Sessió 1

Taula de continguts

Programació Orientada a Objectes

> **Nota**
>
> Aquest tutorial està adaptat del curs CS50P Introduction to Programming with Python.

Plantejament

Suposem que volem programar un eina per gestionar cursos i assignatures.

Per començar, prenem les dades d'un alumne i l'assignatura en què es vol matricular.

```
nom = input("Nom: ")
assignatura = input("Assignatura: ")
print(f"{nom} a {assignatura}")
```

Podem estructurar-ho una mica més, fent funcions que abstreuen cadascun dels passos. Així, si més endavant volem afegir funcionalitat, sabrem quina funció s'encarrega de cada cosa.

```python
def main():
    nom = get_nom()
    assignatura = get_assignatura()
    print(f"{nom} a {assignatura}")

def get_nom():
    return input("Nom: ")

def get_assignatura():
    return input("Assignatura: ")

if __name__ == "__main__":
    main()
```

Com que l'estudiant i l'assignatura haurien d'anar junts, podem organitzar-los en una `tuple`:

```python
def main():
    nom, assignatura = get_estudiant()
    print(f"{nom} a {assignatura}")

def get_estudiant():
    nom = input("Nom: ")
    assignatura = input("Assignatura: ")
    return nom, assignatura

if __name__ == "__main__":
    main()
```

De fet, pensar en a tupla directament:

```python
def main():
    estudiant = get_estudiant()
    print(f"{estudiant[0]} a {estudiant[1]}")

def get_estudiant():
    nom = input("Nom: ")
    assignatura = input("Assignatura: ")
    return (nom, assignatura)

if __name__ == "__main__":
    main()
```

L'avantatge de fer servir `tuple` és que no es pot modificar. Si la funció `main()` intenta modificar el valor de `estudiant`, obtindrem un error.

Una altra possibilitat seria fer servir un diccionari, de manera que no haguem de recordar en quina posició hi ha el nom i en quina l'assignatura:

```
def main():
    estudiant = get_estudiant()
    print(f"{estudiant['nom']} a {estudiant['assignatura']}")

def get_estudiant():
    nom = input("Nom: ")
    assignatura = input("Assignatura: ")
    return {'nom' : nom, 'assignatura' : assignatura}

if __name__ == "__main__":
    main()
```

Desavantatges que podem trobar en aquesta implementació:

- El diccionari és mutable. La funció `main()` pot canviar les dades sense voler.
- Ens cal documentar en algun lloc les claus que farà servir el diccionari.
- A la funció `get_estudiant()` hi ha moltes paraules repetides…

Classes i objectes

Python ens dona una manera de crear els nostres propis tipus. Els diccionaris són un tipus genèric, però si Python ens proporcionés un tipus `Estudiant` que contingués les dades relacionades amb un estudiant, encara seria millor. Aquesta és la funció de les classes.

> **Nota**
>
> Podem pensar una classe com un *plànol*, a partir de la qual es creen *objectes* o *instàncies*. Cadascun d'aquests objectes contindrà dades diferents, però estaran estructurades tal i com dicti la classe.

```
class Estudiant:
    pass

def main():
    estudiant = get_estudiant()
    print(f"{estudiant.nom} a {estudiant.assignatura}")
```

```
def get_estudiant():
    estudiant = Estudiant()
    estudiant.nom = input("Nom: ")
    estudiant.assignatura = input("Assignatura: ")
    return estudiant

if __name__ == "__main__":
    main()
```

> **Nota**
>
> Per convenció, els noms de les classes s'escriuen en Majúscula. Les excepcions són les classes que Python ja ens dona: `list`, `tuple`, `int`, `dict`,...

El codi anterior no és gaire *Pythonic*: encara que hem donat nom als *atributs* que conformen un estudiant, els hem d'assignar manualment. Una millor versió seria la següent:

```
class Estudiant:
    def __init__(self, nom, assignatura):
        self.nom = nom
        self.assignatura = assignatura

def main():
    estudiant = get_estudiant()
    print(f"{estudiant.nom} a {estudiant.assignatura}")


def get_estudiant():
    nom = input("Nom: ")
    assignatura = input("Assignatura: ")
    return Estudiant(nom, assignatura)

if __name__ == "__main__":
    main()
```

Una altra avantatge d'aquest punt de vista és l'*encapsulació*: tot el que estigui relacionat amb la definició d'un nou estudiant hauri de pertanyer a la classe `Estudiant`. Per exemple, podem controlar errors:

```
class Estudiant:
    def __init__(self, nom, assignatura):
        if not nom:
            raise ValueError("Falta el nom")
        if assignatura not in \
            ["Àlgebra", "Estructures", "Galois", "Aritmètica", "Commutativa"]:
            raise ValueError("Assignatura no vàlida")

        self.nom = nom
        self.assignatura = assignatura

def main():
    estudiant = get_estudiant()
    print(f"{estudiant.nom} a {estudiant.assignatura}")


def get_estudiant():
    nom = input("Nom: ")
    assignatura = input("Assignatura: ")
    return Estudiant(nom, assignatura)

if __name__ == "__main__":
    main()
```

Seguim amb la idea d'encapsulació: fixem-nos que la feina de generar un `str` amb les dades de l'estudiant també la podem delegar a la classe `Estudiant`:

```
class Estudiant:
    def __init__(self, nom, assignatura):
        if not nom:
            raise ValueError("Falta el nom")
        if assignatura not in \
            ["Àlgebra", "Estructures", "Galois", "Aritmètica", "Commutativa"]:
            raise ValueError("Assignatura no vàlida")

        self.nom = nom
        self.assignatura = assignatura

    def __str__(self):
        return f"{self.nom} a {self.assignatura}"

def main():
```

```
    estudiant = get_estudiant()
    print(estudiant)


def get_estudiant():
    nom = input("Nom: ")
    assignatura = input("Assignatura: ")
    return Estudiant(nom, assignatura)

if __name__ == "__main__":
    main()
```

Els *mètodes* `__init__()` i `__str__` els proporciona Python per defecte, i són especials. Per això porten la doble barra baixa (*double under* o *dunder* en anglès). Però també podem inventar-nos els nostres propis mètodes. Per exemple, suposem que volem inferir el curs de l'estudiant a partir de l'assignatura que fa. Ho podem fer així:

```
class Estudiant:
    def __init__(self, nom, assignatura):
        if not nom:
            raise ValueError("Falta el nom")
        if assignatura not in \
            ["Àlgebra", "Estructures", "Galois", "Aritmètica", "Commutativa"]:
            raise ValueError("Assignatura no vàlida")

        self.nom = nom
        self.assignatura = assignatura

    def __str__(self):
        return f"{self.nom} a {self.assignatura}"

    def curs(self):
        match self.assignatura:
            case 'Àlgebra':
                return 1
            case 'Estructures':
                return 2
            case 'Galois':
                return 3
            case _:
                return 4
```

```python
def main():
    estudiant = get_estudiant()
    print(estudiant)
    print('Curs probable:', estudiant.curs())


def get_estudiant():
    nom = input("Nom: ")
    assignatura = input("Assignatura: ")
    return Estudiant(nom, assignatura)

if __name__ == "__main__":
    main()
```

> Atenció
>
> Les classes tenen *atributs* (no pas variables) i *mètodes* (no pas funcions). És simplement terminologia.

Propietats

Encara que ens hem esforçat a fer les comprovacions d'errors quan creem una instància d'`Estudiant`, els atributs de la classe es poden modificar a qualsevol lloc del programa. Per exemple:

```python
class Estudiant:
    def __init__(self, nom, assignatura):
        if not nom:
            raise ValueError("Falta el nom")
        if assignatura not in \
            ["Àlgebra", "Estructures", "Galois", "Aritmètica", "Commutativa"]:
            raise ValueError("Assignatura no vàlida")

        self.nom = nom
        self.assignatura = assignatura

    def __str__(self):
        return f"{self.nom} a {self.assignatura}"

def main():
    estudiant = get_estudiant()
```

```
    estudiant.assignatura = 'Anàlisi funcional'
    print(estudiant)

def get_estudiant():
    nom = input("Nom: ")
    assignatura = input("Assignatura: ")
    return Estudiant(nom, assignatura)

if __name__ == "__main__":
    main()
```

Aquest comportament és indesitjable, i hi ha una manera fàcil de millorar-ho. Es tracta d'afegir mètodes que modifiquin els atributs, i amagar d'alguna manera els propis atributs. Hi ha dues maneres de fer-ho:

1. Escriure mètodes `get_nom()` i `set_nom()`.
2. Fent servir el decorador `property`.

El codi queda així, fent servir les dues variants. Podem provar de canviar a una assignatura no vàlida i veurem el resultat.

```
class Estudiant:
    def __init__(self, nom, assignatura):
        self.set_nom(nom)
        self.assignatura = assignatura

    def get_nom(self):
        return self._nom

    def set_nom(self, nom):
        if not nom:
            raise ValueError("Nom invàlid")
        self._nom = nom

    @property
    def assignatura(self):
        return self._assignatura

    @assignatura.setter
    def assignatura(self, assignatura):
        if assignatura not in \
            ["Àlgebra", "Estructures", "Galois", "Aritmètica", "Commutativa"]:
```

```
            raise ValueError("Assignatura no vàlida")
        self._assignatura = assignatura

    def __str__(self):
        return f"{self.get_nom()} a {self.assignatura}"

def main():
    estudiant = get_estudiant()
    print(estudiant)

def get_estudiant():
    nom = input("Nom: ")
    assignatura = input("Assignatura: ")
    return Estudiant(nom, assignatura)

if __name__ == "__main__":
    main()
```

> **Atenció**
>
> Els mètodes i atributs que comencen amb `_` es consideren privats. Hi ha llenguatges de programació que no permeten accedir als mètodes/atributs privats des de fora la classe. Python funciona amb un *pacte de cavallers*: si el programador de la classe hi ha posat una `_`, vol dir *no ho toquis*. Si hi posa dues barres baixes `__` vol dir que *no ho toquis, de veritat*. Però en ambdós casos s'assumeix que l'usuari de la classe és una adult responsable, i no *Python* no s'hi posa.

L'avantatge de com hem implementat `assignatura` és que si la classe ja s'estava utilitzant no haurem de canviar res del codi. Diem que l'API de la nostra classe no canvia. D'altra banda, hem d'anar amb compte amb ells *getters* i els *setters*. Quan l'usuari assigna o llegeix un atribut, no espera que hi pugui haver errors i per tant no programarà els `try...except`. Això vol dir que hem de ser molt curosos amb el codi que hi posem, o acabarem causant més problemes dels què hem resolt. Si el codi fa moltes comprovacions que poden ser problemàtiques, sovint és més expressiu implementar mètodes de la forma `get_nom()` i `set_nom()`.

Mètodes de classe

Si ens fixem en el codi anterior, hi ha una funcionalitat molt relacionada amb estudiants que encara no hem incorporat a la classe. Es tracta de la funció `get_estudiant()`. Aquesta funció crea un estudiant nou a partir de l'entrada de l'usuari. D'una banda hauria de pertanyer a

la classe `Estudiant`, però d'altra banda no té massa sentit haver de crear un estudiant "de mentida" per poder accedir al mètode en qüestió.

Els mètodes de classe s'utilitzen quan el mètode que volem implementar no depèn de les dades de l'objecte en concret, sinó que és comú a tots els objectes. La variable `self` no hi és, i el primer paràmetre s'anomena `cls` i és la pròpia classe. Queda així:

```
class Estudiant:
    def __init__(self, nom, assignatura):
        self.nom = nom
        self.assignatura = assignatura

    @property
    def nom(self):
        return self._nom

    @nom.setter
    def nom(self, nom):
        if not nom:
            raise ValueError("Nom invàlid")
        self._nom = nom

    @property
    def assignatura(self):
        return self._assignatura

    @assignatura.setter
    def assignatura(self, assignatura):
        if assignatura not in \
            ["Àlgebra", "Estructures", "Galois", "Aritmètica", "Commutativa"]:
            raise ValueError("Assignatura no vàlida")
        self._assignatura = assignatura

    def __str__(self):
        return f"{self.nom} a {self.assignatura}"

    @classmethod
    def get(cls):
        nom = input("Nom: ")
        assignatura = input("Assignatura: ")
        return cls(nom, assignatura)

def main():
```

```
    estudiant = Estudiant.get()
    print(estudiant)

if __name__ == "__main__":
    main()
```

Herència

Suposem que ara volem crear una nova classe pels professors. Hauríem de fer com amb l'estudiant, però no tindria una assignatura assignada. Posem que de cada professor en volem desar el departament al qual pertany. La classe començaria com:

```
class Professor:
    def __init__(self, nom, departament):
        self.nom = nom
        self.departament = departament

    @property
    def nom(self):
        return self._nom

    @nom.setter
    def nom(self, nom):
        if not nom:
            raise ValueError("Nom invàlid")
        self._nom = nom
```

Per evitar repetir codi, podem crear una classe `Persona` que s'encarregui del nom. Aleshores, tant `Estudiant` com `Professor` *hereden* les característiques (atributs i mètodes) de `Persona`. La classe `Persona` es considera una *abstracció* o *generalització* de les classes `Estudiant` i `Professor`:

```
class Persona:
    def __init__(self, nom):
        self.nom = nom

    @property
    def nom(self):
        return self._nom

    @nom.setter
```

```
    def nom(self, nom):
        if not nom:
            raise ValueError("Nom invàlid")
        self._nom = nom

    def __str__(self):
        return self.nom

class Professor(Persona):
    def __init__(self, nom, departament):
        super().__init__(nom)
        self.departament = departament

    def __str__(self):
        return f"{self.nom} del departament de {self.departament}"

class Estudiant(Persona):
    def __init__(self, nom, assignatura):
        super().__init__(nom)
        self.assignatura = assignatura

    def __str__(self):
        return f"{self.nom} a {self.assignatura}"

    @classmethod
    def get(cls):
        nom = input("Nom: ")
        assignatura = input("Assignatura: ")
        return cls(nom, assignatura)

print(Persona('Marc'))
print(Professor('Joaquim', 'Matemàtiques'))
print(Estudiant('Jordi', 'Galois'))
```

Qualsevol mètode que accepti objectes de tipus `Persona` podrà treballar amb objectes `Professor` i `Estudiant`. Això és el què es coneix com a *polimorfisme*. Per exemple, una funció que ens busqui una persona en una llista:

```
def busca_persona(nom, llista):
    for pers in llista:
        if pers.nom == nom:
            return pers
```

```
    return None

busca_persona('Jordi', [Professor('Marc', 'Mates'), Persona('Maria'), Estudiant('Jordi', 'Gal
```

```
def busca_persona(llista, nom):
```

Sobrecàrrega d'operadors

Tarjeta moneder

Suposem que estem intentant implementar una tarjeta moneder de la UAB, que pot contenir diners, crèdit per fotocòpies, i viatges de bus. Podem fer la classe `Tarjeta` així:

```
class Tarjeta:
    def __init__(self, diners, fotocopies, viatges):
        self.diners = float(diners)
        self.fotocopies = int(fotocopies)
        self.viatges = int(viatges)

    def __str__(self):
        return f'{self.diners:.2f} € '\
            f'{self.fotocopies} fulls, '\
            f'{self.viatges} viatges'

print(Tarjeta(1.2, 130, 8))
```

Si tenim dues tarjetes com l'anterior, potser ens interessa sumar els valors corresponents. Python ens permet implementar un mètode que es cridarà quan fem servir l'operació `+`:

```
class Tarjeta:
    def __init__(self, diners, fotocopies, viatges):
        self.diners = float(diners)
        self.fotocopies = int(fotocopies)
        self.viatges = int(viatges)

    def __add__(self, other):
        return Tarjeta(self.diners + other.diners,
            self.fotocopies + other.fotocopies,
            self.viatges + other.viatges)
```

```
    def __str__(self):
        return f'{self.diners:.2f} €, '\
        f'{self.fotocopies} fulls, '\
        f'{self.viatges} viatges'

print(Tarjeta(1.2, 130, 8))
print(Tarjeta(2.3, 21, 5))
print('Suma:', Tarjeta(1.2, 130, 8) + Tarjeta(2.3, 21, 5))
```

Hi ha molts altres mètodes *especials* com aquest. Se'ls anomena **mètodes màgics**, o **dunder** (de **d**ouble under), i es poden trobar aquí.