😎 Clases y objetos en Python

- **Una clase es una plantilla para la creación de objetos según un modelo definido previamente. Las clases se utilizan para la definición de atributos (variables) y métodos (funciones).**
- **Un objeto sería una instancia de esa clase, es decir, un objeto sería la llamada a una clase.**

Por ejemplo cuando se importa el módulo random de Python. Este módulo en sí es una clase, al llamar a dicha clase estamos creando una instancia (objeto) de dicha clase.

Cuando llamamos al método randint() realmente estamos llamando al método randint de la clase random a través de un objeto creado para dicho fin.

Viendo esta relación, podemos llegar fácilmente a la conclusión de que para crear un objeto, debemos crear previamente la clase, la cual queremos instanciar.

Definición de una clase

Para crear una clase hay que hacer uso de la palabra class a continuación el nombre que se le asigne y por último dos puntos (:). Ej:

class Restaurante:

El código debe estar indentado, en caso contrario se producirá un error. Después del nombre de la clase no lleva paréntesis, a menos que quisiéramos que herede características de otra clase.

Atributos de una clase

Dentro de la clase podemos definir atributos o propiedades de la clase . Ej:

```
class Restaurante:
    nombre = "Mi Restaurante"
    cuit = "30-12345678-9"
    categoria = 4
    concepto = "Temático"
```

- **Para asignar atributos sólo debemos declarar variables dentro de la clase, estas serían las características principales.**
- **No hay límites en cuanto a los atributos.**
- **Cuando definimos atributos tenemos que estar pendientes de:**

    1. **Asignarle siempre un valor ya que en caso contrario el interprete disparará una excepción.**
    2. **Los nombres de los atributos deberán ser lo más sencillos y descriptivos posibles.**

Instanciar una clase - Objeto de la clase

Una clase no se puede manipular directamente, es por eso que se debe instanciar un objeto de la clase para así modificar los atributos que posea. Para instanciar lo único que debemos hacer es asignarle a una variable el nombre de la clase seguido de paréntesis.

rest_1= Restaurante()

Crear una instancia de la clase y emitir atributos

In [55]:
```
class Restaurante:
    nombre = "Mi Restaurante"
    cuit = "30-12345678-9"
    categoria = 4
    concepto = "Temático"

rest1 = Restaurante()
print(rest1.nombre)
print(rest1.cuit)
print(rest1.categoria)
print(rest1.concepto)
print(f"El restaurante se llama '{rest1.nombre}' su cuit es '{rest1.cuit}' de categoria '{rest1.categoria}' y\
 el concepto es '{rest1.concepto}'")
```

```
Mi Restaurante
30-12345678-9
4
Temático
El restaurante se llama 'Mi Restaurante' su cuit es '30-12345678-9' de categoria '4' y el concepto es 'Temático'
```

Emitimos los valores de los atributos de la clase

Modificar atributos

Si podemos referenciar un atributo como **rest1.nombre** , entonces podemos tratar los atributos como si fueran variables y las podemos modificar.

In [56]:
```
class Restaurante:

    nombre = "Mi Restaurante"
    cuit = "30-12345678-9"
    categoria = 4
    concepto = "Temático"

rest1 = Restaurante()
print(rest1.nombre)
```

```
Mi Restaurante
```

In [57]:
```
rest1.nombre = "Rest_1"
rest1.cuit = "30-11111111-8"
rest1.categoria = 3
rest1.concepto = "Comida rápida"
print(f"El restaurante se llama '{rest1.nombre}' su cuit es '{rest1.cuit}' de categoria '{rest1.categoria}' y\
 el concepto es '{rest1.concepto}'")
```

```
El restaurante se llama 'Rest_1' su cuit es '30-11111111-8' de categoria '3' y el concepto es 'Comida rápida'
```

Métodos

In [58]:
```
class Restaurante:

    '''método con parámetros'''

    def agregar_restaurante(self, nombre, cuit, categoria, concepto):
        print('Agregando restaurante....')
        self.nombre = nombre
        self.cuit = cuit
        self.categoria = categoria
        self.concepto = concepto

    '''método'''

    def mostrar_info(self):
        print('Emitiendo info de restaurante....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.categoria}')
        print(f'Concepto: {self.concepto}')

rest1 = Restaurante()
rest1.agregar_restaurante('Rest_1','30-11111111-8',3,'Comida rápida')
```

```
Agregando restaurante....
```

In [59]:
```
rest1.mostrar_info()
```

```
Emitiendo info de restaurante....
Nombre: Rest_1
Cuit: 30-11111111-8
Categoría: 3
Concepto: Comida rápida
```

In [60]:
```
rest2 = Restaurante()
rest2.agregar_restaurante('Rest_2', '30-22222222-7',2,'Para llevar')
rest2.mostrar_info()
```

```
Agregando restaurante....
Emitiendo info de restaurante....
Nombre: Rest_2
Cuit: 30-22222222-7
Categoría: 2
Concepto: Para llevar
```

Para que el método funcione dentro de una clase debe cumplir con:

- **Extra indentado: todo bloque debe esta extraindentado dentro de la clase para que el interprete de Python lo entienda.**
- **Siempre debe poseer un argumento self para que cuando sea invocado, Python le pase el objeto instanciado y así pueda operar con los valores actuales de la instancia.**
- **Si no se incluye el self, Python emitirá una excepción.**

Métodos constructor de la clase: __init__

def __init__ es la definición de una función como cualquier otra.

El nombre **init** , Python lo reserva para los métodos constructores.

- **Un método constructor de una clase se ejecuta automáticamente cuando se crea un objeto. El objetivo es inicializar los atributos de un objeto.**
- **Es imposible olvidarse de llamarlo porque se llama automáticamente.**
- **Se ejecuta inmediatamente después de la creación del objeto.**
- **No puede retornar datos.**
- **Puede recibir parámetros normalmente para inicializar los atributos.**
- **Es opcional, de todos modos es común declararlo.**
- **Se escribe con dos guiones bajos, la palabra init y a continuación otros dos guiones bajos.**

Abstracción y constructores, se refiere a qué datos son necesarios en la clase y los objetos:

In [61]:
```
class Restaurante:

    def __init__ (self, nombre, cuit, categoria, concepto):
        print('Agregando restaurante....')
        self.nombre = nombre
        self.cuit = cuit
        self.categoria = categoria
        self.concepto = concepto

    def mostrar_info(self):
        print('Emitiendo info de restaurante....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.categoria}')
        print(f'Concepto: {self.concepto}')

rest1 = Restaurante('Rest_1','30-11111111-8',3,'Comida rápida')
'''
Al implementar el constructor pasándole los parámetros inicializamos
el objeto con los datos en el momento de crearlo.
'''
rest1.mostrar_info()
```

```
Agregando restaurante....
Emitiendo info de restaurante....
Nombre: Rest_1
Cuit: 30-11111111-8
Categoría: 3
Concepto: Comida rápida
```

In [62]:
```
rest2 = Restaurante('Rest_2','30-22222222-7',2,'Para llevar')
'''
Al implementar el constructor pasándole los parámetros inicializamos
el objeto con los datos en el momento de crearlo.
'''
rest2.mostrar_info()
```

```
Agregando restaurante....
Emitiendo info de restaurante....
Nombre: Rest_2
Cuit: 30-22222222-7
Categoría: 2
Concepto: Para llevar
```

In [63]:

```
class Restaurante:

    def __init__(self):
        self.nombre = ''
        self.cuit = ''
        self.categoria = 0
        self.concepto = ''

    def agregar_restaurante(self):
        print('Alta restaurante....')
        self.nombre = input("Ingrese el nombre del restaurante\n")
        self.cuit = input("Ingrese el número de cuit\n")
        self.categoria = int(input("Ingrese la categoría (1 a 5)\n"))
        self.concepto = input(
        "Ingrese el concepto:\nGourmet, Especialidad, Familiar, Buffet,\nComida rápida, Buffet, Para llevar\n")

    def mostrar_info(self):
        print('Información del restaurante....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.categoria}')
        print(f'Concepto: {self.concepto}')

rest1 = Restaurante()
rest1.agregar_restaurante()

'''
Tenemos al constructor creado, entonces agregamos el método agregar_restaurante
y lo invocamos inmediatamente después de crear el objeto.
'''
rest1.mostrar_info()
```

```
Alta restaurante....
Ingrese el nombre del restaurante
Mi Resto
Ingrese el número de cuit
30-12345678-5
Ingrese la categoría (1 a 5)
4
Ingrese el concepto:
Gourmet, Especialidad, Familiar, Buffet,
Comida rápida, Buffet, Para llevar
Buffet
Información del restaurante....
Nombre: Mi Resto
Cuit: 30-12345678-5
Categoría: 4
Concepto: Buffet
```

Llamado de métodos desde otro método de la misma clase

In [64]:

```
class Restaurante:

    def __init__(self):
        self.nombre = ''
        self.cuit = ''
        self.categoria = 0
        self.concepto = ''

    def agregar_restaurante(self):
        print('Alta restaurante....')
        self.nombre = input("Ingrese el nombre del restaurante\n")
        self.cuit = input("Ingrese el número de cuit\n")
        self.categoria = int(input("Ingrese la categoría (1 a 5)\n"))
        self.concepto = input("Ingrese el concepto:\nGourmet, Especialidad, Familiar, Buffet,\nComida rápida, \
                              Buffet, Para llevar\n")

    def mostrar_info(self):
        print('Información del restaurante....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.categoria}')
        print(f'Concepto: {self.concepto}')

    def main(self):
        self.mostrar_info()
        '''
        Es importante saber que, llamar un método desde otro método,
        solo se puede hacer dentro de la misma clase.
        '''

rest1 = Restaurante()
rest1.agregar_restaurante()
rest1.main()
```

```
Alta restaurante....
Ingrese el nombre del restaurante
Mi Resto
Ingrese el número de cuit
30-12345678-7
Ingrese la categoría (1 a 5)
4
Ingrese el concepto:
Gourmet, Especialidad, Familiar, Buffet,
Comida rápida,                                  Buffet, Para llevar
Buffet
Información del restaurante....
Nombre: Mi Resto
Cuit: 30-12345678-7
Categoría: 4
Concepto: Buffet
```

Colaboración entre clases

In [65]:

```
class Cliente:

    def __init__(self,nombre):
        self.nombre=nombre
        self.monto=0

    def factura(self, monto):
        self.monto = self.monto + monto

    def impuesto(self):
        self.monto = self.monto + (self.monto * 0.21)

    def retornar_monto(self):
        return self.monto

    def imprimir(self):
        print(self.nombre,": El importe a abonar es de $",self.monto)

class Restaurante:
    def __init__(self):
        self.nombre = ''
        self.cuit = ''
        self.categoria = 0
        self.concepto = ''

    def agregar_restaurante(self):
        print('Alta restaurante....')
        self.nombre = input("Ingrese el nombre del restaurante\n")
        self.cuit = input("Ingrese el número de cuit\n")
        self.categoria = int(input("Ingrese la categoría (1 a 5)\n"))
        self.concepto = input(
        "Ingrese el concepto:\nGourmet, Especialidad, Familiar,Buffet,\nComida rápida, Buffet, Para llevar\n")

    def mostrar_info(self):
        print('Información del restaurante....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.categoria}')
        print(f'Concepto: {self.concepto}')

    def un_cliente(self):
        '''
        La clase Cliente colabora con la clase Restaurante
        '''
        self.cliente1=Cliente("Pepe")
        self.cliente1.factura(1000)
        self.cliente1.impuesto()
        self.cliente1.retornar_monto()
        self.cliente1.imprimir()

    def main(self):
        self.mostrar_info()
        self.un_cliente()

rest1 = Restaurante()
rest1.agregar_restaurante()
rest1.main()
```

```
Alta restaurante....
Ingrese el nombre del restaurante
El Resto
Ingrese el número de cuit
30-12345678-9
Ingrese la categoría (1 a 5)
4
Ingrese el concepto:
Gourmet, Especialidad, Familiar,Buffet,
Comida rápida, Buffet, Para llevar
Buffet
Información del restaurante....
Nombre: El Resto
Cuit: 30-12345678-9
Categoría: 4
Concepto: Buffet
Pepe : El importe a abonar es de $ 1210.0
```

Encapsulamiento

In [68]:
```
class Restaurante:
    def __init__(self,nombre,cuit,categoria,concepto):
        self.nombre = nombre #Default Public
        self.cuit = cuit #Default Public
        self.__categoria = categoria #Private
        self._concepto = concepto #Protected
        '''
        Default Public: quiere decir que se puede modificar en cualquier lugar de la aplicación.
        Protected: un guión bajo dice que está protegido de cambios, sólo puede ser accesible desde una clase deri
        Private: doble guión bajo, sólo es accesible dentro de la clase, no se puede modificar, sólo por algún mét
        getters y setters. También se pueden encapsular métodos.
        '''
    def mostrar_info(self):
        print('Información del restaurante....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.__categoria}')
        print(f'Concepto: {self._concepto}')

    def main(self):
        self.mostrar_info()

    def get_categoria(self, categoria):
        '''
        get: obtiene un valor
        '''
        return self.__categoria

    def set_categoria(self, categoria):
        '''
        set : agrega un valor
        '''
        self.__categoria = categoria

rest1 = Restaurante('Rest 5', '30-55555555', 2, 'Buffet')
rest1.main()
```

```
Información del restaurante....
Nombre: Rest 5
Cuit: 30-55555555
Categoría: 2
Concepto: Buffet
```

In [70]:
```
rest1.set_categoria(5)
rest1.main()
```

```
Información del restaurante....
Nombre: Rest 5
Cuit: 30-55555555
Categoría: 5
Concepto: Buffet
```

In [72]:
```
categoria = rest1.get_categoria(4)
print(categoria)
rest1.main()
```

```
5
Información del restaurante....
Nombre: Rest 5
Cuit: 30-55555555
Categoría: 5
Concepto: Buffet
```

Herencia

En Python, dos clases, además de poder tener una **relación de colaboración**, también pueden tener una **relación de herencia**. A través de la herencia se pueden crear nuevas clases a partir de una clase o de una jerarquía de clases (comprobadas y verificadas), evitando el rediseño, verificación y modificación de la parte implementada.

Implica que una subclase tiene todo el comportamiento (métodos) y los atributos (variables) de su superclase, además, de poder agregar los suyos propios.

Por medio de la herencia, una clase, extiende su funcionalidad, y permite la reutilización y la extensibilidad.

La clase de la que se hereda suele llamarse **clase base, superclase, clase padre, clase ancestro (depende del lenguaje de programación)**

En los lenguaje que tienen un sistema de tipos muy fuerte y restrictivo con el tipo de datos de las variables, la herencia suele ser un requisito fundamental para poder emplear el polimorfismo.

Herencia simple	Herencia Múltiple
Una clase sólo puede heredar de una clase base y de ninguna otra.	Una clase puede heredar las características de varias clases base, es decir, puede tener varios padres. En este aspecto hay discrepancias entre los diseñadores de lenguajes, algunos prefieren no admitir la herencia múltiple por los conflictos entre métodos y variables con igual nombre, y eventualmente con comportamientos diferentes pueda crear un desajuste que va en contra de los principios de la POO. Por ello la mayoría de los lenguajes admiten herencia simple, en contraste, unos pocos admiten la herencia múltiple, entre ellos C++, Python o Eiffel.

Herencia simple

In [73]:
```
class Restaurante:

    def __init__(self,nombre,cuit,categoria,concepto):
        self.nombre = nombre
        self.cuit = cuit
        self.__categoria = categoria
        self.__concepto = concepto

    def mostrar_info(self):
        print('Información....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.__categoria}')
        print(f'Concepto: {self.__concepto}')

    def main(self):
        self.mostrar_info()

    def get_categoria(self, categoria):
        return self.__categoria

    def set_categoria(self, categoria):
        self.__categoria = categoria

class Hotel(Restaurante):
    '''
    Agregando entre paréntesis el nombre de la clase, estamos especificándole que hereda dicha clase.
    Entonces podemos decir que la clase Hotel heredará los atributos de Restaurante.
    '''
    def __init__(self,nombre, cuit, categoria, concepto):
        super().__init__(nombre, cuit, categoria, concepto)
        '''
        Con el método super() hacemos referencia a la clase heredada. De otra forma podemos llamar directamente
        a la clase en lugar de al método super()
        '''

hotel = Hotel('Hotel POO', '30-12341234-9', 5, 'Boutique')
hotel.mostrar_info()
```

```
Información....
Nombre: Hotel POO
Cuit: 30-12341234-9
Categoría: 5
Concepto: Boutique
```

In [74]:
```
rest1 = Restaurante('Rest 5', '30-55555555', 2, 'Buffet')
rest1.main()
```

```
Información....
Nombre: Rest 5
Cuit: 30-55555555
Categoría: 2
Concepto: Buffet
```

In [77]:

```
class Restaurante:

    def __init__(self,nombre,cuit,categoria,concepto):
        self.nombre = nombre
        self.cuit = cuit
        self.__categoria = categoria
        self.__concepto = concepto

    def mostrar_info(self):
        print('Información....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.__categoria}')
        print(f'Concepto: {self.__concepto}')

    def main(self):
        self.mostrar_info()

    def get_categoria(self, categoria):
        return self.__categoria

    def set_categoria(self, categoria):
        self.__categoria = categoria

class Hotel(Restaurante):

    def __init__(self,nombre, cuit, categoria, concepto, pileta):
        super().__init__(nombre, cuit, categoria, concepto)
        self.pileta = pileta
        '''
        Agrego atributo y método exclusivo de la clase Hotel. Pero no sale en la emisión porque debo redefinir
        el método mostrar_info() para Hotel, eso se llama polimorfismo.
        '''
    def get_pileta(self):
        return self.pileta

hotel = Hotel('Hotel POO', '30-12341234-9', 5, 'Boutique', 'Si')
hotel.mostrar_info()
```

```
Información....
Nombre: Hotel POO
Cuit: 30-12341234-9
Categoría: 5
Concepto: Boutique
```

In [78]:

```
rest1 = Restaurante('Rest 5', '30-55555555', 2, 'Buffet')
rest1.main()
```

```
Información....
Nombre: Rest 5
Cuit: 30-55555555
Categoría: 2
Concepto: Buffet
```

Herencia Múltiple

In [79]:
```
class Restaurante:

    def __init__(self, nombre, cuit, categoria, concepto):
        self.nombre = nombre
        self.cuit = cuit
        self.categoria = categoria
        self.concepto = concepto

    def mostrar_info(self):
        print('Información....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.categoria}')
        print(f'Concepto: {self.concepto}')

    def main(self):
        self.mostrar_info()

class Gerente:
    def __init__(self, dni, apellido):
        self.dni = dni
        self.apellido = apellido

    def marcacion(self):
        print(f"Apellido: {self.apellido}: Marca asistencia 1 vez.")

class Hotel(Restaurante, Gerente):

    def __init__(self, nombre, cuit, categoria, concepto, dni, apellido, pileta):
        super().__init__(nombre, cuit, categoria, concepto)
        self.pileta = pileta
        Restaurante.__init__(self, nombre, cuit, categoria, concepto)
        Gerente.__init__(self, dni, apellido)
        '''
        Inicializo el constructor de la clase con los parámetros e inicializo los constructores de las clases here
        '''
    def get_pileta(self):
        return self.pileta

    def mostrar_info(self):
        print(f'Nombre: {self.nombre}, Cuit: {self.cuit}, Categoría: {self.categoria}, Concepto: {self.concepto},
        Gerente: {self.apellido}, Pileta: {self.pileta}')

gerente = Gerente(12341234, 'Python3')
print(gerente.marcacion())
```

```
Apellido: Python3: Marca asistencia 1 vez.
None
```

In [80]:
```
rest1 = Restaurante('Rest 5', '30-55555555', 2, 'Buffet')
rest1.main()
```

```
Información....
Nombre: Rest 5
Cuit: 30-55555555
Categoría: 2
Concepto: Buffet
```

In [81]:
```
hotel = Hotel('Hotel Python', '30-12341234-9', 5,'Boutique', 111111111, 'Python3', 'Si')
hotel.mostrar_info()
```

```
Nombre: Hotel Python, Cuit: 30-12341234-9, Categoría: 5, Concepto: Boutique,         Gerente: Python3, Pileta: Si
```

Polimorfismo

- **Es la cualidad de los objetos de responder de distintos modo al mismo mensaje.**
- **Es el cambio de comportamiento de un objeto de tal manera que una referencia a una clase acepta directivas de objeto de dicha clase y de sus clases derivadas.**

Tipos de Polimorfismo	
Sobrecarga:	**Paramétrico:**
Se encuentra cuando, varias clases independientes entre sí , cuentan con un método con el mismo nombre y la misma funcionalidad.	Es la capacidad para definir varias funciones utilizando el mismo nombre pero usando parámetros diferentes.

In [82]:

```
class Gerente:
    def marcacion(self):
        print("Marca asistencia 1 vez.")

class Encargado:
    def marcacion(self):
        print("Marca asistencia 2 veces.")

class Mozo:
    def marcacion(self):
        print("Marca asistencia 2 veces y firma planilla.")


def marcacionTrabajador(trabajador):
    trabajador.marcacion()
    '''
    El método marcacion esta definido en las clases Gerente, Encargado y Mozo. La función marcacionTrabajador
    recibe el objeto y llama al método marcacion según el objeto creado.
    '''

class Restaurante:
    def __init__(self,nombre,cuit,categoria,concepto):
        self.nombre = nombre
        self.cuit = cuit
        self.__categoria = categoria
        self.__concepto = concepto

    def mostrar_info(self):
        print('Información....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.__categoria}')
        print(f'Concepto: {self.__concepto}')

    def main(self):
        self.mostrar_info()

    def get_categoria(self, categoria):
        return self.__categoria

    def set_categoria(self, categoria):
        self.__categoria = categoria


rest1 = Restaurante('Rest 5', '30-55555555', 2, 'Buffet')
rest1.main()
```

```
Información....
Nombre: Rest 5
Cuit: 30-55555555
Categoría: 2
Concepto: Buffet
```

In [83]:

```
mTrabajador1 = Mozo()
marcacionTrabajador(mTrabajador1)
```

```
Marca asistencia 2 veces y firma planilla.
```

In [84]:

```
mTrabajador2 = Gerente()
marcacionTrabajador(mTrabajador2)
```

```
Marca asistencia 1 vez.
```

In [85]:
```
class Restaurante:
    def __init__(self, nombre, cuit, categoria, concepto):
        self.nombre = nombre
        self.cuit = cuit
        self.categoria = categoria
        self.concepto = concepto

    def mostrar_info(self):
        print('Información....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.categoria}')
        print(f'Concepto: {self.concepto}')

    def main(self):
        self.mostrar_info()

    def get_categoria(self, categoria):
        return self.__categoria

    def set_categoria(self, categoria):
        self.__categoria = categoria

class Hotel(Restaurante):
    def __init__(self, nombre, cuit, categoria, concepto, pileta):
        super().__init__(nombre, cuit, categoria, concepto)
        self.pileta = pileta

    def get_pileta(self):
        return self.pileta

    def mostrar_info(self):
        '''
        Redefino el método mostrar_info() para la clase Hotel
        '''
        print(f'Nombre: {self.nombre} Cuit: {self.cuit} Categoría: {self.categoria} Precio: {self.concepto} \
        Pileta: {self.pileta}')


hotel = Hotel('Hotel POO','30-12341234-9',5,'Boutique','Si')
hotel.mostrar_info()
```

```
Nombre: Hotel POO Cuit: 30-12341234-9 Categoría: 5 Precio: Boutique         Pileta: Si
```

In [86]:
```
rest1 = Restaurante('Rest 5', '30-55555555', 2, 'Buffet')
rest1.main()
```

```
Información....
Nombre: Rest 5
Cuit: 30-55555555
Categoría: 2
Concepto: Buffet
```

Variables de clase

En algunos momentos necesitaremos almacenar datos que sean compartidos por todos los objetos de la misma clase, en esos momentos necesitamos emplear **variables de clase**. Para definir una variable de clase lo hacemos dentro de la clase pero fuera de sus métodos.

La variable de clase es compartida por todos los objetos.

In [87]:
```
class Restaurante:
    restaurantes = []

    def __init__(self, nombre, cuit, categoria, concepto):
        self.nombre = ''
        self.cuit = ''
        self.categoria = 0
        self.concepto = ''
        self.restaurantes.append(nombre)

    def mostrar_info(self):
        print('Información....')
        print(f'Nombre: {self.nombre}')
        print(f'Cuit: {self.cuit}')
        print(f'Categoría: {self.categoria}')
        print(f'Concepto: {self.concepto}')

rest1 = Restaurante('Rest 1', '30-11111111-9', 2, 'Buffet')
rest2 = Restaurante('Rest 2', '30-22222222-7', 3, 'Especialidad')
rest3 = Restaurante('Rest 3', '30-33333333-5', 5, 'Temático')
rest4 = Restaurante('Rest 4', '30-44444444-3', 4, 'Comida rápida')
```

In [88]:
```
print(Restaurante.restaurantes)
```

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

Método str

Python nos permite **redefinir** el método que se debe ejecutar, esto se hace definiendo en la clase.

En este ejemplo, devolvemos un string con el método **__str__** que mostrará el dato que le indiquemos al emitir dicho objeto. Para generar dicho string, debemos concatenar valores fijos como el paréntesis o la coma, dobles comillas y convertir a string los atributos que no lo son.

In [89]:
```
class Restaurante:

    def __init__(self, nombre, cuit, categoria, concepto):
        self.nombre = nombre
        self.cuit = cuit
        self.categoria = categoria
        self.concepto = concepto

    def __str__(self):
        cadena=self.nombre+', número de cuit:'+self.cuit+', de categoría: '+ \
            str(self.categoria)+', tipo: ' + self.concepto
        return cadena

rest1 = Restaurante('Rest 1', '30-11111111-9', 2, 'Buffet')
rest2 = Restaurante('Rest 2', '30-22222222-7', 3, 'Especialidad')
rest3 = Restaurante('Rest 3', '30-33333333-5', 5, 'Temático')
rest4 = Restaurante('Rest 4', '30-44444444-3', 4, 'Comida rápida')
print(rest1)
print(rest2)
print(rest3)
print(rest4)
```

```
Rest 1, número de cuit:30-11111111-9, de categoría: 2, tipo: Buffet
Rest 2, número de cuit:30-22222222-7, de categoría: 3, tipo: Especialidad
Rest 3, número de cuit:30-33333333-5, de categoría: 5, tipo: Temático
Rest 4, número de cuit:30-44444444-3, de categoría: 4, tipo: Comida rápida
```

Method Resolution Order (MRO)

Es el orden en el cual el método debe heredar en la presencia de herencia múltiple.

In [90]:
```
Restaurante.__mro__
```

Out[90]:
```
(__main__.Restaurante, object)
```

In [91]:
```
Gerente.__mro__
```

Out[91]:
```
(__main__.Gerente, object)
```

In [92]:
```
Hotel.__mro__
```

Out[92]:
```
(__main__.Hotel, __main__.Restaurante, object)
```

vars() permite ver las clases:

In [36]:
```
vars()
```

Out[36]:
```
{'__name__': '__main__',
 '__doc__': '\nAsignamos la función a una variable.\n',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': ['',
  'class Restaurante:\n    nombre = "Mi Restaurante"\n    cuit = "30-12345678-9"\n    categoria = 4\n    concepto = "Temático"\n    \nrest1 = Restaurante()\nprint(rest1.nombre)\nprint(rest1.cuit)\nprint(rest1.categoria)\nprint(rest1.concepto)\nprint(f"El restaurante se llama \'{rest1.nombre}\' su cuit es \'{rest1.cuit}\' de categoria \'{rest1.categoria}\' y\\\n el concepto es \'{rest1.concepto}\'")',
  'class Restaurante:\n    \n    nombre = "Mi Restaurante"\n    cuit = "30-12345678-9"\n    categoria = 4\nconcepto = "Temático"\n    \nrest1 = Restaurante()\nprint(rest1.nombre)\nrest1.nombre = "Rest_1"\nrest1.cuit = "30-11111111-8"\nrest1.categoria = 3\nrest1.concepto = "Comida rápida"\nprint(f"El restaurante se llama \'{rest1.nombre}\' su cuit es \'{rest1.cuit}\' de categoria \'{rest1.categoria}\' y\\\n el concepto es \'{rest1.concepto}\'")',
  "class Restaurante:\n    \n    '''método con parámetros''' \n\n    def agregar_restaurante(self, nombre, cuit, categoria, concepto):\n        print('Agregando restaurante....')\n        self.nombre = nombre\n        se
```

dir() permite ver los atributos y métodos de una clase.

In [94]:
```
dir(Cliente)
```

Out[94]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'factura',
 'imprimir',
 'impuesto',
 'retornar_monto']
```

Decoradores

Una función puede ser asignada a una variable, puede ser utilizada como argumento para otra función, o inclusive puede ser retornada:

In [95]:
```
def sub_funcion():
    print('Hola soy sub_funcion.')

def super_funcion(funcion):
    funcion()
    print('Y yo super_funcion.')

'''
Asignamos la función a una variable.
'''
funcion = sub_funcion

super_funcion(funcion)
```

```
Hola soy sub_funcion.
Y yo super_funcion.
```

Un decorador es una función que toma como entrada una función y retorna otra función.

Crear un decorador

In [96]:
```
def decorador(func):
    print("Decorador")
    return func
```

Decorar una función

Con la palabra decorar estamos indicando que queremos modificar el comportamiento de una función ya existente, pero sin tener que modificar su código. Esto es útil cuando queremos asignar nuevas funcionalidades a la función. De allí el nombre decorar.

In [97]:
```
@decorador
def hola():
    print("Hola!")

hola()
```

```
Decorador
Hola!
```

La notación del decorador es mediante el símbolo **@** seguido del nombre de la función que cumple el papel de decorador, es equivalente a:

In [98]:
```
Hola = decorador(hola)
```

```
Decorador
```

Si el decorador no retorna la función, esta se ejecutará y lanzará un Error NoneType

In [100]:
```
def decorador(func):
    print("Decorador")

@decorador
def hola():
    print("Hola!")

hola()
```

```
Decorador

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-100-afdf927cf71e> in <module>
      6     print("Hola!")
      7
----> 8 hola()

TypeError: 'NoneType' object is not callable
```

Decorador mediante dos funciones

In [101]:
```
'''
Función decoradora
'''
def decorador(func):

    def innerFunc(*args, **kwargs):
        print(args)
        print(kwargs)
        return func(*args, **kwargs)
    return innerFunc

'''
Función decorada
'''
@decorador
def foobar(x, y, z):
    return x * y + z

print(foobar(3, 5, z=10))
```

```
(3, 5)
{'z': 10}
25
```

*dentro de la función decoradora se emiten los parámetros que se pasaron al ejecutar la función foobar(…)_*

Decorador simple mediante Clases

Hola = decorador(hola) esta estructura vista en un ejemplo anterior, también se puede utilizar en los decoradores implementados mediante clases, solo que ahora Hola va a ser una instancia de la clase decoradora.

In [102]:
```
class Decorator(object):
    '''
    Clase de decorador simple.
    '''
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Antes de ser llamada la función.')
        retorno = self.func(*args, **kwargs)
        print('Despues de ser llamada la función.')
        print(retorno)
        return retorno

@Decorator
def function():
    print('Dentro de la función.')
    return "Retorno"

function()
```

```
Antes de ser llamada la función.
Dentro de la función.
Despues de ser llamada la función.
Retorno
```

Out[102]:
```
'Retorno'
```

*En este ejemplo el decorador se ejecuta en el método sobrecargado "def __call__(self, *args, **kwargs):" de la clase, este método se ejecuta siempre que se instancia y se hace una llamada a la clase.*

La función decorada "function(…)" ya no es una función sino que es una instancia de la clase "Decorador" como se observa en el siguiente ejemplo:

In [103]:
```
import types
isinstance(function, types.FunctionType)
```

Out[103]:
```
False
```

In [104]:
```
type(function)
```

Out[104]:
```
__main__.Decorator
```

Decorando métodos de una clase

Para decorar el método de una clase es necesario agregar un método adicional, este es el método **get**(self):

In [105]:
```
from types import MethodType

class Decorator(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Dentro del Decorador...')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        '''
        Retorna un método si se llama en una instancia
        '''
        return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        print("Dentro de la función decorada...")

a = Test()
```

```
Dentro del Decorador...
Dentro de la función decorada...
```

Como se observa creamos una clase Decorador con tres métodos obligatorios:

__init__(self)

__call__(self)

__get__(self)

esta es la estructura de la clase decoradora para poder decorar métodos de clases.

Decorador con parámetros mediante clases

Este tipo de decorador es muy utilizado al desarrollar software, ya que tiene más funcionalidades adicionales que las implementaciones de decoradores anteriores:

In [106]:
```
class MyDecorator(object):
    def __init__(self, flag):
        self.flag = flag

    def __call__(self, original_func):
        decorator_self = self
        def wrapper(*args, **kwargs):
            print('En decorador antes de wrapper ->', decorator_self.flag)
            original_func(*args,**kwargs)
            print('En decorador luego de wrapper ->', decorator_self.flag)
        return wrapper

@MyDecorator(flag='flag de MyDecorator')
def bar(a,b,c):
    print('En bar argumentos de llamado en __main__(...)-> : ',a,b,c)

if __name__ == "__main__":
    bar(1, "Hola", True)
```

```
En decorador antes de wrapper -> flag de MyDecorator
En bar argumentos de llamado en __main__(...)-> :  1 Hola True
En decorador luego de wrapper -> flag de MyDecorator
```

La función anidada del decorador tendrá por nombre: wrapper, por convención. De igual forma, el nombre del decorador debe ser muy descriptivo.

Errores y Excepciones

Errores de sintaxis	Errores semánticos	Errores de ejecución
Son detectados por el intérprete (o por el compilador, según el lenguaje que estemos utilizando) al procesar el código fuente y generalmente son consecuencia de errores de tipeo al escribir el programa. En el caso de Python nos informa con un mensaje SyntaxError.	Sucede cuando un programa no produce el resultado esperado pero tampoco hay mensajes de error. Generalmente se debe a un algoritmo incorrecto, error u omisión de una sentencia.	El origen se debe a una o más causas, pueden ser errores de programación o debido a recursos externos, por ejemplo intentar leer un archivo dañado.

El control de errores y excepciones hacen que un programa sea más robusto.

Excepciones

Los errores de ejecución son llamados excepciones. Durante la ejecución de un programa, cualquier línea de código puede generar una excepción. Las más frecuentes son:

Exception	AssertionError	IndexError	KeyError	TypeError	ValueError	NameError	ZeroDivisionError	IOError
Todas las excepciones son de tipo Exception	Una instrucción assert falló	Intento de acceder a una secuencia con un índice fuera de rango	Intento de acceder a un diccionario con una clave inexistente	Aplica una operación a un valor de tipo inapropiado	Aplica una operación con un parámetro de tipo apropiado pero su valor no lo es.	Variable no definida	Intenta dividir un número por 0	Error de entrada/ salida, por ejemplo intento de acceder a un archivo

Manejo de excepciones

Cuando ocurre un error o una excepción, el programa se detendrá y generará un mensaje de error, los bloques de control son: try, except y finally

1. Las excepciones se pueden manejar usando la declaración try. Dentro del bloque try se ubica el código que pueda llegar dar una excepción. A continuación se ubica el bloque except, que se encarga de capturar la excepción y da la posibilidad de controlarla.

In [110]:
```
try:
    print(y)
except:
    print("Ocurrió una excepción")
```

```
Ocurrió una excepción
```

2. Se pueden definir tantos bloques de except, como sea necesario -sólo uno se ejecutará- y se puede utilizar except sin especificar el tipo de excepción a capturar (en cuyo caso captura cualquiera) si es este caso debe ser la última de las instrucciones except:

In [111]:
```
try:
    print(y)

except NameError:

    print("La variable y no está definida")

except:

    print("Algo más salió mal…")
```

```
La variable y no está definida
```

3. Se puede usar else para definir un bloque de código que se ejecutará si no se generaron errores:

In [114]:
```
try:

    print("Hola")

except:

    print("Algo salió mal")

else:

    print("Nada salió mal")
```

```
Hola
Nada salió mal
```

4. Si se especifica el bloque finally, se ejecutará independientemente de si el bloque try genera un error o no. Es posible tener un try sólo con finally:

In [115]:
```
try:

    print(y)

except:

    print("Algo salió mal")

finally:

    print("El 'try except' ha finalizado")
```

```
Algo salió mal
El 'try except' ha finalizado
```

5. Se puede emitir una excepción si se produce una condición. Para emitir (o aumentar) una excepción, hay que usar la palabra raise. Esta también se utiliza para generar una excepción. Se puede definir qué tipo de error generar y el texto a emitirle al usuario.

In [116]:
```
x = -1

if x < 0:
    raise Exception("Perdón, no hay valor válido debajo de cero")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-116-4bbf152f440f> in <module>
      2
      3 if x < 0:
----> 4     raise Exception("Perdón, no hay valor válido debajo de cero")

Exception: Perdón, no hay valor válido debajo de cero
```

In [117]:
```
x = "Hola!"

if not type(x) is int:
    raise TypeError("Sólo son permitidos números enteros")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-117-3bc7ad79172b> in <module>
      2
      3 if not type(x) is int:
----> 4     raise TypeError("Sólo son permitidos números enteros")

TypeError: Sólo son permitidos números enteros
```

Si dentro de una función se emite una excepción pero no es controlada, esta se propaga hacia la función que la invocó; si esta otra tampoco la controla, continúa propagándose hasta llegar a la función inicial del programa, y si ésta tampoco la maneja se interrumpe la ejecución del programa.

Una vez que capturamos las excepciones podemos realizar procesos alternativos, por ejemplo dejar constancia detallada en un archivo .log o emitir un mensaje o incluso ambas acciones. El objetivo de dejar constancia es corregir el programa.

Validaciones

Las validaciones permiten asegurar que los valores con los que se van a operar estén dentro de determinado dominio.

Comprobar contenido	Comprobar por tipo	Comprobar por característica
Significa que comprobaremos que el contenido de las variables (valores ingresados por el usuario, de archivos, etc) a utilizar, estén dentro de los valores con los cuáles se pueden operar. A veces no es posible hacerlo pues es costoso corroborar las precondiciones, por lo tanto se realizan sólo cuando es posible.	Significa que nos interesa el tipo del dato que vamos a tratar de validar, para ello se utiliza la función type(variable).	Significa comprobar si una variable tiene determinada característica. Para comprobar si una variable tiene o no una función se utiliza la función hasattr(variable, atributo), donde atributo puede ser el nombre de la función o de la variable que se quiera verificar.

Documentación y comentarios

En general, en el desarrollo de programas y aplicaciones, la documentación es un trabajo que se posterga. En consecuencia, cuando llega el momento de escribirla, se construye documentación que no refleja en profundidad y con detalles, el trabajo realizado. Posteriormente, cuando el código evoluciona con modificaciones y actualizaciones, la tarea se vuelve mucho más difícil.

Si bien el código fuente transmite el algoritmo, hay descripciones que aportan claridad, por ejemplo determinar él o los problemas, fundamentar el diseño, describir el análisis funcional, detallar las razones en que se basan las decisiones, determinar los objetivos, etc. Un desarrollo bien documentado es una parte importante de todo el proyecto.

Documentación	Comentarios
Se escribe entre """ ó ''' (triples comillas simples o dobles)	Se escribe # al comienzo de la línea de comentario
Explica qué hace el código. Está dirigida a quién necesite utilizar la función o módulo, para que pueda entender cómo usarla sin necesidad de leer el código fuente.	Explica cómo funciona el código y en algunos casos por qué se decidió implementarlo así. Los comentarios están dirigidos a quien esté leyendo el código fuente.

def CalcularFactorial(num):
"""" Función recursiva que devuelve el factorial de un número pasado como parámetro"""
if num == 1:
return 1
else:
return num*CalcularFactorial(num-1)
	def CalcularFactorial(num):
	if num == 1:
	return 1
	else:
	return num*CalcularFactorial(num-1)
	# Recibe un número si es 1 devuelve que el factorial es 1, sino
	# acumula el producto del número con el cálculo
	# del factorial del numero-1.

Código autodocumentado

En esta técnica, el objetivo es elegir nombres de funciones y variables o también agregar comentarios en las líneas del código, de tal manera que la documentación sea innecesaria. La desventaja es que hay que tener en cuenta, al elegir los nombres, que sea descriptivos y cortos, que no siempre es posible. Además, para saber qué hace y cómo, implica leer todo el código y no describe los detalles a otros niveles.

Con comentarios	Con nombres de variables descriptivos

an = 7.80 # ancho de la figura
al = 15.45 # alto de la figura
a = an * al # área de la figura
	ancho_del_rectangulo = 7.80
	alto_del_rectangulo = 15.45
	area_del_rectangulo = ancho_del_rectangulo * alto_del_rectangulo

Contratos

Las pre y postcondiciones son un contrato entre el código invocante y el invocado.

Precondiciones	Postcondiciones	assert
Son las condiciones que deben cumplirse antes de ejecutar el programa y para que se comporte correctamente, es decir cómo deben ser los parámetros que recibe, cómo debe ser el estado global, etc. Por ejemplo, en una función que divide dos números, las precondiciones son que los parámetros son números y que el divisor es distinto de 0 (cero).	Son las condiciones que se cumplirán una vez finalizada la ejecución de la función (asumiendo que se cumplen las precondiciones): es decir cómo será el valor de retorno, si los parámetros recibidos o variables globales son alteradas, si se emiten, si modifican archivos, etc. Para el ejemplo dado, dadas las precondiciones se puede asegurar que devolverá un número correspondiente al cociente.	Precondiciones y postcondiciones son assertions, es decir, afirmaciones. Si llegaran a ser falsas significa que existe algún error en el algoritmo. Es recomendable comprobar estas afirmaciones con la instrucción assert. Esta recibe una condición a verificar, si es verdadera la instrucción no hace nada; en caso contrario produce un error. Puede recibir un mensaje que mostrará en caso que la condición no se cumpla. Se debe implementar en la etapa de desarrollo.

Uso de assert

Se deben usar aserciones para probar las condiciones que nunca deberían ocurrir.

assert() en testing	assert() en funciones	assert() con clases

Es útil para escribir tests unitarios o units tests. Ejemplo:
 	
`def calcula_media(lista):`
`  return sum(lista)/len(lista)`
 	
Es muy importante testear el software, para asegurarse de que está libre de errores. Con assert() podemos realizar estas comprobaciones de manera automática.
 	
`assert(calcula_media([5, 10, 7.5]) == 7.5)`
`assert(calcula_media([4, 8]) == 6)`
	Es útil cuando queremos realizar alguna comprobación dentro de una función. En el siguiente ejemplo tenemos una función que sólo suma las variables si son números enteros:
	
	`# Funcion suma de variables enteras`
	
	`def suma(a, b):`
	`  assert(type(a) == int)`
	`  assert(type(b) == int)`
	`  return a+b`
	
	`# Error, ya que las variables no son int`
	`suma(3.0, 5.0)`
	`# Ok, los argumentos son int`
	`suma(3, 5)`
		Verificar que un objeto pertenece a una clase determinada. Ejemplo:
		
		`class MiClase():`
		`  pass`
		
		`class MiOtraClase():`
		`  pass`
		
		`mi_objeto = MiClase()`
		`mi_otro_objeto = MiOtraClase()`
		`# Ok`
		`assert(isinstance(mi_objeto, MiClase))`
		`# Ok`
		`assert(isinstance(mi_otro_objeto, MiOtraClase))`
		`# Error, mi_objeto no pertenece a MiOtraClase`
		`assert(isinstance(mi_objeto, MiOtraClase))`
		`# Error, mi_otro_objeto no pertenece a MiClase`
		`assert(isinstance(mi_otro_objeto, MiClase))`

Invariantes

Se refieren a estados o situaciones que no cambian dentro de un contexto o código.

Invariante de ciclo	Invariante de clase
Permite conocer cómo llegar desde las precondiciones hasta las postcondiciones, cuando la implementación se compone de un ciclo. El invariante de ciclo es, entonces, una aseveración (assertions) que debe ser verdadera al comienzo de cada iteración.	Son condiciones que deben ser ciertas durante toda la vida de un objeto. Una clase tiene dos características fundamentales que la definen: estado y comportamiento. El estado viene definido por la información de sus propiedades (atributos) y el comportamiento viene definido en sus métodos que utilizarán dichos atributos. Los invariantes de clase son propiedades globales de una clase que tienen que ser conservadas por todas las rutinas que la componen.

https://es.wikipedia.org/wiki/Ciclo_invariante (https://es.wikipedia.org/wiki/Ciclo_invariante)

https://es.wikipedia.org/wiki/Invariantes_de_clase#Clases_invariantes_y_herencia
(https://es.wikipedia.org/wiki/Invariantes_de_clase#Clases_invariantes_y_herencia)

In [ ]: