RESUMEN

December 14, 2023

0.0.1 Propertys

- Una property es como un atributo, pero podemos modificar su comportamiento.
- Podemos leer (**getter**) y modificar (**setter**) el valor de un objeto encapsulado

[ ]:
```
# Ejemplo Properties:

class Puente:
    def __init__(self, maximo):
        self.maximo = maximo
        self._personas = 0      # _personas -> Atributo encapsulado

    @property  # @ -> decorador
    def personas(self):
        return self._personas   # --> Este es nuestro metodo (GETTER): obtenemos␣
 ↪el valor de _personas

    @personas.setter
    def personas(self, p):      # --> Este es nuestro metodo (SETTER):␣
 ↪modificamos el valor de _personas
        if p > self.maximo:
            self._personas = self.maximo
        elif p < 0:
            self._personas = 0
        else:
            self._personas = p
```

0.0.2 Herencia

- Caractreristica importante de OOP, relacion de especializacion y generalizacion entre CLASES
- LA **SUPERCLASE** le hereda atributos, metodos a una **SUBCLASE**

[ ]:
```
# Ejemplo Herencia:

class Auto:
    def __init__(self, marca, modelo, año, color, km):
        self.marca = marca
        self.modelo = modelo
```

```
        self.año = año
        self.color = color
        self._kilometraje = km
    def conducir(self, kms):
        print(f"Conduciendo {kms} kilómetros")
        self._kilometraje += kms

class FurgonEscolar(Auto):
    def __init__(self, marca, modelo, año, color, km):
        Auto.__init__(marca, modelo, año, color, km) # -> No es la mejor manera␣
↪(USAR SUPER !!)
        #super().__init__(marca, modelo, año, color, km)
        self.niños = []

    def inscribir(self, niño):
        self.niños.append(niño)

    """ ---- OVERRIDE DE METODOS: Modificar un metodo heredado ---- """
    def conducir(self, kms):
        print(f"Conduciendo {kms} kilómetros en la furgoneta que hereda de auto␣
↪xd xd xd")
```

[5]:
```
# Ejemplo Multiherencia y problema del diamante:
#    B
# izq der
#    A

# si A es subclase de Izq y de Der, y ambas son una subclase de B, si llamo un␣
↪metodo que comparte ambos
# se llamara 2 veces al metodo, cosa que se puede evitar con una implementacion␣
↪correcta del codigo

class B:
    num_llamadas_B = 0
    def llamar(self):
        print("llamada en B")
        self.num_llamadas_B += 1
class Izq(B):
    num_llamadas_izq = 0
    def llamar(self):
        super().llamar()
        print("llamada en IZQ")
        self.num_llamadas_izq += 1
class Der(B):
    num_llamadas_der = 0
    def llamar(self):
        super().llamar()
```

```
        print("llamada en DER")
        self.num_llamadas_der += 1
class A(Izq, Der):
    num_llamadas_A = 0
    def llamar(self):
        super().llamar()
        print("llamada en A")
        self.num_llamadas_izq += 1

a = A()
a.llamar()
print( A.__mro__ ) # --> MRO: El orden de jerarquía de herencias
```

```
llamada en B
llamada en DER
llamada en IZQ
llamada en A
(<class '__main__.A'>, <class '__main__.Izq'>, <class '__main__.Der'>, <class
'__main__.B'>, <class 'object'>)
```

0.0.3 Polimorfismo

- Propiedad que utliza distintos objetos con la misma interfaz
- **OVERRIDING**: Subclase sobreescribe un metodo de la clase superior
- **OVERLOADING**: Definir un metodo con mismo nombre pero distinto numero de Argumentos (En python solo con Built-ins)

ej : “hola” + “mundo” 3 + 5 [“1”] + [“2”]

operador + funciona dependiendo del tipo de argumento

[1]:
```
# Ejemplo override del metodo __add__

class Carro:
    def __init__(self, pan, leche):
        self.pan = pan
        self.leche = leche

    def __add__(self, otro):
        suma_pan = self.pan + otro.pan
        suma_leche = self.leche + otro.leche
        return Carro(suma_pan, suma_leche)

    def __str__(self) -> str:
        return f"Pan:{self.pan}, Leche:{self.leche}"

carro1 = Carro(1,1)
carro2 = Carro(2,3)
```

```
print( carro1 + carro2 )
```

```
Pan:3, Leche:4
```

str vs **repr**

-> si se implementan ambos se imprime str, si no se printea repr - str : printeo rapido - repr: printeo de info

0.0.4 *Args y **Kwargs

- **kwargs** : keyward arguments -> entrega los argumentos no posicionales,los elementos no se entregan por orden, si no por key (Diccionarios)
- Puede ser usado para entregar cantidades variables de argumentos (SIN ORDEN)
- **args** : mecanismo similar. solo que sin keys, * desempaqueta el contenido de args y da argumentos posicionales, (ORDEN)

[1]:
```
def imprimir(base, *args, **kwargs):
    print(base)
    print(args)
    print(kwargs)

imprimir("hola", "mundo", 1, 2, key="valor" )
```

```
hola
('mundo', 1, 2)
{'key': 'valor'}
```

0.0.5 Clases Abstractas:

- Su intencion es no ser Instanciadas !
- Usamos para modelar otras clases
- Existen los **Metodos Abstractos**: Modelan el comportamiento que deben tener toda clase que herede de ella

[ ]:
```
from abc import ABC, abstractmethod
class Base(ABC):
    @abstractmethod
    def metodo1(self):
        pass
    @abstractmethod
    def metodo2(self):
        pass
class Subclase(Base):
    def metodo1(self):
        print("Primer metodo")
    def metodo1(self):
        print("Primer metodo")
```

```
# SI no se instancian ambos metodos, dara ERROR
```

0.0.6 NamedTuples:

Son como objetos, pero no queremos instanciar metodos, solo atributos

[5]:
```
from collections import namedtuple
Registro = namedtuple("Registro", ["Rut", "name", "age"])
persona = Registro("21014981-3", "Esteban", 21)
print(persona.Rut)
print(persona.name)
print(persona.age)
```

```
21014981-3
Esteban
21
```

1 Diccionarios:

-Estructura de datos no secuencial, del tipo "llave-valor" -Estructura similar a una tabla hash

- Las "key" Tienen que ser HASHEABLES -> Unicas, se le pueden entregar a una funcion hash que retorna un numero unico
- Todos los elementos inmutables de python son hasheables -> Listas/Dict: No / int,str,tuple : Si

[10]:
```
dic = {
    "nombre": "Esteban",
    "edad": 21,
    "universidades": ["USM", "PUC"],
    2002: "Año en que nací"
}
print(dic[2002])

# 1: Get -> Sirve para intentar acceder a una llave, que quizas no existe
print(dic.get("nombre", "no tiene nombre"))
print(dic.get("RUT", "No tiene Rut"))

# 2: del -> Sirve para eliminar keys y values del dict
del dic["edad"]

# 3: in -> comprobar existencia
print( 2002 in dic )
```

```
Año en que nací
Esteban
No tiene Rut
True
```

[11]:
```
# metodos utiles:
print(dic.keys())
print(dic.values())
print(dic.items())
```

```
dict_keys(['nombre', 'universidades', 2002])
dict_values(['Esteban', ['USM', 'PUC'], 'Año en que nací'])
dict_items([('nombre', 'Esteban'), ('universidades', ['USM', 'PUC']), (2002,
'Año en que nací')])
```

1.0.1 Iterables por comprension

[14]:
```
lista = [x for x in range(1,10+1)]
dic   = { str(x): x for x in lista  }
sets = {x for x in range(1,10,2)}
print(lista)
print(dic)
print(sets)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
{'1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10':
10}
{1, 3, 5, 7, 9}
```

1.0.2 Sets:

- No se pueden indexar (set[0] no podria hacerse)
- Pueden recorrerse, no tienen orden en especifico
- No repiten elementos
- Se añaden elementos con add: *set.add(elemento)*
- Se eliminan elementos con remove: *set.remove(elemento)*
- Se eliminan elementos con discard: *set.discard(elemento)* -> No dara error en caso de no encontrarlo
- Union : |
- Interseccion: &
- Diferencia: -
- Diferencia Simetrica: ^

[18]:
```
### Generadores (usan menos espacio que una lista)
generador = (i for i in range(10))
print(generador)
for i in generador:
    print(i)
for i in generador: # Solo se puede iterar 1 vez, la 2da no lo lograra! ->␣
 ↪__iter__
    print(i)

### Funciones generadoras:
```

```python
def conteo_decreciente(n):
    print(f"Contando en forma decreciente desde {n}")
    while n > 0:
        yield n
        n -= 1
```

```
<generator object <genexpr> at 0x110d4a4d0>
0
1
2
3
4
5
6
7
8
9
```

1.0.3 Programacion Funcional:

- Funciones Lambda: forma alternativa de definir una funcion en Python

[ ]:
```python
sucesor = lambda x: x + 1

# Es equivalente a:

def sumar_uno(x):
    return x + 1
```

[23]:
```python
# 1. MAP : Recibe parametros y al menos 1 iterable -> Retorna GENERADOR con␣
 ↪Funcion aplicada a cada elemento del iterable

iterable = ["HOLA", "MUNDO"]
mapeo    = map(lambda x: x.lower(), iterable)
print( mapeo )
print( list(mapeo) )
print( "------" )

# 2. Filter: Recibe una funcion que retorna True o False, y filtra elementos de␣
 ↪un Iterable segun la condicion
iterable = [1, 2, 3, 4, 5]
filtro   = filter(lambda x: x > 3, iterable)
print( filtro )
print( list(filtro) )
print( "------" )

# 3. Reduce: Aplica una funcion a dos elementos, luego lo aplica al resultado,␣
 ↪con el siguiente de la iteracion, asi hasta el final del iterable
```

```
from functools import reduce
iterable = [1, 2, 3, 4, 5]
suma = reduce(lambda x, y: x + y, iterable )
print( suma )
print( "------" )
```

```
<map object at 0x110e3dc60>
['hola', 'mundo']
------
<filter object at 0x110e3ed40>
[4, 5]
------
15
------
```

[27]:
```
# Anexo Funciones de Built-in

# 1. Enumerate: Va enumerando las iteraciones y los valores (tupla)
lista = ["a","b","c","d"]
for indice, elemento in enumerate(lista):
    print(f"{indice}: {elemento}")

# 2. Zip: toma +2 iterables y retorna lista de n° tuplas (cantidad menor de elem␣
 ↪de los iterables)
a = [1,2,3,4,5,6,7,8,9]
b = ["a","b","c","d","e"]
print( list( zip(a, b) ) )
```

```
0: a
1: b
2: c
3: d
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

1.0.4 THREADS

- Es un hilo de ejecucion de codigo
- El programa principal es el **Thread Principal**
- Cada thread lleva su propio registro de VARIABLES locales y en que parte del codigo se encuentra
- Generalmente son funciones que quieren ejecutarse simultaneamente

[34]:
```
import threading

def contar():
    print("Empieza el thread")
    for numero in range(1,6):
        print(numero)
```

```
thread = threading.Thread(target=contar) # TARGET = Entregamos la funcion la␣
 ↪cual queremos ejecutar de fondo
thread.start() # EMPEZAR EL THREAD! no se ejecuta si no se inicia
# thread.start()  -> Dará error pues solo se puede iniciar 1 vez
```

```
Empieza el thread
1
2
3
4
5
```

- join(timeout=None) o join(): el thread que llama al metodo queda bloqueado hasta que termine correctamente
- join(timeout != None) : El programa esperara el tiempo (s) indicado hasta cerrar el thread
- thread.is_alive() : True o False segun si el Thread esta activo o no, nos sirve para cosas de flujo

[ ]:
```
import threading
import time
class Esperar(threading.Thread):
    def __init__(self):
        super().__init__()
    def run(self):
        time.sleep(1)
        print(1)
        print(2)
        print(3)
cuenta1 = Esperar()
cuenta2 = Esperar()
cuenta3 = Esperar()

cuenta1.start()
cuenta2.start()
cuenta3.start()

cuenta1.join()
print("Termino la cuenta 1")
cuenta2.join()
print("Termino la cuenta 2")
cuenta3.join()
print("Termino la cuenta 3")
```

1.0.5 Daemons

- Son aquellos threads que no le impiden al programa principal terminar antes que ellos, se ejecutan de fondo.
- Daemon = True

[ ]:
```
import threading
import time
thread = threading.Thread(target=contar, daemon=True)
```

1.0.6 Concurrencia

- 1. **Seccion critica**: es donde solo un Thread deberia esta accediendo a la vez, se dice que el conjunto de instrucciones debe ser **Atomica**

-> Un ejemplo de esto es alguna seccion donde se actualize un valor general, y dos threads quieran modificar al mismo tiempo, esto podria concurrir en calculos equivocados

- 2. **Mecanismos de Sincronizacion**: Vienen a segurar estas secciones criticas, dejando solo el paso a un Thread

a.- **Lock**: Bloqueado o Desbloqueado -> Cada thread va a requerir tener el lock, para ejecutarse **acquire**: El thread puede pedir el lock, para entrar en dicha seccion critica, BLOQUEANDO el paso (lock) **release**: Una vez el thread termine su ejecucion, puede liberar el lock usando thread.release(), LIBERANDO el paso (aquire)

Cabe destacar, que el Lock debe ser global, no individual de cada thread, todos deben recurrir al mismo para ejecutarse por turnos !!!

b.- **Context Manager**: Realiza por si solo el acquire y el release

[ ]:
```
class Contador():
    def __init__(self) -> None:
        self.valor = 0

import threading

lock_global = threading.Lock()
def funcion_critica(contador, lock):

    for _ in range(10):
        lock.acquire()
        contador.valor += 1
        lock.release()

    # Con context manager:
    #for _ in range(10):
    #    with lock:
    #        contador.valor += 1

cont = Contador()
thread_1 = threading.Thread(target=funcion_critica, args=(cont, lock_global))
thread_2 = threading.Thread(target=funcion_critica, args=(cont, lock_global))
```

1.0.7 Señales entre Threads

1. Event: mecanismo de comunicacion entre Threads, los threads esperan por una señal (flag) True: Activa
    a. wait(): el thread se *Pausa* hasta que otro thread active la señal correspondiente
    b. set() : el thread *Activa* la señal (flag:True)
    c. clear(): el thread *resetea* la señal (flag:False)
    d. is_set(): retorna True o False de la flag, *NO PAUSA* a diferencia de wait()

[35]:
```
import threading
import time
video_cargado = threading.Event()
audio_cargado = threading.Event()
def play_video():
    print("Cargando video")
    time.sleep(10)
    video_cargado.set()
    audio_cargado.wait() # Esperar a que el audio este cargado
    print("Comenzando reproduccion video")

def play_audio():
    print("Cargando audio")
    time.sleep(3)
    audio_cargado.set()
    video_cargado.wait() # Esperar a que el video este cargado
    print("Comenzando reproduccion audio")

thread_1 = threading.Thread(target=play_audio)
thread_2 = threading.Thread(target=play_video)

thread_1.start()
thread_2.start()

thread_1.join()
thread_2.join()
```

```
Cargando audio
Cargando video
Comenzando reproduccion videoComenzando reproduccion audio
```

1.0.8 Deadlocks:

- Se produce cuando por error, dos threads se esperan mutuamente, ninguno avanza
- Es error del programador -> Tratar de evitar

[ ]:
```
import threading
import time
```

```
lock1 = threading.Lock()
lock2 = threading.Lock() # Ya aqui vemos un problema -> varios locks?
def master():
    with lock1:
        with lock2:
            print("Master trabajando")

def worker():
    with lock2:
        with lock1:
            print("Worker Trabajando")

thread_1 = threading.Thread(target=master)
thread_2 = threading.Thread(target=worker)

thread_1.start()
thread_2.start()

# Si se ejecuta -> Solo se estaran consiguiendo locks pero nunca entra a los␣
 ↪prints
```

1.0.9 Serializacion

- Procedimiento de transformar cualquier objeto en *secuencia de bytes*
- Nos permite enviar el objeto a otros programas o computadores

1. **Pickle**: Unico de python, permite guardar casi cualquier objeto solo de *PYTHON*
    a. **dumps**: *serializa* el objeto
    b. **loads**: *deserializa* el objeto, carga el objeto a su esto original
2. **JSON**: formato de texto estandar interpretado por *MUCHOS LENGUAJES* permite serializar: int, str, float, dict, bool, list, tuple y NoneType (NADA DE CLASES Y FUNCIONES) Es leible por humanos, es un diccionario
    a. **dumps**
    b. **loads** Igual que en pickle

[37]:
```
import pickle

tupla = ("1",2,"hola")
tupla_serializada = pickle.dumps(tupla)

print(tupla_serializada)
print(type(tupla_serializada))

tupla_deserealizada = pickle.loads(tupla_serializada)
print(tupla_deserealizada)
```

```
""" Para archivos: usar
    1. dump(objeto, archivo)
    2. load(archivo)
    """

lista = [1,2,3,4,5]

# 1. Escritura: dump
with open("data", "wb") as file:
    pickle.dump(lista, file)

# 2. Lectura: load
with open("data", "rb") as file:
    pickle.load(file)
```

```
b'\x80\x04\x95\x10\x00\x00\x00\x00\x00\x00\x00\x8c\x011\x94K\x02\x8c\x04hola\x94
\x87\x94.'
<class 'bytes'>
----------------
('1', 2, 'hola')
```

[38]:
```
import json

# Convertir un diccionario a JSON
diccionario = {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Ejemplo'}
json_diccionario = json.dumps(diccionario)
print(json_diccionario)

# Convertir una lista a JSON
lista = [1, 2, 3, 4, 5]
json_lista = json.dumps(lista)
print(json_lista)

# Convertir un entero a JSON
numero = 42
json_numero = json.dumps(numero)
print(json_numero)

# Convertir una cadena a JSON
cadena = "Hola, mundo!"
json_cadena = json.dumps(cadena)
print(json_cadena)
```

```
{"nombre": "Juan", "edad": 30, "ciudad": "Ejemplo"}
[1, 2, 3, 4, 5]
42
"Hola, mundo!"
```

1.0.10 Bytes y Encoding

- Cada Byte (8 Bits) puede representar 0 a 255 numeros
- Para representar un caracter, podemos hacerlo con bytes, asociando con Codificacion
- *ASCII* es una codificacion muy conocida, tambien esta *UTF8*
- hex(num) : Transforma num en su representacion hexadecimal
- *UNICODE* Es la mas comun, utiliza 65536 caracteres

1. **Byte**: Es un objeto, de tipo secuencia *INMUTABLE* como un str, se le reconoce con una b"" al inicio ej: = b“636c6963689”
2. **ByteArray**: Lista de Bytes, es del tipo *MUTABLE*, pueden ser indexables, slicing, agregar bytes(extend) agregar int(append)

[44]:
```
caract = b"\x63\x6c\x69\x63\x68\xe9"
frase = caract.decode("latin-1")
# Otra decodificacion podria dar error
#frase_mal = caract.decode("ascii")
print(frase)

frase = "estación"
print(frase.encode("utf-8"))
print(frase.encode("latin-1"))
```

```
cliché
b'estaci\xc3\xb3n'
b'estaci\xf3n'
```

[58]:
```
ba = bytearray(b"hola mundo")
print(ba)
print(ba[0]) # Valor num
print(ba[0:1]) # primer caracter
print(ba[-1:])

ba.extend(b" como estan?")
print(ba)

ba.append(3)
print(ba)
```

```
bytearray(b'hola mundo')
104
bytearray(b'h')
bytearray(b'o')
bytearray(b'hola mundo como estan?')
bytearray(b'hola mundo como estan?\x03')
```

1.0.11 Excepciones

1. SyntaxError : Syntax erronea
2. NameError : No existe variable

3. IdentationError: Error en saltos de linea
4. ZeroDivisionError: obvio XD
5. IndexError: Salirse el largo de un objeto indexable
6. KeyError: Uso invalido de una llave de diccionario
7. AtributeError: Uso incorrecto de metodos o atributos dentro de una clase
8. TypeError: Mezclar 2 tipo de datos distintos
9. ValueError: Manejo erroneo de valor de datos (ej: "Hola Mundo".split(""), no existe string vacio!)

2 Manejo

1. *Try*: intenta ejecutar un codigo, en caso de error, colocar abajo un bloque de:
2. *except* TypeError: aqui manejaremos el error, segun su tipo, *mala practica colocar except Exception*

No se pueden atrapar errores del tipo 1.SyntaxError o 2.IdentationErrror

3. *Raise*: Detiene la ejecucion del programa, y levanta una excepcion con un mensaje determinado

- else y finally

4. *else*: Se ejecutara este bloque de codigo si es que no se capturo excepcion en try
5. *finally*: Se ejecuta siempre, con excepcion o sin (menos con raise que detiene el programa)

2.0.1 Networking

- Cada maquina es un *host*, (maquina que puede ejecutar programas y comunicarse con el exterior)
- Todos los host, reciben una etiqueta llamada *Direccion IP* (id unico)
- Comunicaciones con el host atraves del IP: existen 2 tipos

1. *IPv4* : 4 bytes (32 bits)
2. *IPv6* : 6 bytes (128 bits) -> Muchas mas direcciones

- *Puertos*:
  - Cada programa usa un puerto *distinto* para comunicarse
  - Solo una app puede usar un puerto a la vez
  - El puerto de origen *no tiene que ser igual* al de destino
- **Protocolos de Comunicacion**:

1. **TCP**: - Envia serie de bytes serializados de forma *CONFIABLE*, se asegura de que llegue a salvo - Se consigue mediante funciones, que verifican que no hay alteracion en el mensaje - *handshake* : protocolo de "Establecimiento de conexion" - Esta fiabilidad, nos entrega *LATENCIA*, pero nos aseguramos que lleguen *INTEGRAMENTE* - HTTP, SMTP, BitTorrent usan TCP para no errores

2. **UDP**: - Envia paquetes (datagramas) pero *SIN GARANTIA* de *ENVIO INTEGRO* (NO CONFIABLE) - Ventaja sobre TCP -> Es mas liviano (se *salta handshake*) - *Menor Latencia* - No garantiza la integridad

2.0.2 Sockets

- Objeto del sistema operativo, permite a un programa enviar y recibir datos desde y hacia otra maquina
- family: AF_INET: IPv4
- type: SOCK_STREAM: TCP

[ ]:
```
# 1. Crear socket TCP con IPv4
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 2. Conectar al cliente con un servidor (IP, PUERTO)
sock.connect( ('146.155.123.21', 80) )

# 3. Enviar mensajes (sockets solo transmites bytes)
# - Send(bytes): Trata de enviat mensaje almacenado en los bytes, sabremos␣
 ↪cuantos se alcanzaron a mandar
# - Sendall(bytes): Asegura de enviar todos los bytes del mensaje, en caso de␣
 ↪error no sabremos cuantos se alcanzaron a mandar
```

[1]:
```
""" ---- CLIENTE ---- """

# Ejemplo de conexion:
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = "iic2333.ing.puc.cl"
port = 80

try:
    sock.connect((host, port))
    sock.sendall('GET / HTTP/1.1\nHost: iic2333.ing.puc.cl\n\n'.encode('utf-8'))␣
 ↪# Enviar mensaje al servidor
    data = sock.recv(4096)        # Recibir mensaje del servidor
    print(data.decode('utf-8'))
except ConnectionError as e:
    print("Ocurrió un error.")
finally:
    # ¡No olvidemos cerrar la conexión!
    sock.close()
```

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Thu, 14 Dec 2023 11:54:46 GMT
Content-Type: text/html
Content-Length: 11845
Last-Modified: Wed, 17 Aug 2022 05:04:47 GMT
```

```
Connection: keep-alive
ETag: "62fc76ef-2e45"
Accept-Ranges: bytes

<!DOCTYPE html><html lang="es"><head><!--[if IE]><!--Get a real
browser!--><![endif]--><!--Do NOT read this file, the source is
available--><meta charset="utf-8" /><title>IIC2333 Sistemas Operativos y
Redes</title><meta content="IIC2333 Sistemas Operativos y Redes, 2022-2"
name="description" /><meta content="Cristian Ruz" name="author" /><meta
content="width=device-width, initial-scale=1, shrink-to-fit=no" name="viewport"
/><link href="//cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.0.0-alpha/css/bootstrap.min.css" rel="stylesheet" /><link
rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/highlight.js/9.12.0/styl
es/zenburn.min.css"><script src="../js/highlight.pack.js"></script><script
src="../js/fairyDustCursor.js"></script><link
href="https://cdnjs.cloudflare.com/ajax/libs/KaTeX/0.6.0/katex.min.css"
rel="stylesheet" /><script src="https://cdnjs.cloudflare.com/ajax/libs/KaTeX/0.6
.0/katex.min.js"></script><script
src="https://cdnjs.cloudflare.com/ajax/libs/KaTeX/0.6.0/contrib/auto-
render.min.js"></script><script src="js/countdown.min.js"></script><script>hljs.
initHighlightingOnLoad();</script><style type="text/css">html {
  font-family: sans-serif !important;
  background-color: #FFFFE0;
}
pre code.hljs {
  display: block;
}
code.hljs {
  display: inline;
}

// Hover stuff
code:hover>div {
  display: block;
}
code>div {
  display: none;
}
table {
  margin: auto;
  //border-collapse: separate;
  //border-spacing: 0 0.1em;
}
tr.sep {
  margin-top: 0.2em;
}
td {
  min-width:4em;
```

```
}
td.r2 {
  rowspan: 2;
}
td.full {
  color: #ddd;
  background-color: #282;
}
td.empty {
  color: #ddd;
  background-color: #333;
}

// Fix dl/ dd-dt
dl.inline dd {
  display: inline;
  margin: 0;
}
dl.inline dd:after{
  display: block;
  content: '';
}
dl.inline dt{
  display: inline-block;
  min-width: 100px;
}

section {
  padding-top: 2em;
}
h1, h2, h3, h4, h5, h6 {
  padding-top: 0.4em;
}
.top {
  padding-top: 0.2em;
}
.container {
  display: flex;
  justify-content: center;
}
.container-col {
  display: flex;
  justify-content: center;
  flex-direction: column;
}
.container > img {
  align-content: center;
}
```

```
.container-col > img {
  align-content: center;
}
body {
  font-family: sans-serif !important;
  background-color: #FFFFE0;
  color: #000;

}
.card{
  margin:auto;
  float: left;
  width: 12rem;
  text-align: center;
  margin-left: 5%;
}

.column {
  float: left;
  width: 33.33%;

}

.row {
  margin-top: 2rem;
  margin-bottom: 2rem;

}

.row:after {
  content: "";
  display: table;
  clear: both;

}</style></head><body><a href="/"><header class="col-xs-8 col-md-10 col-md-
offset-1"><br /><a href="https://www.youtube.com/watch?v=FGwevyxoycw"><img
src="images/figures/logo1.gif" width="100%" /></a></header></a><div class="col-
xs-12 col-md-8 col-md-offset-3"><div class="row"><div class="columm"><div
class="card"><div class="card-title"> Sitio Canvas </div><div class="card-
title"><a href="https://cursos.canvas.uc.cl/courses/46706" class="btn btn-
primary btn-sm" role="button">Ir</a> <!-- 2022-2 --></div></div></div><div
class="columm"><div class="card"><div class="card-title"> Planilla de Notas
</div><div class="card-title"><a
href="https://docs.google.com/spreadsheets/d/1TrDycHCSetxSVGx-
lybl1Zz6DA4izGX7_Ct0W6SqXJc/edit?usp=sharing" class="btn btn-primary btn-sm"
role="button">Ir</a></div></div></div></div></div><div class="col-xs-12 col-md-8
col-md-offset-2"><article><section id="slides"><h3> Material de Clases
</h3><ul><li> Videos</li><ul><li><a href="https://www.youtube.com/playlist?li
```

st=PLVTkEjeczcER2aOMHPIfaHCpTx61iKsoV">Playlist Sistemas Operativos
Sistemas Operativos
Procesos
Scheduling<

[ ]:
```
""" --- SERVIDOR --- """
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = socket.gethostname() # Obtener el host
port = 9000                 # Puerto random ojala que no este ocupado

sock.bind( (host, port) )   # Enlazar un socket con el puerto

sock.listen()               # Escuchar conexiones

contador = 0
while contador < 5:                 # Aceptar conexiones
    try:
        socket_cliente, address = sock.accept() # Retorna tupla (IP, PUERTO)
        print("CONEXION ESTABLECIDA")
        socket_cliente.sendall("Gracias por conectarte".encode("utf-8"))
        socket_cliente.close()
        contador += 1
    except ConnectionError:
        print("Error de Conexion")

# Para terminar programa servidor:
sock.close()
```

2.0.3 I/O

1. Archivos:
   - Con *Open* Manejamos la apertura del archivo, al ser una serie de bytes, podemos elegir como codificar (al escribirlo) o decodificar (al leerlo)
   - Parametro "w" escribe y "r" lee el archivo
   - Parametro "a" no sobreescribe, toma el texto y sigue escribiendo al final
   - Podemos en vez de leer contenidos, leer bytes
   - Parametro "wb" y "rb" (write bytes y read bytes)
   - Con *Close* Debemos cerrar el archivo luego de leerlo y/o escribirlo
2. Context Manager:
   - Al igual que con los threads y with lock: podemos arbir y cerrar el archivo de manera rapida
   - Nos ahorramos el open y close
   - Es equivalente a ejecutar *Try y Finally* abriendo y cerrando
   - ejemplo: with open("archivo.txt", "r") as file: conetnido = file.read()
3. Emulacion de archivos I/0:

- Con StringIO y BytesIO se puede emular estar trabajando con archivos, sin tener la necesidad de abrir y leer archivos realmente

2.0.4 Strings

- Son inmutables
- se puede hacer slice, se pueden concatenar
- Sencuencias Escape: (para printear comillas)
  1. " comilla doble
  2. ' comilla simple
- dir(str): Da todos los metodos que tienen los strings
- *f-Strings*:
  - Forma de formatear los stings: f"hola {nombre}, como estas?"
  - Forma de printear varias varibales: print(f""" From: <{from_email}> To: <{to_email}> Subject: {subject} {message} """)
  - str.format() : metodo para ingresar valores a un f-string: ej: frase = "hola {} que tal" print(frase.format("Esteban"))
    -> Se puede realizar con kwargs: "hola {nombre}, {apellido}" frase.format(nombre="Esteban", apellido="Ortega")
- *MEJORAR IMPRESION STRINGS*:
  - Python provee de elementos extra en la formacion de strings, le daran una formacion mas estructurada
  - veamos un ejemplo:

[6]:
```
compra = [('leche', 2, 120), ('pan', 3.5, 800), ('arroz', 1.75, 960)]
print("PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL")

for producto, precio, cantidad in compra:
    subtotal = precio * cantidad
    print( f"{producto:8s} | {cantidad: ^8d} | ${precio: <5.2f} | ${subtotal: >7.
 ↪2f}" )
```

```
PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL
leche    |   120    | $2.00  | $ 240.00
pan      |   800    | $3.50  | $2800.00
arroz    |   960    | $1.75  | $1680.00
```

- *Caracteres especiales*:

1. *:8s* - 8 Caracteres, si el str es mas corto -> rellena de espacios / si es mas largo -> No lo trunca
2. *: ^8d* - Espacios vacios tienen que ser llenado (int rellena con 0s) / ^: centrado / 8d: 8 n°decimal
3. *: <5.2f* - Alineamiento a la izquierda / float de hasta 5 caraceteres, 2 decimales

4. *: >7.2f* - Alineamiento a la derecha / float de hasta 7 caracteres, 2 decimales

2.0.5 Regex

- Son secuencias especiales de caracteres que nos permiten comparar y *buscar strings* o conjuntos de strings
- Se definen como un patron y se describen con una sintaxis especial
- Las expreciones permiten especificar un conjunto de string que hacen *match* con ella
- **Meta-Caracteres**:

1. **[abc]**: hace match con cada uno de los string a, b, c
   - *"-"*: Encuentra todos los elem dentro de una clase de caracteres
   - ej: [0-9]: encuentra 0,1,2,3,4,5,6,7,8 o 9
   - Alphanumeric? : *[a-zA-Z0-9]*
   - *[^]* : Negacion! -> hara match con string que no tengan
   - ej: [^abc]: match con cualquiera que no tenga "a" "b" "c"
2. +: **1 o mas veces** aparece el caracter
   - ej: ab+c -> match con "abc", "abbc", "abbbc", etc (NO CON "ac")
   - ej: a[bc]+d -match con "abd", "acd", "abbd", "abcd", "acbd", etc (pero NO CON "ad")
3. **"*": 0 o mas** veces aparece el caracter
   - ej: ab*c -> match con "abc", "abbc", "abbbc", etc (SI CON "ac")
   - ej: a[bc]*d -match con "abd", "acd", "abbd", "abcd", "acbd", etc (SI CON "ad")
4. **?**: Puede estar **1 vez o 0** (No mas)
   - ej: ab?c -> match con "abc" y "ac"
5. **{m, n}**: *Repetirse* entre *m y n veces*
   - ej: ab{3,5}c -> match con "abbbc", "abbbbc", "abbbbbc"
   - ej: ab{2}c -> match con "abbc"
6. **.**: permite match con **cualquier** caracter, EXCEPTO un *SAlto de Linea*
   - ej: .+ -> match con cualquier str largo mayor a 1
7. ^: permite especificar *primeros caracteres*
8. **$**: permite especificar *ultimos caracteres*
9. **()**: permite especificar *grupos*
   - ej: a(bc)*(de)f -> hace match con : "adef", "abcdef", "abcbcdef", "abcbcbcdef"
10. |: Permite encontrar 2 opciones de match:
    - ej: ab+c|de*f -> hace match con: "abc", "abbc", "abbbc", . . . , "df", "def", "deef", "deeef"
11. **""**: Permite transformar los metacaracteres en expresiones de texto, y no como metacaracteres
    - ej: ab?"> "ab?c" (y no abc o ac)

Modulo Re Python: - **re.match()**: Verifica si la expresión regular hace match en el principio del string - re.matchall(): Verifica si la expresión hace match con todo el string - **re.sub()**: Reemplaza la parte del string con la que hizo match con el otro parámetro - **re.split()**: Retorna una lista y sus elementos son los caracteres del string separados por cada vez que hizo match con la expresión regular que le pasamos - *re.search()*: Entrega el primer match que hace con un string - **re.findall()**: Entrega todos los matchs que hizo en un string - re.finditer(): Lo mismo que findall pero en vez de entregar una lista, entrega un iterador

2.0.6 WebServices

1. **API**: Conjunto de funciones expuestas por un servicio, para ser utilizadas por programas - Generalmente, son servicios que estan en la web (Webservices) - Podemos conectarnos a ella con aplicaciones
2. **HTTP**: Gran parte de arquitecturas de webservices se basan en el protocolo HTTP - Trabaja como protocolo *request-response* -> cliente pide solicitudes y servidor responde - Se incluyen 5 metodos: 1. **GET**: obtener recurso sin cambiar nada 2. **POST**: crear un recurso 3. **PATCH**: modificar parcialmente un recurso 4. **PUT**: reemplazar completamente un recurso 5. **DELETE**: eliminar un recurso *RESPUESTAS*: 1. 200 : OK, solicitud exitosia 2. 403 : Prohibido, solicitud aceptada pero el servidor rechaza 3. 404 : No encontrado, el recurso no fue encontrado 4. 500 : Error interno del servidor

   Tipos de mensajes Http:

   1. **MENSAJE REQUEST**: solitcitud:GET/recursos , header:. . . , body:. . . .
   2. **MENSAJE RESPUESTA**: Status: 200, header:. . . , body:. . . .

[8]:
```
import requests
import json
url = 'https://api.github.com/repos/IIC2233/Syllabus/issues/8'
response = requests.get(url)
print(f"Status: {response.status_code}")
#print(response.json()["body"]) -> Imprimira el contenido en body de la respuesta
```

```
Status: 200
```

[11]:
```
# Endpoints: son diferentes rutas que dispone una API para hacer consultas
import requests
Base = "https://fakerapi.it/api/v1/"
endpoint = "books/"
solicitud = requests.get(Base + endpoint)
print(solicitud.status_code)

data = solicitud.json()
print(f"LLAVES: {data.keys()}")
print(data["data"][0])
```

```
200
LLAVES: dict_keys(['status', 'code', 'total', 'data'])
{'id': 1, 'title': 'Alice kept her.', 'author': 'Noel Rogahn', 'genre':
'Molestiae', 'description': 'KNOW IT TO BE TRUE--" that\'s the queerest thing
```

```
about it.\' (The jury all looked so grave that she ought to have it explained,\'
said the Caterpillar. \'Not QUITE right, I\'m afraid,\' said Alice, in a.',
'isbn': '9795370401144', 'image': 'http://placeimg.com/480/640/any',
'published': '2011-05-08', 'publisher': 'Dolore Non'}
```

[ ]:
```python
# Uso de Post
import json

github_repo = 'juanito-iic2233-20XX-1'
token = "COMPLETAR"

body = {
    'title': "Creando una issue con la API",
    'body': "Ahora tengo el poder para hacer issues desde Python! "
}

my_headers = {
    'Authorization': "MI-TOKEN"
}

url = f"https://api.github.com/repos/IIC2233/{github_repo}/issues"

response = requests.post(url, data=json.dumps(body), headers=my_headers) #␣
 ↪github pide data atravez de json
response.status_code
```

2.0.7 Estructuras de datos

1. Listas ligadas: Almacenan nodos con orden secuencial (listas, stacks, colas)

- Cada nodo tiene un sucesor
- El primer nodo se denomina CABEZA (HEAD)
- El ultimo (no posee sucesor) se denomina nodo COLA (TAIL)

[12]:
```python
class Nodo:
    # Representaremos un nodo dentro de una lista ligada
    def __init__(self, valor=None):
        self.valor = valor
        self.siguiente = None

class ListaLigada:
    def __init__(self):
        self.cabeza = None
        self.cola = None

    def agregar(self, valor):
        # Agrega nodo al final de la cola
        nuevo = Nodo(valor)
```

```
        # Lista vacia (No hay cabeza)
        if self.cabeza is None:
            # El nuevo nodo es CABEZA Y COLA de la lista
            self.cabeza = nuevo
            self.cola = self.cabeza

        else:
            # Agregamos al nuevo nodo como sucesor del actual
            # Si llega aqui, es porque self.cola ya tiene un nodo, por eso se le␣
↪puede poner .siguiente
            self.cola.siguiente = nuevo
            # Actualizamos la referencia
            self.cola = self.cola.siguiente

    def obtener(self, posicion):
        # Buscar el valor del nodo en la posicion
        # Empezar por la cabeza
        nodo_actual = self.cabeza

        for _ in range(posicion):
            # Revisamos no haber llegado al final de la lista
            if nodo_actual is not None:
                nodo_actual = nodo_actual.siguiente
        if nodo_actual is None:
            # En caso de buscar una posicion mayor al largo de la lista
            return None
        else:
            return nodo_actual.valor

    def insertar(self, valor, posicion):
        nodo_nuevo = Nodo(valor)
        nodo_actual = self.cabeza

        if posicion == 0:
            nodo_nuevo.siguiente = self.cabeza
            self.cabeza = nodo_nuevo
            if nodo_nuevo.siguiente is None: # Caso en que la lista era vacia
                self.cola = nodo_nuevo
            return

        for _ in range(posicion - 1): # Encontrar Predecesor
            if nodo_actual is None:
                nodo_actual = nodo_actual.siguiente

        # Caso: No encontrar predecesor
        if nodo_actual is not None:
            nodo_nuevo.siguiente = nodo_actual.siguiente
```

```
            nodo_actual.siguiente = nodo_nuevo
            # Caso: Insertar en la ultima posicion
            if nodo_nuevo.siguiente is None:
                self.cola = nodo_nuevo

    def __repr__(self):
        # Representacion de la lista
        string = ""
        nodo_actual = self.cabeza
        while nodo_actual is not None:
            string = f"{string}{nodo_actual.valor} → "
            nodo_actual = nodo_actual.siguiente
        return string


lista = ListaLigada()
print(lista)
lista.agregar(3)
print(lista)
lista.agregar(4)
print(lista)
lista.agregar(5)
print(lista)

print(f"Posicion 2: {lista.obtener(2)}")
print(f"Posicion 1: {lista.obtener(1)}")

lista.insertar("zero",1)
print(lista)
```

```
3 →
3 → 4 →
3 → 4 → 5 →
Posicion 2: 5
Posicion 1: 4
3 → zero → 4 → 5 →
```

2. Grafos:

- Cada nodo es un Vertice
- Cada relacion entre nodos es una Arista

Existen los grafos NO DIRIGIDOS y los DIRIGIDOS

1. NO DIRIGIDOS: Son relaciones simetricas (a relacionado con b ssi b con a)
2. DIRIGIDOS: No necesariamente simetricas, a -> b no necesariamente a <- b

3 Formas de representar un grafo:

1. Nodos

2. Listas de Adyacencia
3. Matrices de Adyacencia

[14]:
```python
# Forma 1: Con Nodos

class Nodo:

    def __init__(self, valor):
        self.valor = valor
        self.vecinos = []

    def agregar_vecino(self, nodo):
        self.vecinos.append(nodo)

    def __repr__(self) -> str:
        texto = f"[{self.valor}]"
        if len(self.vecinos) > 0:
            textos_vecinos = [f"[{vecino.valor}]" for vecino in self.vecinos]
            texto += " -> " + ", ".join(textos_vecinos)
        return texto



# GRAFO DIRIGIDO:

nodo_1 = Nodo(1)
nodo_2 = Nodo(2)
nodo_3 = Nodo(3)
nodo_4 = Nodo(4)
nodo_5 = Nodo(5)

nodo_1.agregar_vecino(nodo_2)
nodo_2.agregar_vecino(nodo_3)
nodo_3.agregar_vecino(nodo_2)
nodo_3.agregar_vecino(nodo_4)
nodo_3.agregar_vecino(nodo_5)
nodo_4.agregar_vecino(nodo_5)

print(nodo_1)
print(nodo_2)
print(nodo_3)
print(nodo_4)
print(nodo_5)

# Forma 2: Lista Adyacencia
# Aquí usamos diccionarios con llave: int y valor: list porque ofrece más␣
 ↪facilidad de búsqueda.
```

```python
# Cada llave del diccionario es el valor de un vértice, y el valor asociado es␣
 ↪la lista de vértices con conexión.

grafo_no_dirigido = {1: [2],
                     2: [1, 3],
                     3: [2, 4, 5],
                     4: [3, 5],
                     5: [3, 4]
                    }

grafo_dirigido = {1: [2],
                  2: [3],
                  3: [2, 4, 5],
                  4: [5],
                  5: []
                 }


# Froma 3: Matriz de Adyacencia

# Dimension nxn, n numero de vertices o nodos
# Filas: vertices de origen
# Columnas: vertices de llegada

grafo_no_dirigido = [[0, 1, 0, 0, 0],
                     [1, 0, 1, 0, 0],
                     [0, 1, 0, 1, 1],
                     [0, 0, 1, 0, 1],
                     [0, 0, 1, 1, 0]
                    ]

grafo_dirigido    = [[0, 1, 0, 0, 0],
                     [0, 0, 1, 0, 0],
                     [0, 1, 0, 1, 1],
                     [0, 0, 0, 0, 1],
                     [0, 0, 0, 0, 0]
                    ]

# ¿Están relacionados en el grafo dirigido el vértice 4 y 1?
print( 1 == grafo_dirigido[4 - 1][1 - 1] ) # Se le resta 1 porque los índices␣
 ↪van desde 0 hasta N -1
```

```
[1] -> [2]
[2] -> [3]
[3] -> [2], [4], [5]
[4] -> [5]
[5]
False
```

3. Stacks: es literal, una pila de cosas que se van amontonando

Operaciones basicas:

1. Push: Agregar un elemento en el tope del stack
2. Pop: Eliminar un elemento del tope, sacara el ultimo agregado

3. Peek: Mostrar el elemento del tope sin sacarlo

(LIFO): last in, first out (ultimo en entrar, primero en salir)

- En python: Una stack es una lista

stack = [] stack.append(tope) -> Añade un elem al tope stack.pop() -> Elim un elem del tope stack[-1] -> peek: retorna el elem del tope sin sacarlo len(stack) len(stack) == 0 -> is_empty

[15]:
```python
# Ejemplo 1: uso en la vida real:

class Navegador:

    def __init__(self, current_url='https://www.google.com'):
        self.__urls_stack = []
        self.__current_url = current_url

    def __cargar_url(self, url):
        self.__current_url = url
        print(f"Cargando URL: {url}")

    def ir(self, url):
        self.__urls_stack.append(self.__current_url)
        print('Ir ->', end=' ')
        self.__cargar_url(url)

    def volver(self):
        last_url = self.__urls_stack.pop()
        print('Back->', end=' ')
        self.__cargar_url(last_url)

    def mostrar_pagina_actual(self):
        print(f"Página actual: {self.__current_url}")


browser = Navegador()
browser.ir('http://www.uc.cl')
browser.ir('http://www.uc.cl/es/programas-de-estudio')
browser.ir('http://www.uc.cl/es/doctorado')

browser.mostrar_pagina_actual()
browser.volver()
browser.mostrar_pagina_actual()
```

```
browser.ir('https://stackoverflow.com/')
browser.ir('https://github.com/IIC2233/contenidos')
browser.volver()
browser.mostrar_pagina_actual()
```

```
Ir -> Cargando URL: http://www.uc.cl
Ir -> Cargando URL: http://www.uc.cl/es/programas-de-estudio
Ir -> Cargando URL: http://www.uc.cl/es/doctorado
Página actual: http://www.uc.cl/es/doctorado
Back-> Cargando URL: http://www.uc.cl/es/programas-de-estudio
Página actual: http://www.uc.cl/es/programas-de-estudio
Ir -> Cargando URL: https://stackoverflow.com/
Ir -> Cargando URL: https://github.com/IIC2233/contenidos
Back-> Cargando URL: https://stackoverflow.com/
Página actual: https://stackoverflow.com/
```

4. Colas: Estructura de datos secuencial, mantiene orden de acuerdo a llegada

(Fifo) : First in, First Out -> La primera en entrar es la que sale

Operaciones basicas:

1. Enqueue: Agrega un elemento al final de la cola
2. Dequeue: Sacar ek elemento del inicio de la cola (el que lleva mas tiempo)

3. Peek: permite ver el primer elemento en la cola

En python:

from collections import deque cola = deque() cola = deque(lista) -> convertir list a una queue cola.append(elemento) -> agrega al final de la cola cola.popleft() -> Retorna y extrae el elemto del principio cola[0] -> peek: retorna el primer elemento sin sacarlo

- COMPARACION CON LISTAS

LISTAS MAS RAPIDAS: para buscar elementos

DEQUE MAS RAPIDAS: para extraer elementos (mucha diferencia)

[16]:
```
from collections import deque
cola = deque()
cola.append(1)
cola.append(2)
cola.append(3)
#print(cola)

primero = cola.popleft()
#print(primero)
#print(cola)

"""
DEQUE: Colas de doble extremo, podemos sacar elementos de ambos lados
```

```
"""

cola = deque()
cola.append(1)
cola.appendleft("por la izquierda")
cola.append("por la derecha")
print(cola)
cola.popleft()
cola.pop()
print(cola)

cola.clear()
```

```
deque(['por la izquierda', 1, 'por la derecha'])
deque([1])
```

5. Recorridos Conocidos (Algoritmos)

A. **BFS**: Breadth-First Search (breath: AMPLITUD) recorre por nivel

1.Toma un nodo, sacandolo de la cola, y metiendolo a un stack

2.Coloca a sus vecinos dentro de la cola

3.Saca al nodo vecino, y lo añade al stack, preguntando si no lo visito antes

4.Repite el proceso

[17]:
```
from collections import deque

class Persona:

    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __repr__(self):
        return self.nombre


class Nodo:

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

    def __repr__(self):
        return repr(self.valor)
```

```
class Grafo:

    def __init__(self, lista_adyacencia=None):
        self.lista_adyacencia = lista_adyacencia or dict()

    def adyacentes(self, x, y):
        return y in self.lista_adyacencia[x]

    def vecinos(self, x):
        return self.lista_adyacencia[x]

    def agregar_vertice(self, x):
        self.lista_adyacencia[x] = set()

    def remover_vertice(self, x):
        self.lista_adyacencia.pop(x, None)
        for k, v in self.lista_adyacencia.items():
            if x in v:
                v.remove(x)

    def agregar_arista(self, x, y):
        if x in self.lista_adyacencia:
            self.lista_adyacencia[x].add(y)

    def remover_arista(self, x, y):
        vecinos_x = self.lista_adyacencia.get(x, set())
        if y in vecinos_x:
            vecinos_x.remove(y)

    def __repr__(self):
        texto_nodos = []
        for nodo, vecinos in self.lista_adyacencia.items():
            texto_nodos.append(f"Amigos de {nodo}: {vecinos}.")
        return "\n".join(texto_nodos)

coco = Nodo(Persona("Coco", 15))
thor = Nodo(Persona("Thor", 20))
luna = Nodo(Persona("Luna", 21))
kira = Nodo(Persona("Kira", 20))
bon = Nodo(Persona("Bon", 20))
tomas = Nodo(Persona("Tomás", 10))
anya = Nodo(Persona("Anya", 22))

# Definimos las amistades.
amistades = {
    coco: set([thor, luna, kira, bon, tomas, anya]),
    thor: set([kira, tomas, anya]),
```

```
    luna: set([thor, bon, anya]),
    bon: set([luna, tomas, anya]),
    kira: set([thor, luna, bon, anya]),
    anya: set([thor, luna, kira, bon]),
    tomas: set([bon, coco])
}

grafo = Grafo(amistades)
print(grafo)
```

```
Amigos de Coco: {Luna, Kira, Bon, Thor, Tomás, Anya}.
Amigos de Thor: {Tomás, Anya, Kira}.
Amigos de Luna: {Bon, Anya, Thor}.
Amigos de Bon: {Anya, Luna, Tomás}.
Amigos de Kira: {Bon, Luna, Anya, Thor}.
Amigos de Anya: {Bon, Luna, Thor, Kira}.
Amigos de Tomás: {Bon, Coco}.
```

[18]:
```
# 1: BFS !!!!!

def bfs(grafo, inicio):
    visitados = [] # Stack de los visitados
    queue = deque([inicio])

    while len(queue) > 0:
        vertice = queue.popleft() # Sacamos al primero (izquierda)

        # Saltarse en caso de ya haberlo visitado
        if vertice in visitados:
            continue

        print(vertice)
        visitados.append(vertice)
        # Agregar vecinos a la cola que no hayan sido visitados antes
        for vecino in grafo[vertice]:
            if vecino not in visitados:
                queue.append(vecino)
    return visitados
print("--- BFS ---")
print( bfs(amistades, bon) )

# Notar que en este grafo se alcanza a llegar a todas partes desde el nodo␣
 ↪inicial
# -> GRAFO CONEXO
```

```
--- BFS ---
Bon
Anya
```

```
Luna
Tomás
Thor
Kira
Coco
[Bon, Anya, Luna, Tomás, Thor, Kira, Coco]
```

B. **DFS**: Depth-First Search (depth: PROFUNDIDAD) de cada uno de los vecinos

-> Se realiza igual que BFS, solo que utiliza stack en vez de queue !! -> Esa diferencia provoca un cambio en el recorrido

[19]:
```python
def dfs(grafo, inicio):
    visitados = set() # Stack de los visitados
    stack = [inicio]

    while len(stack) > 0:
        vertice = stack.pop() # Sacamos al primero (izquierda)

        if vertice in visitados:
            continue

        print(vertice)
        visitados.add(vertice)
        # Agregar vecinos a la cola que no hayan sido visitados antes
        for vecino in grafo[vertice]:
            if vecino not in visitados:
                stack.append(vecino)

    return list(visitados)

print("--- DFS ---")
print( dfs(amistades, bon) )
```

```
--- DFS ---
Bon
Tomás
Coco
Anya
Kira
Thor
Luna
[Luna, Kira, Bon, Thor, Tomás, Coco, Anya]
```

[20]:
```python
# 3 DFS: RECURSIVO !!!

def dfs_recursivo(grafo, vertice, visitados=None):
    visitados = visitados or set()
```

```
    # Lo visitamos
    print(vertice)
    visitados.add(vertice)

    for vecino in grafo[vertice]:
        # Detalle clave: si ya visitamos el nodo, ¡no hacemos nada!
        if vecino not in visitados:
            dfs_recursivo(grafo, vecino, visitados)

    return list(visitados)

print( dfs_recursivo(amistades, bon) )
```

```
Bon
Anya
Luna
Thor
Tomás
Coco
Kira
[Luna, Kira, Bon, Thor, Tomás, Coco, Anya]
```