CFGS Desenvolupament d'aplicacions multiplataforma
Curs 2021-2022

UD1 - Introducció a la programació amb Python

Desenvolupament d'interfícies

IES Jaume II El Just
Tavernes de la Valldigna

Professor:
Ferran Cunyat Pellicer

Continguts

1. Introducció a Python

1.1. Començant amb Python

1.1.1. Què és Python?

Python és un llenguatge de propòsit general, al igual que Java o C. El seu ús ha augmentat durant els últims temps gràcies a:

- La seua flexibilitat i simplicitat, que el fan fàcil d'aprendre
- El seu ús extens en camps com el "Data Science", la IA i l'aprenentatge de programació.
- Llenguatge d'scripting d'alt nivell.
- Multiplataforma.
- Gran quantitat i varietat de llibreries. Web frameworks, clients correu, gestors de contingut, concurrència, generació de documents, gràfics, intel·ligència artificial, …
- Lliure!! Mantés per la Python Software Foundation

1.1.2. Execució de Python

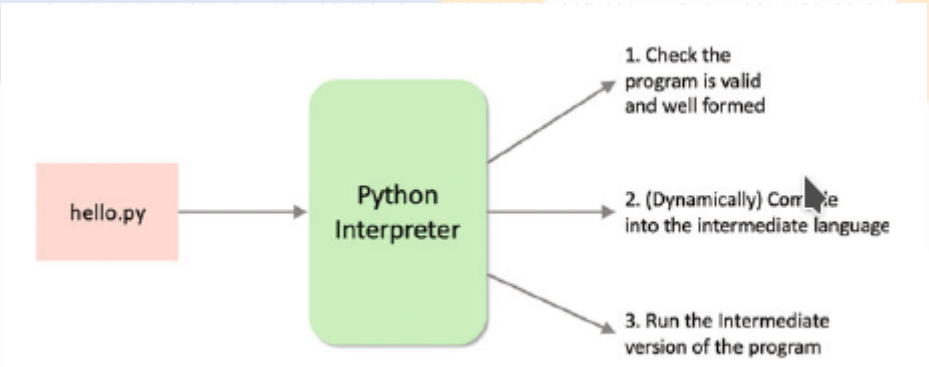

Figura 1: execució_python

1.1.3. Perquè Python

El llenguatge de programació Python és una opció cada vegada més utilitzada per principiants com per desenvolupadors experimentats. Flexible i versàtil, Python té punts forts en scripts, automatització, anàlisi de dades, aprenentatge automàtic i desenvolupament de back-end. Publicat per primera vegada el 1991 amb un nom inspirat en el grup de comèdia britànic Monty Python, l'equip de desenvolupament volia fer de Python un llenguatge divertit d'utilitzar.

C és el llenguatge de programació més popular a l'índex TIOBE, mentre que Python és el llenguatge més buscat a l'índex PYPL. Python i Java segueixen de prop a C al TIOBE. A PYPL, la diferència és més gran, ja que Python, que ocupa el primer lloc, supera en un 12 % a Java, que ocupa el segon lloc.

Aquest curs, aprendrem a utilitzar Python i el framework Qt per desenvolupar aplicacions amb interfície d'usuari.

1.1.4. Instal·lació de Python 3

Anem a instal·lar a l'Ubuntu 20.04 un entorn de programació Python 3, encara que valdrà per a qualsevol distribució basada en Debian Linux, com és el cas. En cas d'utilitzar Windows o MacOS, busqueu la forma d'instal·lar-lo.

Requisits previs Per poder instal·lar, haureu de tindre un usuari amb privilegis sudo en l'Ubuntu 20.04.

Primer pas - Configurant Python 3 Ubuntu 20.04 ja inclou Python 3 preinstal·lat. Per assegurar-nos que les nostres versions estan actualitzades, actualitzem els repositoris i actualitzem el sistema amb l'ordre apt:

```
$ sudo apt update
$ sudo apt -y upgrade
```

L'opció -y confirmarà que estem d'acord per instal·lar totes les actualitzacions, però, segons la vostra versió de Linux, és possible que hàgiu de confirmar les sol·licituds addicionals.

Una vegada finalitzat el procés, podem comprovar la versió de Python 3 que s'instal·la al sistema escrivint:

```
$ python3 -V
Python 3.8.10
```

El terminal vos indicarà el número de versió. No es tracta de l'última versió estable de Python (3.9.7), però si de la última disponibles als repositoris d'Ubuntu.

Per gestionar paquets de programari per a Python, instal·leu pip, una eina que instal·larà i gestionarà paquets de programació que és possible que vulguem utilitzar en els nostres projectes de desenvolupament.

```
$ sudo apt install -y python3-pip
```

Ara podrem utilitzar pip3 per instal·lar paquets de Python3.

```
$ pip3 install "paquet"
```

A més, instl·larem algunes llibreries necessàries per a la construcció dels nostres mòduls i extensions.

```
$ sudo apt install -y build-essential libssl-dev libffi-dev python3-dev
```

Segon pas - Configurem un entorn virtual de desenvolupament Els entorns virtuals ens permeten tenir un espai aïllat per a desenvolupament de projectes Python, cosa que garanteix que cadascun dels vostres projectes puga tindre el seu propi conjunt de dependències que no interrompin cap dels vostres altres projectes.

Configurar un entorn de programació proporciona un major control sobre els projectes de Python i sobre com es gestionen les diferents versions dels paquets. Això és especialment important quan es treballa amb paquets de tercers.

Podeu configurar tants entorns de programació Python com vulgueu. Cada entorn és bàsicament un directori que conté uns quants scripts per fer-lo actuar com a entorn aïllat de la resta de programes i llibreries de l'ordinador.

Tot i que hi ha algunes maneres d'aconseguir un entorn de programació a Python, farem servir el mòdul venv, que forma part de la biblioteca estàndard de Python 3. Instal·lem venv escrivint:

```
$ sudo apt install -y python3-venv
```

Amb açò, estem preparats per crear entorns virtuals *(virtual environments)*. Trieu en quin directori volem situar els nostres entorns de programació Python o bé creeu un directori nou amb mkdir:

```
$ mkdir environments
$ cd environments
```

Una vegada esteu al directori on voleu que es creen els entorns, podeu crear un entorn executant l'ordre següent:

```
$ python3 -m venv my_env
```

Essencialment, pyvenv crea un nou directori que conté alguns elements:

```
bin include lib lib64 pyvenv.cfg share
```

Junts, aquests fitxers funcionen per assegurar-vos que els vostres projectes estiguen aïllats, de manera que els fitxers del sistema i els fitxers de projecte no es mesclen i entren en conflicte. Aquesta és una bona pràctica per al control de versions i per garantir que cadascun dels vostres projectes tinga accés als paquets particulars que necessita.

Per utilitzar aquest entorn, l'heu d'activar, cosa que podeu aconseguir escrivint l'ordre següent que crida a l'script d'activació:

```
$ source my_env/bin/activate
```

El vostre indicador d'ordres ara tindrà el prefix amb el nom del vostre entorn, en aquest cas s'anomena my_env. Depenent de la versió de Debian Linux que utilitzeu, el vostre prefix pot aparèixer de manera diferent, però el nom del vostre entorn entre parèntesis hauria de ser el primer que veieu a la vostra línia:

Aquest prefix ens permet saber que l'entorn my_env està actiu actualment, és a dir, que quan creem programes aquí, només utilitzaran la configuració i els paquets d'aquest entorn concret.

Nota: a l'entorn virtual, podeu utilitzar l'ordre python en lloc de python3 i pip en lloc de pip3 si ho preferiu. Si utilitzeu Python 3 a la vostra màquina fora d'un entorn, haureu d'utilitzar exclusivament les ordres python3 i pip3.

Després de seguir aquests passos, el vostre entorn virtual ja es pot utilitzar.

Per a desactivar l'entorn virtual, simplement tanquem la consola o utilitzem l'ordre **deactivate**. Deapareixerà el nom de l'entorn virtual abans del prompt.

1.1.5. Modes d'execució

1. Interactiva a través de l'intèrpret

```
~$ python3
Python 3.7.3 (default, Jul 25 2020, 13:03:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2. Execució d'un fitxer Python

```
~$ python3 hola_mon.py
Hola món
```

3. Execució d'un script

```
hola_mon.py
---
#!/usr/bin/env python3
print("Hola món")
---

~$ chmod u+x hola_mon.py
```

```
~$ ./hola_mon.py
Hola món
```

4. Des d'un Entorn de Desenvolupament IDE

1.1.6. Activitat 1

Implementa el "Hola món!" i executa-lo de les quatre formes possibles.

1.2. Paraules reservades i identificadors

1.2.1. Paraules reservades

No es poden utilitzar coma identificador de variables ni nom de funcions, ja que s'utilitzen per a definir la sintaxi i l'estructura d'un programa. Les paraules reservades són:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Per a obtindre la llista completa des de l'intèrpret:

```
>>> import keyword
>>> print(keyword.kwlist)
```

1.2.2. Identificadors

Per convenció, els noms de les variables i funcions han de ser:

- combinacions de lletres minúscules i números
- començar per lletra
- descriptius (excepte en bucles o se sol utilitzar *i* i *j*)

- amb paraules separades per guió baix

Per exemple: nom_usuari, numero_telefon, cognom1, sumar(), ...

Recorda que Python és un llenguatge *case sensitive*. Per tant Var i var no són el mateix identificador.

1.3. Instruccions i sagnat

1.3.1. Instruccions

L'intèrpret de Python va executant línia a línia cada instrucció. Si volem que una instrucció ocupe diverses línies ho hem d'indicar amb el caràcter \.

Per exemple:

```
>>> a = 1 + 2 + 3 + \
...     4 + 5 + 6 + \
...     7 + 8 + 9
>>> print(a)
45
```

La continuació de línia és implicita dins de parèntesis (), corxets [] i claus {}.

```
colors = ['red',
          'blue',
          'green']
```

També podem posar diverses sentències en una línia:

```
a = 1; b = 2; c = 3
```

1.3.2. Sagnat

La majoria de llenguatges de programació utilitzen les claus {} per a definir blocs de codi. En canvi, Python utilitza el sagnat (identation).

Un bloc de codi (cos d'una funció, bucle, etc.) comença amb sagnat i acaba amb la primera línia sense sagnat. Depèn de vosaltres la quantitat de sagnat, però ha de ser coherent en tot el bloc. Generalment, s'utilitzen quatre espais en blanc per a sagnat i es prefereixen a les tabulacions. El resultat és un codi net i clar. Exemple:

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

Un sagnat incorrecte llança un *IndentationError*.

1.3.3. Comentaris

Els comentaris són molt importants en escriure un programa. Descriuen el que passa dins d'un programa, de manera que una persona que mira el codi font no tinga dificultats per entendre'l. A més, és possible que oblideu els detalls clau d'implementació del programa que acabeu d'escriure. Per tant, **invertir temps per explicar aquests conceptes en forma de comentaris sempre és fructífer**.

A Python, fem servir el símbol coixinet (#) per començar a escriure un comentari. S'estén fins al caràcter de línia nova. No cal que el comentari estiga a principi de línia, pot estar en mig.

Per fer comentaris multilínia, podem utilitzar el coixinet a principi de cada línia. També podem fer servir les cometes dobles o simples tres vegades.

Exemple:

```
''' Comentari
multilínia'''
```

1.3.4. Docstrings

Una docstring és una abreviatura de text de documentació.

La documentació de Python són els textos que apareixen just després de la definició d'una funció, mètode, classe o mòdul.

S'utilitzen cometes triples durant l'escriptura de la documentació.

Per exemple:

```
def doble (num):
    """Funció per duplicar el valor"""
    return 2 * num
```

La documentació s'associa a l'objecte com a atribut **doc**.

Per tant, podem accedir a la documentació de la funció anterior amb les següents línies de codi:

```
def doble (num):
    """Funció per duplicar el valor"""
    return 2 * num
print(doble .__ doc__)
```

```
def suma_binaria(a, b):
    '''
```

```
    Torna la suma de dos enters en binari.

            Paràmetres:
                    a (int): Un enter
                    b (int): Altre enter

            Torna:
                    suma_binaria (str): String amb els digits binaris de
                       la suma
    '''
    suma_binaria = bin(a+b)[2:]
    return suma_binaria


print(suma_binaria.__doc__)
```

1.4. Variables, constants i tipus

1.4.1. Variables

En Python, quan declarem una variable i li assignem un valor, realment estem creant un objecte i assignant un valor per referència.

```
>>> num = 10
>>> type(num)
<class 'int'>
>>> num = 10.0
>>> type(num)
<class 'float'>
```

Podem inicialitzar múltiples variables en una mateixa línia, ja siga amb el mateix valor o diferent.

```
a, b, c = 5, 3.2, "Hola"
x = y = z = "iguals"
```

1.4.2. Constants

A Python no existeixen les constants a l'estil de *static final* de Java, sinó que simplement es defineix una variable que no es modifica el valor. Normalment es definixen en un mòdul a banda, utilitzant majúscules i guió baix si és necessari, que s'importa a l'arxiu principal.

```
constants.py
---
PI = 3.14
```

```
main.py
---
import constants.py
radi = 5
perimetre = 2 * constants.PI * radi
print(perimetre)
```

1.4.3. Tipus

L'assignació de tipus és dinàmica i pot canviar, per això no declarem els tipus de les variables. Per determinar el tipus d'un objecte, fem servir el mètode **type()**. Els tipus d'objecte definits a Python3 són: 1. Numèrics:

1.1. Integer

```
    a = 0b1010 #Binary
    b = 100 #Decimal
    c = 0o310 #Octal
    d = 0x12c #Hexadecimal
```

1.2. Float

```
    float_1 = 10.5
    float_2 = 1.5e2
```

1.3. Complex

```
    x = 3 + 4j
```

2. Strings

```
    nom = 'Ferran Cunyat'
```

Els principals mètodes sobre un string són **capitalize(), count(), find(), format(), lower(), replace(), split(), title(), translate(), upper()**.

3. Boolean

```
    x = (1 == True) # False pren el valor numéric 1, mentre que False el 0
    y = (1 == False)
    a = True + 4
    b = False + 10
```

4. Especials (None)

S'utilitza per no donar-li valor a una variable.

```
>>> x = None
>>> type(x)
<class 'NoneType'>
```

5. Col·leccions

    1. Llista

Són una seqüència d'elements, no necessàriament del mateix tipus, encara que normalment si que ho són. Es definix amb corxets i els elements separats per comes. Podem accedir a un element o un rang i és mutable.

```
>>> a = [5,10,15,20,25,30,45,40]
>>> print(a)
[5, 10, 15, 20, 25, 30, 45, 40]
>>> a[3] = "Ferran"
>>> print(a[:4])
[5, 10, 15, 'Ferran']
```

Algunes de les principals funcions que podem aplicar sobre llistes són **apend(), clear(), copy(), extend(), insert(), remove(), reverse()**.

2. Tupla Són una seqüència d'elements, no necessàriament del mateix tipus, però esta vegada immutable.

```
>>> tupla = (2,'hola')
>>> tupla[1] = 'clavel'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

3. Rang

Seqüència immutable de números, generalment utilitzada per a iterar sobre for o generar llistes ràpidament.

```
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
```

4. Conjunt (Set)

Col·lecció desordenada d'elements únics. Ja que és una llista desordenada, no la podem indexar i per tant accedir als elements segons la seua posició.

```
>>> a = {2,2,5,5,4,10,1,0}
>>> print(a)
{0, 1, 2, 4, 5, 10}
```

```
>>> a[1]
Traceback (most recent call last):
File "<string>", line 301, in runcode
File "<interactive input>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

5. Diccionari

Diccionari és una col·lecció no ordenada de parells valor-clau.

Generalment s'utilitza quan tenim una gran quantitat de dades. Els diccionaris estan optimitzats per recuperar dades. Hem de conèixer la clau per recuperar el valor.

A Python, els diccionaris es defineixen entre claus {} i cada element és un parell que adopta la forma de clau:valor. La clau i el valor poden ser de qualsevol tipus.

```
>>> d = {1:'valor','clau':2}
>>> print(type(d))
<class 'dict'>
>>> print("d[1] = ", d[1]);
d[1] =  valor
>>> print("d['clau'] = ", d['clau']);
d['clau'] =  2
>>> print("d[2] = ", d[2]);
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
KeyError: 2
```

1.4.4. Activitat 2

Fixa't en el següent fragment de codi:

```
>>> s = [1,2]
>>> r = s[:]
>>> s[0]=2
>>> print(s)
[2, 2]
>>> print(r)
???
>>> print(s)
???
```

Quina serà l'eixida? Per qué?

```
>>> r = s
>>> s[0]= 5
>>> print(r)
???
```

```
>>> print(s)
???
```

I ara? Per qué?

1.4.5. Conversióentre tipus

Conversió implícita Són conversions que fa el mateix llenguatge automàticament. Per exemple:

```
num_int = 123
num_flo = 1.23

num_nou = num_int + num_flo

print("num_int és de tipus:",type(num_int))
print("num_flo és de tipus:",type(num_flo))

print("Valor de num_nou:",num_nou)
print("num_nou és de tipus:",type(num_nou))
```

1.4.6. Conversió explícita

Utilitzem funcions predefinides per a forçar la conversió **int(), float(), str(), ...**.

1.4.7. Activitat 3

Quin és el resultat d'executar el següent fragment de codi?

```
>>> enter = 123
>>> cadena = "456"
>>> suma = enter + cadena
```

Definix dues variables, una per a fer la suma entera (579) i l'altra per a concatrenar com a text (123456).

1.5. Entrada, eixida i import

1.5.1. Entrada

Ja hem vist que per a introduir informació per teclat utilitzem la funció **input([prompt])**.

1.5.2. Eixida

Per a imprimir per ella utilitzarem la funció **print()**. Moltes vegades s'utilitza en combinació amb la funció **format()** dels strings.

```
>>> x = 5; y = 10
>>> print('El valor d\'x és {} i el de y és {}'.format(x,y))
>>> # Fixeu-se que hem escapat el caracter '
El valor d'x és 5 i el de y és 10
>>> print('Hola {nom}, {salutacio}'.format(salutacio = 'Bon dia', nom = '
    Pau'))
```

Activitat 4 Implementa el programa "Hola món" en una sola instrucció.

1.5.3. Import

Quan volem fer ús del codi d'un altre mòdul (.py), l'hem d'importar al programa actual amb **import**.

```
>>> import math
>>> print(math.pi)
```

També podem importar sols alguna de les funcions o atributs amb **from**

```
>>> from math import pi
```

En importar un mòdul, Python analitza diversos llocs definits a sys.path. És una llista de les ubicacions on buscar el mòdul.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python37.zip', '/usr/lib/python3.7', '/usr/lib/python3.7/
    lib-dynload', '/home/ferran/.local/lib/python3.7/site-packages', '/usr/
    local/lib/python3.7/dist-packages', '/usr/lib/python3/dist-packages']
```

Per a importar d'una altra carpeta, ho fem amb **from *paquet* import *modul***. Pots posar ubicacions absolutes o relatives. Si volem importar un codi d'una ubicació que no està al path.

```
>>> import sys
>>> sys.path.append('/path/a/la/carpeta')
```

1.6. Espai de noms i àmbit de variables

1.6.1. Noms

En Python, tot són objectes, inclús les funcions. Un nom és la forma d'accedir als objectes, i amb la funció **id()** podem veure la seua ubicació en memòria. Per exemple:

```
>>> a = 2
>>>print('id(2) =', id(2))
id(2) = 9062656
>>>print('id(a) =', id(a))
d(a) = 9062656
```

Activitat 5 Quin creus que serà el resultat al següent fragment de codi?

```
>>> a = 2
>>> print('id(a) =', id(a))

>>> a = a+1
>>> print('id(a) =', id(a))
>>> print('id(3) =', id(3))

>>> b = 2
>>> print('id(b) =', id(b))
>>> print('id(2) =', id(2))
```

1.6.2. Espais de noms (namespaces)

Els espais de noms a Python, són una col·lecció de noms.

Diferents espais de noms poden coexistir, però estan completament aïllats.

Quan iniciem Python, es crea un espai de noms amb totes les funcions que l'intèrpret reconeix, **espai de noms predefinit**. Aquesta és la raó per la qual funcions integrades com id (), print () etc. sempre estan disponibles per a nosaltres des de qualsevol part del programa.

En canvi, cada mòdul crea el seu propi **espai de noms global**. Aquests espais de noms estan aïllats entre ells. Per tant, podem donar el mateix nom a objectes de mòduls diferents sense que entren en conflicte.

Els mòduls poden contindre funcions i classes. Quan es crida una funció, es crea un **espai de noms local** que té noms propis definits. Similar, és el cas de la classe. El següent diagrama pot ajudar a aclarir aquest concepte.

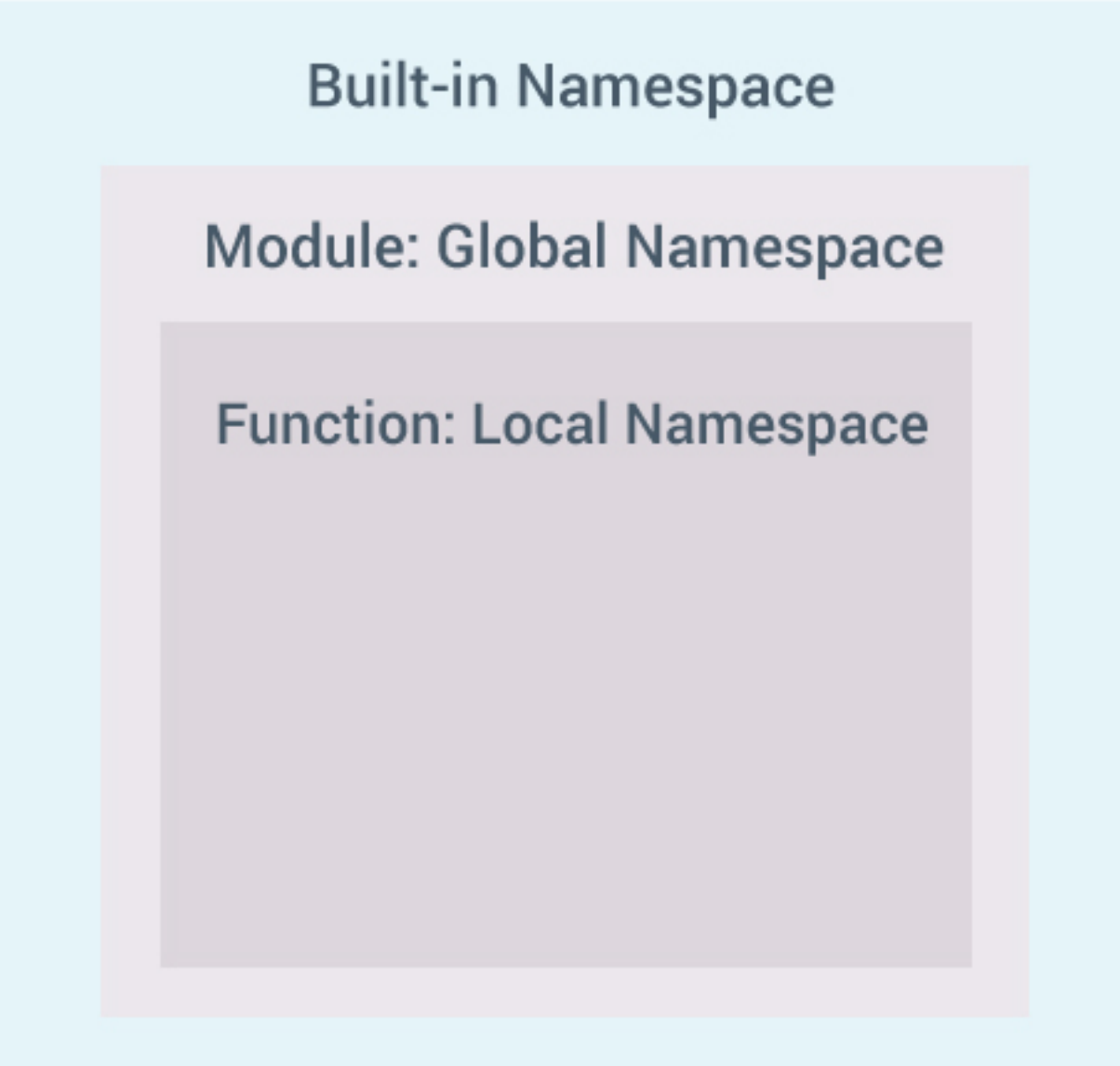

Figura 2: namespaces

1.6.3. Àmbit de les variables

Tot i que hi ha diversos espais de noms definits, és possible que no puguem accedir a tots ells des de totes les parts del programa. El concepte d'àmbit entra en joc.

Sempre hi ha almenys tres àmbits definits.

1. Àmbit local a la funció amb nom locals
2. Àmbit del mòdul que té noms globals

3. Àmbit més extern que té noms predefinits

Quan es fa una referència dins d'una funció, el nom es busca a l'espai de noms local, després a l'espai de noms global i finalment a l'espai de noms predefinit.

Si hi ha una funció dins d'una altra funció, s'anida un nou àmbit dins de l'àmbit local.

Activitat 6 Quin serà el resultat mostrat per consola a l'executar el següent fragment de codi?

```
def funcio_externa():
    a = 20

    def funcio_interna():
        a = 30
        print('a =', a)

    funcio_interna()
    print('a =', a)


a = 10
funcio_externa()
print('a =', a)
```

Activitat 7 Quin serà el resultat mostrat per consola a l'executar el següent fragment de codi?

```
def funcio_externa():
    global a
    a = 20

    def funcio_interna():
        global a
        a = 30
        print('a =', a)

    funcio_interna()
    print('a =', a)


a = 10
funcio_externa()
print('a =', a)
```

2. Control de fluxe

2.1. If ... else

```
num = int(input("Número: "))
if num > 0:
    print("Positiu")
elif num == 0:
    print("Zero")
else:
    print("Negatiu")
```

2.2. For

No existeix un for a l'estil de C o Java. Amés podem afegir un else al final de bucle.

```
nums = [6, 5, 3, 8, 4, 2, 5, 4, 11]

suma = 0

for val in nums:
    suma = suma + val
else:
    print("Hem acabat de sumar")

print("La suma és", suma)
```

Podem combinar el bucle for amb la funció **range(principi, fi, pas)**.

Activitat 8 Fes una aplicació que imprimisca els primers 100 números imparells.

Activitat 9 Fes una aplicació que donada la següent llista, imprimisca els seus membres: aficions = ['esports', 'cine', 'teatre']

2.3. While

```
contador = 0

while contador < 3:
    print("Dins del while")
    contador = contador + 1
else:
    print("Fora del bucle")
```

2.4. Break i continue

S'utilitzen igual que a Java. El continue passa a la següent iteració, mentre que el break ix del bucle. En cas de bucles anidats, ix del bucle intern.

3. Funcions

3.1. Definició de funcions

```
def nom_funcio(paràmetres):
    """docstring"""
    instruccions(s)
```

Nota: Recordeu làmbit de les variables, ja que hi haurà variables locals a la funció.

3.2. Arguments

3.2.1. Valors per defecte

Els arguments de les funcions poden tindre un valor per defecte. En cas de no assignar-li un valor per defecte, necessitem fer la crida passant-li el valor de l'argument.

```
def saluda(nom="desconegut", msg="Benvingut!"):
    """
    Funció per saludar a un usuari

    Entrada:
        nom="desconegut": String, nom de l'usuari
        msg="Benvingut!": String, missatge de salutació

    Si no li proporcionem valors en la crida,
    utilitzarà els valors per defecte
    """

    print("Hola", nom + '.', msg)
```

La crida a la funció la podem fer sense arguments, amb un o amb dos. És una forma de fer una sobrecàrrega de mètodes de forma molt ràpida.

```
saluda("Tomàs", "Qué fas?")
saluda("Pau")
saluda()
saluda(msg="", nom="Artur")
saluda(msg="","Artur")
```

3.2.2. Nombre arbitrari d'arguments

Si no sabem a priori quants arguments rebrà la funció, podem utilitzar el caràcter "*" en la definició de la funció.

```
def saluda(*noms):
    """
    Funció per saludar a un conjunt d'usuaris

    Entrada:
        noms: llistat de noms
    """
    for nom in noms:
        print("Hola", nom)



saluda("Alex","Guillem","Javier")
```

3.3. Funcions recursives

La recursió és el procés de definir alguna cosa en termes d'eixa mateixa cosa. Aleshores, una funció recursiva és aquella que es crida a sí mateix. S'ha d'anar molt en compte en crear correctament la condició d'eixida de la funció, ja que d'altra forma entrariem en bucle infinit.

```
def factorial(x):
    """Funció per a calcular el fatorial d'un enter.

        Entrada:
            - x: int, el nombre del que volem calcular el factorial

        Eixida:
            - x!: int, factorial d'x
    """

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = int(input("Número: "))
print(num, "! =", factorial(num))
```

3.4. Funcions anònimes

Les **funcions anònimes o funcions lambda**, són funcions sense nom. Poden tindre un nombre indeterminat d'arguments, però sols una expressió, que serà avaluada i retornat el seu resultat.

M'interessa crear una agrupació que els incloga als dos per si algun dia vull que algun recurs/activitat els arribe a tots-dos?

```
quadrat = lambda x: x ** 2

print(quadrat(5))
```

Normalment les funcions lambda s'utilitzen en combinació amb altres funcions com filter(), map(), etc.

La ***funció map()*** rep com a arguments una funció i una llista, i torna una llista del mateix tamany on cada element és el resultat d'aplicar la funció sobre l'element que ocupa la mateixa posició a la llista original.

La **funció filter()**, rep una funció i una llista com a arguments, i torna com a resultat una llista amb els elements que avaluen a True la funció.

```
llista = [1, 5, 4, 6, 8, 11, 3, 12]

nova_llista = list(map(lambda x: x ** 2 , llista))

print(nova_llista)
```

3.4.1. Activitat 10

Definix una llista i utilitzant filter, que la separe en dues llistes, una amb els elements parells i l'altra amb els senars.

3.5. Packages

Igual que la informació al disc dur està organitzada en carpetes i subcarpetes, Un programa en Python es pot organitzar en paquets, sub paquets i mòduls. Açò fa un programa més fàcil de gestionar i conceptualment més clar. Una carpeta conté un arxiu anomenat __init__.py. Este arxiu pot estar buit o no, però normalment conté codi d'inicialització.

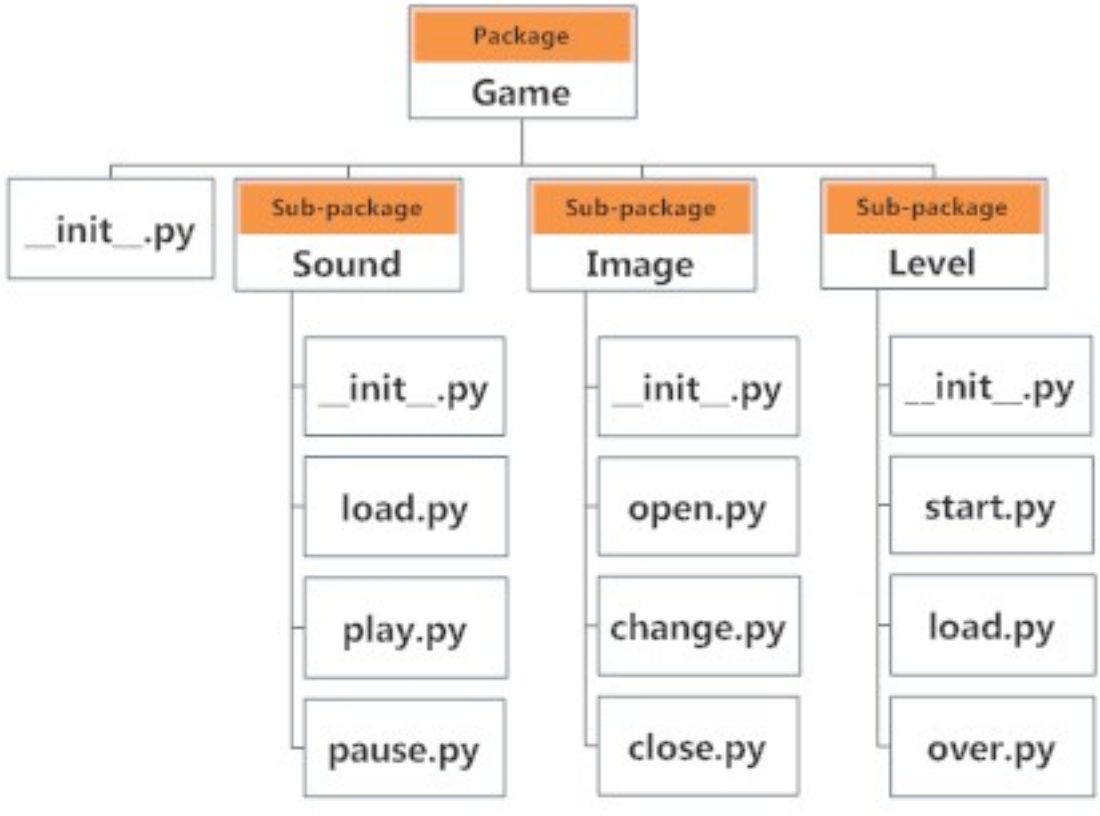

Figura 3: paquets

Per a importar un mòdul d'un paquet utilitzariem **import** o **from ... import**.

```
import Game.Level.start
from Game.Level import start
```

4. Tractament de fitxers

4.1. Fitxers

4.1.1. Entrada eixida utilitzant fitxers

Per a llegir o escriure en un fitxer, primer l'hem d'obrir. Quan acabem, s'ha de tancar perquè s'alliberen els recursos relacionats amb el fitxer.

Per tant, a Python, utilitzem la següent seqüència d'operacions per traballar amb fitxers:

1. Obrir un fitxer

2. Llegir o escriure
3. Tancar el fitxer

4.1.2. Open

Per obrir un fitxer utilitzem la funció **open()**.

```
>>> f = open("test.txt")    # arxiu en el mateix directori
>>> f = open("C:/Python38/README.txt")  # path sencer
```

A més, podem especificar el mode d'apertura i la codificació.

Mode	Descripció
r	lectura
w	escriptura
x	creació exclusiva (falla si ja existeix)
a	afegir al final, el crea si no existeix
t	mode lectura de text (per defecte)
b	mode binari
+	actualització (lectura i escriptura)

```
>>> f = open("test.txt",'w') # obert per a escriptura
>>> f = open("test.txt", mode='r', encoding='utf-8')
```

4.1.3. Close

Python utilitza un *garbage collector* per netejar objectes sense referències, però no hem de confiar per tancar el fitxer.

```
try:
   f = open("test.txt", encoding = 'utf-8')
   # operacions sobre l'arxiu
finally:
   f.close()
```

Altra possibilitat és amb **with**. En este cas no hem de tancar-lo explícitament.

```
with open("test.txt", encoding = 'utf-8') as f:
```

```
    # operacions sobre l'arxiu
```

4.1.4. Escriptura

Per a escriure, necessitem haver-lo obert amb les opcions w, a o x. Compte amb l'opció w, perquè sobreescriu els arxius.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
    f.write("Primer arxiu\n")
    f.write("Este arxiu\n")
    f.write("conté tres línies\n")
```

4.1.5. Lectura

Utilitzarem el mètode **read()** per a llegir. La funció **tell()** ens diu en quina posició tenim el cursor i amb **seek()** el podem modificar.

```
>>> f = open("test.txt",'r',encoding = 'utf-8')
>>> f.read(6)
'Primer'

>>> f.read(6)
' arxiu'

>>> f.read()     # llig fins al final
'\nEste arxiu\nconté tres línies\n'

>>> f.read()  # posteriors lectures tornen la cadena buida
''
```

```
>>> f.tell()
45

>>> f.seek(0)
0

>>> print(f.read())
Primer arxiu
Este arxiu
conté tres línies
```

També podem utilitzar la funció **readline()** per a llegir una línia, o **readlines()** per a que ens torne una llista de línies llegides.

Activitat 11 Crea una aplicació que vaja llegint operacions d'un fitxer (una operació per línia) i afegisca els resultats. Per exemple, si llig: 4 + 4

Haurà de generar: 4 + 4 = 8

Utilitza funcions anònimes per a implementar les operacions.

4.2. Directoris

Si hi ha una gran quantitat de fitxers i directoris amb els que tractar, disposem del mòdul os (operating system), que ens proporciona mètodes per al seu tractament.

Per a veure el directori de treball, utilitzem **getcwd()**.

```
>>> import os

>>> os.getcwd()
```

Per a canviar de directori, **chdir()**.

```
>>> os.chdir('/home/ferran')
```

Per a llistar els directoris ens servim de **listdir()**.

```
>>> os.listdir('/home')
```

Per crear un directori usem **mkdir()**.

```
>>> os.mkdir('Nova_carpeta')
```

Si volem renombrar un directori.

```
>>> os.rename('Nova_carpeta','Vella_carpeta')
```

Per a eliminar un arxiu utilitzarem **remove()**. Si el que volem eliminar és una carpeta buida **rmdir()**.

```
>>> os.remove('arxiu.txt')
>>> os.rmdir('Vella_carpeta')
```

En el cas que la carpeta no estiga buida, hem d'importar el mòdul **shutil** i utilitzar la funció **rmtree()**.

```
>>> import shutil
>>> shutil.rmtree('Carpeta')
```

5. Errors i excepcions

Podem cometre errors mentre programem. Estos errors es poden classificar bàsicament en dos tipus: 1. Errors de sintaxi 2. Errors en la lògica (Excepcions)

Els erros de sintaxi es produixen abans de l'execució, mentre que els erros lògics es produixen en temps d'execució.

5.1. Excepcions

Algun exemple d'excepció són els següents: - Quan intentem obrir un fitxer (per llegir) que no existeix (FileNotFoundError) - Quan intentem dividir un nombre per zero (ZeroDivisionError) - Quan intentem importar un mòdul que no existeix (ImportError).

Sempre que es produeixen aquests tipus d'errors d'execució, Python crea un objecte d'excepció. Si no el tractem, interrumpeix l'execució i imprimeix una traça de l'error juntament amb alguns detalls sobre per què ha produït aquest error.

5.1.1. Excepcions definides en Python

Per a consultar totes les excepcions definides podem utilitzar:

```
>>> print(dir(locals()['__builtins__']))
```

- ArithmeticError
- AssertionError
- AttributeError
- BaseException
- BlockingIOError
- BrokenPipeError
- BufferError
- ChildProcessError
- ConnectionAbortedError
- ConnectionError
- ConnectionRefusedError
- ConnectionResetError
- EOFError
- EnvironmentError
- Exception

- FileExistsError
- FileNotFoundError
- FloatingPointError
- GeneratorExit
- IOError
- ImportError
- IndentationError
- IndexError
- InterruptedError
- IsADirectoryError
- KeyError
- LookupError
- MemoryError
- ModuleNotFoundError
- NameError
- NotADirectoryError
- NotImplementedError
- OSError
- OverflowError
- PermissionError
- ProcessLookupError
- RecursionError
- ReferenceError
- RuntimeError
- StopAsyncIteration
- SyntaxError
- SystemError
- TabError
- TimeoutError
- TypeError
- UnboundLocalError
- UnicodeDecodeError
- UnicodeEncodeError
- UnicodeError
- UnicodeTranslateError
- ValueError

5.1.2. Com funcionen les excepcions

Python llança una d'aquestes excepcions en executar una instrucció que provoca un error.

Quan es produeixen aquestes excepcions, l'intèrpret de Python detén l'execució del procés actual i el passa al procés que ha fet la crida, fins que algú la tracte. Si ningún procés la tracta, el programa es detendrà.

Per exemple, considerem un programa en què tenim una funció A que crida a la funció B, que al seu torn crida a la funció C. Si es produeix una excepció a la funció C però no es tracta a C, l'excepció passa a B i després a A.

Si no es gestiona mai, es mostrarà un missatge d'error i el nostre programa s'aturarà de sobte.

5.2. Capturant excepcions en Python

A Python, les excepcions es poden gestionar mitjançant una sentència try.

L'operació crítica que pot generar una excepció es col·loca dins de la clàusula try. El codi que gestiona les excepcions s'escriu a la clàusula except i s'executarà en produir-se. Per tant, podem escollir quines operacions es realitzaran una vegada que haguem capturat l'excepció.

Exemple:

```
# importem el mòdul sys per veure el tipus d'excepció
import sys

randomList = ['a', 0, 2]

for element in randomList:
    try:
        print("Element val ", element)
        r = 1/int(element)
        break
    except:
        print("Oops! Excepció capturada -->", sys.exc_info()[0])
        print("Següent iteració")
        print()
print("L'invers de ", element, "és", r)
```

Com totes les excepcions hereden de la superclasse Exception, podem reescriure el programa com:

```
# importem el mòdul sys per veure el tipus d'excepció
import sys

randomList = ['a', 0, 2]
```

```
for element in randomList:
    try:
        print("Element val ", element)
        r = 1/int(element)
        break
    except Exception as e:
        print("Oops! Excepció capturada -->", e.__class__)
        print("Següent iteració")
        print()
print("L'invers de ", element, "és", r)
```

A l'exemple anterior, no hem utilitzat cap excepció específica a la clàusula except.

No és una bona pràctica de programació, ja que capturarà totes les excepcions i gestionarà tots els casos de la mateixa manera. Podem especificar quines excepcions hauria de capturar una clàusula except.

Una clàusula try pot tindre un nombre indeterminat de clàusules except per gestionar diferents excepcions, però, només s'executarà una en cas que es produeixi una excepció.

Podem utilitzar una tupla de valors per especificar diverses excepcions en una clàusula except.

Exemple:

```
try:
   # fer alguna cosa
   pass

except ValueError:
   # manegem ValueError
   pass

except (TypeError, ZeroDivisionError):
   # manegem múltiples excepcions
   # TypeError i ZeroDivisionError
   pass

except:
   # manegem totes les altres excepcions
   pass
```

5.2.1. Try ... except ... finally

La sentència try de Python pot tindre una clàusula final opcional. Aquesta clàusula s'executa independentment de si es produix una excepció o no i s'utilitza generalment per alliberar recursos.

Per exemple, podem estar connectats a un centre de dades remot a través de la xarxa o treballar amb un fitxer o una interfície gràfica d'usuari (GUI). En totes aquestes circumstàncies, hem de netejar

recursos abans que el programa ature la seua execució, tant si ho fa de forma controlada com si es para bruscament. Aquestes accions (tancar un fitxer, GUI o desconnectar de la xarxa) es realitzen normalment a la clàusula final.

Exemple:

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # operacions amb el fitxer
finally:
   f.close()
```

5.3. Llançant excepcions en Python

Les excepcions es generen normalment quan es produeixen errors en temps d'execució, però també podem generar/llançar excepcions manualment mitjançant la paraula reservada **raise**. Opcionalment, podem passar arguments a l'excepció per aclarir per què s'ha generat aquesta excepció.

```
import sys
try:
    nombre = int(input("Dona'm un número positiu: "))
    if nombre <= 0:
        raise ValueError(str(nombre) + " no és un número positiu!")
    1/0
except ValueError as ve:
    print("Excepció capturada:", ve)
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

5.3.1. Assert

Amb la paraula reservada **assert** llancem una excepció sempre i quan l'expressió que la segueix s'avalua a *Fals*. Si s'avalua a *True*, es continua l'execució del programa de forma seqüencial.

```
try:
    num = int(input("Introdueix un nombre positiu: "))
    assert num > 0
except AssertionError:
    print("AssertionError: No és positiu!")
except ValueError:
    print ("ValueError: No has introdït un número!")
else:
    print("És positiu")
```

5.4. Excepcions definides per l'usuari

De vegades necessitem definir excepcions que no estan disponibles a Python quan es dóna alguna condició. En este cas, hem de crear les nostres propies excepcions. Per a fer-ho, hem de definir noves classes que **hereden de *Exception***, ja siga directa o indirectament. Esta nova excepció que hem creat també podrà ser llançada amb *raise*.

```
>>>class CustomError(Exception):
...    pass

>>> raise CustomError
Traceback (most recent call last):
...
__main__.CustomError

>>> raise CustomError("S'ha produït un error")
Traceback (most recent call last):
...
__main__.CustomError: S'ha produït un error
```

Quan estem desenvolupant un programa gran, és una bona pràctica col·locar totes les excepcions definides per l'usuari que el nostre programa definix en un fitxer separat *exceptions.py* o *errors.py*.

Activitat 12 Modifica el codi de l'activitat 11 per a que no es produïsquen errors en l'execució, ja siga per introdïur valor no definits per a les funcions, valors que no són numèrics o operacions desconegudes. Controla també que no es produïsquen errors en la lectura/escriptura dels arxius.

Activitat 13 Anem a implementar un xicotet joc per consola. El programa generarà un número aleatori entre 0 i 100 (utilitzeu randint() del mòdul random) i demanarà a l'usuari que introduïsca un número.

Mentre el número siga massa menut, llançarà una excepció ErrorEnterMassaMenut indicant-li-ho. Si per contra és massa gran llançarà ErrorEnterMassaGran.

El joc acabarà quan s'introduïsca un valor no numèric o quan s'introduïsca l'enter buscat, en este cas felicitarà a l'usuari.

6. Programació orientada a objectes amb Python

És un dels paradigmes més populars per resoldre problemes a través de la programació.

Un objecte té dues característiques:

- atributs (estat)
- funcions (comportament)

Per exemple:

Una persona pot ser un objecte, ja que té les propietats següents:

- nom, edat com a atributs
- cantar, ballar com a comportament

El concepte de POO a Python se centra en la reutilització de codi. Aquest concepte també es coneix com DRY (Don't Repeat Yourself).

6.1. Classes

Una classe és una definició d'un objecte abstracte, que representa algun ent de la realitat al nostre programa. Conté tots els detalls comuns sobre tots els objectes del mateix tipus.

L'exemple de classe de lloro pot ser:

class Parrot(): pass Ací fem servir la paraula clau class per definir una classe Loro buida.

6.2. Objectes

Quan es defineix la classe, només es defineix la descripció de l'objecte. Per tant, no s'assignen recursos per a la seua execució, ni s'assignen valors als seus atributs. Quan este fet es produeix, aleshores tenim un objecte en memòria sobre el que podem actuar.

L'exemple d'objecte de classe lloro pot ser:

obj = Loro () Aquí, obj és un objecte de la classe Loro.

Suposem que tenim detalls de lloros. Ara, podem construir la classe i crear un objecte per a cada lloro.

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
```

```
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))

---
Eixida
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

Al programa anterior, hem creat una classe Parrot. A continuació, hem definit uns atributs, que prendran valors diferents en la instanciació d'objectes diferents.

Aquests atributs es defineixen dins del mètode **init** de la classe, que és el mètode inicialitzador que s'executarà només creem objectes.

Després, creem instàncies de la classe Parrot. *blu i woo* són referències (valor) als nostres objectes nous.

Podem accedir als atributs de classe mitjançant `__class__.species`. Els atributs de classe són els mateixos per a totes les instàncies d'una classe. De la mateixa manera, accedim als atributs de la instància mitjançant `objecte.nom_atribut`. Els atributs d'instància (valors) són diferents per a cada instància d'una classe.

6.3. Mètodes

Són funcions definides dins el cos d'una classe. S'utilitzen per a definir el comportament de l'objecte.

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
```

```
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
---
Blu sings 'Happy'
Blu is now dancing
```

Hem definit dos mètodes sing() i dnce(), que són mètodes d'instància, ja que es criden sobre un objecte.

6.4. Herència

L'herència és una forma de reutilitzar codi sense tindre-lo que reescriure. Açò facilita el manteniment de les aplicacions.

Les noves classes s'anomenen classes derivades (o classe filla). La classes de les que deriven són les classes base (o classe pare).

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")
```

```
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
---
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

En l'anterior programa, la classe Penguin hereda de la classe Bird. La classe derivada hereda el mètode *swim()*, modifica el mètode *whoisThis()* i extén amb un nou mètode *run()*.

Utilitzem `super().__init__()` dins de l'`__init()__` per a inicialitzar la classe pare.

6.5. Encapsulament

Podem restringir l'accés a mètodes i variables, és a dir, definir-los com a privtas. Això impedeix que les dades es modifiquen directament accedint als atributs, és el que anomenem encapsulació. Definim atributs o mètodes privats utilitzant el guió baix com a prefix, és a dir, simple _ o doble __.

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

```
Eixida
Selling Price: 900
Selling Price: 900
```

```
Selling Price: 1000
```

Com vegem, per canviar el valor, hem d'utilitzar una funció modificadora *setter*, és a dir, setMaxPrice (), que pren el preu com a paràmetre.

6.6. Polimorfisme

El polimorfisme és la capacitat d'utilitzar una interfície comuna (crides amb els mateixos noms) en diferents classes derivades.

Suposem que hem de pintar una forma i que hi ha diverses opcions: rectangle, quadrat, cercle, … Podríem utilitzar el mateix mètode per a pintar qualsevol forma. Aquest concepte s'anomena polimorfisme.

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

```
Parrot can fly
Penguin can't fly
```

Al programa anterior, hem definit dues classes Parrot i Penguin. Cadascun d'elles té un mètode comú fly(). No obstant això, les seues funcions són diferents.

Per utilitzar el polimorfisme, hem creat una interfície comuna, és a dir, la funció flying_test() que pren com a paràmetre qualsevol objecte i crida al mètode fly() de l'objecte. Així, quan passem els objectes blu i peggy a la funció flying_test(), s'executa el mètode corresponent a cadascuna.