Platzi

Curso de Python

David Aroesti

¿Qué es la programación?

¿Qué es la programación?

- Ciencias de la computación
  - Matemáticas
  - Ingeniería
  - Ciencia
- La habilidad más importante: resolver problemas

¿Qué es la programación?

- Un programa es una secuencia de instrucciones que describe cómo realizar un cómputo.
- Casi todos los programas realizan las siguientes tareas:
  - Input
  - Output
  - Operaciones matemáticas
  - Ejecución condicional
  - Repeticiones

¿Qué es la programación?

- Objetivos:
  - Aprender a pensar como un Científico del cómputo
  - Aprender a utilizar Python
  - Entender las ventajas y desventajas de Python
  - Aprender a construir una aplicación de línea de comandos


4GIFs
.com


YOU'RE ABOUT
TO HACK TIME,
ARE YOU SURE?
YES
NO

1. Básicos del lenguaje

¿Por qué programar con Python?

¿Por qué programar con Python?

- Comunidad
- Facilidad de uso
- Librerías
- Popularidad
- Industria

1. Básicos del lenguaje

Instalación de Python

Operadores matemáticos

Operadores matemáticos

- +
- -
- /
- //
- %
- *
- **

Operadores matemáticos

- +=
- -=
- /=
- //=
- %=
- *=
- **=

1. Básicos del lenguaje

Variables y expresiones

Variables y expresiones

- ¿Qué es una variable?
- Tipos de variables: públicas, privadas, constantes
- Una asignación (assignment statement) crea una variable y le asigna un valor
  - message = ‘How are you?’
  - _age = 20
  - PI = 3.14159
  - __do_not_touch = ‘something important’
- Las variables se pueden reasignar
  - my_var = 2
  - my_var = my_var * 5
  - print(my_var)

Variables y expresiones

- Variables
  - Pueden contener números y letras
  - No deben comenzar con número
  - Múltiples palabras se unen con _
    - multiple_words
  - No se pueden utilizar palabras reservadas

Variables y expresiones

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

Variables y expresiones

- Una expresión (**expression**) es una combinación de valores, variables y operadores
  - El intérprete evalúa expresiones
  - Ej. 2 + 2
- Un enunciado (**statement**) es una unidad de código que tiene un efecto
  - Ej. age = 20

Variables y expresiones

- Orden de operaciones:
  - **P**arentesis
    - **E**xponente
      - **M**ultiplicación
      - **D**ivisión
        - **A**dición
        - **S**ubstracción
- PEMDAS

Variables y expresiones

- Los operadores funcionan de manera contextual según el tipo de los valores
  - 2 + 2
  - ‘H’ + ‘ello’
  - 2 * 2.0
  - ‘Hello’ * 2
  - 10 / 2.5
  - ‘Hello’ / 2

1. Básicos del lenguaje

Funciones

Funciones

- En el contexto de la programación, una función es una secuencia enunciados (*statements*) con un nombre que realizan un cómputo
- Una función tiene un nombre, parámetros (opcional) y valor de regreso (*return value*)(opcional)
- Python incluye varias *built-in functions* en su librería estándar

Funciones

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Funciones

- Otras funciones se pueden encontrar en módulos
  - Para utilizarlas es necesario importar el módulo
    - Ej. import math
- Para declarar una función, utilizamos el keyword **def**
  - Ej. def my_fuction(first_arg, second_arg=None)
- Las funciones se pueden componer.
  - Ej.

```
def sum_two_numbers(x, y):
    return x + y

other_function(sum_two_numbers(3, 4))
```

Funciones

- Los argumentos pueden ser posicionales (positional arguments) o con nombre (named arguments)
  - Los parámetros y variables son locales a la función
    - global keyword
- Orden de ejecución:
  - Arriba para abajo
  - Izquierda a derecha

1. Básicos del lenguaje

Estructuras condicionales

Estructuras condicionales

- Una expresión booleana siempre evalúa como verdadero (True) o como falso (False)
- Operadores de comparación
  - x = 2
  - y = 3
    - x == y
    - x != y
    - x > y
    - x < y
    - x >= y
    - x <= y

Estructuras condicionales

- Los operadores lógicos
  - and, or, not

Estructuras condicionales

- Al escribir software siempre necesitamos de una ejecución condicional
- Python nos ofrece esta habilidad con el keyword **if**
  - if x > y:
        do_something()

- También podemos definir ejecuciones alternas con los keywords **elif** y **else**
  - if x > y:
        do_something()
    elif:
        do_something_else()
    else:
        execute_if_no_other_conditions_are_true()

2. Uso de strings y ciclos

Strings en Python

Strings en python

- Las cadenas (strings) es un tipo con comportamiento diferente a los int, float y bool
  - Las cadenas son secuencias
  - Las secuencias se pueden acceder a través de un índice
    - apple = ‘apple’
      apple[1]
- Las cadenas (al igual que otras secuencias) tienen una longitud
  - Para saber la longitud de una secuencia, se puede usar la función **len**
    - len(apple)
- En Python, los caracteres que componen un string se reutilizan a lo largo del programa
  - Esto ayuda a reducir la cantidad de memoria que necesita el programa
  - También significa que las strings deben ser inmutables
    - x = ‘a’
      y = ‘b’
      id(x) == id(y)

2. Uso de strings y ciclos

Operaciones con strings

Operaciones con strings

- Los strings tienen varios métodos que nos sirven para manipularlas
- Algunos son:
  - upper
  - lower
  - find
  - startswith
  - endswith
  - capitalize

Operaciones con strings

- Operadores de pertenencia
  - **in**
  - **not in**
- Comparaciones entre strings
  - Las comparaciones son lexicográficas
  - Tener cuidado:
    - En Python ‘a’ y ‘A’ son diferentes

2. Uso de strings y ciclos

Slices en Python

Slices en Python

- Python tiene una de las sintaxis más poderosas para manipular secuencias
- Esta sintaxis se llama **slice** (rebanada en español)
  - secuencia[comienzo:final:pasos]
- Ej.

```
my_name = ‘David’
my_name[0]
my_name[-1]
my_name[0:3]
my_name[::2]
```

2. Uso de strings y ciclos

For loops

For loops

- Los **for** loops permiten ciclar a lo largo de una secuencia
- Se usan cuando se quiere ejecutar un conjunto de instrucciones varias veces
  - Esto también se llama **iteration**
- Se puede utilizar el keyword **continue** para saltarse los *statements* restantes y pasar a la siguiente iteración
- Ej.

```
for i in range(1000):
    print(i)
```

2. Uso de strings y ciclos

While loops

While loop

- Al igual que los **for** loops, los **while** loops sirven para iterar a lo largo de una secuencia
- Ej.

```
def cuenta_regresiva(n):
    while n > 0:
        print(n)
        n -= 1
```

While loop

- Los while loops se ejecutan de la siguiente manera:
  - Determinan si la condición es verdadera
  - Si es verdadera, vuelve a ejecutar el loop
  - Si es falsa, sal del bloque y continúa ejecutando el programa
- Se tiene que tener mucho cuidado para evitar un infinite loop
- Se puede utilizar el keyword **break** si se quiere salir anticipadamente del ciclo

3. Estructuras de datos

Listas en Python

Listas en Python

- Como los strings, las listas son secuencias de valores.
  - En las listas, los valores pueden tener cualquier tipo
- Ej.

  [2, 5, 6]
  [‘Colombia’, ‘Mexico’, ‘Argentina’]
  [‘tacos’, 3, ‘arepas’, 6, ‘chorizo’, 9]
- Las listas son mutables, a diferencia de los strings

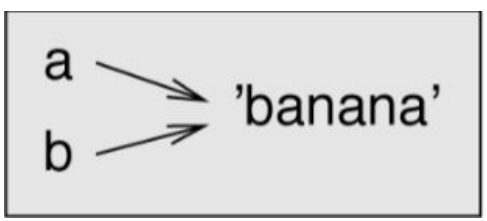
  my_list = [1, 2, 3]
  my_list[2] = 6
- Los índices de las listas, funcionan igual que los de los strings
- Las listas se inician con [] o con la función **list**

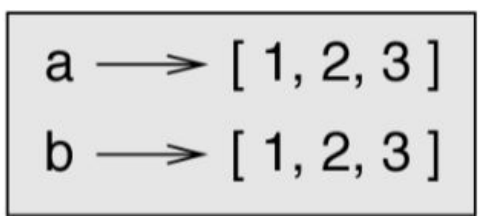
Listas en Python

- Para ciclar a lo largo de los elementos de una lista, normalmente usamos **for** loops
  - Ej.

    ```
    for student in students:
        print(student)
    ```
- Si la lista está vacía, el cuerpo del loop nunca se ejecuta

Listas en Python





3. Estructuras de datos

Operaciones con listas

Operaciones con listas

- El operador **+ (suma)** concatena dos o más listas
  - Ej.
    a = [1, 2]
    b = [2, 3]
    a + b # [1, 2, 2, 3]

- El operador * **(multiplicación)** repite la misma lista
  - Ej.
    a = [1, 2]
    a * 2 # [1, 2, 1, 2]

Operaciones con listas

- Para añadir un elemento al final de la lista, podemos utilizar el método **append**
  - Ej.
    ```
    a = [1]
    a.append(2) # [1, 2]
    ```
- Para eliminar el último elemento de la lista, podemos utilizar el método **pop**
  - Este método, también regresa el valor que fue eliminado
    - Ej.
      ```
      a = [1, 2]
      b = a.pop()
      print(a) # [1]
      print(b) # 2
      ```
- Para ordenar una lista, podemos utilizar el método **sort**
  - Ej.
    ```
    a = [3, 8, 1]
    a.sort() # [1, 3, 8]
    ```

Operaciones con listas

- Para eliminar elementos, también podemos utilizar el keyword **del**
  - **del** también funciona con slices
  - Ej.
    a = [1, 2, 3]
    del a[-1]
- Si sabes qué elemento quieres eliminar, pero no su índice, puedes utilizar el método **remove**

3. Estructuras de datos

Diccionarios

Diccionarios

- Un diccionario es similar a una lista en el sentido de que se puede acceder a través de índices (en el diccionario se llaman llaves)
  - En la lista los índices tienen que ser enteros
  - En el diccionario pueden ser casi cualquier tipo
- Un diccionario es una asociación entre llaves (**keys**) y valores (**values)**
- Los diccionarios se inicializan con {} o con la función **dict**
  - Ej.

    productos = {}

    productos[‘leche’] = 23.50

Diccionarios

- Existen varias formas de ciclar a lo largo de un diccionario
  - Ej.

```
for key in my_dict.keys():
        pass

for value in my_dict.values():
        pass

for key, value in my_dict.items():
        pass
```

3. Estructuras de datos

Tuplas y conjuntos

Tuplas y conjuntos

- Las tuplas (**tuples**) son similares a las listas
  - La mayor diferencia es que son inmutables
- Lo que define a un tuple es que sus valores están separados por comas
  - Es buena práctica utilizar un paréntesis también para ayudar a la legibilidad
  - Ej.

```
tup = 1, 2, 3
tup = (1, 2, 3)
```

- También podemos utilizar la función **tuple**
- Un uso muy común es utilizarlas para regresar más de un valor en una función
  - Ej.

```
return (students, teachers)
```

Tuplas y conjuntos

- Los conjuntos (**sets**) son una colección sin orden que no permite elementos duplicados
- Los sets se inicializan con la función **set**
- Para añadir elementos utilizamos el método **add**
- Y para eliminarlos, el método **remove**

3. Estructuras de datos

Comprehensions

Comprehensions

- Comprehensions son constructos que nos permiten generar secuencias a partir de otras secuencias

- List comprehension
  - **[element** for **element** in *element_list* if element_meets_condition**]**
- Dictionary comprehension
  - **{key**: **element** for **element** in *element_list* if element_meets_condition**}**
- Set comprehension
  - **{element** for **element** in *element_list* if element_meets_condition**}**

3. Estructuras de datos

Búsqueda binaria

Búsqueda binaria

- El módulo random nos permite generar números aleatorios
  - Recibe dos parámetros: el rango inicial y final (inclusive)

```
import random

random.randint(0, 10)
```

4. Uso de objetos y módulos

Manipulación de archivos

Manipulación de archivos

- La función **open** nos permite leer archivos
  - f = open(‘some_file’)
- Es importante siempre cerrar el archivo con la función **close** para que se escriban los datos y no se desperdicie memoria
  - f.close()
- Una mejor manera de manipular archivos es utilizando **context managers,** porque garantizan que el archivo se cierre
  - Ej.

```
with open(filename) as f:
    # do something with the file
```

- Existen varios modos de abrir un archivo. Los más importantes son **r** (read) y **w** (write)

```
with open(filename, mode=’w’) as f:
    # do something with the file
```

Manipulación de archivos

- El módulo **csv** nos permite manipular archivos con terminación *.csv*
  - csv significa comma separated values
  - Es un formato para almacenar datos tabulares
- Para utilizarlo lo importamos con la siguiente declaración:
  Import csv

- Existen dos readers y dos writers
  - **csv.reader** y **csv.writer** nos permiten manipular los valores a través de listas que representan filas
    - Solo se puede acceder por índice a los valores
  - **csv.DictReader** y **csv.DictWriter** nos permiten manipular los valores a través de diccionarios que representan filas
    - Se puede acceder a través de llaves a los valores

Decoradores

Decoradores

- Los decoradores permiten extender y modificar el funcionamiento de las funciones
- Los decoradores envuelven a otra función y permiten ejecutar código antes y después de que es llamada
- Ej.

```
def lower_case(func):
    def wrapper():
        # execute code before
        result = func()
        # execute code after
        return result

     return wrapper
```

¿Qué es la programación orientada a objetos?

¿Qué es la programación orientada a objetos?

- La programación orientada a objetos es un paradigma de programación que otorga los medios para estructurar programas de tal manera que las propiedades y comportamientos estén envueltos en objetos individuales
  - En pocas palabras, es un enfoque que nos permite modelar objetos concretos y del mundo real y las relaciones entre ellos
- Los principios básicos son:
  - **Encapsulation**
  - **Abstraction**
  - **Inheritance**
  - **Polyphormism**
- Todos los objetos son una instancia de una clase

¿Qué es la programación orientada a objetos?

- Los tipos básicos en Python (str, int, bool, etc.) están diseñados para representar cosas simples
- Cuando necesitamos crear estructuras más complejas (por ejemplo, un avión), podemos utilizar clases
- La instancia es el objeto concreto con valores reales

4. Uso de objetos y módulos

OOP en Python

OOP en Python

- Ejemplo Class

```
class Airplane:
    def __init__(self, passengers, seats, pilots=[]):
        self.passengers = passengers
        self.seats = seats
        self._pilots = pilots

    def takeoff(self):
        pass

airplane = Airplane(passengers=20, seats=30, pilots=['Tom', 'Billy'])
airplane.passengers = 31
airplane.takeoff()
```

OOP en Python

- Ejemplo Inheritance

```
class Vehicle:
    def __init__(self, current_speed):
        self.current_speed

class Airplane(Vehicle)
      def __init__ …
            super().__init__(current_speed=0)

       def current_speed():
            return ‘{} km/h’.format(self.current_speed)
```

Introducción a click

Introducción a Click

- Click es un framework que nos permite crear aplicaciones de Command Line
- Click utiliza decoradores para implementar su funcionalidad
- Nos otorga una interfaz que podemos personalizar
  - También autogenera ayuda para el usuario
- Los decoradores más importantes que nos otorga son:
  - @click.group
  - @click.command
  - @click.argument
  - @click.option
- También realiza las conversiones de tipo por nosotros

4. Uso de objetos y módulos

Errores y jerarquía de errores en Python

Errores y jerarquía de errores

- Un programa de Python termina en cuanto encuentra un error
  - Es diferente a un error de sintaxis
- Para “aventar” un error utilizamos el keyword **raise**
  - Ej.

```
def divide(numerator, denominator):
    If denominator == 0:
        raise ZeroDivisionError
```

- También podemos generar nuestros propios error, si extendemos **BaseException**
  - Ej

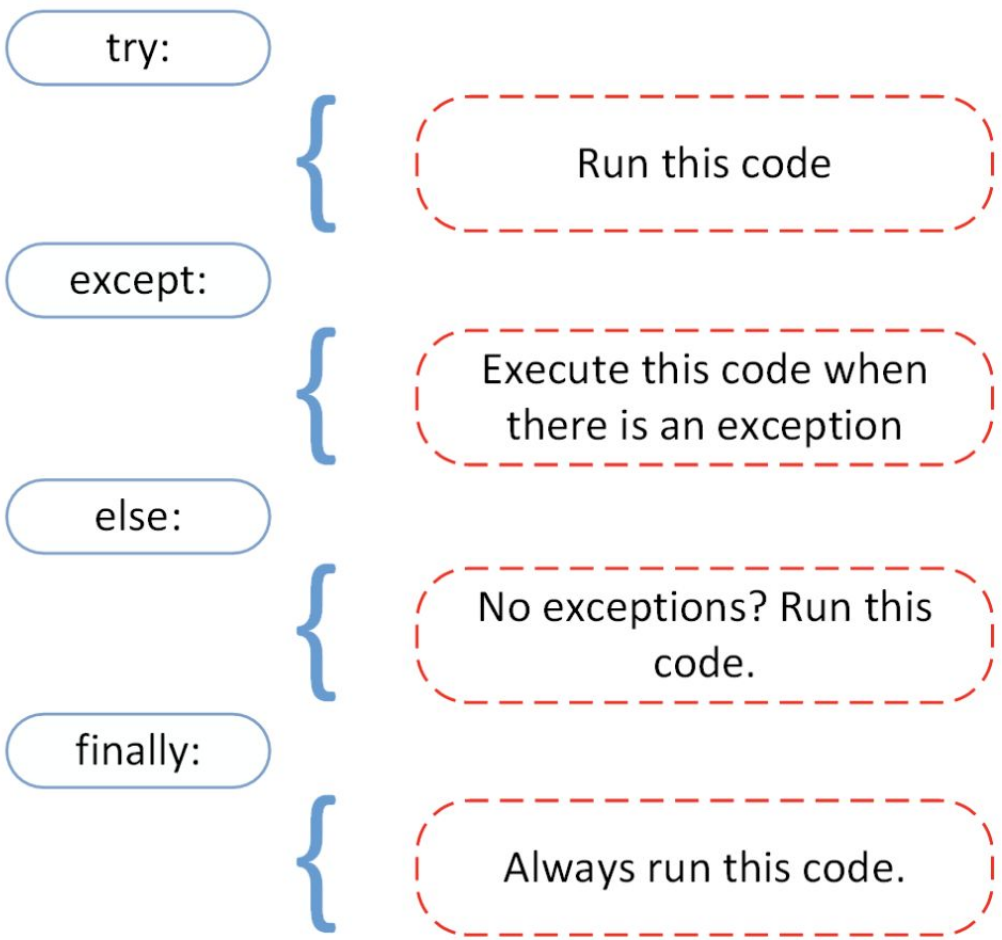
```
class TakeOffError(BaseException)
```

Errores y jerarquía de errores

- Si queremos evitar que termine el programa y tenemos una estrategia para responder al error podemos utilizar los keyword **try / except**
- Ej

```
try:
    airplane.takeoff()
except TakeOffError as error:
    airplane.land()
```

Errores y jerarquía de errores



Errores y jerarquía de errores

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
```

¿Qué es la programación funcional?

¿Qué es la programación funcional?

- La programación funcional consiste en utilizar funciones puras como pieza base de nuestros programas
- **Funciones puras** son aquellas que no tienen estado, no causan efectos secundarios y dependen únicamente de sus parámetros para producir un resultado
- El lenguaje debe implementar **first class functions**
  - Es decir, las funciones deben poderse pasar como argumentos de otras funciones

Python en el mundo real

Ciencias

Ciencias

- Astropy
- Biopython
- Sympy
- Numpy
- Pandas
- Matplotlib
- Scipy
- Sunpy
- Tomopy

6. Python en el mundo real

CLI

CLI

- aws
- gcloud
- rebound
- geeknote

6. Python en el mundo real

Aplicaciones web

Aplicaciones web

- Django
- Flask
- Bottle
- Chalice
- Webapp2
- Gunicorn
- Tornado

7. Conclusiones

Python 2 vs 3

Python 2 vs 3

- print
- integer division
- unicode
- range y xrange
- input vs raw_input
- raising errors
- handling exceptions
- __future__
- six library
  - https://six.readthedocs.io/

7. Conclusiones

PEPs

PEPs

- Los PEPs son Python Enhancement Proposals
  - Describen cambios al lenguaje o a los estándares alrededor
- Pueden ser de tres tipos:
  - Standards
    - Describen un nuevo feature o comportamiento
  - Informational
    - Describen un problema de diseño, una guía general, o información para la comunidad
  - Process
    - Describen un proceso relacionado con Python, pero no al código fuente de Python
      - Ej. cambios en los procesos de toma de decisiones

PEPs

- PEP8
  - Python style guide
- PEP257
  - Python docstrings
- PEP20
  - import this

- https://www.python.org/dev/peps/