

Fonaments de Python

Iloc: [Institut d'Ensenyaments a Distància de les Illes Balears](#)
Curs: Programació d'intel·ligència artificial
Llibre: Fonaments de Python

Imprès per: Carlos Sanchez Recio
Data: dimecres, 2 d'octubre 2024, 20:33

Taula de continguts

1. Introducció

2. Llenguatges de programació per a Intel·ligència Artificial

- 2.1. Compilació i interpretació
- 2.2. Característiques d'un llenguatge per a IA i BD
- 2.3. Per què Python?

3. Entorn de desenvolupament

- 3.1. IPython
- 3.2. Jupyter Notebook
- 3.3. JupyterLab
- 3.4. Google Colab

4. Fonaments del llenguatge Python

- 4.1. Estructura del codi: sagnia, no claus
- 4.2. Tot són objectes
- 4.3. Entrada i sortida bàsiques
- 4.4. Crida a funcions i mètodes
- 4.5. Variables i pas d'arguments
- 4.6. Referències dinàmiques
- 4.7. Objectes mudables i immutables
- 4.8. Atributs i mètodes
- 4.9. Convencions en el codi
- 4.10. Importacions de mòduls
- 4.11. Operadors binaris i comparacions
- 4.12. assert

5. Tipus escalar

- 5.1. Tipus numèrics
- 5.2. Cadenes de caràcters
- 5.3. Booleans
- 5.4. Conversió de tipus
- 5.5. None
- 5.6. Dates i hores

6. Estructures de control

- 6.1. Estructures condicionals: if, elif, else
- 6.2. Expressions ternàries
- 6.3. Estructures iteratives: bucles while
- 6.4. Estructures iteratives: bucles for
- 6.5. El tipus range
- 6.6. L'operació buida: pass

7. Estructures de dades

- 7.1. Tuples, llistes, diccionaris i conjunts

8. Funcions

9. Fitxers i sistema operatiu

10. Classes i objectes

1. Introducció

En aquest primer lliurament del mòdul de Programació d'Intel·ligència Artificial farem primer una breu introducció als llenguatges de programació i veurem perquè Python s'ha convertit en el llenguatge més utilitzat per la comunitat d'intel·ligència artificial i ciència de dades. Python serà també el llenguatge que emprarem al llarg del curs, tant en aquest mòdul com en la resta.

A continuació veurem els entorns que utilitzarem per desenvolupar codi Python. Començarem amb IPython (Python interactiu) i després veurem què són els quaderns Jupyter. I acabarem introduint Colab, l'eina de Google que permet executar els quaderns Jupyter en servidors de Google. Colab s'ha convertit en una eina molt popular, especialment en l'àmbit de la intel·ligència artificial i la ciència de dades, per escriure i compartir codi Python.

A continuació, en la resta del lliurament estudiarem en profunditat els fonaments del llenguatge Python.

2. Llenguatges de programació per a Intel·ligència Artificial



Un **llenguatge de programació** és un llenguatge formal (un llenguatge artificial amb unes regles sintàctiques i semàntiques estrictes) que li proporciona a una persona (el programador) la capacitat d'escriure (o programar) una sèrie d'instruccions o seqüències d'ordres amb la finalitat de controlar el comportament físic o lògic d'un sistema informàtic, de manera que es puguin obtenir diverses classes de dades o executar determinades tasques. A tot aquest conjunt d'ordres escrites mitjançant un llenguatge de programació se'l denomina programa informàtic.

DEFINICIÓ

Des de l'aparició de Fortran, el primer llenguatge de programació d'alt nivell, en 1957, s'han desenvolupat una gran quantitat de llenguatges. Alguns d'ells tenen un propòsit general i d'altres han estat dissenyats per resoldre problemes molt específics. Tradicionalment aquests llenguatges s'han classificat en dos grans grups segons el paradigma de programació que suporten: programació imperativa i programació declarativa.

La **programació imperativa** descriu, passa a passa, un conjunt d'instruccions (seguint un algorisme) que s'han d'executar per resoldre un problema. Alguns exemples molt coneguts són Pascal, C o Basic.

Dins la programació imperativa s'ha desenvolupat un subparadigma: la **programació orientada a objectes**, que es basa en encapsular elements denominats objectes que contenen tant variables (també anomenades propietats o atributs) com funcions (anomenats mètodes) i que es comuniquen a través de missatges, amb algunes característiques addicionals com l'herència i el polimorfisme. Però la base segueix essent la de la programació imperativa. El primer llenguatge orientat a objectes va ser Smalltalk, i alguns dels més coneguts són Java, C++ i C#. Aquests han estat els llenguatges més utilitzats en les dues primeres dècades del segle.

En canvi, en la **programació declarativa** el que es fa és descriure el problema que es vol solucionar però no les instruccions necessàries per solucionar-lo. Dins de la programació declarativa trobam també dos subparadigmes: la programació funcional i la programació lògica.

En la **programació funcional**, els programes es basen en l'ús de vertaderes funcions matemàtiques (que tenen algunes diferències importants amb les funcions dels llenguatges imperatius), fonamentades en el càlcul lambda, un sistema formal desenvolupat a la dècada del 1930. Els llenguatges funcionals prioritzen l'ús de la recursivitat i aplicació de funcions d'ordre superior per resoldre problemes que en un llenguatge imperatiu es faria mitjançant estructures de control. Lisp i Haskell són dos dels llenguatges funcionals més coneguts.

La **programació lògica** es fonamenta en la lògica matemàtica (normalment la lògica de primer ordre) i consisteix en declarar el coneixement mitjançant un conjunt de regles i aplicar mecanismes d'inferència per obtenir les solucions. El llenguatge lògic més conegut és Prolog.

Tot i això, cada vegada més els llenguatges segueixen un enfocament **multi-paradigma**. El cas més representatiu és Python, un llenguatge que suporta la programació imperativa, l'orientació a objectes i la programació funcional.

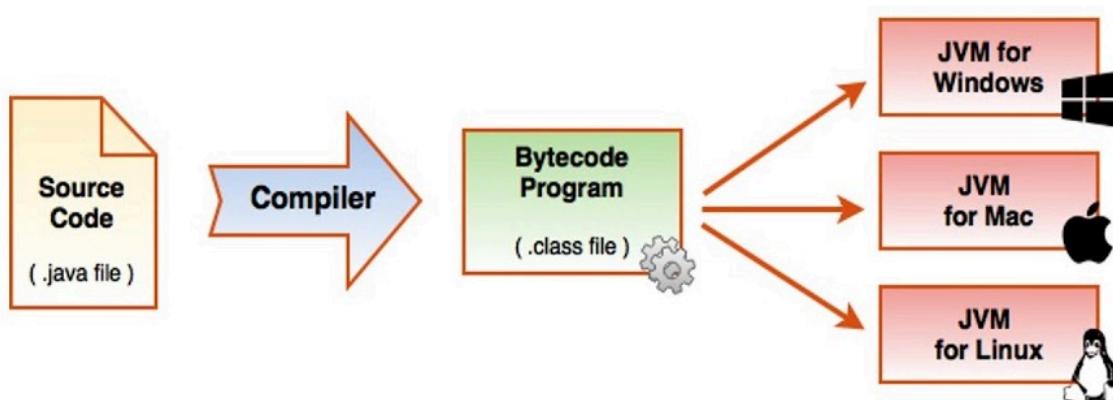
2.1. Compilació i interpretació

Sigui quin sigui el paradigma de programació en què es basa un llenguatge de programació, les instruccions (o funcions o regles) escriptes en un llenguatge d'alt nivell s'han de traduir a llenguatge màquina per poder ser executades. Aquesta traducció pot fer-se mitjançant dos mecanismes diferents: la compilació i la interpretació.

- **Compilació:** abans d'executar el nostre programa, el compilem. El compilador primer reconeix si el nostre codi està escrit seguint les regles del llenguatge i si no hi ha errors, el tradueix, tot sencer, a codi màquina. El que posteriorment s'executa és el codi màquina que s'ha generat. La majoria dels llenguatges imperatius són compilats.
- **Interpretació:** les instruccions del programa, una a una, es van traduint i executant. Qui fa aquest procés és l'intèrpret. Els llenguatges declaratius soLEN ser interpretats.

En general, un programa compilat és més eficient, perquè el codi s'ha traduït a codi màquina abans d'executar-se, mentre que en un programa interpretat, el codi s'ha de traduir mentre es va executant, la qual cosa suposa una sobrecàrrega (*overhead*). Per altra banda, en un programa compilat, l'usuari final no necessita tenir el compilador (és el programador qui el compila), mentre que en un programa interpretat, l'usuari final necessita tenir un intèrpret per poder traduir i executar el codi. En canvi, el programador haurà de compilar el codi font per a tantes plataformes com sigui necessari, perquè el codi màquina obtingut pel compilador serà diferent per a cada una d'elles. Per últim, si un programa interpretat té errors de sintaxi, no els detectam fins que no s'executa.

El llenguatge Java va introduir una solució mixta: primer, un compilador comprova la sintaxi del programa i fa una traducció del codi. Però no ho fa directament al codi màquina de la màquina física, sinó al codi màquina del que s'anomena una màquina virtual, la Java Virtual Machine (JVM). A aquest codi màquina virtual se li anomena **bytecode**. La màquina virtual és en realitat un intèrpret que anirà executant (traduint a codi màquina real) instrucció a instrucció del bytecode. Aquest esquema es basa en que totes les plataformes tindran disponible una JVM. D'aquesta manera, el bytecode del nostre programa es podrà executar, sense haver de fer cap canvi en qualsevol plataforma: és el que s'ha denominat *write once, run everywhere* (WORA).



Imatge: Esquema de compilació-interpretació de Java

Què guanyam amb això? Primer, que els errors de sintaxi els detectam en temps de compilació, no d'execució. Segon, que el programador només ha de compilar una vegada, no ha de preveure totes les possibles plataformes on vol que el seu programa se pugui executar i compilar separatadament per a cada una d'elles. I tercer, que el bytecode ja està optimitzat perquè la seva interpretació sigui més eficient que la dels llenguatges d'alt nivell interpretats. Si que és veritat que hi ha una sobrecàrrega perquè al final, quan s'executa hi ha un procés d'interpretació, però aquesta és menor. Resumint el codi màquina és més ràpid que el bytecode, però el bytecode és més portable i segur. Tot això ha permès que Java sigui el llenguatge més utilitzat per a aplicacions distribuïdes i multi-plataforma.

Aquest esquema mixt basat en bytecode ha estat finalment utilitzat per la majoria dels llenguatges interpretats per tal de millorar la seva eficiència. És el cas de Python, un llenguatge interpretat però que també utilitza bytecode.

Quan l'intèrpret CPython (el més emprat) executa un programa escrit en Python (fitxer .py), primer el tradueix a

bytecode i el guarda en un fitxer .pyc. Després aquest bytecode és executat per la màquina virtual de CPython. És per això que si un programa s'executa diverses vegades, la primera vegada és més lenta perquè ha de generar el bytecode, però les següents ja són més ràpides perquè fa servir el bytecode generat.

Vegem un exemple senzill. Aquest és un exemple d'una funció que escriu el clàssic "Hello, World!" en Python (no entram ara a comentar la sintaxi, es veurà en capítols posteriors):

```
def hello()
    print("Hello, World!")
```

I això és el bytecode generat (en realitat és un fitxer binari, però aquí el mostrem d'una manera llegible):

2	0 LOAD_GLOBAL	0 (print)
	2 LOAD_CONST	1 ('Hello, World!')
	4 CALL_FUNCTION	1



Scala és un llenguatge de programació de propòsit general multi-paradigma, que combina característiques de programació orientada a objectes i programació funcional.

Scala segueix també un model basat en bytecode, amb la particularitat que el bytecode de Scala és pràcticament idèntic al de Java. Això fa que el codi Scala pugui ser executat sobre una màquina virtual Java (JVM). Així doncs, aprofitant la robustesa de l'ecosistema de Java i el fet que hi ha JVMs disponibles per a pràcticament qualsevol plataforma, el codi Scala pot executar-se de manera eficient sobre pràcticament qualsevol sistema. La integració entre aplicacions Java i Scala és molt senzilla, ja que s'executen sobre la mateixa JVM. Per exemple, des de codi Scala es poden emprar biblioteques escrites en Java i viceversa.

D'altra banda, Scala ofereix algunes característiques avançades que milloren algunes de les limitacions de Java, com les funcions d'ordre superior, les col·leccions immutables i les inferències de tipus, cosa que permet desenvolupar aplicacions més elegants i eficients.

El framework Apache Spark per a l'anàlisi de dades massives, que veurem en profunditat en els mòduls de Sistemes de big data i Big data aplicat, està escrit en Scala.

2.2. Característiques d'un llenguatge per a IA i BD

Fins ara hem xerrat de llenguatges de programació en general. Però, quines característiques serien desitjables que tengués un llenguatge per a ser utilitzat en l'àmbit de la intel·ligència artificial i del big data? Aquestes són cinc característiques i la raó per la qual es consideren les més importants en aquest àmbit:

1. **Simplicitat**, a causa de la importància de poder escriure i mantenir codi fàcilment.
2. **Capacitat de prototipatge ràpid**, a causa de la pròpia naturalesa de les aplicacions d'IA i BD, que solen variar considerablement la seva implementació en les primeres etapes.
3. **Llegibilitat**, a causa de la necessitat d'acostar el màxim possible al programador i al codi, considerant que un nombre significatiu de desenvolupaments en aquest àmbit parteixen de pseudocodi o algorismes prèviament dissenyats.
4. **Existència de biblioteques per a IA i BD**, a causa de la importància de reutilitzar codi existent que acceleri la generació de prototips.
5. **Comunitat de desenvolupament**, a causa dels avantatges que ofereix compartir experiències i solucions a problemes prèviament abordats per una altra persona.

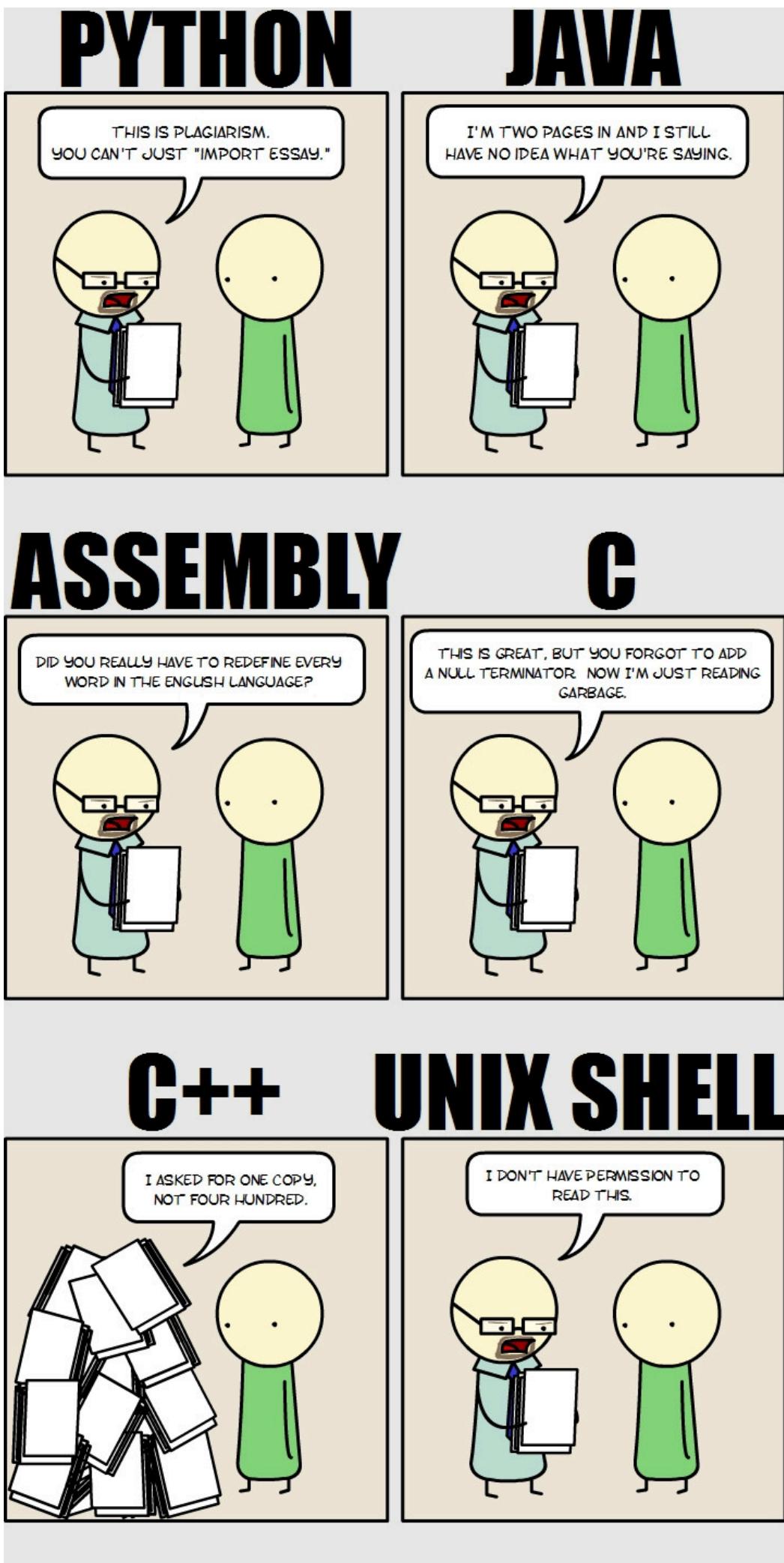
2.3. Per què Python?

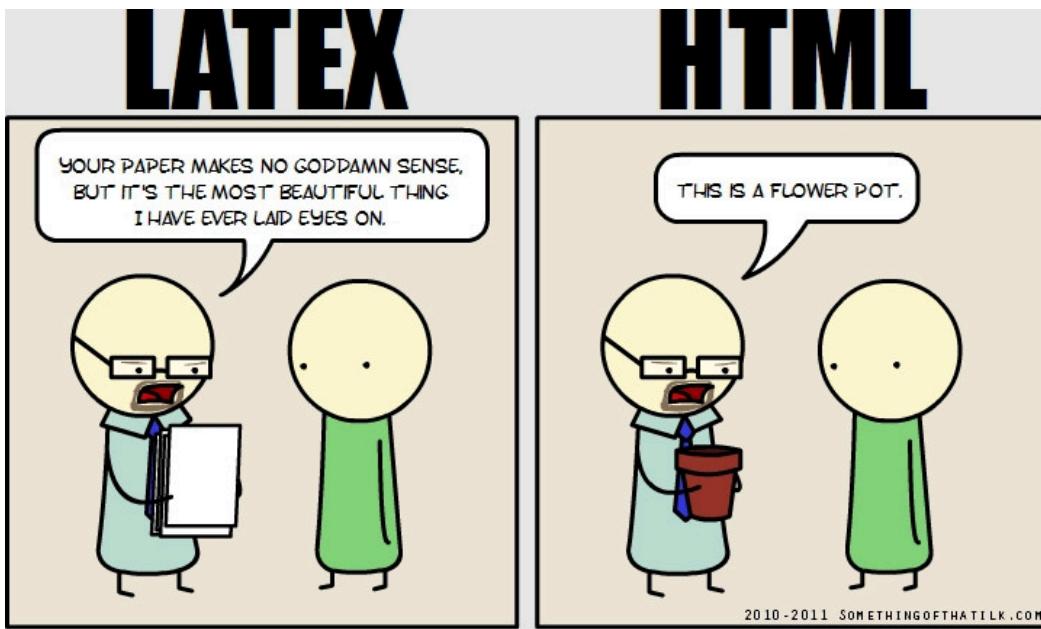
Ja hem dit anteriorment que Python és un llenguatge multi-paradigma que suporta programació imperativa, orientada a objectes i funcional. També hem comentat que és un llenguatge interpretat, tot i que utilitza bytecode per a millorar l'eficiència.

Python és un llenguatge de programació més lent que altres com C, C++ o fins i tot Java. Aquestes són les causes principals:

- És un llenguatge de programació de més alt nivell, més proper a com pensam els humans que, per exemple C o C++. Això fa que hagi d'absttreure alguns detalls com ara la gestió de memòria. Per això és més lent que altres llenguatges de més baix nivell.
- És interpretat, raó per la qual és més lent que llenguatges compilats purs com C o C++.
- Té una definició de tipus dinàmica. En C, C++ o Java hem de declarar les variables i dir de quin tipus són perquè es pugui reservar l'espai corresponent en memòria. Això no passa en Python i per això l'execució és més lenta i també més difícil d'optimitzar.
- Python no suporta multi-threading: només un thread pot prendre el control de l'intèrpret de Python. És el que s'anomena el Global Interpreter Lock (GIL).

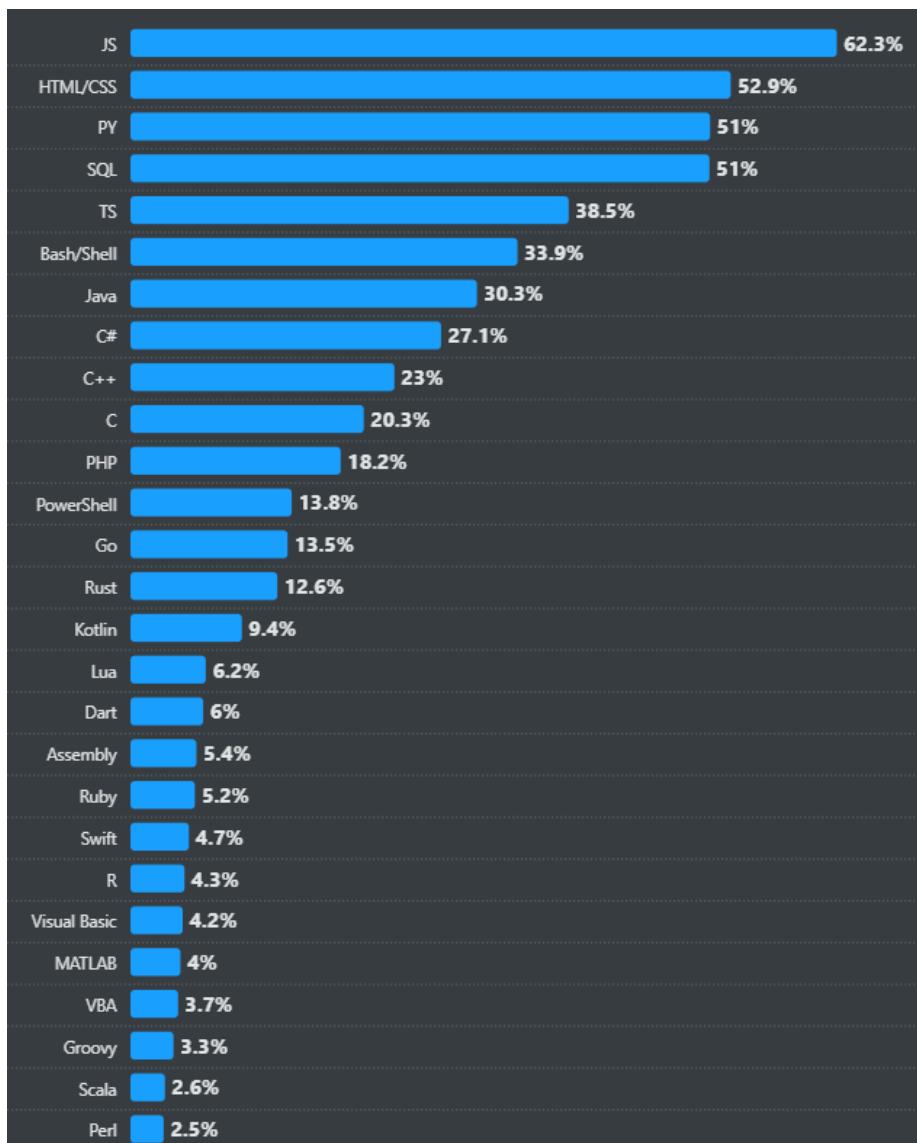
No obstant això, Python s'ha convertit en un dels llenguatges de programació més utilitzats i demandats avui en dia. I és, sens dubte, el principal llenguatge dels àmbits de la intel·ligència artificial i la ciència de dades. Avui en dia, en la majoria de desenvolupaments, les diferències d'eficiència entre els llenguatges (que amb el maquinari actual són realment petites) no són tan importants com la productivitat a l'hora d'escriure el codi. Si el llenguatge permet escriure, llegir i mantenir el codi d'una manera més senzilla, farà que el temps de desenvolupament sigui menor, i per tant l'organització estarà fent un estalvi econòmic important. Resumint: és més important la velocitat de desenvolupament que la d'execució. I és aquí on Python guanya a altres llenguatges com C, C++ o Java.





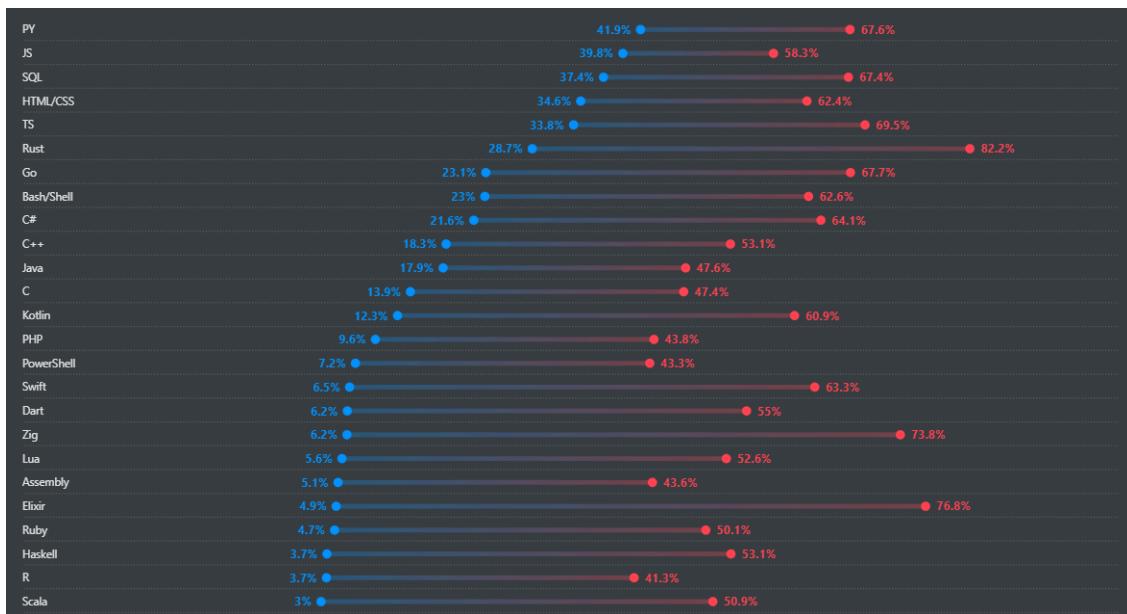
Imatge: Caricatura sobre la facilitat de llegir codi en Python i altres llenguatges. Font: somethingofthatilk.com

Per tenir una idea de la popularitat de Python, val la pena analitzar el darrer [Developer Survey de StackOverflow \(2024\)](#). Python ("PY" en el gràfic) és el tercer llenguatge més popular (després de JavaScript i HTML/CSS), bastant per davant d'altres llenguatges de propòsit general com Java, C, C++, C# o Ruby.



Imatge: Llenguatges de programació més populars. Font: [Stack Overflow 2024 Developer Survey](https://stack Overflow 2024 Developer Survey)

Pel que fa als llenguatges més desitjats pels desenvolupadors, aquells que voldrien introduir en la seva feina, Python apareix en el primer lloc, desbancant JavaScript que era el primer l'any passat. També està entre els primers més admirats, aquells que estan emprant i voldrien seguir emprant-lo l'any següent. Rust és amb diferència el llenguatge més admirat de tots.



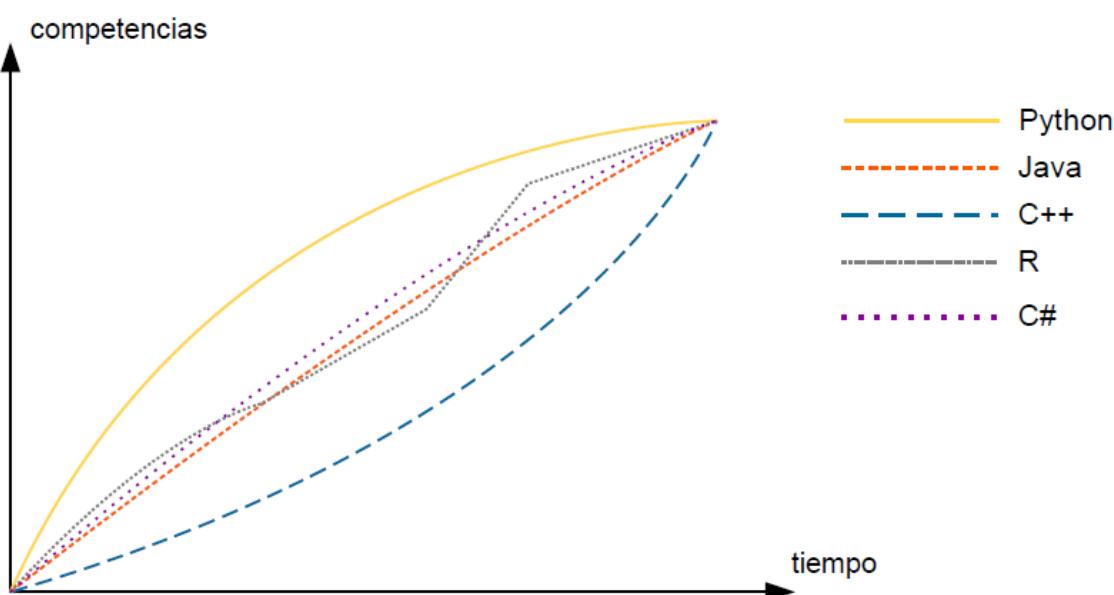
Imatge: Llenguatges de programació més desitjats i admirats. Font: [StackOverflow 2024 Developer Survey](#).

A continuació mencionarem algunes de les principals raons de la popularitat de Python (basant-nos en la llista de [pulumi.com](#)).

1. Python és fàcil d'aprendre

Més que gairebé qualsevol altre llenguatge de programació, Python llegeix i escriu de forma molt similar a l'anglès estàndard. Utilitza una sintaxi simplificada amb un èmfasi en el llenguatge natural, amb una corba d'aprenentatge molt més fàcil per als principiants. I, com que Python és d'utilització lliure i està suportat per un ecosistema extremadament gran de biblioteques i paquets, sovint és el llenguatge de primera elecció per a nous desenvolupadors.

La següent gràfica mostra la corba d'aprenentatge d'alguns dels llenguatges més populars, on podem veure que destaca Python:



Imatge: Corba d'aprenentatge de diversos llenguatges de programació. Font: Universidad de Castilla-La Mancha

2. Python té una comunitat activa i solidària

Cap programador és una illa; depenen de la documentació i el suport essencials perquè quan es troben amb problemes inesperats o problemes nous, puguin anar a cercar respostes. Python ha existit durant més de tres dècades, temps més que suficient perquè una comunitat d'usuaris creixi al seu voltant. La comunitat de Python inclou desenvolupadors de tots els nivells d'habilitat i proporciona un accés fàcil a la documentació, guies d'aprenentatge i més. Al mateix temps, la comunitat de Python és extremadament activa.

3. Python és flexible

Python es descriu sovint com un llenguatge de programació de propòsit general. Això significa que, a diferència dels llenguatges específics de domini que estan dissenyats només per a certs tipus d'aplicació, Python es pot utilitzar per desenvolupar gairebé qualsevol tipus d'aplicació en qualsevol indústria o camp.

Per a què s'utilitza Python? Python s'ha utilitzat molt en el desenvolupament web, l'anàlisi de dades, l'aprenentatge automàtic, la ciència de dades, així com en l'aprenentatge automàtic i la intel·ligència artificial. Moltes de les principals empreses i companyies de programari depenen de Python, incloent Facebook, Google, Netflix, Instagram, i altres. Amb el suport d'una gamma de frameworks i biblioteques, essencialment no hi ha cap tasca de codificació que Python no pugui gestionar.

4. Python ofereix solucions versàtils de desenvolupament web

Encara que Python és una opció efectiva per a molts tipus de projectes de desenvolupament, la seva utilitat en el desenvolupament web mereix un reconeixement específic. Utilitzant les biblioteques de codi obert disponibles, els desenvolupadors de Python poden posar en marxa les seves aplicacions web ràpidament i fàcilment.

5. Python és molt adequat per a la ciència de dades

Molts dels factors que fan que Python sigui una opció atractiva per a principiants també la distingeixen com una opció fiable per a la ciència i l'anàlisi de dades. La facilitat d'ús, suport i flexibilitat de Python l'han convertit en una eina essencial per a aquells que treballen amb l'aprenentatge automàtic, la informàtica en el nigul i les dades massives.

Python és particularment eficaç per analitzar i organitzar conjunts de dades. Les capacitats d'anàlisi de dades que incorpora Python, combinades amb el seu creixent ecosistema de frameworks centrats en dades, ajuden a assegurar que Python continua sent una solució popular de programació de ciència de dades.

6. Python s'utilitza àmpliament amb la tecnologia IoT

A mesura que l'accés sense fil es torna cada vegada més ubiqua, l'Internet de les coses (IoT) continua creixent. Aquests petits dispositius connectats a Internet sovint permeten als usuaris fer petits ajustos al seu codi, personalitzant el seu rendiment per adaptar-se a necessitats específiques. Molts d'aquests dispositius suporten Python o Micropython (una versió reduïda del llenguatge de programació dissenyada per a dispositius més simples).

7. Python s'utilitza per a l'automatització personalitzada

Gràcies a la seva àmplia biblioteca de plugins, Python s'ha convertit en un estàndard d'automatització per a la indústria. De fet, fins i tot treballant amb altres llenguatges de programació, els desenvolupadors sovint escriuen els seus scripts d'automatització utilitzant Python. Per exemple, LibreOffice permet als usuaris escriure macros en diversos llenguatges interpretats, un dels quals és Python. PyUNO és el component que dona accés a l'API de LibreOffice amb Python.

8. Python és el llenguatge acadèmic

Gràcies a la seva creixent dependència en les àrees de la ciència de dades, Python s'ha convertit en el llenguatge de computació en escoles, universitats i altres llocs d'aprenentatge. Així, ensenyant a la pròxima generació de programadors i desenvolupadors com treure el màxim partit de Python, les escoles estan assegurant que Python continuï sent una opció viable i popular durant els pròxims anys.

3. Entorn de desenvolupament

Moltes distribucions de UNIX i Linux ja incorporen un entorn de Python recent. Fins i tot alguns ordinadors Windows (especialment els d'HP) també venen amb una versió de Python instal·lat. En tot cas, si no el tens ja, instal·lar-lo és molt senzill. Pots baixar el codi font i els binaris per a diverses plataformes a python.org. En el cas de Linux, la manera més senzilla és fent servir el teu gestor de paquets com YUM o APT. Actualment (setembre 2024), la versió estable més recent és la 3.11.10.

Si estàs en Windows i vols comprovar si tens Python instal·lat, obre una consola i escriu l'ordre **py** (o **python**). Si Python està instal·lat s'executarà el *shell* de Python.

```
C:\toni>py
Python 3.10.5 (tags/v3.10.5:f377153, Jun  6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

imatge: Executant Python en Windows

En el cas de Linux, per exemple una distribució Ubuntu, pots comprovar la versió instal·lada amb **python3 -V**. Si no el tens, ho pots instal·lar amb **sudo apt install python3** (o **upgrade** per a actualitzar-lo). Una vegada instal·lat, per executar el shell ho has de fer amb l'ordre **python3**.

```
toni@olap1:~$ python3 -V
Python 3.10.6
toni@olap1:~$ python3
Python 3.10.6 (main, Aug 10 2022, 11:40:04) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

imatge: Executant Python en Ubuntu



En Linux, l'ordre **python3** es fa servir per a versions 3.0 i superiors, mentre que **python** s'ha emprat per a les versions anteriors.

Si només tenim una versió instal·lada i és 3.0 o superior, hi ha un link simbòlic de **python** a **python3**, així que podem emprar les dues indistintament.

IMPORTANT

Amb això el que tindrem és l'intèrpret de Python i les biblioteques estàndard.

Una de les eines més importants relacionades amb Python és **pip**, una eina de gestió de paquets que permet estendre Python amb altres biblioteques. Pots veure tots els paquets disponibles a pypi.org. En el cas de Windows, pip ja està inclòs amb la instal·lació de Python. En el cas de Linux, és probable que l'hagis d'instal·lar manualment. Primer, comprova si la tens instal·lada amb l'ordre:

```
pip --version
```

```
toni@olap1:~$ pip --version
pip 22.0.2 from /usr/lib/python3/dist-packages/pip (python 3.10)
```

imatge: Comprovant la versió de pip en Ubuntu

En cas de necessitar instal·lar-lo, ho hauries de fer amb el teu gestor de paquets. Per exemple, en Ubuntu:

```
sudo apt install python3-pip
```

Una vegada tens pip instal·lat, si per exemple necessites instal·lar *pandas*, una biblioteca per a la manipulació i ànalisi de dades que veurem en el segon lliurament, simplement has d'escriure l'ordre:

```
pip install pandas
```



En Linux, igual que hem vist abans amb python i python3, tenim les ordres **pip** (per a versions anteriors a la 3.0) i **pip3** (per a 3.0 o posterior).

Per exemple:

```
pip3 install pandas
```

IMPORTANT

Si només tenim instal·lada una versió 3.0 o superior, hi ha un link simbòlic i podem emprar-les indistintament.

A partir d'aquí existeixen nombroses eines de desenvolupament. En aquest curs, nosaltres treballarem amb **IPython**, **Jupyter** i **Colab**.

3.1. IPython

IPython (abreviatura d'*Interactive Python*) és un projecte que es va iniciar en 2001 per Fernando Pérez com a un intèrpret de Python millorat. Si Python és el motor de la nostra tasca d'intel·ligència artificial o de ciència de dades, podem pensar en IPython com a un panell de control interactiu.

A més de ser una interfície interactiva útil per a Python, IPython també proporciona una sèrie d'afegits sintàctics útils per al llenguatge. A més, molt important, IPython està estretament lligat amb el projecte **Jupyter**, el qual proporciona una interfície web, anomenada quadern (*notebook*), que és molt útil per a escriure i compartir el nostre codi amb Python.

Per instal·lar IPython, podem fer-ho utilitzant pip:

```
pip install ipython
```

Hi ha dues maneres principals d'utilitzar IPython: el shell d'IPython i un quadern de Jupyter (*Jupyter notebook*), que veurem en l'apartat següent. Per executar el shell, ho fem amb l'ordre:

```
python -m IPython
```

```
toni@olap1:~$ python3 -m IPython
Python 3.10.6 (main, Nov  2 2022, 18:53:38) [GCC 11.3.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.6.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

imatge: Shell d'IPython (en Ubuntu)

Una vegada obert el shell podem escriure les nostres sentències de Python i executar-les. Per exemple, anam a fer que escrigui la típica frase "Hello, World":

```
print("Hello, World!")
```

```
In [1]: print("Hello, World!")
Hello, World!
```

imatge: Execució d'una sentència en el shell d'IPython (en Ubuntu)

Podem anar escrivint i executant interactivament totes les sentències que vulguem. Per acabar, hem d'escriure l'ordre `exit` per sortir del shell.

3.2. Jupyter Notebook

Un quadern Jupyter (Jupyter notebook) és una interfície gràfica web per al shell d'IPython. A més d'executar sentències Python/IPython, un quadern permet a l'usuari incloure text amb format, així com visualitzacions estàtiques i dinàmiques, equacions matemàtiques, widgets de JavaScript, entre d'altres.

A més, aquests documents es poden guardar d'una manera que permet a altres persones obrir-los i executar el codi en els seus propis sistemes.

Per instal·lar Jupyter Notebook, ho feim mitjançant pip:

```
pip install notebook
```

Una vegada que l'hem instal·lat, podem executar el servidor de Jupyter Notebook mitjançant l'ordre següent, des del directori on volem tenir els nostres fitxers de codi:

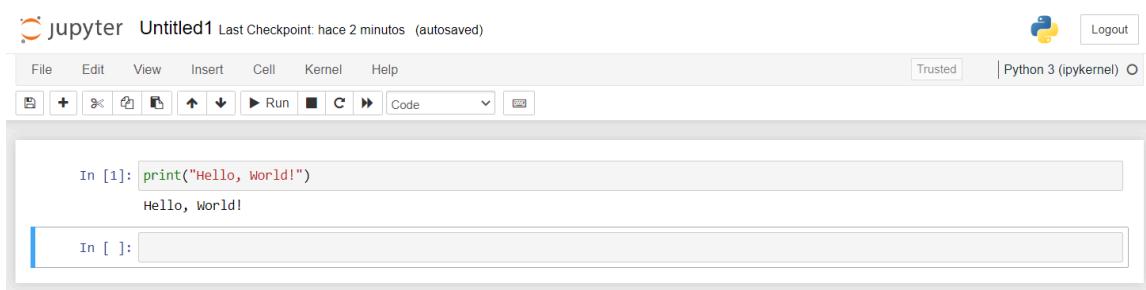
```
jupyter notebook
```

També ho podeu fer amb l'ordre:

```
python -m notebook
```

Amb qualsevol de les dues, a més de posar en marxa el servidor en el port 8888, s'obrirà una finestra del navegador que carregarà <http://localhost:8888/> (si no ho fa, obre tu mateix la finestra i carrega la pàgina).

A continuació podem crear un nou fitxer de tipus Notebook (Python 3 ipykernel). Podem veure que hi ha una cel·la on podem escriure una sentència Python (o un bloc amb varíes sentències). Per exemple, podem escriure-hi `print("Hello, World!")`. Per executar la sentència feim clic a la icona de Run o pitjam les tecles Mayus+Enter. A continuació veurem el resultat de l'execució de la sentència i ens apareixerà una altra cel·la per escriure una altra sentència o bloc:



Imatge: Quadern de Jupyter

A més de codi, també podem escriure text i donar-li format. Aquesta possibilitat és molt útil a l'hora de compartir i publicar els nostres quaderns. Per fer-ho, hem de marcar la cel·la com a Markdown. I escriure el text amb la sintaxi de Markdown, un llenguatge de marcatge lleuger. Per exemple, per escriure en cursiva una paraula, ho hem de fer posant un asterisc davant i un darrera (*cursiva*) i perquè sigui negreta, dos asteriscs (**negreta**). Podem definir encapçalaments de nivell 1 començant la línia amb el caràcter #, de nivell 2 amb ## i així successivament. Podeu trobar una referència al format Markdown a <https://commonmark.org/help/>

Si després d'escriure el text, l'executam (botó Run o Mayus+Enter), veurem el text amb el seu format:

Això és un títol
Això és text en *cursiva* i això **negreta**

Això és un títol

Això és text en cursiva i això negreta

Figura: Text amb format Markdown (abans i després d'executar)

Podem veure que tenim tota una interfície gràfica, amb diversos menús, per fer feina amb els nostres quaderns. Per guardar el nostre quadern ho farem amb l'opció Save As del menú File. Això generarà un fitxer **.ipynb** en un path relatiu al directori des d'on hem iniciat el servidor (amb l'ordre *jupyter notebook*)

3.3. JupyterLab

JupyterLab és una nova interfície gràfica web per als quaderns Jupyter. Té una estructura modular i permet obrir diversos quaderns com a pestanyes de la mateixa finestra. S'assembla més a les IDE habituals com Eclipse o Netbeans.

Per instal·lar JupyterLab amb pip:

```
pip install jupyterlab
```

Per executar el servidor:

```
jupyter-lab
```

O també amb l'ordre:

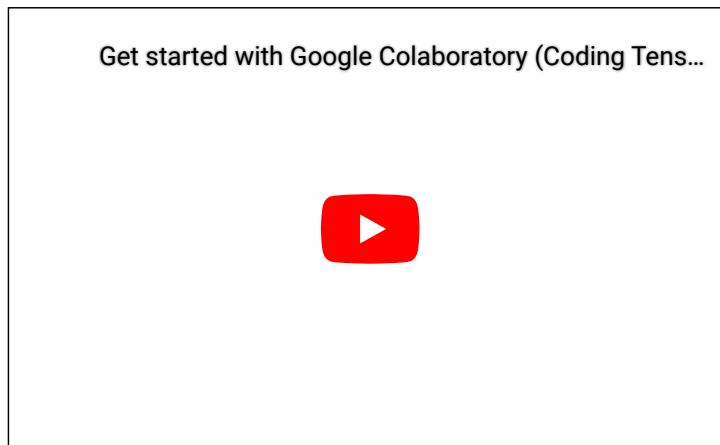
```
python -m jupyterlab
```

No entram en més detalls perquè l'edició dels quaderns és exactament igual que la que hem vist amb Jupyter Notebook.

3.4. Google Colab

Colab, també conegut com "Colaboratory" és una eina de Google Research que permet guardar i executar els quaderns Jupyter en servidors de Google. És a dir, un quadern Colab és en realitat un quadern Jupyter i la manera d'escriure'l és la mateixa que hem vist amb Jupyter Notebook. La diferència és que amb Colab no necessitam configurar el nostre servidor: n'emprarem els de Google.

Aquest vídeo elaborat per Google ens fa una breu introducció:



Vídeo: Introducció a Google Colab

En la següent imatge podem veure el quadern que hem creat anteriorment amb Jupyter Notebook i que hem pujat a Colab.

```
print("Hello, World!")
Hello, World!
```

Això és un títol

imatge: Quadern Colab

Gràcies a que no és necessari fer cap instal·lació i a la facilitat d'edició i compartició dels quaderns, Colab s'ha convertit en una eina molt popular per a les comunitats d'intel·ligència artificial i de ciència de dades. Al llarg del curs, farem servir Colab molt sovint, no només en aquest mòdul.

4. Fonaments del llenguatge Python

El llenguatge Python va ser desenvolupat per Guido van Rossum a finals dels anys 80, quan feia feina al Centrum Wiskunde & Informatica (CWI), als Països Baixos. La primera versió que es va fer pública va ser l'any 1991. L'any 2001 es va fundar la Python Software Foundation, una organització sense ànim de lucre, prenent com a model l'Apache Software Foundation. A partir d'aquell moment (versió 2.1) Python es distribueix com a codi lliure, amb una llicència Python Software Foundation License, compatible amb la llicència GNU GPL.

Existeixen diversos intèrprets de Python, tot i que CPython n'és la implementació de referència. CPython, escrit en el llenguatge C, és programari lliure i de codi obert, i està desenvolupat per la Python Software Foundation, recolzant-se en una àmplia comunitat.

El disseny del llenguatge Python es caracteritza pel seu èmfasi en la **llegibilitat, simplicitat i explicitació**. Algunes persones arriben fins al punt de dir que Python és una mena de pseudocodi executable.

En aquest capítol, donarem una visió general de les principals particularitats del llenguatge Python i que el poden diferenciar d'altres llenguatges imperatius com Java, JavaScript, C o C++.

Serà després en els capítols següents, quan entrarem a estudiar en detall els tipus de dades, el control de flux i la definició de funcions i classes en Python.

4.1. Estructura del codi: sagnia, no claus

Molts de llenguatges de programació, com R, C++, Java, Perl o Javascript, utilitzen les claus {} per estructurar el codi.

Python, en canvi, fa servir els espais (tabulacions o espais en blanc) per organitzar els programes.

Per exemple, vegem com quedaría un bucle simple que escrigui de 0 a 9.

```
for x in range(10):
    print(x)
```

Els dos punts indiquen l'inici d'un bloc de codi amb sagnia. A partir d'aquest punt, totes les línies que formin part del mateix bloc de codi han de tenir la mateixa sagnia. Si heu programat en altres llenguatges, aquesta característica de la sagnia pot ser sorprendent a l'inici, però aporta llegibilitat al codi. Es pot fer servir tabulació, dos espais, quatre espais. Alguns autors recomanen fer servir sistemàticament quatre espais.

Les sentències Python no han d'acabar necessàriament amb punt i coma. Això no obstant, els punts i coma es poden fer servir per separar múltiples sentències en una sola línia. En l'exemple següent, construïm tres variables i els assignam un valor inicial.

```
a=5; b=6; c=7
```

Aquesta pràctica de posar moltes sentències en una sola línia està desaconsellada, ja que fa el codi més poc llegible.

4.2. Tot són objectes

Una característica important del llenguatge Python és la consistència del seu **model d'objectes**. Qualsevol nombre, cadena de caràcters (*string*), estructura de dades, o fins i tot mòdul o funció és un **objecte** en Python. Cada objecte té un **tipus** associat: podria ser, entre d'altres, un nombre enter (**int**), un nombre decimal (**float**), una cadena de caràcters (**string**) o una funció (**function**). És a dir, l'objecte pertany a una **classe**. Una classe és la definició d'un model dels seus objectes, on es defineix quines dades (**propietats** o **atributs**) conté i quines operacions (**mètodes**) es poden fer amb aquestes dades.

A la pràctica, el que tot en Python siguin objectes fa el llenguatge molt flexible, ja que fins i tot les funcions es poden tractar com qualsevol altre objecte.

Si ho comparem amb Java, tot i que també és un llenguatge orientat a objectes, Java incorpora els tipus primitius (*int*, *float*, *double*, *char*, *boolean*, etc.) que no són objectes. En Python no hi ha aquesta excepció i tots els elements del llenguatge s'ajusten al paradigma d'orientació a objectes.

Per exemple, el codi següent compta quantes vegades apareix la lletra **a** dins la paraula **casa**, utilitzant el mètode **count()** de la classe **string**. També mostrarà de quina classe és la variable **mot**.

```
mot = "casa"  
print(type(mot))  
x=mot.count("a")  
print(x)
```

El resultat d'executar aquest codi és el següent, on podem veure que **mot** és un objecte de la classe **str** (*string* o cadena de caràcters) i que el resultat de cridar al seu mètode **count** és 2:

```
<class 'str'>  
2
```

Com veurem més endavant, amb la paraula **class** podem definir les nostres pròpies classes, amb les seves propietats i mètodes.

4.3. Entrada i sortida bàsiques

Ja hem vist en l'apartat anterior que Python té una funció **print** que ens permet escriure per pantalla. Vegem alguns exemples:

```
print("Hola")
```

Això escriu la cadena de caràcters "Hola".

```
x = 3
print(x)
```

Això escriu el valor de la variable x, que en aquest cas és 3.

Podem escriure diverses coses a la vegada separant-les per comes:

```
x = 3
y = 5
print("Valors", x, y)
```

Això escriu "Valors 3 5".

Podem donar un format a la nostra sortida. Definirem els espais que volem emplenar amb valors de variables mitjançant claus {}. I emplenarem aquests espais mitjançant el mètode **format**. Per exemple:

```
x = 3
y = 5
print('El valor de x és {} i el de y és {}'.format(x,y))
```

Això escriu "El valor de x és 3 i el de y és 5".

També podem demanar a l'usuari que introdueixi un text mitjançant la funció **input**. Per exemple:

```
s = input("Introdueix una paraula: ")
print("Has introduït:", s)
```

Això ens obre un camp de text on introduir una cadena de caràcters i quan pitjam <Enter> la guarda a la variable s, que després imprimim amb print.

Si en lloc d'una cadena de caràcters volem un número, hem de fer una conversió de tipus. Tornarem a parlar sobre les conversions de tipus en l'apartat 5.4. Simplement hem de posar el tipus que volem, **int** per a nombres enters o **float** per a nombres reals (amb decimals), seguit de l'expressió que volem convertir entre parèntesis:

```
x = int(input("Introdueix un número: "))
print(x)
```

El valor que introduceix es convertirà a un nombre enter (int). Si el valor no és un nombre enter, donarà un error.

Per als nombres reals (amb decimals), teniu present que es fa servir la notació anglosaxona: els decimals van darrera d'un punt (no d'una coma).

4.4. Crida a funcions i mètodes

Les funcions són el mètode d'organització i reutilització de codi més important en Python. Com a regla general, si preveiem que haurem d'usar el mateix codi, o un de molt semblant, més d'una vegada, pot valer la pena escriure una funció reutilitzable. Les funcions també ajuden a fer el codi més llegible, perquè es dona un nom a un grup de sentències Python.

Les funcions es declaren amb la paraula reservada **def** i el valor que retornen s'especifica amb **return**. Una funció pot tenir diversos arguments o paràmetres, que es posen entre parèntesis després del nom del mòdul i separats per comes. Per exemple, la següent funció *major* retorna el més gran de dos números *a* i *b*:

```
def major(a, b):
    if (a>=b):
        return a
    else:
        return b
```

Invocam les funcions emprant el seu nom i passant entre parèntesis i separats per comes els seus arguments, que poden ser zero o més. Si la funció retorna algun valor, el podem assignar a una variable. Per exemple, per saber el més gran entre 3 i 5 fent servir la nostra funció *major*:

```
x = major(3,5)
print(x)
```

Gairebé tots els objectes de Python tenen funcions associades, conegudes com a **mètodes**, que tenen accés al contingut intern de l'objecte.

Els mètodes es poden invocar amb la sintaxi següent:

```
objecte.metode(x, y, z)
```

Les funcions poden rebre arguments posicionals (com en l'exemple de la funció *major* o els 3 primers de la següent funció *f*) o per paraula clau (els 2 darrers arguments de *f*):

```
resultat = f(a, b, c, d=5, e='abc')
```

En l'apartat 8 aprofundirem en la definició i invocació de funcions i mètodes. Però ens ha de quedar clar que un mètode està associat a un objecte, mentre que una funció no (tot i que en sí mateixa també és un objecte). Les funcions s'escriuen per fomentar la reusabilitat del codi, mentre que els mètodes ofereixen un comportament específic d'un objecte.

4.5. Variables i pas d'arguments

Quan assignam una variable o nom en Python, estam creant una referència a l'objecte del costat dret de la igualtat.

Considerem la següent llista d'enters, i assignem-la a una nova variable *b*.

```
a = [1, 2, 3]
b = a
```

En molts de llenguatges, això provocaria que es copiassin les dades [1,2,3]. En Python, en canvi, les variables *a* i *b* es refereixen al mateix objecte, la llista original [1,2,3].

Podem comprovar-ho afegint un element a la llista *b* i examinant *a*. Podem comprovar que tot i que no hem modificat la variable *a*, el seu valor ha canviat:

```
b.append(4)
a
```

Obtenim aquest resultat:

```
[1, 2, 3, 4]
```

Hem d'entendre bé el significat de les referències quan treballam en Python. Saber quan, com i per què es copien les dades és especialment crític quan feim servir grans conjunts de dades.

L'assignació també rep el nom de **vinculació** (en anglès **binding**), com quan vinculam un nom a un objecte. Els noms de variables que s'han vinculat poden rebre el nom de variables vinculades o lligades.

Aquest mateix mecanisme de vinculació és el que es fa servir quan es crida una funció. Quan es passen objectes a una funció com a arguments, es creen noves variables locals que es vinculen als objectes originals, sense cap còpia. És el que anomenam pas de paràmetres per referència, en contraposició al pas de paràmetres per valor, on es faria una còpia del valor. Si llavors modificam les dades del nou objecte dins de la funció, aquest canvi es reflectirà també a l'àmbit superior. D'aquesta manera es pot modificar el valor d'un argument dins una funció.

El codi següent ho il·lustra:

```
def afegir_element(llista, element):
    llista.append(element)
dades = [1, 2, 3]
afegir_element(dades, 4)
dades
```

Obtenim el resultat següent:

```
[1, 2, 3, 4]
```

4.6. Referències dinàmiques

A diferència de molts de llenguatges compilats, com per exemple Java o C++, les referències a objectes no tenen cap tipus associat. És a dir, una variable pot començar essent d'un tipus i després passar a ser-ne d'un altre. No hi ha cap problema amb el següent codi.

```
a = 5
type(a)
```

```
<class 'int'>
```

I, tot seguit,

```
a = 'foo'
type(a)
```

```
<class 'str'>
```

És per això que, a diferència d'altres llenguatges, en Python no declaran les variables (no diem quin és el seu tipus abans de poder utilitzar-la). Les variables són noms d'objectes dins un espai de noms (*namespace*) particular; la informació del tipus s'emmagatzema dins l'objecte mateix i depèn del valor que se li assigni.

Això no vol dir que Python no sigui un llenguatge tipat. Per això, el codi següent donarà error, perquè no es pot sumar directament un enter a una cadena de caràcters.

```
'5' + 5
```

```
TypeError: must be str, not int
```

En alguns llenguatges, la cadena "5" es pot convertir implícitament, a través d'un cast a un enter, donant com a resultat 10. En uns altres, com per exemple JavaScript, l'enter 5 es converteix a cadena, i s'obté com a resultat 55. En aquest sentit, Python es considera un **llenguatge fortemet tipat**. Això significa que cada objecte és del seu tipus, i les conversions implícites només es duran a terme en casos molt obvis, com el que mostrem a continuació:

```
a = 4.5
b = 2
print('a és ', type(a), ' i b és ', type(b))
```

Podem veure que *a* és un objecte de tipus *float* (número real, amb decimals) i *b* de tipus *int* (número enter). Si ara dividim *a* entre *b*, es fa una conversió automàtica de l'enter 2 al real 2.0:

```
a / b
```

Dona com a resultat 2.25:

És important saber el tipus d'un objecte, i és útil poder escriure funcions capaces de manejar molts de tipus diferents d'entrada. Es pot comprovar que un objecte és una instància d'un tipus particular fent servir la funció *isinstance*.

```
a = 5
isinstance(a, int)
```

```
True
```

La funció `isinstance` pot acceptar una tupla de tipus. Això ens permetrà saber si una variable és d'un d'entre diversos tipus possibles, com en el següent codi que comprova si `a` i `b` són de tipus `int` o `float`:

```
a = 5; b = 4.5  
  
isinstance(a, (int, float))
```

```
True
```

```
isinstance(b, (int, float))
```

```
True
```

4.7. Objectes mudables i immutables

Abans hem dit que per exemple un *string* o cadena de text (*str*) o un número enter (*int*) són objectes. Però és important aclarir que en Python existeixen dos tipus d'objectes: **mudables** i **immutables**.

Qualsevol operació que fem sobre un *string* està dissenyada per produir un nou *string* com a resultat, però no modifica el *string* original. És per això que diem que els strings són immutables en Python, perquè no es poden modificar una vegada que s'han creat. Per exemple, no podem modificar el valor d'una posició d'un *string*, n'hem de crear un de nou amb la nova cadena que volem. Vegem-ho amb un exemple:

```
s = 'abcde'
s[0]
```

Dona com a resultat:

```
'a'
```

Però quan intentam modificar la primera posició del *string*, es llança un error:

```
s[0] = 'A'
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-22-cc37915722b8> in <cell line: 1>()
----> 1 s[0] = 'A'

TypeError: 'str' object does not support item assignment
```

També són immutables els tipus més simples (*int*, *float*, etc.) i les tuples (en xerrarem més endavant). Per exemple:

```
tupla = (3,5,(4,5))
tupla[1] = 4
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-23-ced3792ed467> in <cell line: 2>()
      1 tupla = (3,5,(4,5))
----> 2 tupla[1] = 4

TypeError: 'tuple' object does not support item assignment
```

En canvi, les llistes (*list*), com les que hem vist a algun exemple anterior, diccionaris (*dict*) i conjunts (*set*), els arrays de la biblioteca NumPy, així com la majoria de tipus definits per l'usuari (classes), són **mudables**: es poden modificar lliurament. Sobre llistes, diccionaris i conjunts n'aprofundirem més endavant en aquest lliurament, mentre que la biblioteca NumPy la veurem en el següent. Per exemple, mitjançant el mètode *append* hem vist com afegim un element nou a una llista.

```
a = [1, 2, 3]
b.append(4)
a
```

```
[1, 2, 3, 4]
```

També podem modificar una posició concreta:

```
a[0] = 5  
a
```

```
[5, 2, 3, 4]
```

Ara bé, el fet que puguem canviar un objecte no significa que sempre ho hagim de fer. Hem d'anar alerta quan modificam un objecte que es passa com a argument a una funció o mètode. Això pot provocar que el programador que faci servir aquesta funció o mètode es trobi amb resultats inesperats. És el que es coneix com **efecte lateral**. Per tant, es molt desaconsellable modificar objectes mudables que es passen com a argument; i si fos imprescindible, s'ha de comunicar expressament en la documentació de la funció o mètode.

4.8. Atributs i mètodes

Com ja hem dit anteriorment, els objectes en Python poden tenir atributs o propietats (variables) i mètodes. Els atributs són altres objectes Python inclosos dins l'objecte, i els mètodes són funcions associades amb l'objecte que tenen accés a les dades internes de l'objecte.

Podem accedir als atributs i als mètodes mitjançant la sintaxi **objecte.atribut** i **objecte.mètode**.

La funció **dir** permet consultar la llista d'atributs i mètodes d'un objecte, en aquest cas de tipus *string*:

```
a="abc"  
dir(a)
```

El resultat és:

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'removeprefix',
```

```
'removeprefix',
'replace',
'rfind',
'rindex',
'rjust',
'rpartition',
'rsplit',
'rstrip',
'split',
'splitlines',
'startswith',
'strip',
'swapcase',
'title',
'translate',
'upper',
'zfill']
```

També es pot accedir a atributs i mètodes amb la funció **getattr**.

```
getattr(a, 'split')
```

```
<function str.split>
```

4.9. Convencions en el codi

Hi ha una sèrie de convencions que els programadors segueixen per a escriure codi més fàcil de llegir. Tot i que hi ha similituds, també hi ha diferències considerables depenen del llenguatge de programació. En el cas de Python, s'han elaborat diverses guies d'estil, essent la més reconeguda la de la comunitat de python.org: [Style Guide for Python Code](#), elaborada per Guido van Rossum (el creador del llenguatge Python), Barry Warsaw i Nick Coghlan.

Una de les convencions més importants és la del nom de les variables i mètodes. En Python, a diferència de Java o C, es fa servir l'estil minúscules_amb_guions_baixos (*lower_case_with_underscores*). És a dir, no feim servir majúscules en els noms de variables i mètodes, i si un nom ha de contenir més d'una paraula, empram el guió baix per unir-les. Per exemple, si volem una variable per al codi d'un alumne serà *codi_alumne* (mentre que en Java o C es recomana *codiAlumne*).

En canvi, els noms de les classes sempre comencen per una lletra majúscula i segueixen l'estil *CapitalizedWords*, on ajuntam les paraules emprant una majúscula al principi de cada paraula. Per exemple, si volem definir una classe per als alumnes del curs de programació, podríem emprar el nom *AlumneProgramacio* (aquí sí que és igual que en Java o C).



Les guies d'estil només defineixen recomanacions. L'intèrpret de Python no es "queixarà" si no seguim la seva guia d'estil i, per exemple, definim una variable *codiAlumne*.

Però és útil acostumar-se a fer-les servir perquè és la manera en què la comunitat de Python treballa.

IMPORTANT

4.10. Importacions de mòduls

En Python un mòdul és simplement un fitxer amb l'extensió **.py** que conté codi Python.

Suposem que tenim el mòdul següent en el fitxer *modul_nostre.py*:

```
# modul_nostre.py
PI = 3.14159

def incrementa(x):
    return x + 1

def suma(x, y):
    return x + y
```

Si volem accedir a les variables i funcions que hi ha definides en **modul_nostre.py**, des d'un altre fitxer en el mateix directori, podem importar-lo (amb **import**) i després utilitzar les seves variables i funcions, emprant el nom del mòdul com a prefix:

```
import modul_nostre
a = modul_nostre.incrementa(5)
b = modul_nostre.suma(3,4)
pi = modul_nostre.PI
```

O de manera equivalent:

```
from modul_nostre import incrementa, suma, PI
a = incrementa(5)
b = suma(3,4)
pi = PI
```

És molt habitual assignar un alias a un mòdul, mitjançant la paraula clau **as**, i utilitzar aquest alias com a prefix. Així podem saber en tot moment en quin mòdul està definit una variable o funció.

```
import modul_nostre as mn
a = mn.incrementa(5)
b = mn.suma(3,4)
pi = mn.PI
```

Recordau que un mòdul també és un objecte Python.

4.11. Operadors binaris i comparacions

La majoria d'operacions matemàtiques binàries i comparacions tenen els resultats esperables. Per exemple:

- $5 - 7$ retorna -2
- $12 + 21.5$ retorna 33.5
- $5 \leq 2$ retorna *False*

A les taules següents presentam les operacions aritmètiques i lògiques, i les comparacions.

Operador	Significat
$a + b$	Suma d' a i b
$a - b$	Resta a menys b
$a * b$	Multiplicació d' a i b
a / b	Divisió d' a entre b
$a // b$	Divisió entera d' a entre b , arrodonida cap a baix
$a ** b$	a elevat a la potència b
$a & b$	En booleans, i lògica: <i>True</i> si a i b són <i>True</i> ; en enters, AND bit a bit
$a b$	En booleans, o lògica: <i>True</i> si a o b són <i>True</i> ; en enters, OR bit a bit
$a ^ b$	En booleans, o exclusiva lògica: <i>True</i> si a o b són <i>True</i> , però no tots dos; en enters, XOR bit a bit
$a \text{ and } b$	En booleans, i lògica: <i>True</i> si a i b són <i>True</i>
$a \text{ or } b$	En booleans, o lògica: <i>True</i> si a o b són <i>True</i>
$\text{not } a$	En booleans, negació: <i>True</i> si a és <i>False</i> i <i>False</i> si a és <i>True</i>

Taula: Operacions aritmètiques i lògiques

Comparació	Títol1
$a == b$	<i>True</i> si a és igual a b
$a != b$	<i>True</i> si a no és igual a b
$a \leq b, a < b$	<i>True</i> si a és més petit o igual a b , <i>True</i> si a és més petit que b
$a > b, a \geq b$	<i>True</i> si a és més gran que b , <i>True</i> si a és més gran o igual que b
$a \text{ is } b$	<i>True</i> si a i b es refereixen al mateix objecte Python
$a \text{ is not } b$	<i>True</i> si a i b no es refereixen al mateix objecte Python

Taula: Comparacions

Per comprovar si dues referències es refereixen al mateix objecte, usam la paraula reservada ***is***. Per comprovar que no són el mateix objecte, s'usa ***is not***.

```
a = [1, 2, 3]
b = a
a is b
```

```
True
```



És important tenir clar que la comparació amb `is` és diferent que amb l'operador `==`. Mentre que `is` comprova si les dues variables tenen la mateixa referència, `==` compara si contenen el mateix valor. Per exemple, en aquest codi podem veure que a i b contenen referències a objectes diferents i per això a `is` b retorna `False`:

```
a = [1, 2, 3]
b = [1, 2, 3]
a is b
```

```
False
```

IMPORTANT

En canvi, els valors referenciats per a i per b (tot i ser dos objectes diferents) sí que són iguals:

```
a == b
```

```
True
```

Un ús habitual dels comparadors `is` i `is not` és comprovar si una variable és `None`, ja que hi ha una única instància de `None`.

```
a = None
a is None
```

```
True
```

4.12. assert

La paraula clau **assert** comprova una condició i si aquesta és falsa, llença un error del tipus *AssertionError*. Podem escriure un missatge que doni més informació sobre l'error.

Vegem un exemple:

```
assert x>0, "La variable x hauria de tenir un valor positiu"
```

Si quan arribam aquí x és major que 0, no passarà res i l'execució continuarà amb la següent instrucció. En canvi, si la x és menor o igual, es llançarà el següent error:

```
-----
AssertionError                                     Traceback (most recent call last)
<ipython-input-4-8a190c43c45f> in <cell line: 1>()
----> 1 assert x>0, "La variable x hauria de tenir un valor positiu"

AssertionError: La variable x hauria de tenir un valor positiu
```



Les instruccions amb `assert` són útils per fer comprovacions i detectar possibles errors en la fase de debugging del codi, però s'haurien d'eliminar en un entorn en producció.

5. Tipus escalars

Python juntament amb la seva biblioteca estàndard té un petit conjunt de tipus integrats per manejar dades numèriques, cadenes de caràcters, valors booleans (*True* o *False*, cert o fals) o bé data i hora. Aquests tipus de "valor únic" de vegades s'anomenen tipus escalars, o simplement escalars. A la taula següent tenim una llista dels principals tipus escalars. La gestió de la data i l'hora no la incloem a la taula, ja que ve resolta en el mòdul *datetime* de la biblioteca estàndard.

Tipus	Significat
<i>None</i>	El valor nul de Python. És una única instància d'objecte.
<i>str</i>	Tipus cadena de caràcters; conté cadenes Unicode codificades en UTF-8.
<i>bytes</i>	Bytes ASCII directament, o bé Unicode codificat com a bytes.
<i>float</i>	Número en coma flotant de precisió doble (64 bits). No hi ha un tipus <i>double</i> separat.
<i>bool</i>	Un valor cert o fals, <i>True</i> o <i>False</i> .
<i>int</i>	Un enter amb signe de precisió arbitrària.

Taula: Tipus escalars de Python

5.1. Tipus numèrics

Els tipus principals de Python per als números són ***int*** i ***float***. Un *int* pot emmagatzemar nombres arbitràriament grans:

```
ival = 17239871  
ival ** 6
```

```
26254519291092456596965462913230729701102721
```

Els nombres de coma flotant es representen amb el tipus ***float*** de Python. A diferència d'altres llenguatges no existeix un tipus separat ***double***, ja que internament, un *float* de Python ja té doble precisió (64 bits). També es poden expressar els valors amb notació científica:

```
fval = 7.243  
fval2 = 6.78e-5
```

La divisió entera que no resulta en un nombre enter sempre donarà un nombre de coma flotant:

```
3/2
```

```
1.5
```

Per obtenir una divisió entera d'estil Java o C (que elimina la part fraccionària si el resultat no és un nombre enter), utilitzam l'operador de divisió entera **//**:

```
3 // 2
```

```
1
```

5.2. Cadenes de caràcters

Molta gent utilitza Python per les seves capacitats potents i flexibles de processament de cadenes integrades. Podem escriure literals de cadena utilitzant cometes simples ' o cometes dobles ":

```
a = 'una manera d'escriure una cadena'
b = "una altra manera"
```

Per a cadenes de diverses línies amb salts de línia, podem utilitzar cometes triples: 3 caràcters de cometa simple "" o 3 caràcters de cometa doble """.

```
c = """
Aquesta és una cadena més llarga que
abasta diverses línies
"""
```

Pot ser una sorpresa que aquesta cadena c en realitat conté quatre línies de text; els salts de línia després de les primeres triples cometes """ i després de les línies de text s'inclouen a la cadena. Podem comptar els caràcters de salt de línia ('\n') amb el mètode **str.count()**:

```
c.count('\n')
```

```
3
```

A Des de la versió 3.0, Python fa servir **Unicode** com a codificació de caràcters per defecte. D'aquesta manera podem escriure sense problemes caràcters no anglesos com les vocals amb accents o dièresis, la ç i la ñ. Per exemple:

```
s = 'àüçñ'
s
```

```
àüçñ
```

En la versió 2, podem indicar amb una u davant de la cadena de caràcters que està codificada en Unicode:

```
s = u'àüçñ'
```

També podem utilitzar la representació en Unicode d'un caràcter, emprant el caràcter especial \u (en aquest exemple 00f1 és la representació de la ñ):

```
s = "\u00f1"
s
```

```
ñ
```

Si necessitem fer feina amb alguna altra codificació de caràcters, podem emprar els mètodes *encode* i *decode* per fer la transformació corresponent. Per exemple, per codificar-la amb UTF-8:

```
s = 'àüçñ'
sutf8 = s.encode('utf-8')
sutf8
```

```
b'\xc3\xab\xc3\xbc\xc3\xad\xc3\xb1'
```

Les cadenes de Python són immutables, és a dir, no podem modificar una cadena:

```
a = 'això és una cadena'
a[10] = 'f'
```

```
TypeError Traceback
----> 1 a[10] = 'f'
TypeError: l'objecte 'str' no admet l'assignació d'elements
```

■ Mètodes de str

Una cadena és un objecte de la classe **str**. A la [documentació de Python](#) podeu trobar tots els mètodes d'aquesta classe, molt útils per treballar amb cadenes.

Uns dels mètodes més emprats són **str.upper()** i **str.lower()**, que passen una cadena a majúscules o a minúscules, respectivament.

```
s = "Hola"
s.upper()
```

```
'HOLA'
```

Amb el mètode **str.replace()** podem fer substitucions dins la cadena.

```
a = 'això és una cadena'
b = a.replace('cadena', 'cadena més llarga')
b
```

```
"això és una cadena més llarga"
```

Teniu present que després d'aquesta operació, la variable **a** no es modifica:

```
a
```

```
"això és una cadena"
```

El mètode **str.find()** ens permeten cercar una subcadena dins una cadena i ens retorna la posició de la primera ocurrència (començant per 0). Si no la troba, retorna -1. Per exemple:

```
s = "En Miquel i en Rafel"
s.find("el")
```

Retorna la posició (7) on troba la primera ocurrència de la subcadena "el".



Si només volem saber si una subcadena es troba dins una cadena, i no necessitam la seva posició, és recomanable emprar l'operador **in**:

```
s = "En Miquel i en Rafel"
"el" in s
```

True

Un altre mètode de str força utilitzat és **str.split()** que ens permet dividir una frase en una llista de paraules.

```
s = "En Miquel i en Rafel"
s.split()
```

['En', 'Miquel', 'i', 'en', 'Rafel']

Podem especificar un altre caràcter separador. Per exemple:

```
s = "taronja, pera, poma"
s.split(',')
```

['taronja', ' pera', ' poma']

Fixau-vos que pera i poma inclouen un espai en blanc al principi. Podríem llevar aquest espai amb el mètode **str.strip()**, que elimina el caràcter indicat al principi o final de la cadena, tantes vegades com hi sigui. Per exemple:

```
s = '    pera    '
s.strip(' ')
```

pera

Podem emprar **str.rstrip()** per eliminar només a la part dreta de la cadena o **str.lstrip()** per a la part esquerra.

Molt relacionat amb **str.split()** és el mètode **str.splitlines()** que es fa servir per separar un text llarg en línies, prenent els caràcters de salt de línia (normalment \r o \n).

■ Conversions de tipus

Molts d'objectes de Python es poden convertir en una cadena mitjançant la funció **str**:

```
a = 5.6
s = str(a)
print(s)
```

5.6

Les cadenes són una seqüència de caràcters Unicode i, per tant, es poden tractar com altres seqüències, com ara llistes i tuples (que explorarem amb més detall en un capítol posterior):

```
s = 'python'
list(s)
```

'p', 'y', 't', 'h', 'o', 'n'

```
s[:3]
```

```
'pyt'
```

La sintaxi `s[3:]` s'anomena ***slicing*** i s'implementa per a molts tipus de seqüències de Python. Això s'explicarà amb més detall més endavant, ja que s'utilitza àmpliament en aquests apunts.

■ Caràcter d'escapada

El caràcter de barra invertida \ és un caràcter d'escapada, que s'utilitza per especificar caràcters especials com ara la línia nova \n o els caràcters Unicode. Per escriure un literal de cadena amb barres invertides, n'hem d'escapar amb dues barres invertides \\:

```
s = '12\\34'
print(s)
```

```
12\34
```

Si tenim una cadena amb moltes barres invertides i sense caràcters especials, això pot semblar una mica molest. Afortunadament, podem introduir la cometa inicial de la cadena amb `r` (la r significa *raw*, en cru), que significa que els caràcters s'han d'interpretar com són:

```
s = r'això\no\té\caràcters\especials'
s
```

```
"això\no\té\caràcters\especials"
```

■ Concatenació

Afegir dues cadenes juntes les concatena i produeix una nova cadena:

```
1 1 a = 'aquesta és la primera meitat' b = ' i aquesta és la segona meitat'
```

```
a + b
```

```
"aquesta és la primera meitat i aquesta és la segona"
```

■ Format

La plantilla o el format de cadena és un altre tema important. El nombre de maneres de fer-ho s'ha ampliat amb l'arribada de Python 3, i aquí descriurem breument la mecànica d'una de les interfícies principals. Els objectes de cadena tenen un mètode de format que es pot utilitzar per substituir arguments amb format a la cadena, produint una nova cadena:

```
template = '{0:.2f} {1:s} valen {2:d} {3:s}'
```

En aquesta cadena,

- `{0:.2f}` significa formatar el primer argument com a nombre de coma flotant amb dos decimals.
- `{1:s}` significa formatar el segon argument com a cadena.
- `{2:d}` significa formatar el tercer argument com un nombre enter exacte.

Per substituir arguments per aquests paràmetres de format, passam una seqüència d'arguments al mètode de format:

```
template.format(10.7, 'dòlars', 10, 'euros')
```

```
'10.70 dòlars valen 10 euros'
```

El format de cadena és un tema complex; hi ha diversos mètodes i nombroses opcions i detalls disponibles per controlar com es formen els valors a la cadena resultant. Per obtenir més informació, podem consultar la documentació oficial de Python.

5.3. Booleans

Els dos valors booleans de Python s'escriuen com a **True** i **False**.

El resultat d'avaluar una condició sempre és un booleà. Podem emprar els operadors lògics **and** (i lògica), **or** (o lògica) i **not** (negació).

```
x=7  
x>1 and x<5
```

```
False
```

```
x>1 and x<5
```

```
True
```

```
not x<0
```

```
True
```

5.4. Conversió de tipus

Les paraules **str**, **bool**, **int** i **float** també són funcions que es poden utilitzar per convertir (*cast*) valors a aquests tipus:

```
s = '3.14159'  
fval = float(s)  
type(fval)
```

float

int(fval)

3

bool(fval)

True

bool(0)

False

5.5. None

None és el tipus de valor nul de Python. Si una funció no retorna explícitament un valor, implícitamente retorna **None**:

```
a = None  
a is None
```

```
True
```

```
b = 5
```

```
b is not None
```

```
True
```

None també és un valor predeterminat comú per als arguments de funció:

```
def add_and_maybe_multiply (a, b, c=None):  
    result = a + b  
    if c is not None:  
        result = result * c  
    return result
```

None no és només una paraula clau reservada, sinó també una instància única del tipus **NoneType**:

```
type(None)
```

```
NoneType
```

5.6. Dates i hores

El mòdul predefinit **`datetime`** inclou els tipus **`datetime`, `date` i `time`**.

El tipus **`datetime`** combina la informació de data i hora, **`date`** i **`time`**, i és el més usat.

```
from datetime import datetime, date, time
dt = datetime(2024, 9, 29, 20, 30, 21)
dt.day
```

29

dt.minute

30

Donada una instància **`datetime`**, se'n poden extreure els objectes **`date`** i **`time`** usant els mètodes que tenen aquest mateix nom.

dt.date()

datetime.date(2024, 9, 29)

dt.time()

datetime.time(20, 30, 21)

El mètode **`strftime`** dona format de cadena (*string*) a un **`datetime`**.

dt.strftime('%d/%m/%Y %H:%M')

'29/09/2024 20:30'

Les cadenes es poden convertir a objectes **`datetime`** amb la funció **`strptime`**.

datetime.strptime('20240929', '%Y%m%d')

datetime.datetime(2024, 9, 29, 0, 0)

La taula següent dona la llista completa d'especificacions de format.

Tipus	Descripció
%Y	Any amb quatre díigits
%y	Any amb dos díigits
%m	Mes amb dos díigits [01, 12]
%d	Dia amb dos díigits [01, 31]
%H	Hora (24-hores) [00, 23]
%I	Hora (12-hores) [01, 12]
%M	Minut amb dos díigits [00, 59]

Tipus	Descripció
%S	Segon [00, 61] (els segons 60 i 61 són segons addicionals, que s'afegeixen com a correcció.)
%W	Dia de la setmana com a enter [0 (diumenge), 6 (dissabte)]
%U	Nombre de setmana dins l'any [00, 53]; el diumenge es considera el primer dia de la setmana i els dies abans del primer diumenge són la setmana 0.
%W	Nombre de setmana dins l'any [00, 53]; el dilluns es considera el primer dia de la setmana i els dies abans del primer dilluns són la setmana 0.
%z	Desplaçament UTC de zona horària +HHMM o - HHMM
%F	Abreviatura de %Y-%m-%d (per exemple, 2012-4-18)
%D	Abreviatura de %m/%d/%y (per exemple, 04/18/12)

Taula: Especificacions de format de data i hora.

Quan agrupam dades de sèries temporals, de vegades convé substituir els camps de temps dels *datetimes*, per exemple posant-hi zeros.

```
dt.replace(minute=0, second=0)
```

```
datetime.datetime(2011, 10, 29, 20, 0)
```

Com que *datetime* és un tipus immutable, els mètodes com aquests sempre produueixen nous objectes.

Vegem com podem obtenir la diferència entre dos objectes *datetime*, que ens retorna un objecte de tipus ***datetime.timedelta***:

```
dt2 = datetime(2024, 8, 15, 22, 30)
diferencia = dt - dt2
diferencia
```

```
datetime.timedelta(days=44, seconds=79221)
```

Aquest resultat ens indica que hi ha 44 dies i 79.221 segons de diferència entre les dues dates.

També podem afegir un *timedelta* a un *datetime* per tal d'obtenir un nou *datetime*, desplaçat:

```
dt2 + diferencia
```

```
datetime.datetime(2024, 9, 29, 20, 30, 21)
```

6. Estructures de control

En Python les sentències es van executant una darrera l'altra, de forma seqüencial. El més habitual és tenir una sentència per línia i no és necessari acabar-la amb un punt i coma com en altres llenguatges. Si volem incloure més d'una sentència dins d'una única línia (una pràctica no recomanada), sí que haurem d'emprar el punt i coma per separar-les.

La sentència més habitual és la d'assignació, que utilitza l'operador `=`, on el resultat d'avaluar l'expressió de la dreta s'emmagatzema en la variable de l'esquerra. En Python tenim alguns operadors (`+=`, `-=`, `*=`, `/=`, entre d'altres) que combinen una operació aritmètica amb l'assignació. Per exemple:

```
x += 1
```

és equivalent a escriure:

```
x = x + 1
```

Comentaris en el codi

L'intèrpret de Python ignorarà qualsevol text que vagi precedit del caràcter `#` (coixinet).

Això es fa servir sovint per afegir comentaris al codi, per explicar-lo. De vegades, també és útil comentar certes línies de codi, mentre s'estan fent proves.

També es poden posar comentaris al final d'una línia de codi.

Teniu present que en Python, a diferència de Java o C, no hi ha una marca especial per definir comentaris de múltiples línies: cada una de les línies ha de començar pel coixinet.

```
# Això és un comentari  
x = 5 # això és un comentari incrustat en una línia  
  
# Si volem comentar diverses línies  
# hem d'emprar el caràcter #  
# al principi de cada una d'elles.
```

IMPORTANT

D'altra banda, les estructures de control ens permeten rompre la seqüencialitat i definir blocs de sentències que s'executaran només quan s'acompleixi una certa condició (estructures de control condicionals) o que es repetiran diverses vegades (estructures de control iteratives o bucles).

En els apartats següents veurem com definir estructures condicionals i iteratives en Python.

6.1. Estructures condicionals: if, elif, else

La instrucció **if** és un dels tipus d'instruccions de flux de control més coneguts i utilitzats. Comprova una condició i si és certa, executa un bloc de codi. Recordem que en Python no es fan servir les claus per emmarcar un bloc de codi, sinó que s'empra la sagnia.

```
if x < 0:  
    print('És negatiu')
```

Una instrucció **if** pot anar seguida opcionalment d'un o més blocs **elif** i un bloc final **else** per al cas en què totes les condicions siguin falses.

```
if x < 0:  
    print('És negatiu')  
elif x == 0:  
    print('És zero')  
elif 0 < x < 10:  
    print('Positiu però menor que 10')  
else:  
    print('Positiu i major o igual que 10')
```

Si alguna de les condicions és *True*, ja no s'avaluarà cap més bloc **elif** o **else**. En el cas d'una condició composta que usi *and* i *or*, les condicions s'avaluen d'esquerra a dreta.

```
a = 5; b = 7  
c = 8; d = 4  
if a < b or c > d:  
    print('Llestos')
```

Llestos

A l'exemple anterior, la condició $c>d$ mai no s'avalua perquè $a<b$ ja ha estat certa.

Les comparacions també es poden encadenar.

```
4 > 3 > 2 > 1
```

True

6.2. Expressions ternàries

Una expressió ternària en Python ens permet combinar un bloc *if-else* que produceix un valor en una única línia o expressió. La sintaxi corresponent a Python és la següent:

```
valor = expressió-si-vertader if condició else expressió-si-fals
```

Això és el mateix que escriure el codi següent, més llarg:

```
if condició
    valor = expressió-si-vertader
else
    valor = expressió-si-fals
```

Vegem-ho amb un exemple:

```
x = 5
'Positiu' if x > 0 else 'No positiu'
```

```
Positiu
```

6.3. Estructures iteratives: bucles while

Un bucle **while** especifica un bloc de codi que s'executarà mentre una condició sigui certa. Així doncs, el bucle acabarà quan la condició sigui falsa. També es pot acabar explícitament mitjançant la sentència **break** (tot i que fer-ho així no és una bona pràctica).

El següent exemple mostra un bucle while per sumar els números enters des d'1 fins a 10, tots dos inclosos:

```
i = 1
suma = 0
while i<=10:
    suma += i
    i += 1
suma
```

55

6.4. Estructures iteratives: bucles for

Els bucles **for** serveixen per a iterar sobre una col·lecció (com una llista o una tupla) o un iterador. La sintaxi estàndard per a un bucle **for** és la següent:

```
for valor in col·lecció:
    # fes alguna cosa amb el valor
```

Per exemple, el següent codi suma tots els valors d'una llista:

```
a = [1, 2, 3, 4, 5]
suma = 0
for x in a:
    suma += x
suma
```

15

Podem avançar un bucle **for** a la següent iteració, saltant la resta del bloc, utilitzant la paraula clau **continue**. Considerem aquest codi, que suma els nombres enters que hi ha en una llista i omet els valors *None*.

```
a = [1, 2, None, 4, None, 6]
suma = 0
for x in a:
    if x is None:
        continue
    suma += x
suma
```

13

També es pot sortir del tot d'un bucle **for** amb la paraula clau **break**. Per exemple, aquest codi suma els elements de la llista fins que troba un 5.

```
a = [1, 9, 0, 4, 6, 5, 3, 2]
suma = 0
for x in a:
    if x == 5:
        break
    suma += x
suma
```

20

Si tenim diversos bucles **for** niats (un dins d'altre), la paraula clau **break** només acaba el bucle **for** més interior; qualsevol bucle **for** exterior continuarà funcionant amb normalitat.

6.5. El tipus range

El tipus **range** representa una seqüència immutable de nombres, usada normalment per a definir bucles que s'han d'iterar un nombre determinat de vegades.

Per obtenir un objecte *range*, empram la funció **range**, que retorna una seqüència de nombres enters espaiats uniformement. L'expressió següent retorna una seqüència dels nombres enters des de 0 fins al número anterior al 10 (9): [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Fixau-vos que el 10 no hi entra.

```
range(10)
```

range es fa servir molt sovint en bucles *for*. Per exemple, si volem sumar els enters de 0 a 10, escriuríem:

```
suma = 0
for i in range(11):
    suma += i
suma
```

55

Es pot especificar un valor inicial diferent de 0. Per exemple, l'expressió següent retorna una seqüència [3, 4]:

```
range(3,5)
```

I també podem fer que en lloc d'anar d'un en un, ho faci amb altra passa. Per exemple, el següent codi suma només els valors senars entre 1 i 10 (va de 2 en 2):

```
suma = 0
for i in range(1,11,2):
    suma += i
suma
```

25

Fixau-vos que també ho haguéssim pogut escriure així, anant d'un en un i comprovant si el número és senar abans de sumar:

```
suma = 0
for i in range(11):
    if i%2 != 0:
        suma += i
suma
```

Si volem fer-ho en ordre invers, des del valor més gran fins al més petit restant cada vegada, emprarem una passa negativa. El següent exemple suma els imparells de 9 a 1 (la passa és -2):

```
suma = 0
for i in range(9,0,-2):
    suma += i
suma
```

Recordau que el valor especificat com a final del *range* no s'inclou en la seqüència. Per tant, *range(9,1,-2)* no inclouria l'1 i per això hem emprat *range(9,0,-2)*.

Podem convertir el resultat de *range* a una llista amb la funció *list*. Per exemple:

```
list(range(9,0,-2))
```

Retorna la llista [9, 7, 5, 3, 1].

Una aplicació freqüent de *range* és iterar al llarg de llistes per índex.

```
llista = [20, 57, 4, 18, 33]
for i in range(len(llista)):
    valor = llista[i]
    print(valor)
```

6.6. L'operació buida: pass

pass és la sentència *no-operació* en Python. Es pot usar en blocs on no cal executar cap acció, com un lloc reservat per a un codi que encara no s'ha implementat. Només fa falta perquè Python usa l'espai en blanc per delimitar blocs.

```
if x < 0:  
    print('negatiu')  
elif x == 0:  
    pass # pendent d'implementar  
else:  
    print('positiu')
```

7. Estructures de dades

Una vegada vists els tipus de dades escalars, aquest apartat el dedicarem a estructures de dades més complexes. En concret, ens centrarem en les seqüències, els conjunts i els diccionaris o tipus associatius. Totes aquestes estructures de dades estan definides en el propi llenguatge Python i les utilitzarem àmpliament durant tots els mòduls del curs. En el lliurament següent veurem que biblioteques complementàries com *pandas* i *NumPy* afegeixen noves estructures de dades optimitzades per treballar amb grans conjunts de dades i que farem servir especialment en els mòduls de big data.

En aquest apartat dels apunts definirem què són aquestes estructures. A més, hem preparat una sèrie de quaderns de Google Colab que inclouen molts exemples per entendre millor com funcionen. Els trobareu enllaçats des dels apunts. Podeu executar els quaderns i anar introduint tots els canvis que considereu oportuns per entendre millor el codi. Feu-ho sense por, no modificareu el quadern públic, la resta de companys no veuran aquests canvis. Si després voleu guardar els vostres canvis, heu de fer una còpia al vostre Google Drive.

7.1. Tuples, llistes, diccionaris i conjunts

■ Seqüències

En Python tenim tres tipus bàsics de seqüències: **tuple**, **list** i **range**. Ja hem vist el tipus *range* en el capítol d'estructures de control, ja que normalment es fan servir per a definir bucles *for*. Recordem que *range* és una seqüència immutable de nombres enters. A continuació veurem en detall les tuples i les llistes.

■ Tuples

Les **tuples** són objectes Python de seqüència de longitud fixa i immutables. La forma més fàcil de crear-ne una és amb una seqüència de valors separats per comes:

```
tupla = 1, 2, 'a', 13, True
tupla
```

El resultat ve tancat entre parèntesis:

```
(1, 2, 'a', 13, True)
```

Per això, també és freqüent definir una tupla emprant els parèntesis:

```
tupla = (1, 2, 'a', 13, True)
```

De fet, quan empram tuples dins expressions més complexes, convé emprar els parèntesis. Per exemple, quan volem definir una tupla niada, és a dir, una tupla que conté altres tuples:

```
tupla = (1, 2, 3), (4, 5, 6, 7)
```

Seguiu ara amb el [quadern de Colab](#), on trobareu més exemples i més conceptes relacionats amb les tuples.

■ Llistes

En contrast amb les tuples, les **llistes** són de longitud variable i el seu contingut es pot modificar (són mudables). Es poden definir usant claudàtors [] o amb la funció de tipus *list*.

```
llista = [1, 2, 3, 4, 5]
```

Com que una llista és mudable, podem fer (amb una tupla no podríem):

```
llista[0]=2
```

Les llistes tenen diversos mètodes que ens permeten operar amb els seus elements. Alguns dels més habituals són:

- *count()*: compta els elements de la llista
- *insert()*: insereix un nou element a una posició concreta de la llista
- *append()*: insereix un nou element al final de la llista
- *remove()*: elimina la primera ocurredoria d'un valor a la llista
- *index()*: retorna la posició del primer element que conté un cert valor

Les llistes i les tuples són semànticament semblants (tot i que les llistes són mudables i tenen més mètodes associats) i en moltes funcions es poden usar de forma intercanviable.



Les llistes es poden emprar per implementar el tipus de dades **array**, molt freqüent a altres llenguatges de programació com Java o C i derivats. En un **array** tots els elements són del mateix tipus de dades i es poden organitzar en una dimensió (vectors), dues (matrius), o més (arrays multidimensionals o N-dimensionals).

Per exemple, la següent instrucció:

```
m = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

crea una matriu (array de 2 dimensions, files i columnes) amb dues files i 4 columnes com aquesta:

1	2	3	4
5	6	7	8

Podem accedir a una posició concreta de la matriu emprant dos índexs (recordau que començam per 0). Per exemple:

```
m[1][2]
```

accedeix a la fila 1 (la segona) i columna 2 (la tercera). Per tant, recupera el valor 7.

IMPORTANT

Emprarem àmpliament **arrays** en el lliurament 2, mitjançant l'objecte **ndarray** de la llibreria NumPy.

Seguiu ara amb el [quadern de Colab](#).

■ Operadors i funcions de seqüències

Python incorpora un conjunt d'operadors i funcions per a les seqüències. La següent taula els mostra, ordenats de major a menor nivell de prioritat.

Operador o funció	Retorna
<code>x in s</code>	True si un element de s és igual a x, i False en cas contrari
<code>x not in s</code>	False si un element de s és igual a x, i True en cas contrari
<code>s + t</code>	La concatenació de s i t
<code>s * n or n * s</code>	L'equivalent a afegir s a si mateix n times
<code>s[i]</code>	<i>i</i> -èssim element de s, començant per 0
<code>s[i:j]</code>	Fragment (<i>slice</i>) de s des d' <i>i</i> fins a <i>j</i>
<code>s[i:j:k]</code>	Fragment (<i>slice</i>) de s des d' <i>i</i> fins a <i>j</i> amb una passa <i>k</i> (de <i>k</i> en <i>k</i>)
<code>len(s)</code>	Longitud de s
<code>min(s)</code>	Valor més petit de s
<code>max(s)</code>	Valor més gran de s
<code>s.index(x[, i[, j]])</code>	Índex de la primera ocurrència de x en s (a partir de la posició <i>i</i> i abans de la <i>j</i>)
<code>s.count(x)</code>	Número total d'ocurrències de x en s

A més, les següents quatre funcions de seqüència ens poden ser molt útils per fer feina amb tuples i llistes:

- **enumerate** s'usa en bucles for i retorna una seqüència de tuples (índex, valor)
- **sorted** retorna una llista ordenada a partir dels elements d'una seqüència
- **zip**, com una mena de cremallera, aparella els elements d'un nombre de llistes, tuples, o d'altres seqüències per crear una llista de tuples.
- **reversed** itera sobre els elements d'una seqüència en ordre invers.

Podeu trobar més detalls d'aquestes funcions en el [quadern de Colab](#).

■ Conjunts

Els **conjunts** són col·leccions **no ordenades d'elements únics**. Precisament, el fet que els elements no estiguin ordenats i no es puguin repetir és la principal diferència amb les altres seqüències. Els conjunts són una important eina matemàtica, sobre la qual hi ha definida multitud d'operacions, com per exemple la unió, la diferència o la comprovació de si un conjunt és subconjunt d'un altre. Totes aquestes operacions matemàtiques estan implementades mitjançant mètodes de l'objecte **set**.

Es poden crear de dues formes: a través de la funció **set** o bé com a literal amb claus {}. Els conjunts són un tipus mudable.

Seguiu ara amb el [quadern de Colab](#), on trobareu els detalls sobre els conjunts.

■ Diccionaris o tipus associatius

Un diccionari (**dict**), que també rep el noms de *hash map* o taula associativa, és una col·lecció de mida variable de parelles **clau-valor** (key-value), on la clau i el valor són objectes Python. Es poden crear usant les claus {} i els dos punts : per separar claus i valors.

Seguiu ara amb els detalls de com funcionen els diccionaris en el [quadern de Colab](#).

■ Comprehensions de llistes, conjunts i diccionaris

Les comprehensions (comprehensions en anglès) de llistes són una de les característiques de Python més apreciades. Permeten crear una nova llista de forma concisa filtrant els elements d'una col·lecció, transformant els elements que passen el filtre en una expressió concisa.

Ho podem veure en el [quadern de Colab](#).

8. Funcions

Les funcions són el mètode d'organització i reutilització de codi més important en Python. Com a regla general, si preveiem que haurem d'usar el mateix codi, o un de molt semblant, més d'una vegada, pot valer la pena escriure una funció reutilitzable. Les funcions també ajuden a fer el codi més llegible, perquè es dona un nom a un grup de sentències Python.

Recordem que declarem les funcions amb la paraula reservada `def` i especificuem el valor retornat amb `return`. Per exemple, la següent funció retorna el més gran de dos números:

```
def major(x, y):
    if (x>=y):
        return x
    else:
        return y
```

Es cridaria especificant els arguments en ordre (posicionalment):

```
a = major(3,2)
```

On se li assignaria a a el valor retornat (3). O també especificant el nom (keyword) dels arguments:

```
a = major(x=3,y=2)
```

Podeu veure molts més detalls sobre les funcions al [quadern de Colab](#).

9. Fitxers i sistema operatiu

Moltes vegades s'usen funcions d'alt nivell proporcionades per biblioteques diverses, com per exemple `pandas.read_csv`, per llegir fitxers de dades del disc cap a estructures de dades de Python. No obstant això, és important entendre les bases de com treballar amb fitxers en Python. Per sort, és molt senzill, i això és una de les raons que fan que Python sigui tan popular per manipular text i fitxers.

Vegem-ho en detall en aquest [quadern de Colab](#).

10. Classes i objectes

Python és un llenguatge orientat a objectes i com a tal, podem definir i instanciar les nostres pròpies classes.

Per declarar una classe, ho fem amb la paraula reservada **class** i especificant els seus atributs i mètodes. Vegem un primer exemple, molt senzill, on definim una classe *Salutacio* que té dos mètodes *hola* i *adeu* que imprimeixen respectivament "Hola" i "Adéu".

```
class Salutacio:  
    def hola(self):  
        print("Hola")  
  
    def adeu(self):  
        print("Adéu")
```

IMPORTANT

self fa referència al propi objecte i és obligatori declarar-lo de forma explícita com a primer argument en tots els mètodes d'una classe.

Vegem com cream un objecte o instància (*sal*) d'aquesta classe *Salutacio*:

```
sal = Salutacio()
```

I ara ja podem invocar els mètodes de l'objecte *sal*:

```
sal.hola()  
sal.adeu()
```

Que dona com a resultat:

```
Hola  
Adéu
```

Normalment, però, a més de mètodes, també voldrem declarar atributs (també anomenades propietats o variables) dins la classe.

Per definir atributs, ho farem mitjançant el mètode **__init__** (dos guions baixos abans i dos després de la paraula *init*). Aquest és el mètode constructor, el que es crida quan es crea un objecte de la classe.

Vegem un exemple, on volem definir una classe *Rectangle* (tot i que no és obligatori, en el nom de les classes normalment posam la primera lletra en majúscula), que té dues propietats *costat1* i *costat2* i dos mètodes per calcular l'àrea (*area*) i el perímetre (*perimetre*).

```
class Rectangle:  
    def __init__(self, costat1, costat2):  
        self.costat1 = costat1  
        self.costat2 = costat2  
  
    def area(self):  
        return self.costat1*self.costat2  
  
    def perimetre(self):  
        return 2*self.costat1 + 2*self.costat2
```

Podem crear una instància (*r*) i invocar els seus mètodes, *area()* en aquest cas:

```
r = Rectangle(3,5)  
r.area()
```

Podeu trobar molts més detalls en el [quadern de Colab](#).