PythonBasicCourse-es003

October 11, 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 Variables y funciones, control de flujo

- Variables, Variables locales, globales y no locales, Funciones, Args y kwargs en una función, Funciones internas, Función input, Funciones productivas y/o nulas, Funciones lambda, Control de flujo, Operadores de igualdad, La sentencia IF, Asignar funciones, La sentencia WHILE, La sentencia FOR, Sentencias break, continue y else en bucles, Switch / case en Python, Módulos.

2.1 Variables

- Una variable es un nombre que apunta a un valor guardado en la memoria.
- En Python, a diferencia de otros lenguajes, no tienen asociado un tipo y no es necesario declararlas antes de usarlas, lo que se denomina *tipado dinámico*.
- Los nombres de las variables pueden contener letras y números pero no pueden comenzar con un número.
- Se pueden usar mayúsculas pero es convencional usar solo minúsculas y separar las palabras con guiones bajos, lo que se denomina en inglés *snake_case*.
- No están permitidos los caracteres especiales (&, %, etc.) ni los nombres reservados de Python (*class, def, if*, etc.).
- Las mayúsculas y minúsculas se tratan de forma distinta.
- La guía de estilo PEP 8 de Python (https://peps.python.org/pep-0008/) recomienda la siguiente convención de nomenclatura para variables y funciones en Python:

  - Nombres de las variables en minúsculas y con palabras separadas por guiones bajos (variable, mi_variable).
  - Idéntica convención para las funciones (funcion, mi_función).

- Algunos ejemplos:

[ ]:
```
lenguaje = 'Python'
x = 3.14
y = 3 + 2
# Asigación múltiple
a1, a2 = 1, 2
# Intercambio de valores
a1, a2 = a2, a1
# Valor no definido
x = None
del x
```

[ ]:
```
# Operadores de asignación compuesta (operadores abreviados)
# Incremento (equivale a x = x + 2)
x = 10
x += 2
# Decremento (equivale a x = x - 1)
x -= 1
```

[ ]:
```
# Ejemplos de asignación compuesta
y = 10
x = x + 2 * y
x += 2 * y
```

[ ]:
```
var = 50
var = var / 2
var /= 2
var = var % 10
var %= 10
```

[ ]:
```
rem = 4
y = y - (x + var + rem)
y -= (x + var + rem)
x = x ** 2
x **= 2
```

- La prioridad de los operadores en Python es la siguiente:
- https://docs.python.org/es/3/reference/expressions.html#operator-precedence

[ ]:
```
"""
A partir de Python 3.6 se utiliza un F string (formatted string literals)
para dar formato e incluir variables y valores dentro de una cadena
"""
```

```
nombre_variable = "valor"
print(f'El texto y el {nombre_variable}')
```

[ ]:
```
"""
 Se puede imprimir texto junto con una variable
 utilizando el operador + o una coma
"""
nombre = 'María'
print('Mi nombre es ' + nombre)
print('Mi nombre es', nombre)
```

[ ]:
```
# Agrega puntos a un valor
total = 158000000
print(f'El monto total es {total:,} de euros'.replace(',','.'))
```

[2]:
```
# Agrega decimales y formato de moneda
total_decimales = 158000000.57462
print(f'El monto total es: {total_decimales:,.2f} €')
```

```
El monto total es: 158,000,000.57 €
```

[ ]:
```
# Realiza una operación matemática
minutos = 2.5
print(f'El proceso finalizará en {minutos * 60} segundos')
```

2.2 Formato de texto

[ ]:
```
nombre = "Juan"
edad = 23

# Concatenar variables y cadena de texto
print("Mi nombre es "+nombre+" y tengo "+str(edad) + " años.")

# F-strings (Python 3.6+)
print(f"Mi nombre es {nombre} y tengo {edad} años.")
```

[ ]:
```
# Módulo
print("Mi nombre es %s y tengo %d años." % (nombre, edad))

# Metodo format (Python 2 y 3)
print("Mi nombre es {} y tengo {} años.".format(nombre, edad))

# Metodo join
print(''.join(["Mi nombre es ", nombre, " y tengo ", str(edad), " años."]))
```

- Documentación: https://docs.python.org/es/3/library/string.html.

2.3 Variables locales, globales y no locales

- Python distingue tres tipos de variables:
  - las variables locales,
  - las variables libres globales, y
  - las variables no locales.
- Las variables locales pertenecen al ámbito de la subrutina,
- las variables globales pertenecen al ámbito del programa principal, y
- las variables no locales son las que pertenecen a un ámbito superior al de la subrutina, pero que no son globales.
- Para identificar a las variables globales y no locales se utilizan las palabras reservadas *global* y *nonlocal*.
- La variable local por otro lado no necesita ser identificada.
- Para una mejor comprensión del tema podemos ejecutar el código en la aplicación web http://pythontutor.com/visualize.html.

[ ]:
```
# Ejemplo de variable local
def subrutina():
    variable = 10
    print(variable)
    return

variable = 20
subrutina()
print(variable)
subrutina()
```

[1]:
```
# Acceder desde fuera de la función a la variable da error
def subrutina():
    variable_local = 10
    print(variable_local)
    return

subrutina()
print(variable_local)
```

```
10
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-4b1b75bbb9f0> in <module>
      6
      7 subrutina()
----> 8 print(variable_local)

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

- Utilizar la palabra reservada *global* dentro de una función con el nombre o nombres de las variables separados por comas hace que la función emplee la variable que se puede acceder desde el exterior.

[ ]:
```
# Ejemplo variable global
def subrutina():
    global var_global
    print(var_global)
    var_global = 10
    return

var_global = 5
subrutina()
print(var_global)
```

[ ]:
```
# Ejemplo variable nonlocal
def subrutina():
    def sub_subrutina():
        nonlocal x
        print(x)
        x = 10
        return

    x = 20
    sub_subrutina()
    print(x)
    return

x = 30
subrutina()
print(x)
```

- Comunidad de Python para despejar dudas: https://www.pythondiscord.com/
- Stackoverflow: https://stackoverflow.com/questions/tagged/python

2.4 Funciones

- Una función es un bloque de código (secuencia de enunciados) que se ejecuta únicamente cuando es llamada.
- Se le puede pasar parámetros y la función devolverá datos como resultado.

- Para definir una función en Python se utiliza la palabra reservada *def*, incluyendo los argumentos entre paréntesis y usando sangría respecto a la primera línea.
- Aunque se pueda indicar el final de la función con la palabra *return*, esto no es obligatorio, basta salir de la sangría.
- Los paréntesis vacíos después del nombre indican que no tiene ningún argumento.
- Dentro de la función, los argumentos son asignados a variables llamadas parámetros.

[ ]:
```python
# Ejemplo función
# Pregunta el nombre y saluda al usuario
def preguntar_nombre():
    nombre = input("Cuál es tu nombre? ")
    print("Hola", nombre)
    return

print(preguntar_nombre())
```

- Crear funciones en programación tiene una serie de ventajas:
  - Hacen un programa más pequeño ya que eliminan código repetido.
  - Ayudan a eliminar errores al hacer *tests* específicos para las mismas.
  - Se pueden reutilizar en varios programas.
- Python en este sentido sigue el principio de desarrollo de software *DRY (Don't repeat yourself)*.
- https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

2.5 Args y kwargs en una función

- `*args` y `**kwargs` en una función se usan para pasar argumentos posicionales.
- https://python-intermedio.readthedocs.io/es/latest/args_and_kwargs.html.

[ ]:
```python
"""
Ejemplo de función admitiendo valores.
Como argumento de entrada utiliza *args ya
que se desconoce la cantidad de los mismos
"""
def calcular_media(*args):
    total = 0
    for i in args:
        total += i
    resultado_media = total / len(args)
    return resultado_media

a, b, c = 5, 15, 10
media = calcular_media(a, b, c)

print(f"La media de {a}, {b} y {c} es: {media}")
print("Programa terminado")
```

[ ]:
```python
# Ejemplo de función admitiendo pares de valores
# Como argumento de entrada utiliza **kwargs
def test_kwargs(**kwargs):
    for clave, valor in kwargs.items():
        print("{0} = {1}".format(clave, valor))


kwargs = {"tres": 3, "cinco": 5}
```

```
test_kwargs(**kwargs)
```

2.6 Funciones internas

- Python cuenta con funciones internas (*built-in functions*) y tipos como por ejemplo:
  - *type* (devuelve el tipo de dato),
  - *id* (identidad),
  - *input* (cadena del texto leído del teclado),
  - *int* (convierte a entero el valor), y
  - *print* (imprime el valor en pantalla).
- El listado completo se encuentra en https://docs.python.org/es/3/library/functions.html.
- Las siguientes funciones convierten un dato de un tipo en otro, siempre y cuando la conversión sea posible:

[ ]:
```
int() # convierte a entero
int('12') 12
int(True) 1
int('c') Error
```

[ ]:
```
float() # convierte a real
float('3.14') 3.14
float(True) 1.0
float('III') Error
```

[ ]:
```
str() # convierte a cadena
str(3.14) '3.14'
str(True) 'True'
```

[ ]:
```
bool() # convierte a valor lógico
bool('0') False
bool('3.14') True
bool('') False
bool('Hola') True
```

2.7 Función input

- La función input() obtiene datos de la consola, lo contrario de la función print(), que envía datos a la consola.
- Tiene un parámetro inicial: un mensaje de tipo cadena para el usuario.

[ ]:
```
nombre = input("Ingresa tu nombre: ")
print("Hola, " + nombre + ". ¡un placer conocerte!")

# Puede ser utilizada para finalizar el programa
print("\nPresiona la tecla Entrar para finalizar el programa.")
```

```
input()
print("FIN.")
```

- El resultado de la función input() es una cadena.

[ ]:
```
num_1 = int(input("Ingresa un número entero: "))
num_2 = int(input("Ingresa el segundo número entero: "))

print(num_1 + num_2)
```

En el modo interactivo, el intérprete de Python imprime puntos (…) para mostrar que la definición no está completa:

```
>>> def MiFuncion():
...     print("lo que sea")
...     print("más")
...
```

- Para salir de la función se ingresa una línea vacía.

2.8 Funciones productivas y/o nulas

- Las funciones que devuelven un valor son llamadas funciones productivas (*fruitful functions*), aquellas que no devuelven ningún valor se denominan nulas (*void functions*).
- Si por ejemplo se intenta asignar una función nula a una variable se mostrará el valor *None*.

[5]:
```
# Ejemplo de función nula
def funcion_nula():
    print("Python")
    return
```

- Cuando se llama a una función productiva, casi siempre se quiere hacer algo con el resultado, por ejemplo, asignarla a una variable o usarla como parte de una expresión.

[ ]:
```
# Ejemplo de función productiva
def funcion_prod(a,b):
    print("suma de a y b", a + b)
    return

funcion_prod(5,2)
```

2.9 Funciones lambda

- En Python las funciones *lambda* son simplemente una manera corta de definir una función:
  - No tienen un nombre
  - No pueden contener declaraciones
- https://docs.python.org/3/faq/design.html#why-can-t-lambda-expressions-contain-statements

[ ]:
```
# Ejemplo de función
def multiplicar(x, y):
    return x * y

print(multiplicar(5,5))
```

[ ]:
```
# Ejemplo función lambda
multiplicar = lambda x, y: x * y
print(multiplicar(5, 5))
```

[2]:
```
# Ejemplo lambda dentro de una expresión - Adaptado de PEP 498
print(f'{(lambda x, y: x * y)(5, 5)}')
```

25

2.10 Control de flujo

- El control de flujo de un programa es el orden en que el código se ejecuta.
- Esto se realiza mediante declaraciones condicionales (*conditional statements*), bucles (*loops*) y llamadas a funciones (*function calls*).
- En Python se utilizan las denominadas sentencias compuestas con construcciones de control de flujo tradicionales tipo *if, while* y *for*; la sentencia *with* que permite la ejecución del código dentro de un bloque de código; y las definiciones de función y clase.

2.11 Operadores de igualdad

- Repasemos primero los operadores de igualdad:
- `=` es un operador de asignación.
- `==`es un operador de igualdad binario con enlazado del lado izquierdo. Compara dos valores.
- `=!` el operador *no es igual a* si los valores no son iguales, el resultado es True.
- `>` y `<` son operador estrictos de igualdad.
- `>=` y `<=` son operadodes no estrictos y tienen prioridad mayor que `==` y `!=`.
- `<` *menor que* y su versión no estricta: `<=` *menor o igual que*.
- Todos son operadores binarios con enlazado del lado izquierdo.

[ ]:
```
# Programa que recibe un entero como entrada, e imprime
# False si es menor que 100 y mayor o igual que 500.

var = int(input("Escribe un número entero: "))
print(int(var) < 100 or int(var) >= 500)
```

2.12 La sentencia IF

- Python cuenta con una instrucción condicional (o sentencia condicional) para establecer condiciones.
- Para aplicar una condición lógica dentro de nuestro código se utiliza como en otros lenguajes de programación la condición *if*:

[ ]:
```
if verdadero_o_no:
    hacer_esto_si_verdadero
```

[ ]:
```
if condicion_verdadera_o_falsa:
    hacer_si_condicion_verdadero
else:
    hacer_si_condicion_falsa
```

[ ]:
```
# Se obtienen dos números
numero1 = int(input("Ingresa el primer número: "))
numero2 = int(input("Ingresa el segundo número: "))

# Se elige el número mayor
if numero1 > numero2:
    numero_mayor = numero1
else:
    numero_mayor = numero2

print("El número mayor es:", numero_mayor)
```

- Se puede también utilizar esta instrucción condicional anidada:

[ ]:
```
password = input("Ingresa la clave: ")

if len(password) >= 8:
    print("Clave verificada.")

    if password == "12345678":
        print("Clave correcta.")
    else:
        print("Clave incorrecta.")
else:
    password != "12345678"
    print('Tu clave tiene menos de 8 caracteres o es incorrecta.')
```

- Para verificar más de una condición se utiliza *elif*, que se detiene en la primera sentencia verdadera.

[ ]:
```
# Se obtienen dos números
numero1 = int(input("Ingresa el primer número: "))
numero2 = int(input("Ingresa el segundo número: "))

# Se elige el número mayor
if numero1 == numero2:
    print("Los números tienen idéntico valor")
elif numero1 > numero2:
    numero_mayor = numero1
else:
```

```
    numero_mayor = numero2
    print("El número mayor es:", numero_mayor)
```

- *else* no es obligatorio.
- Pero si no hay un *else* puede que no se ejecute ninguna de las opciones disponibles.
- *else* siempre es la última sentencia.
- Siempre le debe preceder un *if*.

[ ]:
```
x = 100
y = 500
if y > x:
    print(f'{y}, (y) es mayor que {x} (x)')
elif x == y:
    print(f'{x}, (x) e (y) {y} tienen el mismo valor')
else:
    print(f'{x} (x) es mayor que {y} (y)')
```

2.13 Asignar funciones

- Se puede asignar el resultado de una función a una variable:

[7]:
```
def deseos():
    return "¡Que tengas un buen día!"

d = deseos()
print(d)
```

```
¡Que tengas un buen día!
```

[1]:
```
def deseos():
    print("Mis deseos")
    return "Que tengas un buen día"

deseos()
```

```
Mis deseos
```

[1]:
```
'Que tengas un buen día'
```

[9]:
```
def deseos():
    print("Mis deseos")
    return "Que tengas un buen día"

print(deseos())
```

```
Mis deseos
Que tengas un buen día
```

- Para obtener una lista como resultado de la función:

[ ]:
```
def crear_lista(n):
    lista = []
    for i in range(n):
        lista.append(i)
    return lista


print(crear_lista(5))
```

- Para utilizar una lista como argumento de una función:

[ ]:
```
def hola(lista):
    for nombre in lista:
        print("Buen día", nombre)


hola(["María", "Juan", "Marta"])
```

2.14 La sentencia WHILE

- La sentencia *while* en Python itera sobre un bloque de código mientras que la condición sea verdadera.
- Una instrucción o conjunto de instrucciones ejecutadas dentro de *while* se llama el cuerpo del bucle.
- Un bucle infinito, también denominado bucle sin fin se repite indefinidamente.

[ ]:
```
i = 1
while i <= 1000:
    print(i)
    i = 5 * i + 1
print("Programa terminado")
```

[ ]:
```
numero_mayor = -999999999

numero = int(input("Introduce un número entero o escribe 0 para detener: "))

while numero != 0:
    if numero > numero_mayor:
        numero_mayor = numero
    numero = int(input("Introduce un número entero o escribe 0 para detener: "))

print(f"El número mayor es: {numero_mayor}")
```

- Ejemplo de como escribir utilizando *while* una parte de la serie de Fibonacci (https://en.wikipedia.org/wiki/Fibonacci_number) en Python:

[ ]:
```
# Fibonacci series:
# the sum of two elements defines the next
```

```
a, b = 0, 1
while a < 1000:
    print(a)
    a, b = b, a + b
```

- `while numero != 0:` es equivalente a `while numero:`

[ ]:
```
contador = 5

while contador != 0:
    print("Dentro del bucle.", contador)
    contador -= 1

print("Fuera del bucle.", contador)
```

[ ]:
```
# Ejecuta el bucle exactamente diez veces.
i = 0
while i < 10:
    print("Hola mundo!")
    i += 1
```

2.15 La sentencia FOR

- La sentencia *for* itera sobre los ítems de una lista o una cadena de texto en el orden que aparecen en la secuencia.

[ ]:
```
frutas = ['banana', 'manzana','pera', 'naranja']
for f in frutas:
    print("La palabra", f, "tiene", len(f), "letras")
```

- Para iterar sobre una secuencia de números en cambio se utiliza la función *range()*:

[ ]:
```
for i in range(10):
    print("Imprime del 0 al 9 ",i)
```

[ ]:
```
for i in range(5, 10):
    print("Imprime de 5 a 9", i)
```

[ ]:
```
for i in range(0, 50, 3):
    print("Imprime de 0 a 9, de a tres unidades", i)
```

- Para ver las distintas técnicas de iteración con *for*, un buen recurso es el tutorial de python 5.6 https://docs.python.org/es/3/tutorial/datastructures.html#looping-techniques.

2.16 Sentencias break, continue y else en bucles

- La sentencia *break*, termina el bucle *for* o *while*.
- La sentencia *continue* por otro lado, continúa con la siguiente iteración del ciclo.

- "Las sentencias de bucle pueden tener una cláusula`!else` que es ejecutada cuando el bucle termina, después de agotar el iterable (con *for*) o cuando la condición se hace falsa (con *while*), pero no cuando el bucle se termina con la sentencia *break*."
- Extraído de:
  https://docs.python.org/es/3/tutorial/controlflow.html#
  break-and-continue-statements-and-else-clauses-on-loops.

[ ]:
```
# Ejemplo break
# Verifica si un número es primo
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equivale a', x, '*', n//x)
            break
    else:
        # El bucle sigue sin encontrar un factor
        print(n, 'es un número primo')
```

[ ]:
```
# Ejemplo continue
for num in range(2, 10):
    if num % 2 == 0:
        print("Encontrado un número par", num)
        continue
        # Con break sale de la iteración al encontrar el primer número par
    print("Encontrado un número impar", num)
```

2.17 Switch / case en Python

- En Python no existe un constructor tipo *switch* o *case* como en otros lenguajes de programación.
- Lo que utiliza para reemplazarlos es el uso de diccionarios.

[ ]:
```
# Ejemplo simulando el constructor switch
def semana(i):
    dias={
            0:'Domingo',
            1:'Lunes',
            2:'Martes',
            3:'Miércoles',
            4:'Jueves',
            5:'Viernes',
            6:'Sábado'
        }
    return dias.get(i,"No corresponde a un día de la semana")


print(semana(0))
```

```
print('***')
print(semana(10))
```

- A partir de Python 3.10 se han implementado las PEP 634 y 635 para coincidencia de patrones estructurales (*Match-Case*).
- Fuente: https://docs.python.org/es/3/whatsnew/3.10.html.
- La PEP 636 contiene un tutorial con esta nueva funcionalidad https://peps.python.org/pep-0636/.

[ ]:
```
# Ejemplo adaptado de Match-Case PEP 636
def http_status(cod_estado):
    match cod_estado:
        case 301:
            return "Se movió permanentemente"
        case 400:
            return "Mala solicitud"
        case 401 | 403:
            return "No autorizado o Prohibido"
        case 404:
            return "No encontrado"
        case _:
            return "Otro error"

print(http_status(450))
```

2.18 Módulos

- La "biblioteca Python" (*Python library*) contiene las funciones y excepciones anteriormente mencionadas.
- Igualmente, la mayor parte de su contenido son los módulos que se pueden importar al programa que se está desarrollando para ampliar sus funcionalidades.
- Un módulo se puede considerar una biblioteca de código disponible para incorporar al programa.
- En líneas generales un módulo define funciones, clases y variables; y un script las ejecuta, pero esto no es taxativo.
- Un ejemplo puede ser el módulo *math* (matemáticas) que proporciona numerosas funciones matemáticas:

[1]:
```
# Se importa todo el módulo
import math


raiz_cuadrada = math.sqrt(10)
logaritmo_base10 = 20*math.log10(2)

print(raiz_cuadrada)
print(logaritmo_base10)
```

```
3.1622776601683795
6.020599913279624
```

[ ]:
```python
# Se importa solo algunas funciones del módulo
from math import sqrt, log10


raiz_cuadrada = sqrt(10)
logaritmo_base10 = 20*log10(2)
```

[3]:
```python
# Devuelve el logaritmo natural de un número
print(math.log(10))
```

```
2.302585092994046
```