Curso de Python

David Aroesti

¿Qué lograrás con este curso?

- Entender qué es la programación
- Aprender a utilizar el lenguaje de programación Python
- Crear software sencillo que aplique los conceptos aprendidos
- Estar listo para estudiar temas más avanzados como la programación de servidores web (django, flask, por ejemplo) o la ciencia de datos.

Platzi

¿Qué es un programa?

Un programa es una secuencia de instrucciones que especifican cómo realizar un cómputo.

Todos los programas se componen de “partes” básicas que se utilizan para crear “partes” más complejas.

- Input, Output, Estructuras de datos, Condicionales, Iteraciones y Matemáticas.

Operadores básicos

- Los operadores especifican la operación que debe realizar la computadora.
- +
- -
- *
- /
- //
- %
- **
- >
- <
- ==

Orden de operaciones

- PEMDAS
  - Paréntesis
  - Exponentes
  - Multiplicación / División
  - Adición / Sustracción

Valores y Tipos

- Los valores son uno de los componentes básicos con los que trabaja un programa, como una letra o un número.
- Todos los valores tienen un tipo.
- Los tipos básicos son:
  - Integer <int>
  - Float <float>
  - String <str>
  - Boolean <bool>

Variables y expresiones

- Las expresiones se componen de valores conectados por operadores.
- Cuando la computadora evalúa una expresión, el resultado es otro valor.
- Los valores pueden guardarse en variables.
- Las variables se pueden reasignar

Nombres de variables

- Tienen que tener nombres significativos.
- Relacionados con lo que representa la variable.
- No deben ser Keywords (palabras reservadas):

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Cadenas (strings)

- Python puede procesar texto al igual que procesa números
- Para expresar texto, se utilizan comillas. Por ejemplo:
  - ‘Hola’
  - “Adiós”
- Las cadenas pueden asignarse a variables y algunos operadores pueden utilizarse con ellas.
- Si se requiere utilizar una comilla, es necesario escapar el carácter.

Comentarios

- Leer código propio y ajeno puede resultar una tarea difícil.
- Los comentarios son una excelente forma de hacer más legible tu código.
- En Python, los comentarios van después de los siguientes símbolos:
  - #
  - “””Otra forma de comentar llamada docstring”””

Platzi

Funciones

- Una función es una secuencia de comandos que realizan un cómputo.
- Para definirla, se utiliza el keyword **def,** se le asigna un nombre y se define la secuencia de comandos.
- Las funciones son llamadas por su nombre y regresan un resultado.
- Pueden recibir cero o más parámetros.

```
def suma(num1, num2):

    return num1 + num2


suma(2, 3) # El resultado es 5
```

Platzi

Limitantes

- El nombre de la función no puede empezar con un dígito.
- No puede ser una palabra reservada (keyword).
- Las variables y las funciones deben tener nombres distintos (la última definición es la que gana).

Flujo de ejecución

- Comienza con la primera declaración de arriba hacia abajo.
- Cada declaración se ejecuta de izquierda a derecha, siguiendo el orden de operaciones.
- Cuando una declaración contiene una llamada a una función, el cuerpo de la función se ejecuta con las mismas reglas.

Parámetros y argumentos

- Para ejecutar ciertas funciones, es necesario “pasarles” argumentos.
- Dichos argumentos son convertidos en parámetros (variables locales) dentro de la función.
- El campo de aplicación de los parámetros es local (las funciones también pueden declarar variables locales).

Diagramas de Pila

¿Por qué utilizar funciones?

- Te permite agrupar declaraciones; lo que tiene como resultado que el código sea más fácil de leer.
- Los programas se vuelven más compactos.
- Te permite reutilzar código.
- Es más fácil encontrar un error y cambiarlo (sólo se tiene que cambiar en un lugar).

Condicionales

- Una expresión booleana se evalúa como verdadera o falsa (True, False)
- Los operadores relacionales son:
  - ==
  - !=
  - >
  - >=
  - <
  - <=
- Los operadores lógicos son:
  - and
  - or
  - not

Tablas de verdad **and**

- Evaluación de **and**

A	B	Resultado
True	True	True
True	False	False
False	True	False
False	False	False

Tablas de verdad **or**

- Evaluación de **and**

A	B	Resultado
True	True	True
True	False	True
False	True	True
False	False	False

Tablas de verdad **not**

- Evaluación de **not**

A	Resultado
True	False
False	True

Evaluación en Python

- False
  - False
  - 2 > 5
  - 0
  - []
  - {}
  - ‘’
- True
  - True
  - 5 > 2
  - 1
  - [1] [1,2]
  - {‘a’: ‘b’}
  - ‘Hola’

Declaraciones condicionales

```
if x > 5:

    print(‘x es mayor que 5’)


if x < 5:

    print(‘x es menor que 5’)
```

Declaraciones condicionales

```
If x > 5:
    print('x es mayor que 5')
else:
    print('x es menor que 5')
```

Declaraciones condicionales

```
If nombre == ‘erika’::
    print(‘Hola, Erika’)
elif nombre == ‘david’:
    print(‘Hola, David’)
else:
    print(‘Hola, amigo’)
```

Declaraciones condicionales

```
If genero == ‘masculino’::
    If edad > 18:
        print(‘Hola, señor’)
     else:
        print(‘Hola niño’)
else:
    If edad > 18:
        print(‘Hola, señora’)
    else:
         print(‘Hola niña’)
```

Recursión

- Una función puede llamarse a sí misma dentro de la misma función.
- Siempre es necesario un caso base para evitar una recursión infinita.
- Ejemplo:

```
def factorial(num):

    if num == 0:

        return 1

    else:

        return num * factorial(num - 1)
```

Platzi

Cadenas (strings)

- Un string es una secuencia de caracteres.
- Los caracteres se pueden acceder por índice
  - Los índices empiezan en cero

0	1	2	3	4	5
p	**l**	**a**	**t**	**z**	**i**

mi_string = ‘platzi’

mi_string[2] # a

Longitud de una string

- La función **len** es una de las más útiles.
- Regresa el tamaño de la string.
- Ejemplo:

```
mi_string = ‘david’

len(mi_string) # 5



longitud = len(mi_string)

mi_string[longitud]

# Error común
```

Platzi

Métodos de strings

- Cuando tienes una string, existen muchos métodos que hacen fácil trabajar con ellas.
  - upper
  - isupper
  - lower
  - islower
  - find
  - isdigit
  - endswith
  - startswith
  - split
  - join

Slices (rebanadas)

- Se puede obtener una substring utilizando la notación de slices.
- Se definen los rangos que se requieren y los saltos necesarios.
- Ejemplo:

string = ‘platzi’

string[1:] # ‘latzi’

string[1:3] # ‘la’

string[1:6:2] # ‘lti’

string[::-1] # ‘iztalp’

Las strings son inmutables

- Una vez que se crea una string, no puede modificarse.
- Para modificarla, se tiene que crear una nueva.

```
mi_string = ‘tomas’

mi_string[0] = ‘l’ # error


nueva_string = ‘l’ + mi_string[1:]

# ‘lomas’
```

Comparación de strings

- Una string es igual a otra si contienen los mismos caracteres.
- Una string es menor que otra si sus caracteres se encuentran antes en un orden lexicográfico.
- Ejemplo

‘tomas’ == ‘diana’ # False

‘tomas’ == ‘tomas’ # True

‘alberto’ < ‘zaira’ # True

‘botella’ > ‘aldea’ # True

ASCII vs Unicode

- Ambos son codificadores de caracteres
- ASCII (American standard code for information interchange)
- UNICODE incluye la mayoría de los alfabetos del mundo

Iteración

- Las iteraciones permiten realizar la misma secuencia de pasos varias veces.
- También permiten recorrer una secuencia (como una string).
- Es una de las herramientas clave de cualquier programador.

for loop

- La función range nos permite definir cuántas veces se ejecutará una iteración.

```
for i in range(3):
    print('hola, mundo')
# 'hola, mundo'
# 'hola, mundo'
# 'hola, mundo'
```

for loop

- Se puede utilizar para recorrer strings (una string es una secuencia)
- Se necesita el keyword **in**
- Si se requiere salir antes de una iteración se utiliza el keyword **break**
- Si se requiere pasar a la siguiente iteración se utiliza el keyword **continue**

```
string = ‘david’

for letter in string:

    print(letter)
```

for / else

- La cláusula **else** se ejecuta si el loop no termina con un **break**

while loop

- Similar a un **for** loop; pero en lugar de recorrer una secuencia, se ejecuta hasta que una condición se convierta en falsa.
- Se debe tener mucho cuidado de no caer en un **infinite loop**.

```
i = 10

while i > 0:

    print('ando en un loop')

    i -= 1
```

while / else

- Cuando la condición se vuelve falsa y si no se terminó el loop con un break, se ejecuta la cláusula else.

Listas

- Una lista es una secuencia de elementos.
- Cuando se asigna a una variable, permite agrupar varios elementos en un solo lugar.
- Se crean con los corchetes [] o con la keyword **list**
- Por ejemplo:

```
supermercado = [‘apio’, ‘tomate’, ‘queso’]

temperaturas = [24, 26, 18, 20, 21]
```

- Las listas son mutables

```
supermercado[0] = ‘salsa’
```

Platzi

Acceso a elementos

- Se puede acceder a los elementos de una lista con su índice.
- Los índices se comienzan a contar desde cero.
- Ejemplo:

mi_lista = [‘juan’, ‘pedro’, ‘pepe’]

juan	pedro	pepe
0	1	2

mi_lista[0] # juan

mi_lista[1] # pedro

mi_lista[-1] # pepe

mi_lista[5] # IndexError

Operaciones de lista

- Las listas soportan los opeadores + y *

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

lista_b = [4]

lista_a + lista_b # [1, 2, 3, 4]

lista_b * 3 # [4, 4, 4]
```

List slices

- Las listas, al igual que los strings, pueden “rebanarse”.

```
lista = [1, 2, 3, 4, 5, 6]

lista[1:] # [2, 3, 4, 5, 6]

lista[1:3] # [2, 3]

lista[1:6:2] # [2, 4, 6]

lista[::-1] # [6, 5, 4, 3, 2, 1]
```

Modificación de lista

```
mi_lista = [‘juan’, ‘pedro’, ‘pepe’]

mi_lista[0] = ‘laura’

# [‘laura’, ‘pedro’, ‘pepe’]

mi_lista.append(‘estela’)

# [‘laura’, ‘pedro’, ‘pepe’, ‘estela’]

nombre = mi_lista.pop()

# [‘laura’, ‘pedro’, ‘pepe’]

nombre == ‘estela’

# True
```

Modificación de lista

```
mi_otra_lista = [4, 3, 7, 1]

mi_otra_lista.sort()

# [1, 3, 4, 7]

mi_lista.extend(mi_otra_lista)

# [‘juan’, ‘pedro’, ‘pepe’, 4, 3, 7, 1]

del mi_otra_lista[0]

# [3, 7, 1]
```

Listas y strings

- Como un string es una secuencia de caracteres, se puede utilizar para inicializar una lista.

un_string = ‘cama’

una_lista = list(un_string)

[‘c’, ‘a’, ‘m’, ‘a’]

Iteración de lista

```
hamburgueserias = [‘burger king’, ‘mcdonalds’, ‘carls jr.’]


for hamburgueseria in hamburgueserias:
    print(hamburgueseria)


for idx, hamburgueseria in enumerate(hamburgueserias):
    ...
```

Conjuntos (sets)

- Similares a una lista, pero no permiten repetidos.
- Se crean con el keyword **set**

```
conjunto = set()

conjunto.add(2)

# {2}

conjunto.add(2)

 conjunto.add(3)

# {2, 3}

2 in conjunto

# True
```

Métodos del objeto set

- X in S
- X not in S
- s.union(t)
- s.intersection(t)
- s.difference(t)
- s.issubset(t)
- s.issuperset(t)

Diccionarios

- Un diccionario es un mapa de llaves a valores. Los valores pueden ser de cualquier tipo.
- Se crea con llaves {} o con el keyword **dict**
- Se añaden elementos al diccionario señalando la llave y el valor.
- Ejemplo

```
mi_dict = {}

mi_dict['llave'] = 'valor'
```

- También se pueden añadir valores al momento de inicializarlo

```
mi_dict {'llave': 'valor'}
```

Iteración en diccionarios

- Al iterar un diccionario, de manera predeterminada, se obtendrán las llaves.
- Es posible también iterar a través de los valores
- También se permite obtener las llaves y los valores al mismo tiempo

Iteración en diccionarios

```
for llave in diccionario:
    …
for llave in diccionario.keys():
    …
for valor in diccionario.values():
    …
for llave, valor in diccionario.iteritems():
    ...
```

Platzi

Obtener valores

diccionario[‘llave’]

valor

diccionario.get(‘llave’, ‘default’)

si la llave existe, valor

si no, default

List comprehensions y Dictionary comprehensions

- Python permite crear e inicializar listas y diccionarios con una sintaxis más natural.
- Convierte una secuencia en otra.

```
pares = [x for x in range(100) if x % 2 == 0]

nones = [x for x in range(100) if x %2 != 0]

cuadrados = {x: x**2 for x in range(100) }
```

Tuplas

- Las tuplas son una secuencia de valores indexada por enteros (integers).
- Similares a las listas, con la gran diferencia de que son inmutables.
- Las tuplas se crean separando los valores con comas.
  - Es práctica común encerrar los valores en paréntesis
- Se pueden utilizar para devolver más de un valor en una función o crear estructuras de datos ligeras.

```
mi_tupla = 1, 2, 3, 4
```

```
mi_tupla = (1, 2, 3, 4)
```

Operadores en tuplas

- Al igual que los strings, si se quiere modificar una tupla, es necesario crear una nueva.

```
mi_tupla = (1,)

nueva_tupla = mi_tupla + (2, 3, 4)

# (1, 2, 3, 4)
```

Manejo de errores

- Cuando se avienta (**throw**) un error, si el error no se “atrapa”, entonces el programa se detiene.
- Hay veces que queremos evitar este comportamiento porque sabemos como arreglar el error.
- Para manejar el error se utilizan los keywords **try / except**

```
mi_string = ‘una string’

try:

    int(mi_string)

except ValueError:

    … # maneja el error

Finally:

    … # Se ejecuta pase lo que pase
```

Platzi

Jerarquía de errores en Python

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

try / except / else / finally

- La cláusula **else** se ejecuta si no han ocurrido excepciones y antes de la cláusula **finally.**

Errores propios

- Python permite declarar errores específicos para cada programa.
- Es buena práctica extender el tipo de error del que se trate.
- Los errores se “avientan” con el keyword **raise**

```
class ErrorDePosicion(Exception):

    “””Explicación del error”””

    pass


if not posicion:

    raise ErrorDePosicion(‘mensaje’)
```

Manejo de archivos

- Python puede leer y escribir archivos con la función **open**
- La función **open** regresa un objeto archivo (file)
- Los archivos pueden ser de texto o binarios
- Se tiene que especificar el modo en que se maneja el archivo
  - ‘r’ = read
  - ‘w’ = write
  - ‘a’ = append
  - ‘r+’ = read and write
- Se debe cerrar el archivo con el método **close**
- La mejor manera de manejar archivos es con el keyword **with**

Lectura de archivos

- El objeto archivo tiene dos métodos para leerlo: **read** y **readlines**

```
with open('file.txt', 'r') as f:
    for line in f:
        print(line)
```

Escritura de archivos

- Para escribir a un archivo, se utiliza el método **write**

```
with open('file.txt', 'w') as f:
    for i in range(5):
        print('Hello')
```

CSV

- Coma separated values
- reader
  - readlines
- writer
  - write

Clases y objetos

- Programación orientada a objetos.
- Permite definir tipos propios.
- Permite manejar datos y lógica en un solo contenedor.
- Las clases son como fábricas (moldes) para crear otros objetos.

Clases y objetos

- Los objetos tienen atributos que se pueden definir al momento de *inicializar* un nuevo objeto o directamente en la *instancia*.
- Las clases pueden tener variables de clase, variables de instancia y variables locales.
- Aunque Python no tiene el concepto de variables privadas integrado al lenguaje, es práctica común definirlas con un guión bajo.
- **isinstance** y **hasattr**

Clases y métodos

- Los métodos son como funciones que tienen sentido únicamente en el contexto de una clase.
- Al igual que las variables, los métodos privados se definen con un guión bajo.
- La **encapsulación** es un concepto clave de la programación orientada a objetos.
  - La forma práctica de aplicar este principio es declarar todas las variables y métodos como privados, a menos de que sea necesario exponerlos a otros programadores.
- Un método clave que casi todas las clases deben tener es **__init__**
- Otro es **__str__**
- Existen varios tipos de métodos, estáticos, de clase, de instancia, getters y setters.

Herencia, polimorfismo y composición

- La herencia (inheritance) permite definir una clase (la subclase) basada en otra clase (la superclase).
- Polimorfismo permite modificar métodos preexistentes en la superclase.
- Composición permite crear una clase que depende de instancias de otra clase.

Paquetes y módulos

- Un módulo permite agrupar funcionalidad común en un sólo archivo.
- Cuando varios módulos agrupan funcionalidades comunes, se pueden agrupar, a su vez, en paquetes.
- Python reconoce que un directorio es un paquete porque contiene un archivo llamado __init__.py

Paquetes de terceros

- PyPi (python package index) es un repositorio de paquetes de terceros que se pueden utilizar en proyectos de python.
- Para instalar un paquete, es necesario utilizar la herramienta pip.
- La forma de instalar un paquete es ejecutando el comando **pip install paquete.**
- También se puede agrupar la instalación de varios paquetes a la vez con el archivo **requirements.txt**

Ambientes virtuales

- Es una buena práctica crear un ambiente virtual por cada proyecto de Python en el que se trabaje.
- Esto evita conflictos de paquetes en el intérprete principal.
- **pip install virtualenv**
- **virtualenv venv**
- **source venv/bin/activate**
- **deactivate**

Pruebas automatizadas

- Las pruebas automatizadas son una forma de verificar que el software:
  - Cumple con los requisitos técnicos y de negocio de la especificación
  - Funciona
- Python incluye un módulo específico para realizar esta tarea: **unittest**
- Los tests funcionan realizando afirmaciones (assertions) sobre el valor de regreso de una función o el estado de una clase.
- Las pruebas se corren con *test runners.* El más común en Python se llama nose.
- Los programadores profesionales escriben su software, especificando primero las pruebas. Esto se conoce como test driven development.

Debugging

- En Python, la manera más sencilla de debuguear es utilizando el módulo **pdb**

Decoradores

- Un decorador es una función que recibe como parámetro a otra función y modifica su comportamiento.
- Un decorador se aplica a una función o método con el símbolo @
- Súper útil para definir si una función debe ejecutarse o no.
    - Por ejemplo, en servidores web, existen ciertas funciones que nada más deben ejecutarse si un usuario se encuentra autenticado.

Python 2 vs Python 3

- print
- strings
- Integer division
- raising exceptions
- Input vs raw_input

Aplicaciones de Python en el mundo real

- Desarrollo web
- Ciencia de datos
- Machine learning e Inteligencia artificial
- Programas de escritorio
- Línea de comando
- Internet de las cosas
- Web scraping y bots
- Criptografía
- Finanzas
- Biología computacional
- Ciencias exactas y academia

Conclusión y despedida