PythonBasicCourse-es004

October 12, 2022

1 Curso básico de Python

1.1 Apuntes

Curso básico de Python. Apuntes por Marcelo Horacio Fortino. Versión 2.3. Octubre 2022.

Versión	Autor/es	Fecha	Observaciones
1.0	Marcelo Horacio Fortino	2021/Marzo	Curso Python
1.1	Marcelo Horacio Fortino	2021/Junio	Convertido a markdown - ipynb
1.2	Marcelo Horacio Fortino	2021/Agosto	Actualizados contenidos
1.3	Marcelo Horacio Fortino	2021/Octubre	Agregado Flask microframework
1.4	Marcelo Horacio Fortino	2021/Noviembre	Agregado Pandas - datascience
1.5	Marcelo Horacio Fortino	2021/Diciembre	Agregado Devops - Ansible
2.0	Marcelo Horacio Fortino	2022/Abril	Nueva estructura: core / module
2.1	Marcelo Horacio Fortino	2022/Junio	Módulo apuntes intermedio
2.2	Marcelo Horacio Fortino	2022/Agosto	Actualizado temario y ejercicios
2.3	Marcelo Horacio Fortino	2022/Octubre	Actualizado despliegue a render.com

cluso sin la garantía MERCANTIL implícita o sin garantizar la CONVENIENCIA PARA UN PROPÓSITO PARTICULAR. El autor no asume ninguna responsabilidad si el lector hace un mal uso de la misma.

Estos apuntes se basan en: - La documentación oficial de Python, https://docs.python.org/es/3/tutorial/index.html, - La bibliografía presentada al final de este documento, y - Documentación propia recogida a lo largo de los años de diversas fuentes.

1.2 Objetivo del curso

- Objetivo general del curso:
- Aprender a usar Python para crear scripts y programas simples plenamente funcionales, desarrollar aplicaciones web y realizar análisis de datos.

- Objetivos específicos:

- Reconocer las características principales de Python y su utilidad práctica.
- Identificar tipos de datos (simples y compuestos) y operadores.
- Aplicar variables y estructuras de control de flujo.
- Construir funciones y clases (POO).
- Clasificar los módulos y paquetes por sus funcionalidades y objetivos.
- Comparar el lenguaje con otros similares de scripts, procedimentales y orientados a objetos.
- Probar bibliotecas para conexiones REST a aplicaciones web y bases de datos.
- Resolver problemas y errores en el código fuente proponiendo soluciones alternativas (refactoring).
- Utilizar la biblioteca pandas junto con matplotlib y numpy para realizar análisis estadísticos de datos y gráficos.

- Como resultado práctico al final del curso cada estudiante habrá creado una aplicación web utilizando el microframework de Python Flask.

1.3 Temario

- Introducción, instalación y compilación
- Datos, expresiones y sentencias
- Variables y funciones, control de flujo
- Clases y objetos, herencia, polimorfismo
- Entradas y salidas con Python
- Gestión de módulos, paquetes y bibliotecas
- Servicios y programas en red, REST API
- Desarrollo de aplicaciones web con Flask
- Análisis de datos con pandas, matplotlib y numpy

- Módulos opcionales
  - SQL ejemplos con pandas
  - Plotting con Python
  - Machine Learning con Python
  - DevOps con Ansible
  - SDK de GCP para Python
  - Kubernetes con Python y Docker

1.4 Bibliografía

- Downey, A., Elkner, J., Meyers, C. Aprenda a Pensar Como un Programador con Python. (2015).
  Recuperado de https://argentinaenpython.com/quiero-aprender-python/aprenda-a-pensar-como-un-programador-con-python.pdf
- Kent D. Lee. Python, Programming Fundamentals Second Edition. 2014.
- Marzal Varó, A., Gracia Luengo, I., García Sevilla, Pedro. Introducción a la programación con Python 3. (2014).
  Recuperado de http://repositori.uji.es/xmlui/handle/10234/102653
- Miller, B., Ranum, D. Solución de problemas con algoritmos y estructuras de datos usando Python. Traducido por Mauricio Orozco-Alzate, Universidad Nacional de Colombia - Sede Manizales.
  Recuperado de https://runestone.academy/ns/books/published/pythoned/index.html?mode=browsing
- Shaw, Z. A., Learn Python 3 the Hard Way. (2016).
  Recuperado de https://learnpythonthehardway.org/
- Van Rossum, G. and the Python development team. Documentación de Python en español. (2020).
  Recuperado de https://python-docs-es.readthedocs.io/es/3.10/

2 Operadores lógicos

- Conjunciones y disyunciones, Operadores bit a bit, Operadores de cambio, Variables escalares, Ordenamiento Burbuja, Compresión de listas.
- Clases y objetos en Python: Tipos de métodos, Polimorfismo, Herencia, Importar clases.

2.1 Conjunciones y disyunciones

- Python tiene operadores lógicos para construir conjunciones *and* (y) y disyunciones *or* (o).
- También cuenta con *not*, que es un operador unario que realiza una negación lógica (su prioridad es muy alta).
- Por su parte *and* es un operador binario con prioridad inferior a la expresada por los operadores de comparación.

[ ]:
```
contador = 1
valor = 10
contador > 0 and valor == 10
contador < 0 or valor == 10
```

- *or* es un operador binario de disyunción con una prioridad más baja que *and*.
- Leyes de De Morgan en Python
- https://es.wikipedia.org/wiki/Leyes_de_De_Morgan

[ ]:
```
# La negación de una conjunción es la separación de las negaciones

not (a and b) == (not a) or (not b)

# La negación de una disyunción es la conjunción de las negaciones

not (a or b) == (not a) and (not b)
```

- Los operadores lógicos toman sus argumentos como un todo, sin tener en cuenta la cantidad de bits.
- Los operadores solo conocen los valores:
  - cero: cuando todos los bits se restablecen, es *False*.
  - no cero: cuando se establece al menos un bit, es decir, *True*.
- Importante es aclarar que los operadores lógicos no ingresan al nivel de bits de su argumento.
- Solo trabajan con el valor entero final.

2.2 Operadores bit a bit

- Los operadores que permiten manipular bits de datos individuales se denominan operadores bit a bit.
- Además de la conjunción, disyunción y negación, existe un operador adicional llamado *xor* (significa *o exclusivo* ^).
- `&` (*ampersand*) - conjunción a nivel de bits.
- `|` (barra vertical) - disyunción a nivel de bits.
- `~` (tilde) - negación a nivel de bits.
- `^` (signo de intercalación) - o exclusivo a nivel de bits (*xor*).

[ ]:
```
# Operador lógico de negación.
# La variable logico_neg se establecerá en False

logico_neg = not valor

# Operador bit a bit de negación.

bit_neg = ~valor
```

[2]:
```
# la función bin devuelve en formato binario
# el número que se incluya como argumento
bin(7)
```

[2]:
```
'0b111'
```

[4]:
```
bin(4)
```

[4]:
```
'0b100'
```

- Con el operador `&` (*y* lógico) se comparan los bits de los números 7 y 4 aplicándose el operador *y*.
- Cuando ambos bits sean 1, se devuelve 1; en cualquier otro caso, 0.

[6]:
```
7 & 4
```

[6]: 4

[7]:
```
bin(7 & 4)
```

[7]: '0b100'

- Con el operador `|` (operador lógico *o*) devuelve 1 cuando alguno de los bits comparados es 1, y 0 cuando ambos bits son 0.

[8]:
```
4 | 7
```

[8]: 7

[9]:
```
bin(4 | 7)
```

[9]: '0b111'

2.3 Operadores de cambio

- Para los valores de número entero Python permite el *shifting* o desplazamiento:
  - Desplazamiento izquierdo binario `<<`
  - Desplazamiento derecho binario `>>`
- El argumento izquierdo de estos operadores es un valor entero cuyos bits se desplazan.

[ ]:
```
SISTEMA DECIMAL        SISTEMA BINARIO
0         0000
1         0001
2         0010
3         0011
4         0100
5         0101
6         0110
7         0111
8         1000
9         1001
10         1010
```

[10]:
```
# Desplazarse hacia la izquierda un bit
# es lo mismo que multiplicar números enteros por dos
4 << 1
```

[10]: 8

[11]:
```
bin(4 << 1)
```

[11]:
```
'0b1000'
```

[12]:
```
# Desplazarse hacia la derecha un bit
# equivale a la división de números enteros por dos
7 >> 1
```

[12]:
```
3
```

[13]:
```
# Hacia la derecha se pierden los bits que haya a la derecha del número
bin(7 >> 1)
```

[13]:
```
'0b11'
```

2.4 Variables escalares

- Las variables que pueden almacenar un valor cada vez se denominan escalares por analogía con las matemáticas.
- Si creamos una lista con cinco valores, todos ellos números, podemos decir que es una lista de longitud igual a cinco.
- La lista es una colección de elementos, pero cada elemento es un escalar.

[14]:
```
numeros = [1050, 455, 780, 920, 1001]
```

[ ]:
```
# Acceder al primer elemento de la lista mediante su índice
print(numeros[0])
# acceder al último elemento
print(numeros[-1])
```

[ ]:
```
# Verificar la longitud actual de la lista
print(len(numeros)
```

- Se pueden utilizar métodos para añadir elementos a una lista:

[ ]:
```
# Añade al final de la lista existente
numeros.append(745)

# Añade un nuevo elemento en cualquier lugar de la lista
numeros.insert(0, 745)
```

[ ]:
```
lista = []    # Creando una lista vacía

for i in range(5):
    lista.append(i + 1)

print(lista)
```

[ ]:
```
lista_inversa = []    # Una lista en orden inverso

for i in range(5):
    lista_inversa.insert(0, i + 1)

print(lista_inversa)
```

[ ]:
```
lista = [10, 1, 8, 3, 5] # Sumando los valores de una lista
total = 0

for i in range(len(lista)):
    total += lista[i]

print(total)
```

[ ]:
```
lista = [25, 71, 44, 59, 45]    # Cambiar los valores de la lista

lista[0], lista[4] = lista[4], lista[0]
lista[1], lista[3] = lista[3], lista[1]

print(lista)
```

2.5 Ordenamiento Burbuja

- Se puede ordenar la lista de dos maneras:
  - Ascendente: si en cada par de elementos adyacentes, el primer elemento no es mayor que el segundo.
  - Descendente: si en cada par de elementos adyacentes, el primer elemento no es menor que el segundo.
- Se comparan el primer y el segundo elemento:
  - Si están en el orden incorrecto (el primero es mayor que el segundo) se intercambian.
  - Si su orden es correcto, restan tal cual.

[ ]:
```
lista = [72, 100, 51, 13, 40]
# Necesario como verdadero (True) para ingresar al bucle while
intercambio = True

while intercambio:
    intercambio = False    # no hay intercambios hasta ahora
    for i in range(len(lista) - 1):
        if lista[i] > lista[i + 1]:
            intercambio = True    # se intercambian
            lista[i], lista[i + 1] = lista[i + 1], lista[i]

print(lista)
```

- La función *sorted* devuelve una copia de la lista, dejando el original intacto.

[ ]:
```
print(sorted(lista, reverse=True))
```

- Por su parte, con el método *sort* se puede cambiar el orden de la lista.

[ ]:
```
lista = [72, 100, 51, 13, 40]
lista.sort()
print(lista)
```

- El método *reverse()* invierte el orden de la lista.

[ ]:
```
lista = [72, 100, 51, 13, 40]
print(lista)

lista.reverse()
print(lista)
```

- El nombre de una variable ordinaria es el nombre de su contenido.
- El nombre de una lista es el nombre de una ubicación de memoria donde se almacena la lista.

[ ]:
```
lista = [72, 100, 51, 13, 40]
lista_2 = lista
lista[0] = 2
print(lista_2)
print(lista)
```

- Para copiar una lista:

[ ]:
```
lista_3 = lista.copy()
lista_3[0] = 20
print(lista_3)
```

- Para verificar si algunos elementos existen en una lista o no se utilizan las palabras clave *in* y *not in*:

[17]:
```
lista = ["x", "y", 1, 2, 3]

print("x" in lista)
print("z" not in lista)
print(2 not in lista)
```

```
True
True
False
```

2.6 Compresión de listas

- Una comprensión de lista (*list comprehension*) es en realidad una lista que se crea durante la ejecución del programa y no se describe de forma estática.
- Es un ejemplo de los conceptos de programación funcional aplicados en Python.

[ ]:
```
minusculas = ['no', 'me', 'gusta', 'el', 'spam']
mayusculas = [minusc.upper() for minusc in minusculas]
```

- La parte del código colocada dentro de los paréntesis especifica la cantidad de datos que se utilizarán para completar la lista.

[ ]:
```
duplicar_valor = [x * 2 for x in range(10)]
print(duplicar_valor)
```

[ ]:
```
# La potenciación se realiza con el operador **
potencias = [2 ** x for x in range(8)]
print(potencias)
```

[1]:
```
cuadrados = [2, 3, 4, 5]
impares = [x for x in cuadrados if x % 2 != 0 ]
print(impares)
```

```
[3, 5]
```

- Python también cuenta con la función *pow()*:

[ ]:
```
pow(4, 2) # Elevar al cuadrado
pow(4, 3) # Elevar al cubo
```

3 Clases y objetos en Python

- Python toma del paradigma de la programación estructurada el uso de estructuras de control de flujo, como por ej. declaraciones condicionales (*if, elif, else*), bucles (*while* y *for*), estructuras de bloque, y subrutinas.
- Del paradigma de la programación orientada a objetos (*OOP - Object Oriented Programming*) por su parte, toma el uso de objetos y sus clases junto con sus técnicas: polimorfismo, herencia, cohesión, abstracción, acoplamiento y encapsulamiento.
- Fuente: https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos.
- Todos los tipos de datos *built-in* en Python (*strings, integers*, y *lists*) son en realidad clases.
- Para Python todo es un objeto y una clase es como ese objeto se define, siendo que una clase también es considerada un objeto.
- Fuente: https://docs.python.org/es/3/tutorial/classes.html

[ ]:
```
>>> x = 5
>>> x.__class__
```

- Los objetos tienen comportamientos asociados, llamados métodos, dependiendo de su tipo.
- Las características de la clase se guardan en variables que a su vez, se denominan atributos.

[ ]:
```
class NombreDeLaClase:
    # Se utiliza la sentencia pass para evitar que Python
    # no genere un error por ser una clase vacía
```

```
    pass
```

- Las clases por convención se suelen escribir en singular y utilizando mayúscula en la primera letra de cada palabra, lo que se denomina en inglés *CamelCase*.
- Los módulos y objetos usan en cambio guion bajo (*snake_case*) para separar palabras.

- El método *__ *init* _()* es una función que es parte de la clase como cualquier otra, salvo que es ejecutado automáticamente cada vez que se crea un objeto de la clase.
- Es una especie de constructor, por lo tanto es necesario agregarlo al inicio de la clase.

- La variable *self* sirve para habilitar la disponibilidad de otros objetos y variables en cualquier lugar de la clase.
- Ella es pasada automáticamente a cada método que es llamado a través de un objeto.
- Es el primer argumento obligatorio de cualquier método.

[ ]:
```python
# Fichero festivalMusical.py
# Ejemplo de clase y creación de una instancia de la misma
class FestivalMusical:
    # Inicializa el método __init__ de la clase
    def __init__(self, nombre, pais, estilo_musical):
        # Inicializa los parámetros
        self.nombre = nombre
        self.pais = pais
        self.estilo_musical = estilo_musical
```

[ ]:
```python
# Se crea una instancia de la clase FestivalMusical
festival1 = FestivalMusical('Creamfields', 'UK', 'Dance')

# Se accede a los atributos del objeto
print(festival1.nombre)

# Se accede a la posición del objeto en memoria
print(festival1)
```

- Por otro lado la clase puede tener un comportamiento, el cual se guarda en una función llamada método.

[ ]:
```python
# Ejemplo de clase y creación de una instancia de la misma
class FestivalMusical:
    def __init__(self, nombre, pais, estilo_musical):
        self.nombre = nombre
        self.pais = pais
        self.estilo_musical = estilo_musical

    # Los métodos regulares toman la instancia como primer argumento
    def festival_metodo(self):
        print('El mejor festival es:', self)
        return self
```

[ ]:
```
# Se crea una instancia de la clase FestivalMusical
festival1 = FestivalMusical('Creamfields', 'UK', 'Dance')

# Se accede a los atributos del objeto
print(festival1.nombre)

# Se accede a la posición de los objetos en memoria
print(festival1)
print(FestivalMusical.festival_metodo(festival1))

# Se accede al método y a los atributos del objeto
print(FestivalMusical.festival_metodo(festival1.nombre.upper()))
```

- También se pueden modificar y borrar objetos y sus propiedades:

[ ]:
```
festival1.nombre = ("Primavera Sound")
print(festival1.nombre)
```

[2]:
```
# Eliminando el objeto
del festival1
print(festival1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-225d44be720f> in <module>
      1 # Eliminando el objeto
----> 2 del festival1
      3 print(festival1)

NameError: name 'festival1' is not defined
```

- Para ver los atributos de la instancia:

[ ]:
```
print(festival1.__dict__)
```

- Para ver los atributos y métodos de la clase:

[ ]:
```
print(FestivalMusical.__dict__)
```

- Para ver la ayuda de la clase:

[ ]:
```
print(help(FestivalMusical))
```

3.1 Tipos de métodos

- En Python se pueden dividir los métodos en tres tipos:
  - Métodos de instancia: *metodo()*,
  - Métodos de clase: *@classmethod*, y

  - Métodos estáticos: *@staticmethod*.

- *@classmethod* y *@staticmethod* son llamados en inglés *decorators*.
- Como se ha visto previamente en los ejemplos, los métodos de instancia pueden acceder y modificar los atributos del objeto, acceder a otros métodos, y pueden modificar el estado de la clase.
- Toman la instancia como primer argumento (*self*).
- Los métodos de clase no pueden acceder a los atributos de la instancia pero pueden modificar los atributos de la clase. Toman la clase como primer argumento (*cls*).
- Un método de clase puede ser llamado en la clase como *Clase.metodo())* o en la instancia como *Clase().metodo()*.

[ ]:
```python
# Los métodos de clase toman la clase como primer argumento
@classmethod
def valor_ticket(cls, valor):
    cls.valor_ticket = valor
    return print('El valor del ticket es:', valor)
```

[ ]:
```python
...
FestivalMusical.valor_ticket(100)

# Accede a la clase
print(f'Accede a la Clase: {FestivalMusical.valor_ticket}')
# Accede a la instancia
print(f'Accede a la instancia: {festival1.valor_ticket}')
```

- Finalmente los métodos estáticos no pueden modificar el estado de la clase o de la instancia.
- Trabajan como funciones normales pero se incluyen en la clase porque tienen alguna conexión lógica con la misma.
- Los métodos estáticos no toman ningún argumento.

[ ]:
```python
# Confirmar si el evento es un día de la semana o un fin de semana
# Los métodos estáticos no toman ningún argumento
@staticmethod
def dia_evento(dia):
    if dia.weekday() == 5 or dia.weekday() == 6:
        print('es un final de semana')
        return
    print('es un dia laboral')
    return
```

[ ]:
```python
import datetime
...
dia1 = datetime.date(2022, 5, 9)

# Ejemplo usando f-strings para acceder al objeto
print(f'El festival {festival1.nombre} se realizará el {dia1}')
```

```
# Accede al método estático
FestivalMusical.dia_evento(dia1)
```

- *Decorators* como *@staticmethod* y *list comprehensions* (`[x*x for x in range(10)]`) son considerados en python como azúcar sintáctico (*syntactic sugar*).
- https://es.wikipedia.org/wiki/Az%C3%BAcar_sint%C3%A1ctico

3.2 Polimorfismo

- Otro concepto importante de la *OOP* en Python es el polimorfismo que significa que objetos de diferentes clases pueden ser instanciados tomando una forma u otra dependiendo de la manera en que se accede a ellos.

- Es una característica que tienen los objetos (clases) que comparten atributos entre sí pero con valores distintos.
- El clásico ejemplo para entender el concepto es crear una clase "coche", allí podemos crear un objeto "motor", otro "carrocería", etc., definiendo el comportamiento general de esa categoría de objetos.

[ ]:
```
# Fichero coche.py
# Ejemplo de clase
class Coche:
    def __init__(self, marca, modelo, tipo):
        self.marca = marca
        self.modelo = modelo
        self.tipo = tipo
        self.capacidad_gasolina = 15
        self.nivel_gasolina = 0

    def gasolina_completo(self):
        self.nivel_gasolina = self.capacidad_gasolina
        print('El depósito de gasolina está lleno')

    def conducir(self):
        print(f'El {self.modelo} se está conduciendo.')


# Creando las instancias de la clase Coche
objeto_coche = Coche('SEAT', 'Ateca', '1.0')

# Acceder a los atributos de ese objeto
print(objeto_coche.marca, objeto_coche.modelo, objeto_coche.tipo)

# Llamando a los métodos de la clase
objeto_coche.gasolina_completo()
objeto_coche.conducir()
```

- Para verificar si la instancia pertenece a una clase:

[ ]:
```
print(isinstance(objeto_coche, Coche))
```

3.3 Herencia

- El concepto de herencia en las clases es fundamental en la *OOP*: Cuando una clase hereda de otra, tiene acceso a todos los atributos y métodos de la clase padre.
- Esto permite declarar nuevos atributos y métodos sobrescribiendo los de la clase superior.
- Además se puede heredar de clases que heredan de otras, lo que se denomina herencia múltiple.

[ ]:
```
# Herencia: extendiendo la clase
class CocheElectrico(Coche):
    # El método __init__() para crear una clase hija
    def __init__(self, marca, modelo, tipo):
        super().__init__(marca, modelo, tipo)
        self.tamanio_bateria = 100
        self.nivel_carga = 0

    # Agregar un nuevo método a la clase
    def cargar(self):
        self.nivel_carga = 100
        print('El coche está cargado.')

    # Sobrescribir un método del padre
    def gasolina_completo(self):
        print('El coche no tiene depósito de gasolina!')


# Usar el método padre e hijo
obj_coche_electrico = CocheElectrico('TESLA', 'Modelo 3', 'Berlina')

print(obj_coche_electrico.marca, obj_coche_electrico.modelo,
      obj_coche_electrico.tipo)
obj_coche_electrico.cargar()
obj_coche_electrico.conducir()

# Usar el método sobreescrito
obj_coche_electrico.gasolina_completo()
```

- El método *issubclass* verifica si una clase hereda de otra:

[ ]:
```
print(issubclass(CocheElectrico, Coche))
```

- Para verificar la herencia en las clases se utiliza el MRO (*Method Resolution Order*):

[ ]:
```
# Mostrar la o las clases de la cual se está heredando
print(CocheElectrico.__mro__)
```

3.4 Importar clases

- Una forma de mantener el código simple y ordenado evitando redundancia es guardar las clases como ficheros separados y después importarlas cuando sea necesario.
- Esto se realiza como con cualquier biblioteca o módulo de Python.

[ ]:
```
# el fichero es festivalMusical.py
Import festivalMusical
```

[ ]:
```
# Importando solamente un módulo de la clase
From festivalMusical import MusicFestival
```

[ ]:
```
# Importando el módulo y creando un alias
import festivalMusical as fm
```

- Para ver los métodos del módulo:

[ ]:
```
print(dir(FestivalMusical))
print(dir(festival1))
```