Talento Digital para Chile:

Módulo 3
Programación Avanzada en Python

nodovirtual.awakelab.cl

jELOU futurejob by Adalid Chile

3.4.- Contenido 4: Codificar un programa utilizando el concepto de herencia para resolver un problema de baja complejidad acorde al lenguaje Python

Objetivo de la jornada

1. Describe el concepto de polimorfismo bajo el paradigma de orientación a objetos para resolver un problema
2. Utiliza herencia de clases como medio de implementación de polimorfismo para resolver un problema
3. Codifica un programa utilizando herencia y sobrescritura de métodos para resolver un problema

3.4.1.- Herencia y Polimorfismo

En las primeras clases de este módulo, en la sección Encapsulamiento, mencionamos que hay cuatro conceptos fundamentales de la programación orientada a objetos encapsulamiento, abstracción, herencia y polimorfismo. Los primeros dos fueron vistos en tal clase, ahora veremos herencia y polimorfismo.

3.4.1.1. Qué es el Polimorfismo

Polimorfismo Es un nombre elegante que describe un concepto simple; ocurren diferentes comportamientos dependiendo de la subclase que se esté utilizando, sin tener que saber explícitamente cuál es la subclase realmente. Como ejemplo, imagine un programa que reproduce archivos de audio. Un reproductor multimedia puede necesitar cargar un objeto **AudioFile** y luego **reproducirlo**. Pondríamos un método **reproducir()** en el objeto, que es responsable de descomprimir o extraer el

audio y enrutarlo a la tarjeta de sonido y los altavoces. El acto de reproducir un **AudioFile** podría ser tan simple como:

```
audio_file.reproducir()
```

3.4.1.2. Qué es la Herencia

Sin embargo, el proceso de descomprimir y extraer un archivo de audio es muy diferente para diferentes tipos de archivos. Los archivos **.wav** se almacenan sin comprimir, mientras que los archivos **.mp3**, **.wma** y **.ogg** tienen algoritmos de compresión muy diferentes.
Podemos usar herencia con polimorfismo para simplificar el diseño. Cada tipo de archivo se puede representar mediante una subclase diferente de **AudioFile**, por ejemplo, **WavFile**, **MP3File**. Cada uno de estos tendría un método **reproducir()**, pero ese método se implementaría de manera diferente para cada archivo para garantizar que se siga el procedimiento de extracción correcto. El objeto del reproductor multimedia nunca necesitaría saber a qué subclase de **AudioFile** se refiere; simplemente llama **reproducir()** y "polimórficamente" permite que el objeto se encargue de los detalles reales de la reproducción. Veamos un esqueleto rápido que muestra cómo se vería esto:

```
class AudioFile:

    def __init__(self, nombre_archivo):
        if not nombre_archivo.endswith(self.ext):
        raise Exception("formato de archivo invalido")
        self.nombre_archivo = nombre_archivo

class MP3File(AudioFile):
    ext = "mp3"
    def reproducir(self):
        print("reproduciendo {} como mp3"
            .format(self.nombre_archivo))

class WavFile(AudioFile):
    ext = "wav"
    def reproducir(self):
        print("reproduciendo {} como wav"
            .format(self.nombre_archivo))

class OggFile(AudioFile):
    ext = "ogg"
    def reproducir(self):
        print("reproduciendo {} como ogg"
            .format(self.nombre_archivo))
```

Todos los archivos de audio se comprueban para asegurarse de que se proporcionó una extensión válida al inicializar. ¿Pero observe cómo el método **__init__** en la clase

principal puede acceder a la variable de **ext** de las otras subclases? Eso es "polimorfismo" en acción. Si el nombre del archivo no termina con el nombre correcto, genera una excepción (las excepciones se tratarán en detalle en la clase siguiente). El hecho de que **AudioFile** en realidad no almacene una referencia a la variable **ext** no impide que pueda acceder a ella en la subclase.
Además, cada subclase de **AudioFile** implementa **reproducir()** de una manera diferente (este ejemplo en realidad no reproduce la música; ¡los algoritmos de compresión de audio realmente merecen un curso aparte!). Esto también es polimorfismo en acción. El reproductor multimedia puede usar exactamente el mismo código para reproducir un archivo, sin importar el tipo que sea; no le importa qué subclase de AudioFile esté mirando. Los detalles de la descompresión del archivo de audio están encapsulados. Si probamos este ejemplo, funciona como esperamos:

```
ogg = OggFile("myarchivo.ogg")
ogg.reproducir()
reproduciendo myarchivo.ogg como ogg

mp3 = MP3File("myarchivo.mp3")
mp3.reproducir()
reproduciendo myarchivo.mp3 como mp3

no_mp3 = MP3File("myfile.ogg")

Exception Traceback (most recent call last)
<ipython-input-49-7f9ca5038385> in <module>
----> 1 no_mp3 = MP3File("myfile.ogg")

<ipython-input-45-3aa47387d105> in __init__(self, nombre_archivo)

      3     def __init__(self, nombre_archivo):
      4     if not nombre_archivo.endswith(self.ext):
----> 5         raise Exception("formato de archivo invalido")
      6     self.nombre_archivo = nombre_archivo
      7

Exception: formato de archivo invalido
```

¿Ve cómo **AudioFile.__init__** puede verificar el tipo de archivo sin saber realmente a qué subclase se refiere?
El polimorfismo es en realidad una de las cosas más interesantes de la programación orientada a objetos y hace que algunos diseños de programación sean obvios lo que no era posibles en paradigmas anteriores. Sin embargo, Python hace que el polimorfismo sea menos interesante debido al "duck typing". "Duck typing" en Python nos permite usar cualquier objeto que proporcione el comportamiento requerido sin forzarlo a ser una subclase. La naturaleza dinámica de Python hace que esto sea trivial. El siguiente ejemplo no amplía AudioFile, pero se puede interactuar con él en Python utilizando exactamente la misma interfaz:

```
class FlacFile:
    def __init__(self, nombre_archivo):
        if not nombre_archivo.endswith(".flac"):
            raise Exception("Invalid file format")
        self.nombre_archivo = nombre_archivo

    def play(self):
        Print(reproduciendo {} como flac"
                  .format(self.nombre_archivo))
```

Nuestro reproductor multimedia puede reproducir este objeto con la misma facilidad que uno que amplía **AudioFile**.
El polimorfismo es una de las razones más importantes para utilizar la herencia en muchos contextos orientados a objetos. Debido a que cualquier objeto que proporcione la interfaz correcta se puede usar indistintamente en Python, reduce la necesidad de superclases comunes polimórficas. La herencia aún puede ser útil para compartir código, pero si todo lo que se comparte es la interfaz pública, todo lo que se requiere es duck typing. Esta menor necesidad de herencia también reduce la necesidad de herencia múltiple; a menudo, cuando la herencia múltiple parece ser una solución válida, simplemente podemos usar el duck typing para imitar una de las múltiples superclases.
Por supuesto, el hecho de que un objeto satisfaga una interfaz particular (proporcionando los métodos o atributos requeridos) no significa que simplemente funcionará en todas las situaciones. Tiene que cumplir con esa interfaz de una manera que tenga sentido en el sistema en general. El hecho de que un objeto proporcione un método **reproducir()** no significa que funcionará automáticamente con un reproductor multimedia.
Otra característica útil del duck typing es que el objeto de duck type solo necesita proporcionar el método y los atributos a los que realmente se accede. Por ejemplo, si necesitamos crear un objeto de archivo falso para leer datos, podemos crear un nuevo objeto que tenga un método **leer()**; no tenemos que sobrecargar el método de **escribir** si el código que va a interactuar con el objeto solo se leerá del archivo. De manera más sucinta, duck typing no necesita proporcionar la interfaz completa de un objeto que está disponible, solo necesita cumplir con la interfaz que se utiliza realmente.

3.4.1.3. Qué es la Herencia

En el mundo de la programación, el código duplicado se considera maligno. No deberíamos tener múltiples copias del mismo código o similar en diferentes lugares. Hay muchas formas de fusionar piezas de código u objetos similares con una funcionalidad similar. En este apartado, cubriremos el principio orientado a objetos más famoso: la herencia. La herencia nos permite crear relaciones "es una"/"es un" entre dos o más clases, abstrayendo detalles comunes en superclases y almacenando otros específicos en la subclase. En particular, cubriremos la sintaxis y los principios de Python para:

Herencia Simple

Técnicamente, cada clase que creamos usa herencia. Todas las clases de Python son subclases de la clase especial denominada objeto. Esta clase proporciona muy poco en términos de datos y comportamientos (los comportamientos que proporciona son todos métodos de doble subrayado destinados solo para uso interno), pero permite que Python trate todos los objetos de la misma manera. (Ver ultima sección de la clase anterior)
Si no heredamos explícitamente de una clase diferente, nuestras clases heredarán automáticamente de **objeto**. Sin embargo, podemos afirmar abiertamente que nuestra clase se deriva de un **objeto** utilizando la siguiente sintaxis:

```
class MiSubClase(object):
    pass
```

¡Esto es herencia! Este ejemplo, técnicamente, no es diferente de los que hemos venido usando, ya que Python 3 hereda automáticamente del objeto si no proporcionamos explícitamente una superclase diferente. Una superclase, o clase madre, es una clase de la que se hereda. Una subclase es una clase que hereda de una superclase. En este caso, la superclase es objeto y MiSubClase es la subclase. También se dice que una subclase se deriva de su clase principal o que la subclase extiende a la principal.
Como probablemente haya descubierto en el ejemplo, la herencia requiere una cantidad mínima de sintaxis adicional sobre una definición de clase básica. Simplemente incluya el nombre de la clase madre dentro de un par de paréntesis después del nombre de la clase, pero antes de que los dos puntos terminen la definición de la clase. Esto es todo lo que tenemos que hacer para decirle a Python que la nueva clase debe derivarse de la superclase dada.
¿Cómo aplicamos la herencia en la práctica? El uso más simple y obvio de la herencia es agregar funcionalidad a una clase existente. Comencemos con un administrador de contactos simple que rastrea el nombre y la dirección de correo electrónico de varias personas. La clase de contacto es responsable de mantener una lista de todos los contactos en una variable de clase, y de inicializar el nombre y la dirección, en esta clase simple:

```
class Contacto:
    contactos = []

    def __init__(self, nombre, email):
        self.nombre= nombre
        self.email = email
        Contacto.contactos.append(self)
```

Este ejemplo nos presenta las variables de clase. La lista **contactos**, porque es parte de la definición de clase, en realidad la comparten todas las instancias de esta clase.

Esto significa que solo hay una lista **Contacto.contactos**, y si llamamos a **self.contactos** en cualquier objeto, se referirá a esa lista única. El código en el inicializador asegura que cada vez que creamos un nuevo contacto, automáticamente se agregará el nuevo objeto a la lista. Tenga cuidado con esta sintaxis, porque si alguna vez establece la variable usando self.contactos, en realidad estará creando una nueva variable de instancia en el objeto; la variable de clase seguirá sin cambiar y será accesible como **Contacto.contactos**.
Esta es una clase muy simple que nos permite rastrear un par de datos sobre nuestros contactos. Pero ¿qué pasa si algunos de nuestros contactos también son proveedores a los que necesitamos pedir suministros? Podríamos agregar un método **orden** a la clase **Contacto**, pero eso permitiría a las personas ordenar accidentalmente cosas de contactos que son clientes o amigos familiares. En su lugar, creemos una nueva clase **Proveedor** que actúa como un contacto, pero tiene un método de pedido adicional:

```
class Proveedor(Contacto):

    def orden(self, orden):
        print("Si este fuera un sistema real enviariamos la {} orden a {}"
                    .format(orden, self.nombre))
```

Ahora, si probamos esta clase en intérprete, vemos que todos los contactos, incluidos los proveedores, aceptan un nombre y una dirección de correo electrónico en su **__init__**, pero solo los proveedores tienen un método de orden funcional:

```
c = Contacto("Alguien", "alguien@ejemplo.net")
p = Proveedor("Proveedor", "prov@ejemplo.net")
print(c.nombre, c.email, p.nombre, p.email)

Alguien alguien@ejemplo.net Pro veedor prov@ejemplo.net

c.contactos
[<__main__.Contacto    at    0x11276f590>,    <__main__.Proveedor    at
0x11264aed0>]



c.orden("alicates")
---------------------------------------------------------------------------
------
AttributeError                            Traceback (most recent call
last)
<ipython-input-59-16bff1c856e0> in <module>
----> 1 c.orden("alicates")

AttributeError: 'Contacto' object has no attribute 'orden'

p.orden("alicates")

Si este fuera un sistema real enviariamos la alicates orden a Pro
veedor
```

Así que ahora nuestra clase de **Proveedor** puede hacer todo lo que un **Contacto** puede hacer (incluido agregarse a sí mismo a la lista de **contactos**) y todas las cosas especiales que necesita manejar como proveedor. Ésta es la belleza de la herencia .

3.4.1.4. Herencia Múltiple

La herencia múltiple es un tema delicado. En principio, es muy simple: una subclase que hereda de más de una clase principal puede acceder a la funcionalidad de ambas. En la práctica, esto es mucho menos útil de lo que parece y muchos programadores expertos recomiendan no usarlo. Entonces comenzaremos con una advertencia:
Como regla general, si cree que necesita una herencia múltiple, probablemente esté equivocado, pero si sabe que la necesita, probablemente tenga razón.
La forma más simple y útil de herencia múltiple se llama "mixin". Un "mixin" es generalmente una superclase que no está destinada a existir por sí sola, sino que está destinada a ser heredada por alguna otra clase para proporcionar una funcionalidad adicional. Por ejemplo, digamos que queríamos agregar una funcionalidad a nuestra clase **Contacto** que permite enviar un correo electrónico a **self.email**. Enviar correo electrónico es una tarea común que podríamos querer usar en muchas otras clases. Entonces podemos escribir una clase mixin simple para enviarnos el correo electrónico:

```
class EnviaMail:
    def enviar_mail(self, mensaje):
        print("Mandar mail a " + self.email)
        # Logica adicional para e-mail aqui
```

Por brevedad, no incluiremos aquí la lógica adicional para enviar correo electrónico; Si está interesado en estudiar cómo se hace, consulte el módulo **smtplib** en la biblioteca estándar de Python.
Esta clase no hace nada especial (de hecho, apenas puede funcionar como una clase independiente), pero nos permite definir una nueva clase que es tanto un **Contacto** como un **EnviaMail**, usando herencia múltiple:

```
class EmailableContacto(Contacto, EnviaMail):
    pass
```

La sintaxis de la herencia múltiple parece una lista de parámetros en la definición de clase. En lugar de incluir una clase base dentro del paréntesis, incluimos dos (o más), separadas por una coma. Podemos probar este nuevo híbrido para ver cómo funciona el mixin:

```
e = EmailableContacto("Pedro Jaque", pj@jaque.cl)
Contacto.contactos
[<__main__.Contacto at 0x11276f590>,
```

```
 <__main__.Proveedor at 0x11264aed0>,
 <__main__.EmailableContacto at 0x112a88050>]

e.enviar_mail("Hola,email de prueba")
Mandar mail a pj@jaque.cl
```

El inicializador de **Contacto** todavía está agregando el nuevo contacto a la lista **contactos**, y el mixin puede enviar correo a **self.email** para que sepamos que todo está funcionando.

Eso no fue tan difícil, y probablemente se esté preguntando cuáles son las advertencias sobre la herencia múltiple. Entraremos en las complejidades en un minuto, pero consideremos qué opciones teníamos, además de usar un mixin aquí:

- Podríamos haber usado herencia única y agregar la función **enviar_mail** a la subclase. La desventaja aquí es que la funcionalidad del correo electrónico debe duplicarse para cualquier otra clase que necesite correo electrónico.

- Podemos crear una función de Python independiente para enviar correo, y simplemente llamarla, con la dirección de correo electrónico correcta proporcionada como parámetro, cuando sea necesario enviar un correo electrónico.

- Podríamos monkey-patch (ampliar o modificar el software del sistema de soporte localmente) la clase **Contacto** para tener un método **enviar_mail** después de que se haya creado la clase. Esto se hace definiendo una función que acepte el argumento self y configurándolo como un atributo en una clase existente.

La herencia múltiple funciona bien cuando se mezclan métodos de diferentes clases, pero se vuelve muy complicado cuando tenemos que trabajar con métodos de llamada en la superclase. ¿Por qué? Porque hay múltiples superclases. ¿Cómo sabemos a cuál llamar? ¿Cómo sabemos en qué orden llamarlos?

Exploremos estas preguntas agregando una dirección de casa a nuestra clase **Amigo**. ¿De qué maneras podríamos hacer esto? Una dirección es una colección de cadenas que representan la calle, la ciudad, el país y otros detalles relacionados del contacto. Podríamos pasar cada una de estas cadenas como parámetros al método **__init__** de la clase **Amigo**. También podríamos almacenar estas cadenas en una tupla o diccionario y pasarlas a **__init__** como un solo argumento. Este es probablemente el mejor curso de acción si no hay ninguna funcionalidad adicional que deba agregarse a la dirección.
Otra opción sería crear una nueva clase **Direccion** para mantener juntas esas cadenas y luego pasar una instancia de esta clase a **__init__** en nuestra clase **Amigo**. La ventaja de esta solución es que podemos agregar comportamiento (digamos, un

método para dar direcciones a esa dirección o para imprimir un mapa) a los datos en lugar de simplemente almacenarlos estáticamente. Esto sería utilizar la composición, la relación "tiene una" que discutimos anteriormente. La composición es una solución perfectamente viable a este problema y nos permite reutilizar las clases de direcciones en otras entidades como edificios, negocios, u organizaciones.

Sin embargo, la herencia también es una solución viable, y eso es lo que queremos explorar, así que agreguemos una nueva clase que contenga una dirección. Llamaremos a esta nueva clase **HabitanteDeDireccion** en lugar de **Direccion**, porque la herencia define una relación "es un". No es correcto decir que un **Amigo** es una **Direccion**, pero dado que un amigo puede tener una **Direccion**, podemos argumentar que un **Amigo** es un **HabitanteDeDireccion**. Posteriormente, podríamos crear otras entidades (empresas, edificios) que también tengan direcciones. Aquí está nuestra clase **HabitanteDeDireccion**:

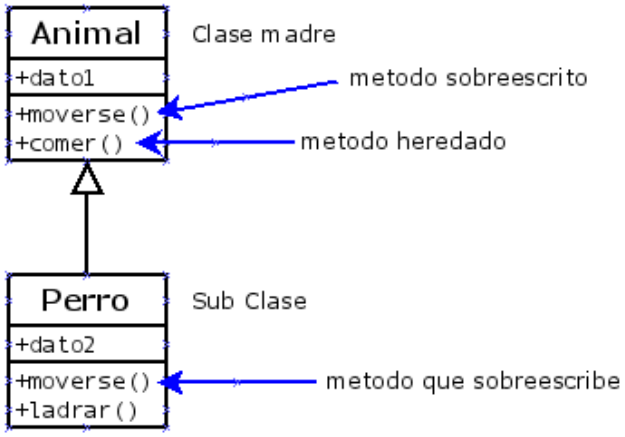
```
class HabitanteDeDireccion:

    def __init__(self, calle, ciudad, comuna, codigo):
        self.calle = calle
        self.ciudad = ciudad
        self.comuna = comuna
        self.codigo = codigo
```

Muy simple; simplemente tomamos todos los datos y los arrojamos a variables de instancia al inicializar.

3.4.1.5. Sobrescritura de métodos

Ahora que ya tenemos claro lo que significa Herencia discutiremos la sobreescritura de métodos. La sobreescritura de métodos es una capacidad de cualquier lenguaje de programación orientado a objetos que permite que una subclase o clase secundaria proporcione una implementación específica de un método que ya es proporcionado por una de sus superclases o clases principales. Cuando un método de una subclase tiene el mismo nombre, los mismos parámetros o firma y el mismo tipo de retorno (o subtipo) tiene un método en su superclase, se dice que el método de la subclase anula el método de la superclase.



La versión de un método que se ejecuta será determinada por el objeto que se utiliza para invocarlo. Si se usa un objeto de una clase principal (madre) para invocar el método, entonces se ejecutará la versión de la clase principal, pero si se usa un objeto de la subclase para invocar el método, se ejecutará la versión de la clase secundaria. En otras palabras, es el tipo de objeto al que se hace referencia (no el tipo de variable de referencia) lo que determina qué versión de un método sobreescrito se ejecutará.

```
# Definiendo clase principal
class Principal():

    def __init__(self):
        self.valor = "Dentro de Principal"

    # Metodo mostrar de Principal
    def mostrar(self):
        print(self.valor)

# Definiendo clase secundaria (sub clase)
class Secundaria(Principal):

    def __init__(self):
        self.valor = "Dentro de Secundaria"

    # Metodo mostrar de Secundaria
    def mostrar(self):
        print(self.valor)

obj1 = Principal()
obj2 = Secundaria()

obj1.mostrar()
obj2.mostrar()
```

Resultado

```
Dentro de Principal
Dentro de Secundaria
```

Sobrescritura de métodos con herencia múltiple

Acabamos de ver lo que es la herencia múltiple, ahora a través veamos cómo funciona la sobreescritura en este caso:

```
# Principal 1
class Principal1():

    def mostrar(self):
        print("Dentro de Principal1")

# Principal 2
class Principal2():

    def desplegar(self):
        print("Dentro de Principal2")

# Secundaria
class Secundaria(Principal1, Principal2):

    def mostrar(self):
        print("Dentro de secundaria")

obj = Secundaria()

obj.mostrar()
obj.desplegar()
```

Resultado:

```
Dentro de secundaria
Dentro de Principal2
```

Llamando al método de la clase principal dentro del método sobreescrito

Los métodos de la clase principal también se pueden llamar dentro de los métodos sobreescritos. Esto generalmente se puede lograr de dos maneras.

1.- Uso de **Classname**: los métodos de clase de los padres se pueden llamar mediante el método de Principal **nombre_clase.metodo** dentro del método sobreescrito.

```
class Principal():

    def mostrar(self):
        print("Dentro de Principal")

class Secundaria(Principal):

    def mostrar(self):
```

```
        # Llamando al metodo de la clasePrincipal
        Principal.mostrar(self)
        print("Dentro de Secundaria")

obj = Secundaria()
obj.mostrar()
```

Resultado:

```
Dentro de Principal
Dentro de Secundaria
```

2.- Usando **Super()**: La función Python **super()** nos proporciona la posibilidad de referirnos explícitamente a la clase madre. Es básicamente útil cuando tenemos que llamar a funciones de superclase. Devuelve el objeto proxy que nos permite hacer referencia a la clase principal por "super".

```
class Principal():

    def mostrar(self):
        print("Dentro de Principal")

class Secundaria(Principal):

    def mostrar(self):
        # Llamando al metodo de la clase Principal
        super().mostrar()
        print("Dentro de Secundaria")

obj = Secundaria()
obj.mostrar()
```

Resultado:

```
Dentro de Principal
Dentro de Secundaria
```

3.4.1.6. Utilizando la función isinstance()

La función **isinstance()** comprueba si el objeto (primer argumento) es una instancia o subclase de la clase **classinfo** (segundo argumento).

La sintaxis de isinstance () es:

```
isinstance(object, classinfo)
```

- Parámetros de **isinstance()**

isinstance() toma dos parámetros:

```
- object - objecto a examinar- classinfo
- clase, tipo, or tupla de clases y tipos
```

- Valor de retorno de **isinstance()**

isinstance() devuelve:

```
- True si el objeto es una instancia o subclase de una clase o
cualquier elemento de la tupla
- False de lo contrario
```

Si classinfo no es un tipo o una tupla de tipos, se genera una excepción **TypeError**.

Ejemplo:

```
class Foo:
    a = 5

instanciaFoo = Foo()

print(isinstance(instanciaFoo, Foo))
print(isinstance(instanciaFoo, (list, tuple)))
print(isinstance(instanciaFoo, (list, tuple, Foo)))
```

Resultado:

```
True
False
True
```

Otro ejemplo:

```
numeros = [1, 2, 3]

resultado = isinstance(numeros, list)
print(numeros,'instancia de list?', resultado)

result = isinstance(numeros, dict)
print(numeros,'instancia de dict?', resultado)

resultado = isinstance(numeros, (dict, list))
print(numeros,'instancia de dict o list?', resultado)

numero = 5

resultado = isinstance(numero, list)
print(numero,'instancia de list?', resultado)

resultado = isinstance(numero, int)
print(numero,'instancia de int?', resultado)
```

Resultado:

```
[1, 2, 3] instancia de list? True
[1, 2, 3] instancia de dict? True
[1, 2, 3] instancia de dict o list? True
5 instancia de list? False
5 instancia de int? True
```

3.4.2. Referencias.

[1] Mark Lutz, Python Pocket Reference, Fifth Edition 2014.

[2] Matt Harrison, Ilustrated Guide to Python3, 2017.

[4] Magnus Lie Hetland, Beginning Python: From Novice to Professional, 2017.

[5] Python Tutorial.
https://www.w3schools.com/python/