Curso Intensivo Python - Día 3

Gabriel Valenzuela

@elestudianteclases

Verano 2021

Agenda

- Los temas a ver el día de hoy comprenden:
  - Funciones
  - Archivos y Excepciones
  - Métodos sobre cadenas
  - Clases y objetos: Introducción

f(x)

- Una función es un grupo de sentencis que existen con un programa para llevar a cabo una tarea específica.

f(x)

- Una función es un grupo de sentencis que existen con un programa para llevar a cabo una tarea específica.

- El porque de usar funciones se basa en **dividir y conquistar**. Esto es el principio de *Single responsability*

f(x)

- Una función es un grupo de sentencis que existen con un programa para llevar a cabo una tarea específica.
- El porque de usar funciones se basa en **dividir y conquistar**. Esto es el principio de *Single responsability*
- Üna función o método debe hacer una sola cosa"

f(x)

- Las funciones pueden venir de dos sabores: **void functions y value-returning functions**

f(x)

- Las funciones pueden venir de dos sabores: **void functions y value-returning functions**

- Las del primer tipo, simplemente ejecutan las sentencias que contiene y termina

f(x)

- Las funciones pueden venir de dos sabores: **void functions y value-returning functions**
- Las del primer tipo, simplemente ejecutan las sentencias que contiene y termina
- Las del segundo tipo, retorna un valor cuando termina de llamarse.

f(x)

- El código para una función se conoce como **definición de la función**

f(x)

- El código para una función se conoce como **definición de la función**
- La ejecución de la misma se realiza cuando **se llama a la función**

f(x)

- El código para una función se conoce como **definición de la función**
- La ejecución de la misma se realiza cuando **se llama a la función**
- El nombre de la función sigue la misma convención que las variables.

Definición y llamada

```
1  def funcionTipoUno(): #¡Ojo! Aqui van dos puntos y hay que indentar
2      #Diremos Tipo 1 a la función que no recive ni devuelve nada
3      print('Hola, no recibo ni devuelvo nada')
4
5  def funcionTipoDos(mensaje):
6      #Diremos Tipo 2 a la función que recive N args ni devuelve nada
7      print('Hola, yo recibo {0}'.format(mensaje.upper()))
8
9  def funcionTipoTres():
10     #Diremos Tipo 3 a la función que no recive nada pero devuelve algo
11     print('Voy a devolver algo...')
12     return 'algo'
13
14 def funcionTipoCuatro(mensaje):
15     #Diremos Tipo 4 a la función que recive N args y devuelve algo
16     print('Hola, yo recibo {0}'.format(mensaje.upper()))
17     return 'algo'.lower()
18
19 #Llamadas a las funciones
20 mensaje = 'Mi MeNsAjE'
21 funcionTipoUno()
22 funcionTipoDos(mensaje)
23 mensaje = funcionTipoTres()
24 mensaje = funcionTipoCuatro(mensaje)
25 print(mensaje)
```

f(x)

- Las variables en las funciones tienen una tratativa especial, se denominan **variables locales**. Estas variables son creadas **dentro** de la función

f(x)

- Las variables en las funciones tienen una tratativa especial, se denominan **variables locales**. Estas variables son creadas **dentro** de la función

- Y **No** pueden ser accedidas desde afuera de la función. Diferentes funciones pueden tener los mismos nombres de variables locales, dado que no pueden verse entre sí.

Variable local

```
1 def main():
2     obtenerNombre()
3     print('Hola {0}'.format(nombre)) #¡Error!
4
5 def obtenerNombre():
6     nombre = input('¿Como te llamas?')
7
8 main()
```

f(x)

- Algunas veces resulta útil no solo llamar a una función, sino enviar información, como vimos en la función Tipo 2 y Tipo 4.

f(x)

- Algunas veces resulta útil no solo llamar a una función, sino enviar información, como vimos en la función Tipo 2 y Tipo 4.

- Un **argumento** es cualquier pieza de dato que se pasa dentro a una función cuando se la llama. En cambio, un **parámetro** es una variable que recibe ese argumento pasado.

f(x)

- Algunas veces resulta útil no solo llamar a una función, sino enviar información, como vimos en la función Tipo 2 y Tipo 4.

- Un **argumento** es cualquier pieza de dato que se pasa dentro a una función cuando se la llama. En cambio, un **parámetro** es una variable que recibe ese argumento pasado.

- Se puede pasar más de un argumento a una función. Los argumentos son pasados por **asignación**, es decir, el parámetro pasado es una referencia a un objeto, pero la referencia es pasada por valor (copia)

f(x)

- Hay que tener en cuenta que algunos tipos de datos son **muteables**, pero otros no.

f(x)

- Hay que tener en cuenta que algunos tipos de datos son **muteables**, pero otros no.

- Por lo que si pasamos un **objeto muteable** a un método, este consigue una referencia al mismo objeto y puede cambiarse. En cambio, si pasamos un **objeto inmutable**, no podemos modificar la referencia.

Argumentos

```
1  def tratarModificarLista(lista):
2      print('Obtuve',lista)
3      lista.append(4)
4      print('Ahora es',lista)
5
6  listaOriginal = list(range(3))
7  print('Previo a llamarse',listaOriginal)
8  tratarModificarLista(listaOriginal)
9  print('Luego de modificarse',listaOriginal)
10
11 def usamosParametro(lista):
12     print('Obtuve',lista)
13     lista = list(range(5))
14     print('Ahora es',lista)
15
16 listaOriginal = list(range(3))
17 print('Previo a llamarse',listaOriginal)
18 usamosParametro(listaOriginal)
19 print('Luego de modificarse',listaOriginal)
```

f(x)

- Las funciones pueden definirse en un **módulo**, que es simplemente un archivo Python que contiene el código.

Modulos

```
1 import math as m
2
3 def area(radio):
4     return m.pi * radio**2
5
6 def circunferencia(radio):
7     return 2*m.pi*radio
```

Modulos

```
1  import circulo as c
2
3  def main():
4      for r in range(1,5):
5          print('Circulo de radio {0},tiene area {1:.2f} y circunferencia {2:.2f}
6              r,
7              c.area(r),
8              c.circunferencia(r)
9              ))
10
11 main()
```

archivo.txt

- Cuando un programa necesita guardar información para un uso posterior, o porque cumple su objetivo, escribe dicha información en un archivo

archivo.txt

- Cuando un programa necesita guardar información para un uso posterior, o porque cumple su objetivo, escribe dicha información en un archivo

- Así mismo, la información puede leerse desde un archivo para llevar a cabo operaciones de un programa.

archivo.txt

- Cuando un programa necesita guardar información para un uso posterior, o porque cumple su objetivo, escribe dicha información en un archivo
- Así mismo, la información puede leerse desde un archivo para llevar a cabo operaciones de un programa.
- Python tiene dos tipos de archivos: **texto y binario**. Un archivo de *texto* contiene información codificada en texto plano como ASCII o Unicode mientras que un *binario* los datos no son convertidos a texto (Solo un programa puede leerlo)

Trabajando con archivos I

```
1  def leerArchivo(ubicacion_nombre):
2      archivo = open(ubicacion_nombre,"r")
3      #open(path,modo) permite abrir el archivo en diversos modos
4      #r -> Lectura            w-> Escritura            a-> Añadir
5      #rb -> Lectura binario  wb-> Escritura binario
6      #El path en Windows debe hacerse con r'LocacionArchivo'
7      todo_contenido = archivo.read()
8      print('Todo el contenido:')
9      print(todo_contenido)
10     print('Se imprime linea por linea:')
11     archivo.seek(0) #Se resetea el valor de la posicion de lectura
12     for linea in archivo:
13         print(linea,end='')
14     archivo.seek(0)
15     print('\nLectura mediante while y readline():')
16     linea = archivo.readline() #Se lee hasta un '\n'
17     while (linea != ''):
18         print(linea,end='')
19         linea = archivo.readline()
20     archivo.close() #FUNDAMENTAL ¡NO OLVIDARSE DE CERRAR EL ARCHIVO!
21
22 def main():
23     archivo = "archivo.txt"
24     leerArchivo(archivo)
25     print('\nFin programa')
```

Trabajando con archivos II

```
1  import math as m
2
3  def leerArchivoContextManager(ubicacion_nombre):
4      with open(ubicacion_nombre,"r") as archivo:#El context manager automaticamte
5          #cierra el archivo
6          for linea in archivo:
7              print(linea,end='')
8
9  def escribirArchivo(ubicacion_archivo,nuevo_append,lista_datos):
10     modo = "w" if nuevo_append == True else "a"
11     archivo = open(ubicacion_archivo,modo)
12     for dato in lista_datos:
13         archivo.write(dato)
14     archivo.close()
```

Trabajando con archivos II

```
15 def main():
16     archivo = "archivo.txt"
17     leerArchivoContextManager(archivo)
18     print('\nAhora se añaden datos...')
19     datos = [
20         '\n***\n',
21         'Esto es la nueva linea y debe añadirse\n',
22         'Siempre debe ser una cadena\n',
23         '¡Ojo!\n',
24         'Para Python pi='+str(m.pi)+'\n'
25         ]
26     escribirArchivo(archivo,False,datos)
27     print('Se lee con nueva funcion:')
28     leerArchivoContextManager(archivo)
29     print('\nFin programa')
30 main()
```

archivo.txt

- Cuando se escribe un archivo que ya existe, su contenido se borra completamente

archivo.txt

- Cuando se escribe un archivo que ya existe, su contenido se borra completamente

- Al leer un archivo, siempre se toma el **n** por lo que se pueden usar funciones de cadenas como *strip()*

archivo.txt

- Cuando se escribe un archivo que ya existe, su contenido se borra completamente
- Al leer un archivo, siempre se toma el **n** por lo que se pueden usar funciones de cadenas como *strip()*
- Los archivos binarios son útiles para la serialización de objetos...

archivo.txt

archivo.txt

- Si pensamos al objeto (lista,conjunto, objeto definido por el desarrollador,etc) como un **flotador**, el mismo es útil cuando está inflado

archivo.txt

- Si pensamos al objeto (lista,conjunto, objeto definido por el desarrollador,etc) como un **flotador**, el mismo es útil cuando está inflado

- Cuando se desifla, sigue siendo en este caso un pato, pero ocupa menos espacio

archivo.txt

- En Python esto se llama **pickling**

archivo.txt

- En Python esto se llama **pickling**
- Con **pickle.load(archivo)** se carga el objeto

archivo.txt

- En Python esto se llama **pickling**
- Con **pickle.load(archivo)** se carga el objeto
- Con **pickle.dum(archivo, obj)** se guarda el objeto

Serialización

```
1  import pickle #Librería para serializar los objetos
2  SUELDO_BASE = 80000
3  """
4      Definiremos un diccionario
5      Empleado que tiene los siguientes
6      atributos:
7      +------------+
8      |  Empleado  |
9      +------------+
10     |IDEmpleado  |
11     +------------+
12     |Nombre      |
13     +------------+
14     |Apellido    |
15     +------------+
16     |Antiguedad  |
17     +------------+
18     |Sueldo*     | (Base = 80.000, 15% por aca año de antiguedad)
19     +------------+
20 """
21 def serializar(lista_empleados,nombre_archivo,escribir_agregar):
22     modo = 'wb' if escribir_agregar == True else 'a' #SE DEBE ESCRIBIR EN BINAR
23     with open(nombre_archivo,modo) as archivo:
24         for empleado in lista_empleados:
25             pickle.dump(empleado,archivo) #Serializa el objeto
```

Serialización

```
26
27 def deserilizar(nombre_archivo,lista_empleados):
28     with open(nombre_archivo,'rb') as archivo:
29         fin_archivo = False
30         while (not fin_archivo):
31             try:
32                 empleado = pickle.load(archivo)
33                 lista_empleados.append(empleado)
34             except EOFError:
35                 fin_archivo = True
```

Serialización

```
36
37 def imprimir_datos(empleado):
38     print('Empleado ID: {0}'.format(empleado.get('IDEmpleado','Error en key')))
39     print('Nombre: {0} \t Apellido: {1}'.format(
40         empleado.get('Nombre','Error en key'),
41         empleado.get('Apellido','Error en key')
42         ))
43     print('==========================')
44     print('Antiguedad: {0} años \t Sueldo Neto: {1:.2f}'.format(
45         empleado.get('Antiguedad','Error en key'),
46         SUELDO_BASE*0.15*empleado.get('Antiguedad','Error en key')
47         ))
48     print('\t\t---o---')
```

Serialización

```
50 def main():
51     empleado = {}
52     lista_empleados = []
53     empleado['IDEmpleado'] = '00000001'
54     empleado['Nombre']     = 'Homero'
55     empleado['Apellido']   = 'Simpson'
56     empleado['Antiguedad'] = 25
57     lista_empleados.append(empleado)
58     empleado = {}
59     empleado['IDEmpleado'] = '00000010'
60     empleado['Nombre']     = 'Peter'
61     empleado['Apellido']   = 'Griffin'
62     empleado['Antiguedad'] = 10
63     lista_empleados.append(empleado)
64     empleado = {}
65     empleado['IDEmpleado'] = '00000011'
66     empleado['Nombre']     = 'Stan'
67     empleado['Apellido']   = 'Smith'
68     empleado['Antiguedad'] = 30
69     lista_empleados.append(empleado)
```

Serialización

```
70      print('Datos:')
71      for empleado in lista_empleados:
72          imprimir_datos(empleado)
73      archivo = 'empleado.data'
74      serializar(lista_empleados,archivo,True)
75      print('Serializacion finalizada')
76      lista_deserilizacion = []
77      deserilizar(archivo,lista_deserilizacion)
78      print('Objetos leidos... Imprimiendo datos')
79      for empleado in lista_deserilizacion:
80          imprimir_datos(empleado)
81  main()
```

¡Oh no!

- En el código anterior hemos usado dos palabras reservadas **try/except**, pero... ¿Para qué sirven?

¡Oh no!

- En el código anterior hemos usado dos palabras reservadas **try/except**, pero... ¿Para qué sirven?

- Una excepción es un error que ocurre mientras un programa se ejecuta, provocando que el programa termine de forma abrupta. Pero podemos tratar estos errores mediante el *manejo de excepciones*

¡Oh no!

- En el código anterior hemos usado dos palabras reservadas **try/except**, pero... ¿Para qué sirven?

- Una excepción es un error que ocurre mientras un programa se ejecuta, provocando que el programa termine de forma abrupta. Pero podemos tratar estos errores mediante el *manejo de excepciones*

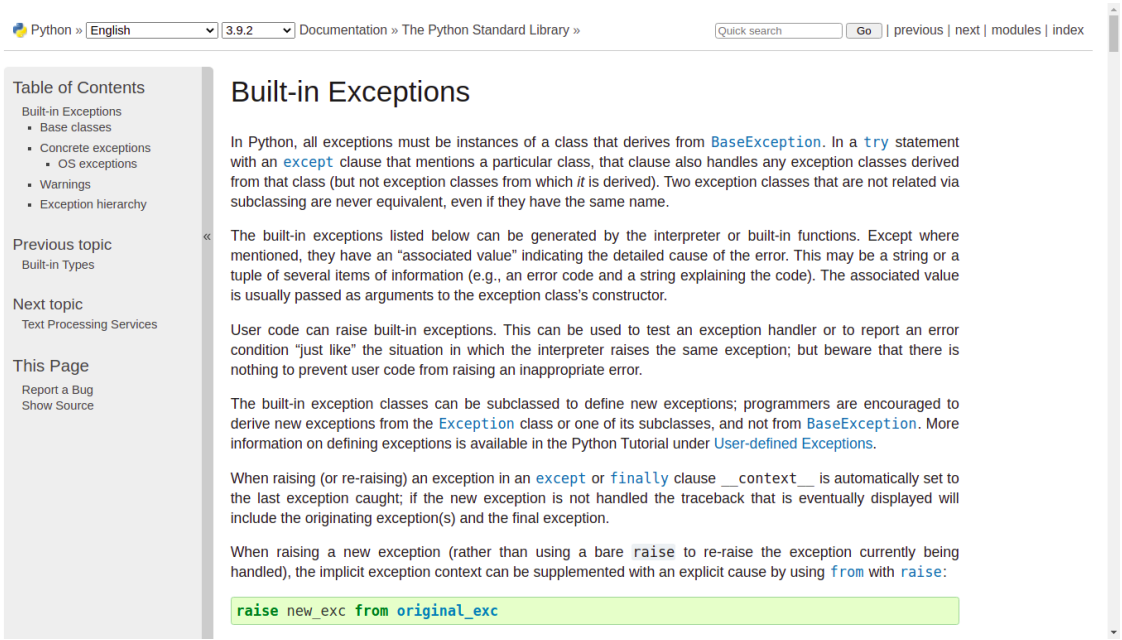
- El bloque try/except se compone principalmente de estos dos bloques:
  - El bloque try donde se ponen las sentencias que ***pueden*** generar una excepción
  - Uno o mas bloques except que **manejan** las excepciones.

¡Oh no!

- La listas de excepciones puede encontrarse en la documentación oficial

¡Oh no!

- La listas de excepciones puede encontrarse en la documentación oficial



Python » English 3.9.2 Documentation » The Python Standard Library » Quick search Go | previous | next | modules | index

Table of Contents

Built-in Exceptions
- Base classes
- Concrete exceptions
  - OS exceptions
- Warnings
- Exception hierarchy

Previous topic
Built-in Types

Next topic
Text Processing Services

This Page
Report a Bug
Show Source

Built-in Exceptions

In Python, all exceptions must be instances of a class that derives from `BaseException`. In a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class's constructor.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, and not from `BaseException`. More information on defining exceptions is available in the Python Tutorial under User-defined Exceptions.

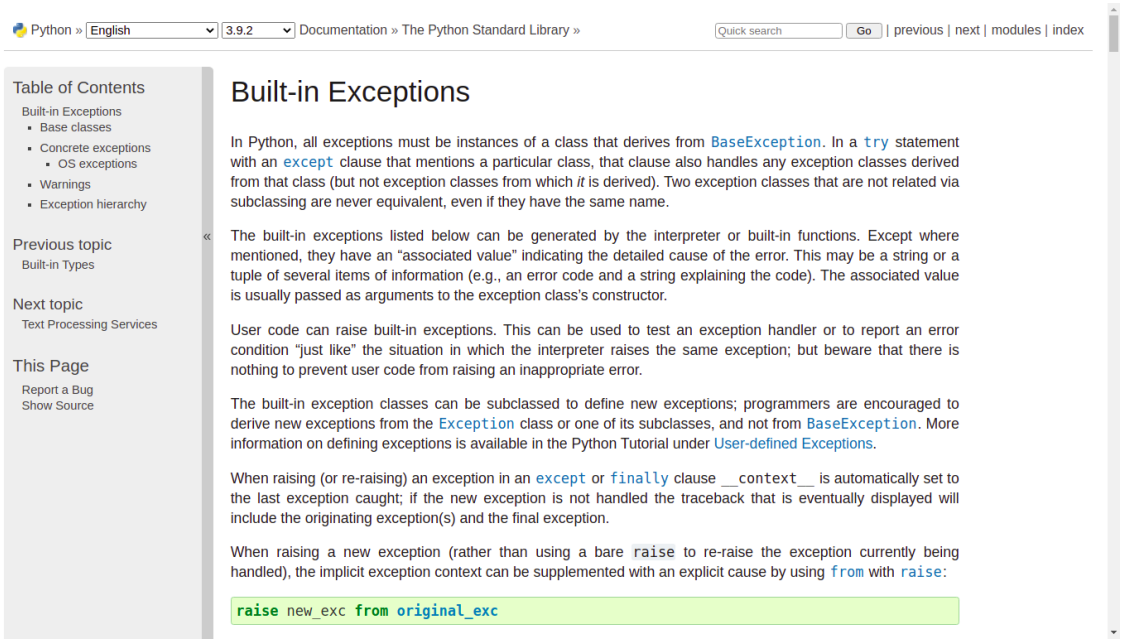
When raising (or re-raising) an exception in an `except` or `finally` clause `__context__` is automatically set to the last exception caught; if the new exception is not handled the traceback that is eventually displayed will include the originating exception(s) and the final exception.

When raising a new exception (rather than using a bare `raise` to re-raise the exception currently being handled), the implicit exception context can be supplemented with an explicit cause by using `from` with `raise`:

```
raise new_exc from original_exc
```

¡Oh no!

- La listas de excepciones puede encontrarse en la documentación oficial



Enlace excepciones

¡Oh no!

- Pueden añadirse dos bloque adiciones:

¡Oh no!

- Pueden añadirse dos bloque adiciones:

- El bloque **else** que se ejecuta ***siempre*** que no se alcance una excepción

¡Oh no!

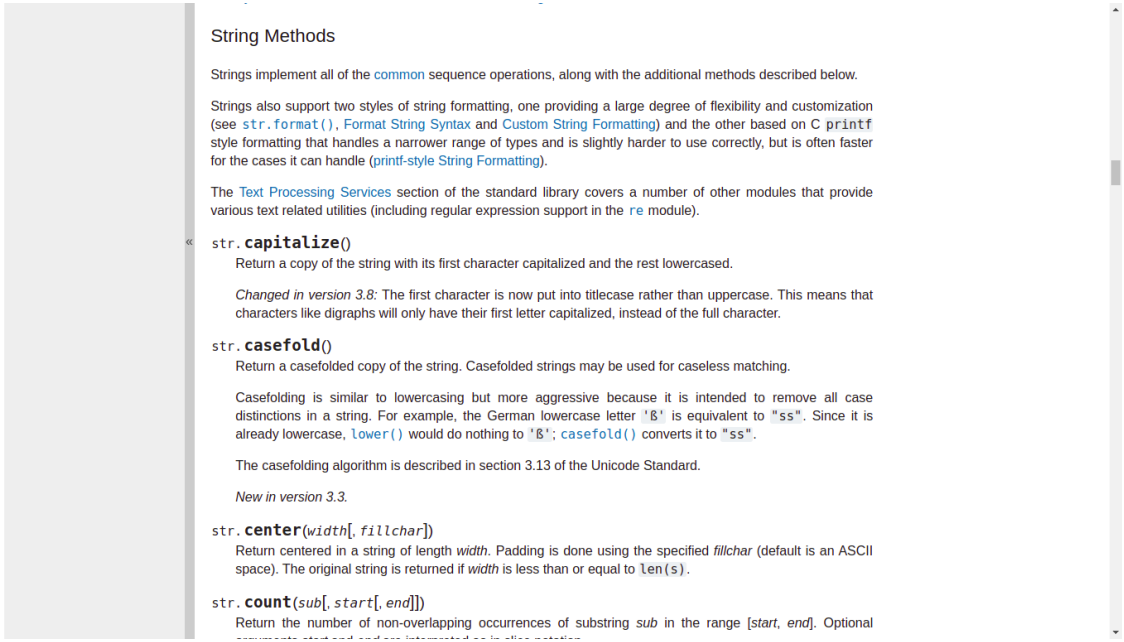
- Pueden añadirse dos bloque adiciones:
- El bloque **else** que se ejecuta ***siempre*** que no se alcance una excepción
- El bloque **finally** que se ejecuta ***siempre***, independiente si se alcanza o no una excepción.

Mas sobre cadenas

- Las cadenas tienen una serie de funciones predefinidas muy útiles que podemos encontrar en la documentación oficial

Mas sobre cadenas

- Las cadenas tienen una serie de funciones predefinidas muy útiles que podemos encontrar en la documentación oficial



String Methods

Strings implement all of the common sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, Format String Syntax and Custom String Formatting) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (printf-style String Formatting).

The Text Processing Services section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.` **capitalize**()
Return a copy of the string with its first character capitalized and the rest lowercased.

Changed in version 3.8: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

`str.` **casefold**()
Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter `'ß'` is equivalent to `"ss"`. Since it is already lowercase, `lower()` would do nothing to `'ß'`; `casefold()` converts it to `"ss"`.

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

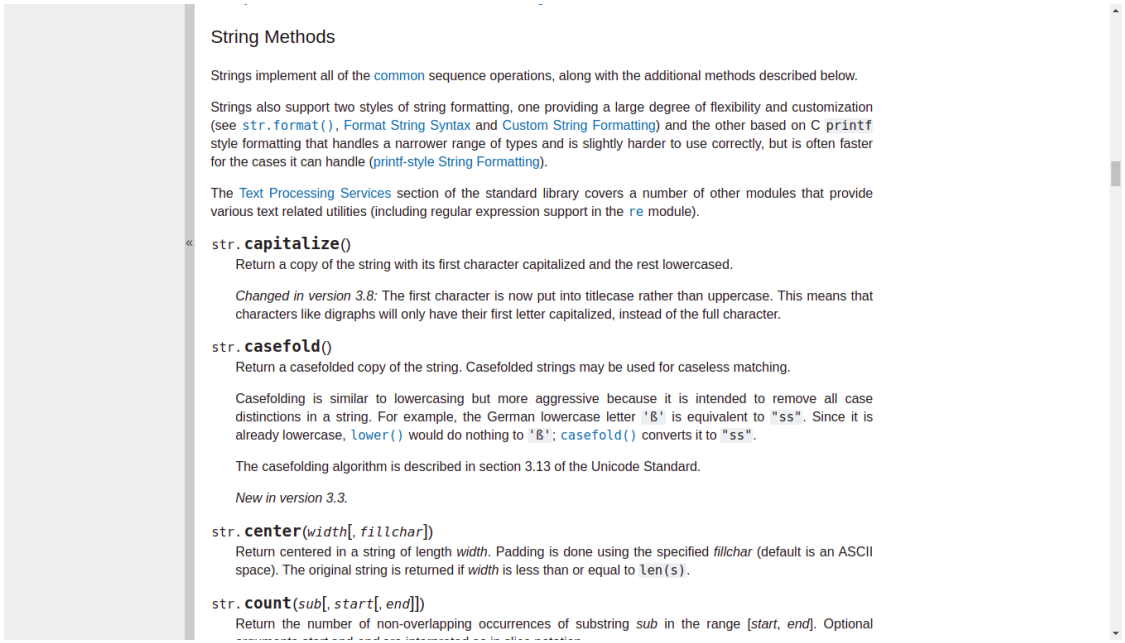
New in version 3.3.

`str.` **center**(*width*[, *fillchar*])
Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.` **count**(*sub*[, *start*[, *end*]])
Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional

Mas sobre cadenas

- Las cadenas tienen una serie de funciones predefinidas muy útiles que podemos encontrar en la documentación oficial



String Methods

Strings implement all of the common sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, Format String Syntax and Custom String Formatting) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (printf-style String Formatting).

The Text Processing Services section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.` **`capitalize`**()
Return a copy of the string with its first character capitalized and the rest lowercased.

Changed in version 3.8: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

`str.` **`casefold`**()
Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter `'ß'` is equivalent to `"ss"`. Since it is already lowercase, `lower()` would do nothing to `'ß'`; `casefold()` converts it to `"ss"`.

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

`str.` **`center`**(*width*[, *fillchar*])
Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.` **`count`**(*sub*[, *start*[, *end*]])
Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional

Enlace cadenas

Cadenas

```
1  cadena = 'una cadena cualquiera'
2  print(cadena.capitalize())#Capitalizacion
3  print(cadena.split())#Separa por espacios, devuelve una lista
4  print(cadena.upper())#MAYUSCULAS!
5  print(cadena.lower())#minusculas!
6  cadena = 'una cadena          '
7  print(cadena.strip()+'.')#Remeuve espacios o caracteres
8  caracter = 'a'
9  print(caracter.isalpha())
10 print(caracter.isdigit())
11 digito = '10'
12 print(digito.isdigit())
```

Obj.o

- Hasta ahora hemos llevado un estilo de programación conocido como **procedural**, hemos enfocado nuestro esfuerzo en describir los procedimientos o acciones que se llevan a cabo

Obj.o

- Hasta ahora hemos llevado un estilo de programación conocido como **procedural**, hemos enfocado nuestro esfuerzo en describir los procedimientos o acciones que se llevan a cabo

- Sin embargo, existen otros estilos como la **programación funcional, orientado a datos, orientado a objetos, etc**. Este último va a ser nuestro pequeño caso de estudio

Obj.o

- Hasta ahora hemos llevado un estilo de programación conocido como **procedural**, hemos enfocado nuestro esfuerzo en describir los procedimientos o acciones que se llevan a cabo

- Sin embargo, existen otros estilos como la **programación funcional, orientado a datos, orientado a objetos, etc**. Este último va a ser nuestro pequeño caso de estudio

- El paradigma orientado a objetos se basa en 4 pilares:
  - **Abstracción:** Nos abstraemos del funcionamiento interno del objeto y tomamos un modelo de caja negra, es decir, nos enfocamos en las entradas y salidas
  - **Encapsulación:** Se basa en la idea de esconder los datos y métodos en una sola unidad, la clase. Es decir, ningún objeto debería conocer toda la información de otro objeto.
  - **Herencia:** En resumidas palabras, podemos crear clases a partir de una clase padre.
  - **Polimorfismo:** Aquí un objeto es polimórfico

Obj.o

- Dado el alcance de este curso, sólo veremos la aplicación de abstracción y encapsulación.

Obj.o

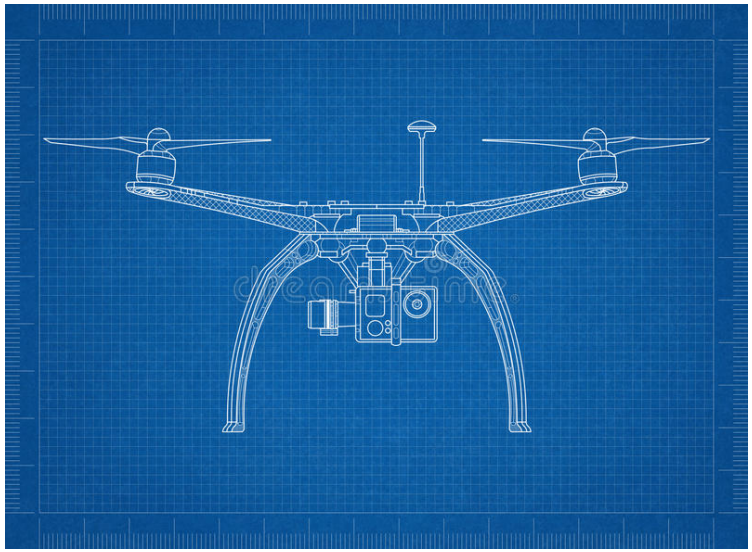
- Dado el alcance de este curso, sólo veremos la aplicación de abstracción y encapsulación.

- Un **objeto** podemos decir que es una colección de datos con un comportamiento asociado

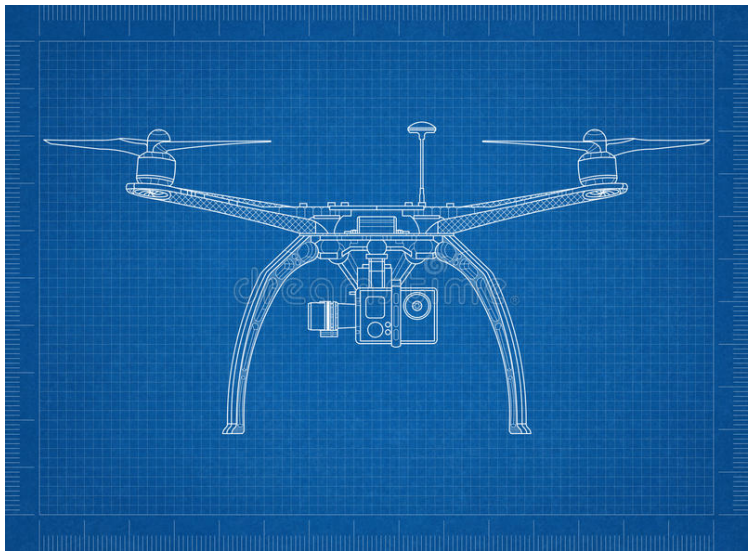
Obj.o

- Dado el alcance de este curso, sólo veremos la aplicación de abstracción y encapsulación.
- Un **objeto** podemos decir que es una colección de datos con un comportamiento asociado
- Pero a la hora de creer un objeto, necesitamos de un *plano*

Obj.o

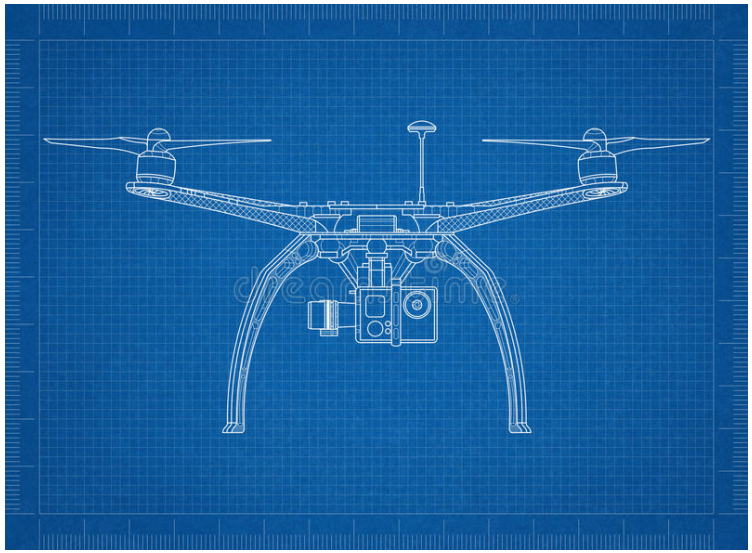


Obj.o



- Este *plano* se denomina de forma técnica, **clase**. ¿Cuál es la diferencia entre objeto y clase?

Obj.o



- Este *plano* se denomina de forma técnica, **clase**. ¿Cuál es la diferencia entre objeto y clase?

- Las clases se usan para *describir* los objetos (Es por esto la idea de plano) y cada instancia de la clase es un objeto.

Clases y objetos

```
1  import random as rd
2  class Moneda:
3      #Para crear una clase se usa la palabra
4      #reservada class seguida del nombre de la clase.
5      def __init__(self):
6      #Este método es una característica de Python que se usa
7      #para dos razones:
8      #Primero inicializa el objeto, poniendolo en un estado apropiado
9      #cuando se crea
10     #Segundo es que __init__ puede tomar muchas formas, permitiendo
11     #poder definir como se crea el objeto o inicializa
12         self.cara_cruz = 'cara'
```

Clases y objetos

```
13      def lanzar(self):
14          '''
15          Método que genera un número aleatorio y determina el estado de la moneda
16          @param Ninguno
17          @return Nada
18          '''
19          #self es lo que marca la diferencia entre método o función
20          # (Si bien son prácticamente lo mismo).
21          #Es un análogo al "this" de otros lenguajes,
22          # y lo que hace es una referencia al objeto que
23          # el método hace cuando se invoca.
24          #Podemos acceder a atributos y métodos
25          # de un objeto como si fuera cualquier otro objeto.
26          if (rd.randint(0,1)):
27              self.cara_cruz = 'cara'
28              print('Toss a coin to your Witcher...')
29          else:
30              self.cara_cruz = 'cruz'
31              print('O\' Valley of Plenty...')
```

Clases y objetos

```
32      def getEstado(self):
33          '''
34          Devuelve el estado de la moneda
35          @param Ninguno
36          @return Estado de la moneda 'cara' o 'cruz'
37          '''
38          return self.cara_cruz
```

Clases y objetos

```
39      def setEstado(self,estado):
40          '''
41          Establece el estado de la moneda
42          @param Nuevo estado
43          @return Nada
44          '''
45          self.cara_cruz = estado
46  #Los métodos get y set se llaman mutadores y son la principal herramienta
47  #para poder establecer (set) u obtener (get) el estado de un objeto
```

Clases y objetos

```
48 def main():
49     moneda_geralt = Moneda() #Con esta linea, creamos una nueva instancia de
50     #un objeto moneda
51     print(moneda_geralt.getEstado()) #Invocamos el metodo get
52     for l in range(3):
53         moneda_geralt.lanzar()
54         print(moneda_geralt.getEstado())
55     if moneda_geralt.getEstado() == 'cara':
56         moneda_geralt.setEstado('cruz')
57     else:
58         moneda_geralt.cara_cruz = 'cara'
59     print('Fin programa')
60 main()
```

Obj.o

- Dos comentarios sobre el código anterior

Obj.o

- Dos comentarios sobre el código anterior
- Los comentarios entre ”’ se escriben después del método y constituye la documentación de la clase. Es una práctica a emplementar siempre que podamos

Obj.o

- Dos comentarios sobre el código anterior
- Los comentarios entre ”’ se escriben después del método y constituye la documentación de la clase. Es una práctica a emplementar siempre que podamos
- @param indica los parámetros del método y @return lo que este devuelve. Podemos leer esto haciendo en consola **python -i archivo.py** y luego **help(NombreClase)**

Obj.o

- Lo otro es que hemos roto la encapsulación

Obj.o

- Lo otro es que hemos roto la encapsulación
- En la línea 58 hemos accedido directamente a la variable y evitamos el mutador set, lo cual está mal porque el objeto tiene sus datos de forma pública.

Obj.o

- Lo otro es que hemos roto la encapsulación
- En la línea 58 hemos accedido directamente a la variable y evitamos el mutador set, lo cual está mal porque el objeto tiene sus datos de forma pública.
- Para remendar esto, debemos hacer que la variable **cara_cruz** sea **privada**. Para python, todos los métodos y variables son **públicos** por defecto

Obj.o

- Si deseamos mantener la encapsulación, debemos añadir un prefijo de dos ___ lo cual lleva a cabo un **name mangling** sobre el atributo

Obj.o

- Si deseamos mantener la encapsulación, debemos añadir un prefijo de dos ___ lo cual lleva a cabo un **name mangling** sobre el atributo

- Esto significa que el atributo es "privado"

Obj.o

- Si deseamos mantener la encapsulación, debemos añadir un prefijo de dos ___ lo cual lleva a cabo un **name mangling** sobre el atributo
- Esto significa que el atributo es "privado"
- ¡Ojo! Las reglas de name mangling están diseñadas principalmente para evitar accidentes; todavía es posible acceder o modificar una variable que se considera privada. Esto incluso puede resultar útil en circunstancias especiales, como en el debugger.

Clases y objetos

```
31 def main():
32     moneda_geralt = Moneda()
33     print(moneda_geralt.getEstado())
34     for l in range(3):
35         moneda_geralt.lanzar()
36         print(moneda_geralt.getEstado())
37     if moneda_geralt.getEstado() == 'cara':
38         moneda_geralt.setEstado('cruz')
39     else:
40         moneda_geralt.__cara_cruz = 'cara'
41     print('Estado final',moneda_geralt.getEstado())
42     print('Fin programa')
```

¿Preguntas?