UTN.BA | Centro de e-Learning
Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Diplomatura en Python:

Python nivel avanzado

UTN.BA | Centro de e-Learning
Universidad Tecnológica Nacional
Facultad Regional Buenos Aires
Secretaría de Cultura y Extensión Universitaria

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo 1:

Aplicaciones

Unidad 1:

POO – Patrones de desarrollo.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

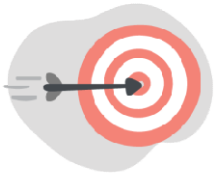
Presentación

Cuando trabajamos con POO muchas veces nos encontramos realizando una y otra vez la misma tarea. Dedicar tiempo a realizar lo que ya alguien ha realizado de forma correcta puede resultar una pérdida de tiempo innecesario que y en muchas ocasiones se suele utilizar el término de “no reinventar la rueda” para decir que no es necesario intentar hacer algo que ya está hecho.

Los patrones de desarrollo son soluciones específicas a problemas concretos, que contienen descripciones de clases y objetos interactuando con la finalidad de resolver un problema de diseño aplicado a una solución particular.

En el transcurso de esta unidad veremos algunos de los patrones más utilizados por los diseñadores.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning



Objetivos

Que los participantes logren…

Implementar algunos de los patrones más conocidos.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Bloques temáticos

1.- Introducción.

2.- Notación UML.

3.- Patrón Singleton.

4.- Factory.

5.- Adapter.

6.- Observer.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

1.- Introducción

A lo largo de esta unidad y la siguiente iremos viendo desde los patrones más simples a los más complejos con los cuales no podemos encontrar en el desarrollo diario. Algunos estarán orientados a la creación de objetos como el singleton, otros tratan sobre la composición de clases y otros caracterizan la forma en la cual interactúan y reparten responsabilidades las clases u objetos.

A lo largo del curso ya hemos utilizado cuatro patrones muy utilizados que no repetiremos en esta unidad:

- Patrón MVC.
- Patrón Delegación.
- Patrón Decorador.
- Patrón Iterador.

2.- Notación UML

El lenguaje de modelado UML (Unified Modeling Language) es uno de los más utilizados en el modelado de sistemas al utilizar el paradigma de programación orientada a objetos. Su objetivo es poder visualizar gráficamente la arquitectura de nuestro sistema, permitiéndonos especificar los atributos, métodos o procesos utilizados de forma independiente del lenguaje de programación utilizado.

Se puede utilizar especificando diferentes diagramas, si utilizamos el diagrama de clases, podemos describir los tipos de objetos en el sistema y las distintas relaciones estáticas que existen entre ellos, así como los atributos y operaciones que realiza una clase, junto con sus restricciones.

En esta notación la visibilidad de una propiedad o un método lo indicaremos por:

+	**Público**
-	Privado

Los métodos estáticos aparecen subrayados.

Los nombres de las propiedadedes inician en minúscula.

Los nombres de los métodos inician en mayúscula.

Nota: En los casos que sea necesario para comprender mejor la implementación introduciremos esta notación esquemática.

3.- Patrón Singleton

Comencemos a trabajar con uno de los patrones más simples, el cual se utiliza para que exista un solo objeto a la vez perteneciente a una determinada clase. Una forma conveniente de hacer esto es delegar a una sola instancia de una clase interna anidada privada:

Debido a que la clase interna se nombra con un subrayado doble, es privada, por lo que el usuario no puede acceder directamente a ella. La clase interna contiene todos los métodos que normalmente se pondrían en la clase si no fuera un singleton, y luego se envuelve en la clase externa que controla la creación utilizando su constructor. La primera vez que crea un "**MiSingleton**", inicializa la instancia, pero después de eso simplemente lo ignora.

singleton1.py

```
1   class MiSingleton(object):
2
3       class __MiSingleton:
4           def __init__(self):
5               self.usuario = None
6
7           def __str__(self):
8               return repr(self) + ' ' + self.usuario
9
10      instancia = None
11
12      def __new__(cls):
13          if not MiSingleton.instancia:
14              MiSingleton.instancia = MiSingleton.__MiSingleton()
15          return MiSingleton.instancia
16
17  juan = MiSingleton()
18  juan.usuario = "Juan Perez2"
19  print(juan)
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

```
20  pedro = MiSingleton()
21  pedro.usuario = "Pedro Correas"
22  print(pedro)
23  Ana = MiSingleton()
24  Ana.usuario = "Ana Gomez"
25  print(Ana)
```

Nos retorna:

```
<__main__.MiSingleton.__MiSingleton object at 0x00323210> Juan Perez2
<__main__.MiSingleton.__MiSingleton object at 0x00323210> Pedro Correas
<__main__.MiSingleton.__MiSingleton object at 0x00323210> Ana Gomez
```

Nota: Como podemos ver la posición de memoria es la misma.

```
<__main__.MiSingleton object at 0x00C33230>
<__main__.MiSingleton object at 0x00C33290>
<__main__.MiSingleton object at 0x00C33290>
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

4.- Factory.

Algunas veces en el diseño en base a clases se requiere que los objetos sean creados en respuesta a condiciones que no pueden ser previstas cuando el programa se escribe. En estos casos podemos utilizar una función como generadora de los objetos de diferentes clases.

factory.py

```
def factory(Clase, *pargs, **kargs):
    return Clase(*pargs, **kargs)

class Auto:
    def doit(self, mensaje):
        print(mensaje)

class Moto:
    def __init__(self, color, marca='Fiat'):
        self.color = color
        self.marca = marca

if __name__ == '__main__':
    object1 = factory(Auto)
    object2 = factory(Moto, "Azul", "Chevrolet")
    object3 = factory(Moto, color='Roja')

    object1.doit('Tipo de auto')
    print(object2.color, object2.marca)
    print(object3.color, object3.marca)
```

Retorna:

```
Tipo de auto
Azul Chevrolet
Roja Fiat
```

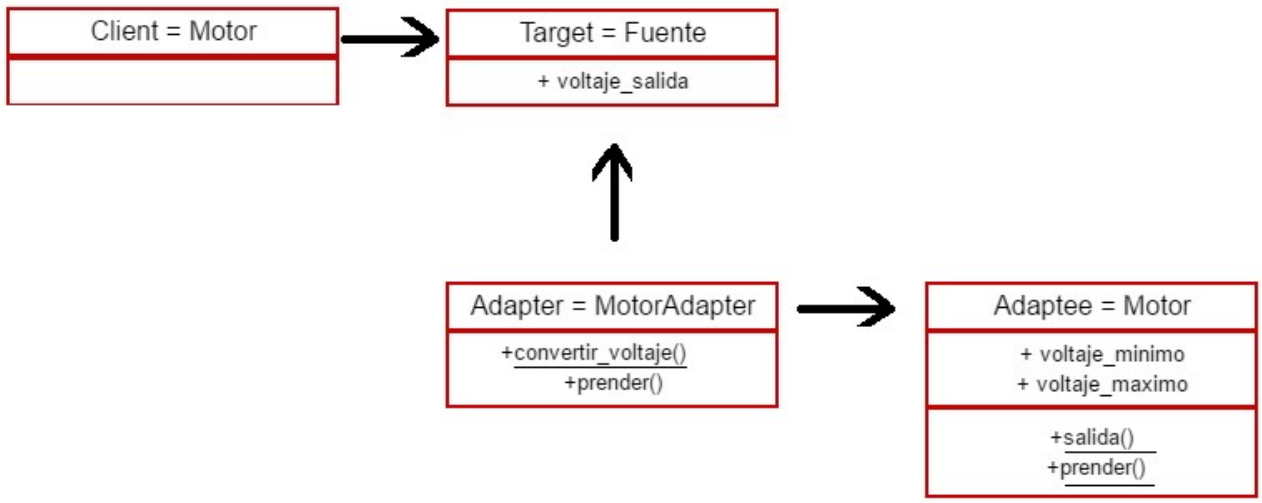
Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

5.- Adapter.

El patrón de adaptador funciona como un puente entre dos interfaces incompatibles. A veces, puede haber un escenario en el que dos objetos no encajan, ya que deben estar en orden para realizar el trabajo. Esta situación puede surgir cuando intentamos integrar un código heredado con un código nuevo, o al cambiar una API de terceros en el código. Esto se debe a las interfaces incompatibles de los dos objetos que no encajan entre sí. En este caso podemos utilizar el patrón de diseño del adaptador. Los elementos que lo conforman son:

- **Target:** Define la interfaz específica del dominio que el Cliente usa.
- **Client:** Colabora con la conformación de objetos para la interfaz Target.
- **Adaptee:** Define una interfaz existente que necesita adaptarse.
- **Adapter:** Adapta la interfaz de Adaptee a la interfaz Target.

Esquemáticamente vamos a llegar a:



El patrón del adaptador se debe utilizar cuando:

- Existe una clase y su interfaz no coincide con la que necesitamos.
- Deseamos crear unas clases reutilizable que coopere con clases no relacionadas o imprevistas, es decir, clases que no necesariamente tienen interfaces compatibles.

- Existen varias subclases para usar, pero no es práctico adaptar su interfaz subclasificando cada una. Un adaptador de objetos puede adaptar la interfaz de su clase padre.

El patrón adaptador puede ser implementado de dos formas, como:

- Adaptador de objetos
- Adaptador de clase

Consideremos un ejemplo en el cual estamos queriendo manejar un motor de corriente continua como el de la figura, que requiere un voltaje de entrada de 1.5V a 3V de corriente continua, pero solo contamos con una batería de 9 V o el enchufe de 220 V.

Motor DC 9000RPM 1.5V a 3V Modelo RE280 RE-280RA-2865 Nubbeo

https://www.nubbeo.com.ar

Podemos crear una clase para el motor "**Motor**", que considere el rango de entrada de voltaje en el cual puede funcionar correctamente, no funcionar o quemarse, y una clase "**Fuente**" que pasamos como parámetro de un objeto de la clase motor de forma de determinar el estado de funcionamiento del motor. Pará hacerlo un poco más real implementaremos una herencia de clases de la clase "**Fuente**" para considerar una fuente de energía de 9v y de 220v.

adaptador1.py

```
1   # ------------ CLIENTE --------------------------------
2   class Motor():
3       voltaje_minimo = 1.5
4       voltaje_maximo = 3
5
6       @classmethod
7       def salida(cls, voltage_entrada):
8           if  voltage_entrada > cls.voltaje_maximo:
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

```
9               print("Con voltaje de entrada de : %s V Se quema!" %(voltage_entrada))
10          elif voltage_entrada < cls.voltaje_minimo:
11              print("Con voltaje de entrada de : %s V el Voltage es insuficiente!!!" %(voltage_entrada))
12          else:
13              print("Con voltaje de entrada de : %s V está funcionando" %(voltage_entrada))
14
15      def prender(self, voltage_entrada):
16          self.salida(voltage_entrada)
17
18  # ---------------FUENTE-----------------------------
19  class Fuente():
20      voltaje_salida = None
21
22  class FuenteAr(Fuente):
23      voltaje_salida = 220
24
25  class FuenteVateria9V(Fuente):
26      voltaje_salida = 9
27
28  # ---------------PRUEBA-----------------------------
29      motor = Motor()
30      print(motor.prender(FuenteAr.voltaje_salida))
31      print(motor.prender(FuenteVateria9V.voltaje_salida))
32      print('---'*25)
33
34      print(motor.prender(FuenteArAdapter.voltaje_salida))
35      print(motor.prender(FuenteVateria9VAdapter.voltaje_salida))
36
37  if __name__ == "__main__":
38
39      motor = Motor()
40      print(motor.prender(FuenteAr.voltaje_salida))
41      print(motor.prender(FuenteVateria9V.voltaje_salida))
```

Retorna

```
Con voltaje de entrada de : 220 V Se quema!
None
```

```
Con voltaje de entrada de : 9 V Se quema!
None
```

Cómo podemos ver en ambos casos el motor se quema debido a que la tensión de entrada es muy alta, por lo que necesitamos un adaptador de corriente que nos permita tomar la tensión y convertirla al rango aceptado por el motor. Está será nuestra clase "**adapter**" (adaptadora).

Objeto adaptador

Podemos pensar al adaptador como una entidad independiente el motor y de la fuente, que encapsule a ambos (cliente = Motor, proveedor = Fuente) y le perita llamar al método prender sin realizar cambios en las clases. Esta clase adaptadora podría convertir la tensión alterna de 200 V o la continua de 9 V en continua dentro del rango utilizado por el motor.

adaptador1.py

```
27  ----------------------
28  #------- CLASES ADAPTADORAS ------------------
29  class FuenteArAdapter():
30
31      voltage_entrada = FuenteAr.voltaje_salida
32      voltaje_salida = (Motor.voltaje_maximo + Motor.voltaje_minimo)/2
33
34  class FuenteVateria9VAdapter():
35
36      voltage_entrada = FuenteAr.voltaje_salida
37      voltaje_salida = (Motor.voltaje_maximo + Motor.voltaje_minimo)/2
38
39  ----------------------

46      print(motor.prender(FuenteArAdapter.voltaje_salida))
47      print(motor.prender(FuenteVateria9VAdapter.voltaje_salida))
```

Retorna:

```
Con voltaje de entrada de : 2.25 V está funcionando
None
Con voltaje de entrada de : 2.25 V está funcionando
None
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Clase adaptador

También podriamos pensar que tanto el motor como la fuente definen un sistema único, el cual puede ser "UsarFuente". Vamos a utilizar un sistema de herencia múltiple que herede atributos y métodos tanto del motor como de la fuente. De esta forma no pensaremos en crear una nueva entidad entre el motor y la fuente, sino que pensamos en redefinir al cliente. A partir de este punto ya no tendremos un motor, sino una nueva entidad que es la combinación de un motor y una fuente.

En el caso de que tengamos un objeto que pensamos que vamos a cargar con una fuente, y la misma no posee el mismo voltaje que pensamos, podemos implementar un mensaje de error que nos permita identificar este problema.

adaptador2.py

```
28  #------- CLASES ADAPTADORAS ------------------
29
30  class ImposibleTransformaVoltage(Exception):
31      pass
32
33  class MotorAdapter(Motor, Fuente):
34
35      @classmethod
36      def convertir_voltaje(cls, voltage_entrada):
37          if voltage_entrada == cls.voltaje_salida: #Se está fijango en las fuentes
38              return ((cls.voltaje_maximo + cls.voltaje_minimo)/2)
39
40          else:
41              raise ImposibleTransformaVoltage(
42                  "No se puede transformar {0} en {1}V. Este adaptador transforma:  {2} en
    {1}V.".format(
43                      voltage_entrada,    ((cls.voltaje_maximo    +    cls.voltaje_minimo)/2),
    cls.voltaje_salida
44                  )
45              )
46
47      @classmethod
48      def prender(cls, voltage_entrada):
49          try:
```

```
50              voltage = cls.convertir_voltaje(voltage_entrada)
51              cls.salida(voltage)
52          except ImposibleTransformaVoltage as e:
53              print(e)
54
55  class FuenteArAdapter(MotorAdapter, FuenteAr):
56      pass
57
58  class FuenteVateria9VAdapter(MotorAdapter, FuenteVateria9V):
59      pass
60
61  if __name__ == "__main__":
62
63      print('---'*25)
64      motor_220 = FuenteArAdapter()
65      print(motor_220.prender(FuenteAr.voltaje_salida) )
66      print(motor_220.prender(FuenteVateria9V.voltaje_salida))
```

Retorna:

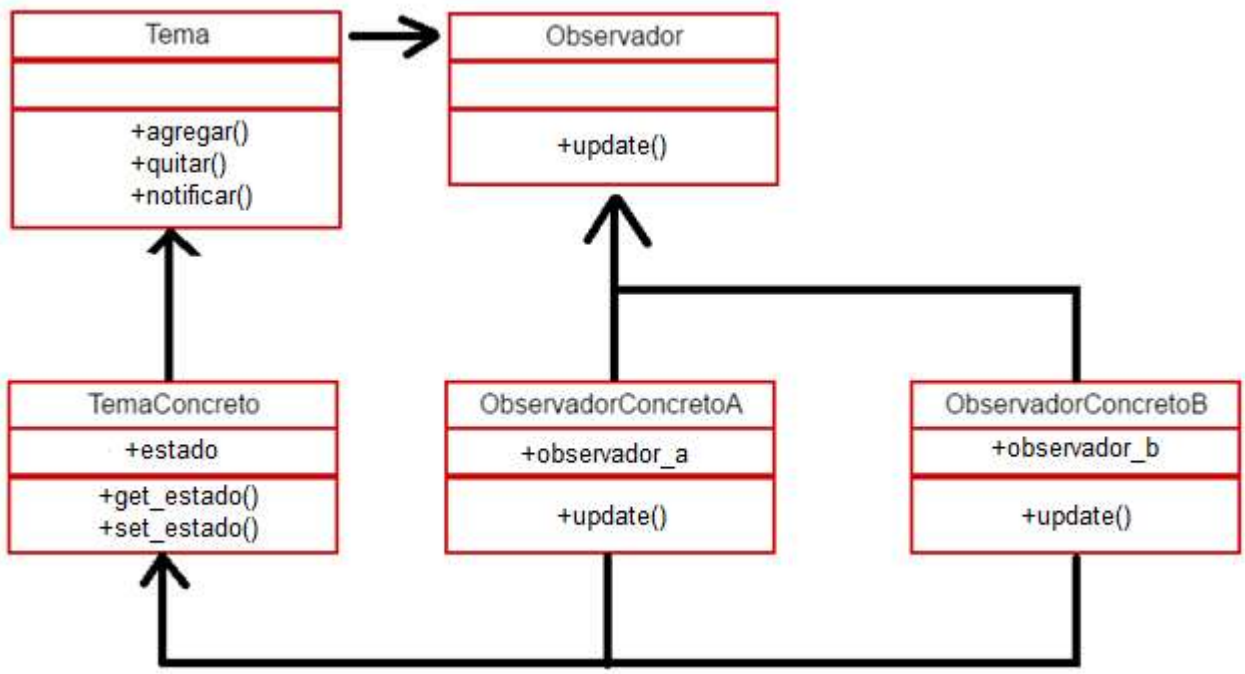
```
---------------------------------------------------------------------------
Con voltaje de entrada de : 2.25 V está funcionando
None
No se puede transformar 9 en 2.25V. Este adaptador transforma:  220V en 2.25V.
None
```

6.- Observador.

El patrón observador resuelve un problema bastante común: ¿**qué sucede si un grupo de objetos necesita actualizarse cuando algún objeto cambia de estado**?

Supongamos que tenemos una aplicación en el cual se están cargando datos de usuario en una agenda y necesitamos que cada vez que un dato es agregado, se actualice la base de datos, los datos en pantalla, y se envíe un mail informándole de evento al administrador de la plataforma. El patrón observador cumple con estas necesidades.

El diagrama UML del patrón se puede ver de la siguiente forma:



En el diagrama anterior, la clase Tema no actualiza el estado de los objetos dependientes directamente. En su lugar, utiliza la interfaz del observador (update ()) para actualizar el estado, lo que hace que el Tema sea independiente de cómo se actualiza el estado de los objetos dependientes. Las clases ObserverConcretoA y ObserverConcretoB implementan la interfaz Observador al sincronizar su estado con el estado del Tema.

En el diagrama anterior hemos agregado un tema específico heredado del tema principal de forma de que un determinado tema pueda ser desglosado en partes y realizar un seguimiento del estado de cada una de sus partes.

Llevado a la práctica lo podemos representar de la siguiente manera:

observador.py

```
1   class Tema:
2
3       observadores = []
4
5       def agregar(self, obj):
6           self.observadores.append(obj)
7
8       def quitar(self, obj):
9           pass
10
11      def notificar(self):
12          for observador in self.observadores:
13              observador.update()
14
15
16  class TemaConcreto(Tema):
17      def __init__(self):
18          self.estado = None
19
20      def set_estado(self, value):
21          self.estado = value
22          self.notificar()
23
24      def get_estado(self):
25          return self.estado
26
27
28  class Observador:
29      def update(self):
30          raise NotImplementedError("Delegación de actualización")
31
32
33  class ConcreteObserverA(Observador):
34      def __init__(self, obj):
35          self.observador_a = obj
36          self.observador_a.agregar(self)
37
```

```
38      def update(self):
39          print("Actualización dentro de ObservadorConcretoA")
40          self.estado = self.observador_a.get_estado()
41          print("Estado = ", self.estado)
42
43
44  class ConcreteObserverB(Observador):
45      def __init__(self, obj):
46          self.observador_b = obj
47          self.observador_b.agregar(self)
48
49      def update(self):
50          print("Actualización dentro de ObservadorConcretoB")
51          self.estado = self.observador_b.get_estado()
52          print("Estado = ", self.estado)
53
54
55  tema1 = TemaConcreto()
56  observador_a = ConcreteObserverA(tema1)
57  observador_b = ConcreteObserverB(tema1)
58  tema1.set_estado(1)
59  # print(observador_a.__dict__)
60  print("---" * 25)
61  print(Tema.__dict__)
```

Nos retorna:

```
Actualización dentro de ObservadorConcretoA
Estado =  1
Actualización dentro de ObservadorConcretoB
Estado =  1
{'observadorA': <__main__.TemaConcreto object at 0x01303430>, 'estado': 1}
---------------------------------------------------------------------------
{'__module__': '__main__', 'observadores': [<__main__.ConcreteObserverA object at
0x01303470>, <__main__.ConcreteObserverB object at 0x01303490>],
{'__module__': '__main__', 'observadores': [<__main__.ConcreteObserverA object at
0x021F2DA8>, <__main__.ConcreteObserverB object at 0x021F2EB0>], 'agregar':
<function Tema.agregar at 0x0207D610>, 'quitar': <function Tema.quitar at
0x0207D388>, 'notificar': <function Tema.notificar at 0x0207D2F8>, '__dict__':
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

<attribute '__dict__' of 'Tema' objects>, '__weakref__': <attribute '__weakref__' of 'Tema' objects>, '__doc__': None}

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Bibliografía utilizada y sugerida

Libros y otros manuscritos:

Programming Python 5th Edition – Mark Lutz – O'Reilly 2013

Programming Python 4th Edition – Mark Lutz – O'Reilly 2011

Manual online

https://docs.python.org/3.7/tutorial/

https://docs.python.org/3.7/library/index.html

https://docs.python.org/3/distutils/introduction.html