POO EN PYTHON

A Programación Orientada a Obxectos (POO) é un paradigma de programación máis próximo ao xeito de expresar os conceptos que empregamos na vida real.

Na POO o problema descomponse en obxectos. Este paradigma céntrase en reproducir o escenario real o máis fielmente posible.

A POO considera a un programa como unha colección de axentes autónomos, chamados obxectos. Mediante a interacción dos obxectos avanza a execución do programa.

Características principais da POO

As características principais da Programación Orientada a obxectos son:

- Calquera cousa é un obxecto. Podemos pensar nun obxecto como un tipo especial de variable que almacena datos, pero ao que tamén se lle poden facer peticións para que leve a cabo operacións.
- Un programa é un conxunto de obxectos dicíndose uns a outros o que deben facer mediante o envío de mensaxes. Para facer unha petición a un obxecto, se lle envía a ese obxecto unha mensaxe.
- Cada obxecto ten a súa propia memoria constituída por outros obxectos. Pódese crear un novo obxecto empaquetando obxectos existentes.

- Cada obxecto ten o seu tipo, que, en terminoloxía de POO se denominaría clase.
- Todos os obxectos dun tipo particular poden recibir as mesmas mensaxes.

---

1. DEFINICIÓN DUNHA CLASE

Antes falabamos de que o mundo real agrupa aos individuos en grupos. Cada tipo de grupo correspóndese a unha **clase** na terminoloxía de POO, e cada individuo, cun **obxecto**. Unha clase é un modelo para un obxecto, no que se define tanto as súas características (mediante os **atributos**) como o seu comportamento ou capacidades (mediante **métodos**).

Definimos unha **clase** como un Tipo Abstracto de Datos equipado cunha implantación, que pode ser parcial. Unha clase é un tipo, xa que describe un conxunto potencial de estruturas de datos, denominadas instancias da clase. Unha clase tamén é un módulo, é dicir, unha unidade de descomposición do software.

Unha clase componse de **atributos** que determinan unha estrutura de almacenamento para cada obxecto da clase e de funcións (**métodos**) que definen as operacións aplicables aos obxectos e constitúen o único medio de acceder aos atributos. Os atributos dunha clase poden ser, á súa vez, clases; ou poden ser atributos simples, que son aqueles que declaran tipos de datos simples.

Vexamos a continuación un exemplo. *Persoa* sería unha clase, que podería ter unhas características almacenadas nos seus atributos: *nome*, *data de nacemento*, *peso* e *altura*, así coma uns comportamentos, que constituirían os seus métodos: *durmir*, *comer* e *camiñar*. As instancias desa clase persoa (os obxectos) serían cada unha das persoas individuais, no gráfico do exemplo, Antía, Xurxo e Olalla. Cada un ten os seus propios valores dos atributos, como se pode ver na figura.

1.1. ESTRUTURA

Para definir unha clase en Python debemos utilizar a palabra reservada `class`.

```
class ClaseExemplo:
    # Atributos de clase
    atributo_clase = "Son un atributo de clase"

    # Método constructor
    def __init__(self, parametro1, parametro2):
        # Atributos de obxecto
        self.parametro1 = parametro1
        self.parametro2 = parametro2

    # Outros métodos da clase
    def mostrar_parametros(self):
        print(f"Parámetro 1: {self.parametro1}")
        print(f"Parámetro 2: {self.parametro2}")
```

Neste exemplo, `ClaseExemplo` é unha clase simple cun atributo de clase chamado `atributo_clase` (digamos que é un atributo estático), un método construtor `__init__`, dous parámetros de obxecto (`parametro1` y `parametro2`) e un método de nome `mostrar_parametros`.

1.2. CONSTRUTORES

O método `__init__` é un método especial que se utiliza como construtor dunha clase. Este método chámase cando se crea un obxecto da clase.

O propósito xeral é inicializar os atributos dun obxecto cando este se crea.

```
class Peroa:
    def __init__(self, nome, idade):
        ....
```

Neste exemplo, o construtor toma 3 parámetros: `self`, `nome` y `idade`.

1.2.1. self

`self` é unha convención que se utiliza como o nome do primeiro parámetro dun método ou construtor dunha clase. Este parámetro representa a instancia actual da clase e proporciona un xeito de acceder aos atributos e métodos dese instancia.

Inda que se utiliza `self` por convención, podes utilizar calquera nome que desexes.

1.2.2. Sobrecarga de construtores

A sobrecarga de métodos é un mecanismo que permite definir nunha clase varios métodos co mesmo nome. Isto resulta útil cando non sempre se lle pasan os mesmos parámetros a un mesmo método e queremos que, segundo o número e tipo de parámetros, se comporte dunha forma ou doutra. Por exemplo, a clase *Empregado* pode ter un método chamado *incrementarSalario* que se comporta de dúas formas diferentes: se non se lle pasa ningún parámetro, incrementa o salario do empregado nun 1%, se se lle pasa un parámetro, incrementará o salario na porcentaxe que se lle indique nese parámetro.

En Python non podemos contar con varios construtores con diferentes parámetros, tan só podemos utilizar un. O xeito de simular esta funcionalidade e utilizando parámetros chave.

```
class Peroa:
    def __init__(self, nome="Manuel", idade=35):
        ....
```

1.3. ATRIBUTOS

Para definir un atributo dunha clase, podemos facelo dentro do construtor ou dende calquera método. O recomendable e definilos todos bo construtor. Para iso utilizamos o parámetro `self`.

```
class ClaseExemplo:
    def __init__(self, _atributo):
        self.atributo = _atributo
        self.numero = 10
```

Neste caso, definimos o atributo `self.atributo` e `self.numero`. O primeiro asignámoslle o valor do parámetro `_atributo` e o segundo asignámoslle sempre o valor 10.

É dicir, con `self` podemos acceder aos atributos dunha clase en Python, tanto para definilos como para acceder a eles.

1.3.1. Atributos estáticos

Os atributos estáticos son variables que pertences a clase en lugar de a un obxecto da clase. Isto significa que estes atributos son compartidos por tódalas instancias de clase.

```
class ClaseExemplo:
    atributo_estatico = 0

    def __init__(self, valor):
        self.atributo_obxecto = valor
```

1.4. MÉTODOS

Os métodos defínense dentro dunha case como se foran funcións. Debemos utilizar a indentación correspondente.

```
class ExemploClase:
    def metodo_exemplo(self):
        # Corpo do método....
```

Neste exemplo, o método `metodo_exemplo` pertence a clase `ExemploClase`. Tódolos métodos deben ter como primeiro parámetro a `self`. Este parámetro permítenos acceder aos atributos do obxecto.

```
class ExemploClase:
    # Construtor
    def __init__(self, nome):
        self.nome = nome

    # Método
    def saudar(self):
        print("Ola,", self.nome)
```

Pódense definir tamén atributos dende un método, aínda que esta práctica non é recomendable.

```
class ExemploClase:
    # Construtor
    def __init__(self, nome):
        self.nome = nome

    # Método
    def set_idade(self, idade):
        self.idade = idade
```

1.4.1. Métodos estáticos

Podemos definir un método estático dunha clase co decorador `@staticmethod`. Un método estático pertence a clase en lugar de a un obxecto da propia clase. Polo tanto non debe recibir o parámetro `self`.

```
class ExemploClase:
    atributo_clase = "Atributo clase"

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

    def metodo_normal(self):
        pass

    @staticmethod
    def metodo_estatico():
        pass
```

2. INSTANCIAR OBXECTOS

Na vida real, un **obxecto** defínese por unha serie de características, chamadas **propiedades** e que realiza unha serie de operacións (**métodos**).

Cada obxecto é un exemplar dalgunha **clase**, é dicir, unha instancia dunha clase e o seu comportamento queda determinado pola clase á que pertence.

O **estado** dun obxecto comprende todas as súas propiedades e os seus valores actuais.

Os obxectos comunícanse mediante mensaxes. A resposta á mensaxe recibida será a invocación dun método.

Resumo

Clases	**Obxectos**
Non teñen existencia real, son só o molde de creación dos obxectos.	**Teñen existencia** real e unhas propiedades con valores concretos.
Son elementos **estáticos**, non evoluciona no tempo.	Son elementos **dinámicos**, o seu estado evoluciona durante a marcha do programa.
Dunha **clase** pódense crear moitos obxectos.	Un **obxecto** só pode ser creado a partir **dunha clase**.

Podemos decir que a clase é o tipo de datos dun obxecto.

2.1. INICIALIZACIÓN DUNHA CLASE

En Python, para crear un obxecto dunha clase utilizamos o nome da clase como se fose unha función que devolve un obxecto desta clase. Cando se realiza isto, chámase ao construtor da clase, é dicir, ao método `__init__()` desta.

```
class ExemploClase:
    def __init__(self, parametro1, parametro2, parametro3=False):
        self.atributo1 = parametro1
        self.atributo2 = parametro2
        self.atributo3 = parametro3

# Creando obxectos dunha clase
obxecto1 = ExemploClase("Hola", 42)
obxecto1 = ExemploClase("Hola", 42, True)
```

Na chamada do construtor, vemos que o parámetro `self` non se ten en conta.

2.2. ACCESO A ATRIBUTOS

Para acceder aos atributos dun obxecto, utilizamos a notación punto (`.`) seguida do nome do atributo.

```
class Persoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

# Creamos un obxecto
persoa = Persoa("Manuel", 35)

# Acceiendo aos atributos do obxecto persoa
print(persoa.nome)  # Imprime "Manuel"
print(persoa.idade)    # Imprime 35

# Modificando un atributo do obxecto persoa
persoa.idade = 36
```

2.2.1. Acceso a atributos estáticos

Para acceder a un atributo estático utilizamos de igual xeito o punto (`.`) pero en lugar de utilizar como variable o nome do obxecto, utilizamos o nome da clase.

```
class ExemploClase:
    atributo_estatico = "Atributo estatico"

# Accedendo ao atributo estático utilizando o nome da clase
print(ExemploClase.atributo_estatico)

# Modificando o atributo estático
ExemploClase.atributo_estatico = "Modificado"

# Accedendo ao atributo estático despois da modificación
print(ExemploClase.atributo_estatico)  # Imprime "Modificado"
```

2.3. CHAMADA A MÉTODOS

Do mesmo xeito que os atributos dun obxecto, para chamar a un método utilizamos a notación punto (`.`) despois da variable que apunta ao obxecto.

```
class ExemploClase:
    def saudar(self):
        print("Ola!")

    def despedir(self, nome):
        print("¡Adiós,", nome, "!")

# Creando un obxecto da clase ExemploClase
obxecto = ExemploClase()

# Chamando ao método saudar()
obxecto.saludar()  # Imprime "Ola!"

# Chamando ao método despedir() cun argumento
obxecto.despedir("Manuel")  # Imprime "¡Adiós, Manuel !"
```

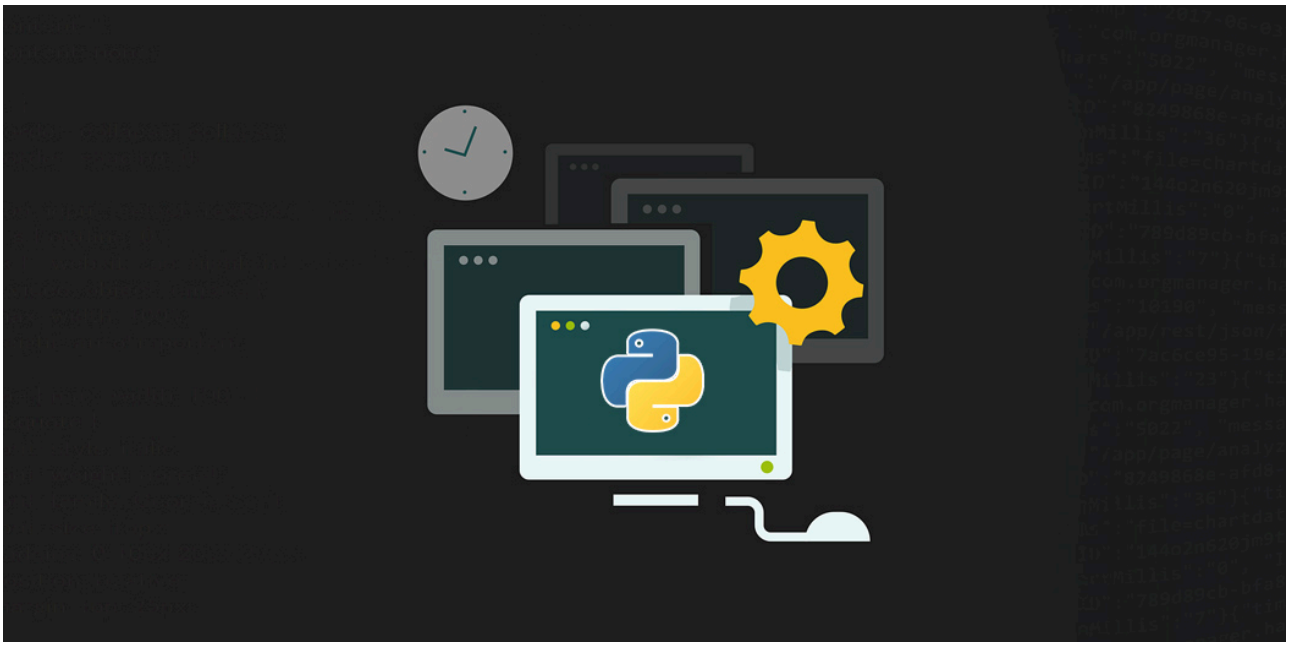
Na chamada dun métodos, vemos que o parámetro `self` non se ten en conta.

2.3.1. Chamada a métodos estáticos

Para chamar a un método estático, realizase do mesmo xeito que a un método pero utilizando o nome da clase en lugar dunha variable que apunte a un obxecto concreto.

```
class ExemploClase:
    @staticmethod
    def metodo_estatico():
        print("Este e un método estático")

# Cahmando ao método estático utilizando o nome da clase
ExemploClase.metodo_estatico()  # Imprime "Este e un método estático"
```

3. ENCAPSULACIÓN



A encapsulación é a forma de protexer os atributos e métodos nas clases, indicando que obxectos poden acceder a eles. Tamén é unha forma de expoñer ao programador unicamente as funcionalidades da clase que lle poden interesar, ocultando a súa complexidade. Existen 3 niveis principais de acceso:

- **Público**: Calquera obxecto de calquera clase pode acceder ao atributo ou método.
- **Protexido**: Só os obxectos desa clase ou de clases descendentes dela poden acceder ao atributo ou método.
- **Privado**: Só o propio obxecto pode acceder ao atributo ou método.

A encapsulación é a maneira que ten unha clase de manter os seus datos seguros, de xeito que non se modifiquen e conteñan datos incoherentes. Por exemplo, a clase *Persoa* podería ter un atributo DNI. O DNI non pode ser unha cadea de caracteres calquera, se este atributo fose público, quen instancia e emprega o obxecto podería escribir valores erróneos nel, por exemplo “Hola Pepe” ou “1Ae$39”, ao non selo, a clase non permite que se escriba directamente nel, senón que o fará a través dun método que comprobará que o formato e a letra correspondentes son as adecuadas antes de actualizar o valor.

En Python, a encapsulación refírese a capacidade de ocultar detalles internos dunha clase e protexer os seus atributos e métodos. En Python non hai un control de acceso estricto como noutras linguaxes como Java ou C++.

3.1. CONVENCIÓNS DE NOMENCLATURA

Podemos utilizar as convencións de nomenclatura existentes para indicar a visibilidade dos atributos e métodos. Por convención, utilizamos un guión baixo para indicar que un atributo ou método é **privado** e non se debería acceder dende fora da clase.

```
class ExemploClase:
    def __init__(self):
        self._atributo_privado = 10

    def _metodo_privado(self):
        return "Son un método privado"
```

Isto non impide o acceso directo dende fora da clase, só é unha sinal para outros programadores de que certos elementos non deben de ser utilizados.

3.2. PROPERTY: GETTERS E SETTERS E INMUTABILIDADE

O decorador `@property` en Python utilízase para converter un método dunha clase nunha propiedade. A propiedade compórtase como un atributo, pero por detrás execútase como un método canda vez que se intenta acceder a esta propiedade. Isto permite un acceso controlado e a posibilidade de realizar operacións adicionais ao obter ou establecer o valor da propiedade.

Isto permítenos definir *getters* e *setters* para acceder a atributos ou modificalos.

```
class ExemploClase:
    def __init__(self):
        self._atributo_privado = 10

    @property
    def atributo_privado(self):
        return self._atributo_privado

    @atributo_privado.setter
    def atributo_privado(self, novo_valor):
        if nuevo_valor > 0:
            self._atributo_privado = nuevo_valor
        else:
            print("O valor debe de ser maior que 0")
```

- `@property` aplícase ao método `atributo_privado`. Isto significa que podes acceder a `obxecto.atributo_privado` como se fose un atributo. O que se fará cando se acceda a este "atributo" e executar o método `atributo_privado`.
- `@atributo_privado.setter` utilízase para definir un método que se chamará cando intentes establecer un valor na propiedade. Neste caso, o método `atributo_privado` actúa como un *setter*, e só permitirá establecer o valor se este é maior que 0.

3.2.1. Propiedades de só lectura

Podemos utilizar `@property` para evitar a modificación de atributos despois da creación do obxecto.

```
class Punto:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y

# Crea un obxecto inmutable
punto = Punto(1, 2)
print(punto.x, punto.y)  # Saida: 1 2
```

Os obxectos creados da clase `Punto` son **immutables** porque os seus atributos non poden ser modificados despois da creación do obxecto.

3.3. LIMITACIÓN DE ATRIBUTOS

En Python, `__slots__` é un característica que permite limitar os atributos que unha instancia pode ter.

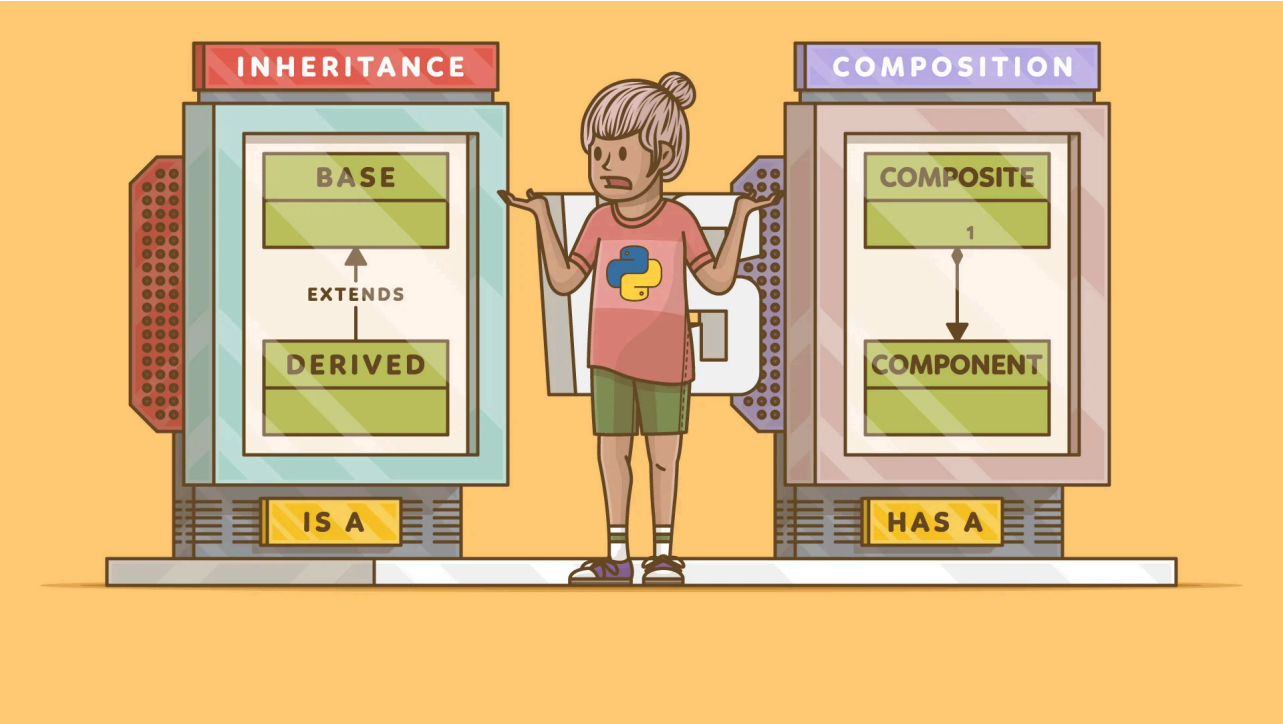
As instancias dunha clase en Python poden ter calquera número de atributos. Isto débese a que Python reserva espazo para un dicionario que contén tódolos atributos da instancia. Sen embargo, para clases que utilizan un número fixo e limitado de atributos isto pode ser innecesario e ineficiente en termos de uso de memoria.

Cando definimos a variable `__slots__` nunha clase, estamos especificando os nomes dos atributos que se permiten nas instancias desa clase. Isto significa que Python só reserva espazo para eses atributos.

```
class ExemploClase:
    __slots__ = ['atributo1', 'atributo2']

    def __init__(self, valor1, valor2):
        self.atributo1 = valor1
        self.atributo2 = valor2

# Creamos un obxecto da clase
objeto = ExemploClase(10, 20)

# Accedemos os atributos da clase
print(objeto.atributo1)
print(objeto.atributo2)

# Intentamos gardar un novo atributo na clase
objeto.atributo3 = 30  # Prodúcese un erro.
```

Utilizando `__slots__` en combinación con `@property` para que os atributos non se poidan modificar, podemos crear unha clase **Inmutable**.

4. HERDANZA



A **herdanza** é unha das grandes cualidades da POO. Cando unha clase herda doutra, adquire os seus atributos e métodos visibles, permitindo reutilizar o código e funcionalidades, que nas clases herdadas se poden ampliar.

A clase da que se herda chámase **superclase** e a clase que herda **subclase**.

Unha subclase dispón de atributos e métodos herdados da superclase. Xeralmente engádense máis atributos e métodos. Isto aumenta a funcionalidade e evita a repetición de código. Na API a maioría de clases non se definen dende cero. Construír clases herdando doutras simplifica o desenvolvemento.

4.1. HERDANZA SIMPLE

A herdanza en Python permite a creación dunha nova clase que toma coma base unha clase xa existente. A nova clase herda atributos e métodos da clase base, e pode agregar ou modificar o seu comportamento segundo sexa necesario.

```
# Clase pai
class Animal:
    # Construtor
    def __init__(self, nome):
        self.nome = nome

    # Metodo
    def facer_son(self):
        print("")

# Clase filla de Animal
class Can(Animal):
    def facer_son(self):
        return "Guau!"

# Clase filla de Animal
class Gato(Animal):
    def facer_son(self):
        return "Miau!"

# Crear instancias das clases derivadas
can = Can(nome="Buddy")
gato = Gato(nome="Whiskers")

# Acceder aos atributos da clase base
print(can.nome)  # Saida: Buddy

# Chama aos metodos da clases derivadas
print(can.facer_son())  # Saida: Guau!
print(gato.facer_son())   # Saida: Miau!
```

- `Animal` é a clase pai cun construtor e un método `facer_son`.
- `Can` e `Gato` son clases fillas da clase `Animal`. Estas fan unha implantación específica para o método `facer_son`.

4.1.1. Pertenza a unha clase

A función `isinstance()` utilízase para verificar se un obxecto é unha instancia dunha clase dada ou dalgunha das súas superclases. A súa sintaxe é a seguinte:

```
isinstance(objeto, clase)
```

Esta devolve `True` en caso afirmativo e `False` en caso contrario.

```
class Persoa:
    pass

class Estudante(Persoa):
    pass

p = Persoa()
e = Estudante()

print(isinstance(p, Persoa))  # Saída: True
print(isinstance(e, Estudante))  # Saída: True
```

```
print(isinstance(e, Persoa))  # Saída: True, xa que Estudante é unha subclase de Persoa
print(isinstance(p, Estudante))  # Saída: False,
```

4.2. SUPER

A chamada da clase pai realízase para invocar o comportamento da clase pai dende unha clase filla. Isto faise utilizando o método `super()`. A chamada a clase pai é útil cando se quere herdar o comportamento da clase pais nunha filla, pero tamén se desexa modificar en parte este comportamento.

4.2.1. Inicialización da clase filla

```
# Definimos a clase pai
class Persoa:
    # Definimos o seu constructor
    def __init__(self, nome):
        self.nome = nome

# Definimos a clase filla
class Estudante(Persoa):
    # Definimos o seu cosntrutor
    def __init__(self, nome, grao):
        super().__init__(nome)  # Chamamos o construtor da clase pai
        self.grao = grao
```

4.2.2. Redifinición de métodos herdados

```
class Animal:
    def facer_son(self):
        return "O animal fixo un son!"

class Can(Animal):
    def facer_son(self):
        son_animal = super().facer_son()  # Chamada ao mñetodo da clase pai
        return "Guau! " + son_animal
```

4.3. HERDANZA MÚLTIPLE

Python permite a herdanza múltiple. Isto significa que unha clase pode herdar de máis dunha clase base. Podes especificar múltiples clases base separándoas por comas na definición da clase.

```
# Clase base 1
class Mamifero:
    def amamantar(self):
        print("Amamantando")

# Clase base 2
class Voador:
    def voar(self):
        print("Voando")

# Clase derivada que herda de Mamifero y Voador
class Morcego(Mamifero, Volador):
    def sonar(self):
        print("Facendo sonar")

# Crear una instancia de la clase derivada
morcego = Morcego()

# Utilizar métodos de ambas clases base
morcego.amamantar()     # Saida: Amamantando
morcego.voar()          # Saida: Voando
morcego.sonar()         # Saida: Facendo sonar
```

A herdanza múltiple pode ser poderosa, pero tamén pode complicar o deseño das clases e aumentar a posibilidade de conflitos en nomes de métodos ou atributos das clases pai. Neste caso utilízase o coñecido como **MRO, Orde de resolución de métodos**, para determinar a orde na que se buscan os métodos nas clases base.

Incluso se poden atopar problemas coñecidos como “problema do diamante” que acontece cando unha clase herda de dúas clases que a súa vez herdan dunha mesma clase,

4.3.1. MRO

O MRO calcúlase seguindo o algoritmo de *C3 linearization*, que é un algoritmo específico de Python para calcular a orde de resolución de métodos en herdanza múltiple.

1. Constrúese unha xerarquía de clases baseados na herdanza, incluídos todas as clases base e a derivada.
2. Determínase a orde de herdanza baseada na declaración da clase derivada e as súas clases base.
3. Aplicación do algoritmo C3.
4. O algoritmo devolve unha lista ordenada de clases que se utilizará para buscar e resolver métodos e atributos.

```
class B():
    def metodo(self):
        print("Método de B")

class C():
    def metodo(self):
        print("Método de C")

class D(B, C):
    pass

# Crea unha instancia da clase derivada D
```

```
objeto_d = D()

# Chama ao método dende a instancia de D
objeto_d.metodo()
```

Neste exemplo, a clase `D` herda das clases `B` e `C`.

Cando se chama a `metodo()`, Python seguira a orde da MRO, que neste caso é `[D, B, C]`. Ista construíuse do seguinte xeito:

1. Engade a clase filla `D`.
2. Engade a primeira clase pai que aparece na definición de `D`, que neste caso é `B`.
3. Engade a seguinte clase pai que aparece na definición de `D`, que neste caso é `C`.

Polo tanto o método da clase `B` será o utilizado, porque `metodo()` non esta definida na clase `D`.

4.4. POLIMORFISMO

En programación orientada a obxectos, o polimorfismo é a propiedade pola que é posible enviar mensaxes sintacticamente iguais a obxectos de tipos distintos. O único requisito que deben cumprir os obxectos que se utilizan de maneira polimórfica é saber responder a mensaxe que se lles envía.

Por exemplo, un obxecto da clase *Liña* e un obxecto da clase *Polígono* poderían ter ambos un método *éMaior* ao que se lle pasaría un número enteiro e terían que devolver verdadeiro ou falso. Como se ve, a chamada a ese método das dúas clases distintas é sintácticamente iguais, pero a súa función é diferente. Mentres que o obxecto da clase *Liña* comprobaría se a súa lonxitude é ou non maior que o número pasado como parámetro, o obxecto da clase *Polígono* comprobaría se a súa área é maior que dito número.

Como Python é unha linguaxe de tipado dinámico, **en Python non é necesario que os obxectos compartan unha mesma interface**. Tan só deben ter métodos co mesmo nome.

```
class Animal:
    def facer_son(self):
        pass

class Can(Animal):
    def facer_son(self):
        print("Guau!")

class Gato(Animal):
    def facer_son(self):
        print("Miau!")

lista_animais =  [Can(), Can(), Can(), Gato(), Gato()]

for animal in lista_animais:
    animal.facer_son()
```

Este é un caso similar ao que se faría noutras linguaxes como Java. Sen embargo, aínda que a clase `Can` e `Gato` non herdaran da mesma clase, funcionaría igual xa que teñen un método co mesmo nome.

4.5. CLASE ABSTRACTA

Para utilizar clases abstractas en Python, utilizamos o módulo `abc` (*Abstract Base Classes*). Unha clase abstracta non pode ser instanciada directamente e xeralmente contén un ou máis métodos abstractos. Estes métodos deben de ser implantados polas clases fillas.

```python
from abc import ABC, abstractmethod

# Para definir unha clase como abstracta debe de herdar da clase ABC
class FiguraXeometrica(ABC):
    # para definir un metodo abstracto utilizamos o decorador abstractmethod
    @abstractmethod
    def calcular_area(self):
        pass

    @abstractmethod
    def calcular_perimetro(self):
        pass

# Clase filla que herda da clase abstracta
class Circulo(FiguraXeometrica):
    def __init__(self, radio):
        self.radio = radio

    # Implantacion do metodo abstracto
    def calcular_area(self):
        return 3.14 * self.radio**2

    # Implantacion do metodo abstracto
    def calcular_perimetro(self):
        return 2 * 3.14 * self.radio

# Intentar instanciar unha clase abstracta xerará un erro
try:
    figura = FiguraXeometrica()
except TypeError as e:
    print(e)

# Crea unha instancia da clase derivada Circulo
circulo = Circulo(radio=5)

# Llamar a los métodos de la clase derivada
print("Área do círculo:", circulo.calcular_area())
print("Perímetro do círculo:", circulo.calcular_perimetro())
```

4.6. INTERFACES

En Python non hai un **concepto directo de interfaces** como noutras linguaxes como Java.

No seu lugar, podemos definir unha clase definindo métodos e deixándoos sen implantación. Outras clases poden entón implantar estes métodos herdando desta.

```
class FiguraXeometrica:
    def calcular_area(self):
        raise NotImplementedError("Este método debe ser implantado nas clases derivadas.")

    def calcular_perimetro(self):
        raise NotImplementedError("Este método debe ser implantado nas clases derivadas.")

class Circulo(Figuraxeometrica):
    def __init__(self, radio):
        self.radio = radio

    def calcular_area(self):
        return 3.14 * self.radio**2

    def calcular_perimetro(self):
        return 2 * 3.14 * self.radio

class Cuadrado(FiguraXeometrica):
    def __init__(self, lado):
        self.lado = lado

    def calcular_area(self):
        return self.lado**2

    def calcular_perimetro(self):
        return 4 * self.lado
```

Outro xeito de utilizar interfaces é definindo unha clase abstracta con métodos abstractos.

4.7. ENUMERANDOS

Podemos utilizar a clase `Enum` do módulo `enum` para crear enumeracións. As enumeracións proporcionan un xeito de presentar valores discretos como nomes e permítete traballar con estes dun xeito claro e lexible.

```python
from enum import Enum

# Definir unha enumeración chamada Cor
class Cor(Enum):
    VERMELLO = 1
    VERDE = 2
    AZUL = 3

# Acceder aos valores da enumeración
print(Cor.VERMELLO)   # Saida: Cor.VERMELLO
print(Cor.VERDE)      # Saida: Cor.VERDE
print(Cor.AZUL)       # Saida: Cor.AZUL

# Comparar valores da enumeración
if Cor.VERMELLO == Cor.VERMELLO:
    print("Son iguais.")
else:
    print("No son iguais.")

# Iterar sobre os membros da enumeración
for color in Cor:
    print(color)

# Obter o nome da enumeración a partir do seu valor
print(Cor(2))  # Saida: Color.VERDE
```

4.8. EXCEPCIONS

Podemos crear excepcións personalizadas en Python definindo unha clase que herde da clase base `Exception` ou dalgunha das súas subclases.

```
class ExcepcionExemplo(Exception):
    def __init__(self, mensaxe="Aconteceu un erro"):
        self.mensaxe = mensaxe
        super().__init__(self.mensaxe)

# Exemplo de uso
def funcion_personalizada(valor):
    if valor < 0:
        raise ExcepcionExemplo("O valor non pode ser negativo")

try:
    funcion_personalizada(-5)
except ExcepcionExemplo as e:
    print(f"Excepción capturada: {e}")
```

4.9. CLASE OBJECT

A clase `object` é a clase base de tódalas clases en Python. Isto significa que tódalas clases herdan directa ou indirectamente desta clase. Esta proporciona métodos e atributos fundamentais que son herdados por tódalas clases.

Se cando se define unha clase, non se especifica ningunha clase pai, esta nova clase será filla de `object`.

4.9.1. Atributos

O atributo `__dict__` é un atributo que contén un dicionario cos atributos dun obxecto. Neste caso as chaves son os atributos e os valores son os propios valores dos atributos.

```
class Persoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

manuel = Persona("Manuel", 35)
print(manuel.__dict__) # Imprime: {'nome': 'Manuel', 'idade': 35}
```

4.9.2. Métodos

Ademais, `object` proporciona varios métodos especiais que se utilizan para implantar comportamentos específicos nas instancias das clases. Estes poden ser sobrecargados. Algúns destes métodos son os seguintes:

- **`__init__(self)`**: Construtor.
- **`__str__(self)`**: Devolve unha cadea que representa o obxecto. Se se personaliza, indicaremos como se verá o obxecto a imprimilo ou a convertelo a unha cadea.
- **`__eq__(self, other)`**: Chámase cando se utiliza o operador de igualdade (`==`). Debe devolver `True` cando dúas instancias son consideradas iguais.

  ```
  class Persoa:
      def __init__(self, nome, idade):
          self.nome = nome
          self.idade = idade

      def __eq__(self, outro):
          if isinstance(outro, Persoa):
              return self.nome == other.nome and self.idade == other.idade
          return False
  ```

- **`__hash__(self)`**: Devolve un valor *hash* utilizado para indexar o obxecto en estruturas de datos como dicionarios.
- **`__len__(self)`**: Devolve a lonxitude da instancia. Utilízase cando se chama a función `len()`.
- **`__copy__(self)` e `__deepcopy__(self)`**: Este método chámase cando se realiza unha copia superficial do obxecto utilizando a función `copy()` do módulo `copy` ou cando se fai unha copia en profundidade ca función do mesmo módulo `deepcopy()`. A diferencia e que `copy()` crea un novo obxecto, pero si este contén outros obxectos(listas, dicionarios, etc.) non crea copias deses obxectos. En cambio se utilizamos a copia completa, `deepcopy()`, estes obxectos internos tamén se copian.

```
import copy

class ExemploClase:
    def __init__(self, x):
        self.x = x

    def __copy__(self):
        # Creamos uhna nova instancia da clase e copiamos o valor de x
        copia = ExemploClase(self.x)
        return copia

# Creamos un obxecto de ExemploClase
obx= ExemploClase(10)

# Realizamos unha copia utilizando copy.copy()
obx2 = copy.copy(obj1)

# Modificamos o valor de x na copia
obx2.x = 20

# Imprimimos os valores para ver que obx1 non se modificou
print(obx1.x)  # Saída: 10
print(obx2.x)  # Saída: 20
```

5. EXERCICIOS

5.1. ENUNCIADOS

Exercicio 1: Lectura de ficheiro CSV

Lee o seguinte ficheiro CSV nun script de Python. O nome do ficheiro recibirase por argumento do *script*. Almacena os datos nunha lista de dicionarios. Imprime por pantalla unicamente o apelido de cada elemento (o campo "nome e apelido" tan só conta cun único e nome e un único apelido).

```
nome e apelido,dni,data contrato,soldo mensual bruto,IRFP
Manuel Varela,12345678a ,12-03-2024 11:03,1200,18
Joe Doe. 98765432b.16-05-2023 11:03 .1500.60.21
Jose Domínguez,123456c,02-11-2023 15:17,800,102
```

Exercicio 2: Procesamento de datos

A partir do exercicio anterior, crea unha función `fila_to_dict(fila)` que a partir de cada fila cree un dicionario cas seguintes claves:

- "nome": tan só o nome. Sen apelidos.
- "apelido": tan só o apelido. Sen o nome.
- "dni": o DNI pero ca letra en maiúscula.
- "data_contrato": a data do contrato pero como tipo de dato `datetime`.
- "soldo_bruto_mensual": o soldo bruto mensual como un `float`.
- "IRPF": o dato d entrada está proporcionado nun valor porcentual e deberemos almacenalo como un `float`, polo que debemos dividir o seu valor entre 100.

NOTA: elimina todos os espazos en branco de cada valor tanto ao comezo como ao final.

Proba a utilizar dita función na lectura do ficheiro realizada no exercicio anterior imprimindo cada dicionario.

Exercicio 3: Expresións regulares

A partir do exercicio anterior, crea unha función `test_dni(dni)` que devolva verdadeiro se un DNI ten o formato de 8 número e unha letra maiúscula. Devolvera falso en caso contrario.

Utilizando dita función e partindo do exercicio anterior, mete só os elementos cun DNI válido nunha lista e por último imprime dita lista.

Exercicio 4: Escribir nun ficheiros JSON

Garda a lista do exercicio anterior nun ficheiro de nome `datos.json`. Terás que primeiro que recorrer a lista para cambiar o tipo de dato `datetime` por unha cadea de texto debido a que este non é *seriazable*. O formato desta cadea debe de ser "ano-mes-dia hora:minuto".

Exercicio 5: Definición de clase pai

Crea o ficheiro `contrato.py` e dentro define a seguinte clase `IRPFValueError` e `Contrato` tendo en conta:

- `IRPFValueError`:
    - Este é unha clase que herda de `Exception`.

    - A mensaxe que se lanzará cando se lance será a seguinte: “Valor de IRPF incorrecto”.

- `Contrato`:
    - Esta clase debe de ser abstracta para que non poida ser instanciada.
    - Necesitamos gardar o número de contrato, data e hora da firma de contrato (como un `datetime`), DNI, nome, apelidos, salario bruto mensual e o seu IRPF (un valor entre 0 e 1). Crea os atributos e utiliza como nome destes a convención para indicar que son privados.
    - Crea o atributo estático `n_contrato_seguinte` co valor inicial 1. Este serviranos para saber que número de contrato asignar cada vez que se cree un novo contrato.
    - Crea un método estático `get_numero_contrato()` que devolva o valor de `n_contrato_seguinte` e aumente o valor deste atributo en estático nunha unidade.
    - Crea un construtor que:
        - Reciba como parámetros a data de firma do contrato (nunha cadea de texto en formato “ano-mes-dia hora:minuto”), o DNI, nome, apelidos, salario bruto mensual e o seu IRPF.
        - O parámetro do IRPF debe de ser opcional. O seu valor por defecto será 0,20. Deste xeito poderemos utilizar sobrecarga de métodos.
        - Débense asignar ditos parámetros aos seus correspondentes atributos.
        - Para asignar o número de contrato utilizaremos o método estático `get_numero_contrato()`.
        - Comproba que se o valor do IRPF non está entre 0 e 1 lance a excepción `IRPFValueError`.
    - Crea os *getters* para tódolos atributos menos para o nome e apelidos. Utiliza `property`.
    - Crea un *getter* para o nome completo co formato “Apelidos, nome”. Utiliza `property`.
    - Crea os *setters* tan so para o atributo do IRPF onde comprobes que o valor está entre 0 e 1, senón lánzase a excepción `IRPFValueError`. Utiliza `property`.
    - Crea o método `salario_neto_anual()` que devolva o salario neto gañado nun ano.
    - Crea o método abstracto `aumentar_soldo_bruto_mensual()` que devolva o salario neto gañado nun ano.
    - Sobrecarga o método `__string__` devolvendo unha cadea que mostre o número de contrato xunto o nome, apelidos e DNI deste.

Exercicio 5: Definición das clases fillas

Crea o ficheiro `tipos_contrato.py` e dentro define a seguinte enumeración:

- `Titulacion`.
    - Este é unha clase que herda de `Enum`.
    - Debemos gardar catro valores: “SMR”, “ASIR”, “DAM” e “DAW”.

Nese mesmo ficheiro crea as seguintes clases:

- `ContratoIndefinido`.
    - Esta é unha subclase de `Contrato`.
    - Necesitamos gardar como atributo para se as pagas extras están ou non prorateadas.
    - Redefine o construtor para incluír os atributos propios desta clase.
    - Implanta o método `aumentar_soldo_bruto_mensual()` para que cada vez que se chame aumente nun 10% o soldo bruto mensual.

- `ContratoTemporal`.
  - Esta é unha subclase de `Contrato`.
  - Necesitamos gardar como atributo os meses de duración do contrato.
  - Redefine o construtor para incluír os atributos propios desta clase.
  - Redefine método `salario_neto_anual()` para que devolva o salario neto que se cobraría só nos meses de duración do contrato. Utiliza o método da clase pai.
  - Implanta o método `aumentar_soldo_bruto_mensual()` para que cada vez que se chame aumente en 100€ o soldo bruto mensual.
- `ContratoPrácticas`.
  - Esta é unha subclase de `Contrato`.
  - Necesitamos gardar como atributo a titulación que está estudando. Este será un valor da enumeración `Titulacion`.
  - Redefine o construtor para incluír os atributos propios desta clase.
  - Implanta o método `aumentar_soldo_bruto_mensual()` para que cada vez que se chame aumente o soldo bruto mensual segundo a titulación. Se é de "SMR" subiráselle 50€ e en caso contrario 75€.

Exercicio 6: Proba de POO

Crea un programa que lea o seguinte JSON:

```
[
    {
        "nome": "Manuel",
        "apelido": "Varela",
        "dni": "12345678A",
        "data_contrato": "2024-03-12 11:03",
        "soldo_bruto_mensual": 1200.0,
        "IRPF": 0.18,
        "tipo": "Temporal",
        "duracion": 9
    },
    {
        "nome": "Joe",
        "apelido": "Doe",
        "dni": "98765432B",
        "data_contrato": "2023-05-16 11:03",
        "soldo_bruto_mensual": 1500.6,
        "IRPF": 0.21,
        "tipo": "Indefinido",
        "prorateado": true
    },
    {
        "nome": "Diego",
        "apelido": "Tristán",
        "dni": "23456789C",
        "data_contrato": "2022-07-11 15:12",
        "soldo_bruto_mensual": 1510.6,
        "IRPF": 0.12,
        "tipo": "Prácticas",
        "titulacion": "DAM"
    },
    {
        "nome": "Pedro",
        "apelido": "Sánchez",
        "dni": "23456780D",
        "data_contrato": "2024-01-11 19:14",
        "soldo_bruto_mensual": 2002.31,
        "IRPF": 0.40,
        "tipo": "Temporal",
        "duracion": 6
    },
```

```
    {
        "nome": "Fernando",
        "apelido": "Alonso",
        "dni": "33333333E",
        "data_contrato": "2023-03-03 15:03",
        "soldo_bruto_mensual": 5000.33,
        "tipo": "Indefinido",
        "prorateado": false
    }
]
```

A partir deste JSON, crea os obxectos correspondentes e gárdaos nunha lista. Podes crear unha función `dict_to_contrato(contrato_dict)` que realice dita tarefa. Recorda capturar a excepción `IRPFValueError` cada vez que crees un obxecto novo.

Recorre a lista e aumenta o soldo de todos aqueles que leven máis de tres meses.

Por último imprime os datos de todos aqueles que cobren anualmente máis de 25000€ utilizando a función `str`.
