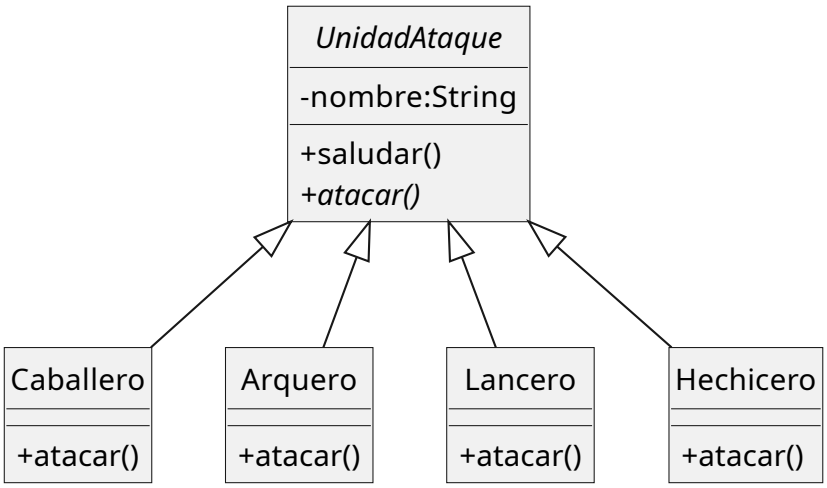
3.2. Polimorfismo por inclusión

El *polimorfismo por inclusión*, también conocido como *polimorfismo por herencia* o *polimorfismo de subclases* (o de *subtipos*), es la variante más común de polimorfismo encontrada en la Programación Orientada a Objetos y, por tal motivo, la mayoría de las veces se lo denomina directamente *polimorfismo* a secas. Se basa en el hecho de que las superclases *incluyen* a todas las instancias de sus subclases o, dicho de otra forma, todos los objetos de una subclase son también objetos de las superclases de ésta. Por ello, aunque se declare un atributo, una variable local, un parámetro o el contenido de una colección como siendo de una superclase, en realidad puede referirse a cualquier instancia de esa clase o de alguna sus subclases. Los objetos no pierden sus características (estado

y comportamiento) al ser referenciados de esta manera, por lo que es posible invocar los métodos que éstos hayan sobrescrito, y su comportamiento será el esperado.

Programa 4: Polimorfismo por inclusión

```python
from abc import ABC, abstractmethod

class UnidadAtaque(ABC):
  def __init__(self, nombre: str) -> None:
    self.__nombre = nombre
  def saludar(self) -> None:
    print(f"Hola, soy un {self.__nombre}")
  @abstractmethod
  def atacar(self) -> None:
    ...

class Caballero(UnidadAtaque):
  def __init__(self) -> None:
    super().__init__("Caballero")
  def atacar(self) -> None:
    print("Envisto con una espada...")

class Arquero(UnidadAtaque):
  def __init__(self) -> None:
    super().__init__("Arquero")
  def atacar(self) -> None:
    print("Lanzo una flecha...")

class Lancero(UnidadAtaque):
  def __init__(self) -> None:
    super().__init__("Lancero")
  def atacar(self) -> None:
    print("Estoqueo con una lanza...")

class Hechicero(UnidadAtaque):
  def __init__(self) -> None:
    super().__init__("Hechicero")
  def atacar(self) -> None:
    print(f"Lanzo un hechizo de fuego...")

if __name__ == "__main__":
  unidades : list[UnidadAtaque] = []
  unidades.append(Caballero())
  unidades.append(Arquero())
  unidades.append(Lancero())
  unidades.append(Hechicero())

  for unidad in unidades:
    unidad.saludar()
    unidad.atacar()
```



En este ejemplo se observa una clase abstracta `UnidadAtaque` que deja el método `atacar` sin implementar y delega la responsabilidad a las subclases que la hereden. Es así, como `Caballero`, `Arquero`, `Lancero` y `Hechicero` podrán atacar como corresponde. Pero en este ejemplo, además, serán accedidos polimórficamente a través de la lista `unidades` que es una lista de `UnidadAtaque` a la cual se les agregan un objeto de cada una de sus subclases. Y luego es recorrida pidiendo que saluden y ataquen respectivamente. Cada cual tendrá el comportamiento esperado y producirá la siguiente salida.

```
Hola, soy un Caballero
Envisto con una espada...
Hola, soy un Arquero
Lanzo una flecha...
Hola, soy un Lancero
Estoqueo con una lanza...
Hola, soy un Hechicero
Lanzo un hechizo de fuego...
```

Al utilizar la herencia podemos obtener diseños robustos, pero poco flexibles y a veces muy acoplados. Es mejor utilizar interfaces que una herencia en general, porque produce código más flexible. Esto será analizado en el próximo apunte **principios de diseño**.

3.3. Duck Typing

En `Python` existe un tipo de polimorfismo universal debido a su dinamismo llamado *duck typing*. Este consiste en que no es necesario que un objeto herede un método o atributo

para poder ser llamado o accedido, con que este exista es suficiente para poder ser invocado en tiempo de ejecución.

Programa 5: Duck typing

```
1  class Marinero:
2    def limpiar() -> None:
3      print("Marinero trapeando la cubierta del barco.")
4
5  class EmpleadoDomestico:
6    def limpiar() -> None:
7      print("Empleado plumereando el living.")
8
9  if __name__ == "__main__":
10   limpiadores = [Marinero(), EmpleadoDomestico()]
11   for limp in limpiadores:
12     limp.limpiar()
```

Como se observa, entre `Marinero` y `EmpleadoDomestico` no hay relación alguna, pero comparten la misma interfaz publica, porque a ambos se les puede enviar el mensaje `limpiar`.

Esta es una práctica bastante común en `Python`, ya que permite desacoplar totalmente el código y permite agilizar el mismo en ensayos rápidos sin diseño previo, es decir, que el programador no necesita estar pensando en interfaces comunes para diversos objetos. Pero aún así, tiene sus desventajas. Esta alta flexibilidad podría llevar a los programadores a cometer errores al carecer de una estructura o interfaz común que compartan los objetos que deseen ser accedidos polimorficamente. Estos errores solo son detectables al ejecutar el programa, es decir en tiempo de ejecución. Pueden ser muy difíciles de detectar.

En este curso no recomendaremos esta práctica, ya que no todos los lenguajes la soportan y puede conllevar a malos diseños de software si no es empleada con cautela.

3.4. Protocolos

Para dar un poco de estructura al *duck typing* llamado también *static duck typing*, a partir de `Python 3.8` se añadió una clase llamada `Protocol` que se encuentra en el módulo `typing`. Es similar a declarar una interfaz, pero tiene la particularidad que las clases no necesitan realizarla (o implementarla) solo hace falta *cumplir con el protocolo*, es decir tener la API pública que coincida con la declarada.

Programa 6: Uso de protocolos

```python
from typing import Protocol

class Empleable(Protocol):
  def trabajar(self) -> None:
    ...

class Obrero:
  def trabajar(self) -> None:
    print("Obrero toma la pala y empieza a excabar")

class Robot:
  def trabajar(self) -> None:
    print("El robot excaba con sus brazos")

if __name__ == "__main__":
  empleados: list[Empleable] = [Obrero(), Robot()]
  for e in empleados:
    e.trabajar()
```

En los protocolos, al igual que en las interfaces, solo se declaran métodos, no se implementan. Pero las clases que cumplen con este protocolo no necesitan heredarla, sino que con cumplir con el protocolo es suficiente. En este caso cualquier clase que tenga el mensaje `trabajar` es compatible con el protocolo.

Con este mecanismo, se pueden prever posibles errores antes de la ejecución del programa con herramientas como `mypy` que analiza los tipos de datos en módulos de `Python`. Para más información puede acceder a la documentación en `https://mypy.readthedocs.io/en/stable/`.

4. Referencias

Para más información puede consultar los siguientes enlaces:

- Documentación oficial: `https://docs.python.org/3/library/typing.html`
- Sobre *Duck Typing*: `https://realpython.com/duck-typing-python/`
- Sobre `Protocol`: `https://realpython.com/python-protocol/`
- Interprete Visual de Python: `http://www.pythontutor.com/visualize.html`