
python
TM

Introducción

Introducción

- Su nombre es por los Monty Python (creado en los 90)
- Sintaxis simple, clara y sencilla
- Lenguaje interpretado o de script
- Tipado dinámico
- Fuertemente tipado
- Multiplataforma
- Orientado a objetos

¿Quienes lo usan?

¿Quienes lo usan?

- Gigantes de Internet: Google, Yahoo!, Instagram
- Juegos: Battefield 2, Civilization 4…
- NASA, Nokia, Thawte, IBM…
- Está presente en todo Linux

https://wiki.python.org/moin/OrganizationsUsingPython

Python 3

La increíble pero triste historia

Python 3: la increíble pero triste historia

- En 2009 se lanza Python 3 estando en la 2.6
- No tiene retrocompatibilidad, por lo que requiere reescribir código para ser compatible…FAIL
- La más utilizada es la 2.x (la última es 2.7.6) aunque se dejará de dar definitivamente soporte a 2.X para por fin saltar a Python 3

http://docs.python.org/3.1/whatsnew/3.0.html

PyCharm

Nuestro IDE de desarrollo

http://www.jetbrains.com/pycharm/download/

Tipos básicos

Tipos básicos

- Números enteros: **77**
- Números enteros long: **77L**
- Representación octal: **077**
- Representación hexadecimal: **0x77**

Tipos básicos

- Números reales (coma flotante): **3.1416** o **1e-2**
- Números complejos: **7 + 5j**
- Cadenas de texto: **“Hola mundo”** o **‘Hola mundo’**
- Booleanos: **True** y **False**

Variables

entero = 77

entero_long = 77L

octal = 077

hexadecimal = 0x77

coma_flotante = 3.1416

Variables

numero_complejo = 7 + 5j

verdadero = True

falso = False

cadena_doble = “A guan ban buluba, balán ban bú”

cadena_simple = ‘Tutti-frutti, au-rutti’

Cadenas de texto

quijote = “En un lugar de la Mancha…”

quijote = ‘En un lugar de la Mancha…’

quijote = “En un lugar de la \“Mancha\”…”

quijote = ‘En un lugar de la \’Mancha\’…’

Cadenas multilínea

quijote = """En un lugar de la Mancha, de cuyo nombre no quiero acordarme[3], no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor…."""

quijote = """
En un lugar de la Mancha, de cuyo nombre no quiero acordarme[3], no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor….
"""

Colecciones

Listas

["Apple", "Orange", "Watermelon"]

Las listas son los arrays de Python

Listas

fruits = [“Apple”, “Orange”, “Watermelon”]

apple = fruits[0]

orange = fruits[1]

watermelon = fruits[2]

Listas

```
fruits = [“Apple”, “Orange”, “Watermelon”]

apple = fruits[-3]

orange = fruits[-2]

watermelon = fruits[-1]
```

Listas

things = [“Lorem”, 22, True, [“Ipsum”, 3.1416]]

things[1] = 22

things[3][1] = 3.1416

Slicing de listas

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

from_2_to_6 = numbers[2:7]

greater_than_4 = numbers[5:]

less_than_4 = numbers[:4]

even = numbers[::2] # [0, 2, 4, 6, 8]

odd = numbers[1::2] # [1, 3, 5, 7, 9]
```

Operaciones con listas

```
fighters = [“Bud”, “Chuck”, “Bruce”]
```

Operaciones con listas

fighters.append(something)

fighters.append(“Goku”)

print fighters # devuelve ['Bud', 'Chuck', 'Bruce', ‘Goku’]

Operaciones con listas

fighters.count(something)

fighters.count(“Chuck”) # devuelve 1

fighters.count(“Alberto”) # devuelve 0

Operaciones con listas

fighters.extend(iterable)

```
fighters.extend([‘Goku’, ‘Mike’])

print fighters # devuelve ['Bud', 'Chuck', 'Bruce', ‘Goku’. ‘Mike’]
```

Operaciones con listas

fighters.index(value [, start [, stop]])

fighters.index(‘Chuck’) # devuelve 1

fighters.index(‘Alberto’) # lanza excepción ValueError

Operaciones con listas

fighters.insert(index, something)

```
fighters.insert(2, ’Goku’) # devuelve 1

print fighters # devuelve ['Bud', 'Chuck', 'Goku', 'Bruce']
```

Operaciones con listas

fighters.pop([index])

```
fighters.pop(2) # devuelve Bruce

fighters.pop() # devuelve Bud
```

Operaciones con listas

fighters.remove(value)

```
fighters.remove('Chuck')

print fighters # devuelve [‘Bud', ‘Bruce']

fighters.remove('Kirk') # lanza excepción ValueError
```

Operaciones con listas

fighters.reverse()

```
fighters.reverse()

print fighters # devuelve [‘Bruce', 'Chuck', 'Bud']
```

Operaciones con listas

fighters.sort(cmp=None, key=None, reverse=False)

```
fighters.sort()

print fighters # devuelve ['Bruce', 'Bud', 'Chuck']
```

Tuplas

(“Apple”, “Orange”, “Watermelon”)

Como las listas, pero inmutables

Diccionarios

me = {“name” : “Alberto Casero”, “country” : “Spain”}

Los diccionarios o arrays asociativos son colecciones que contienen elementos relacionados mediante una clave y un valor.

Diccionarios

me = {“name” : “Alberto Casero”, “country” : “Spain”}

name = me[“name”] # devuelve Alberto Casero

country = me[“country”] # devuelve Spain

me[“name”] = “Michael Jordan”

me[“country”] = “USA”

Operaciones con diccionarios

fighter = {“name”: “Bud Spencer”, “country” : “Italy”}

Operaciones con diccionarios

fighter.get(key [, d])

fighter.get(“name”) # devuelve “Bud Spencer”

fighter.get(“wins”, 0) # devuelve 0

Operaciones con diccionarios

fighter.has_key(key)

fighter.has_key(“name”) # devuelve True

fighter.has_key(“wins”) # devuelve False

Operaciones con diccionarios

fighter.items(key)

```
fighter.items()

# devuelve

# [(‘country', 'Italy'), ('name', 'Bud Spencer')]
```

Operaciones con diccionarios

fighter.keys()

```
fighter.keys() # ['country', 'name']
```

Operaciones con diccionarios

fighter.values()

```
fighter.values() # ['Italy', 'Bud Spencer']
```

Operaciones con diccionarios

fighter.pop(key [, d])

fighter.pop(“country”) # ‘Italy'

fighter.pop(“wins”, 0) # 0

Operadores

Operadores aritméticos

Suma	a + b	res = 2 + 3 # r es 5
Resta	a - b	res = 2 - 3 # r es -1
Multiplicación	a * b	res = 2 * 3 # r es 6
División	a / b	res = 2 / 3 # r es 0.66

Operadores aritméticos

Exponente	a ** b	res = 2 ** 3 # res es 8
Divisón entera	a // b	res = 2 // 3 # res es 0
Módulo	a % b	res = 2 % 3 # res es 2
Negación	-a	res = -5 # res es -5

Operadores booleanos

¿se cumple a y b?	a and b	res = True and False # res es False
¿se cumple a o b?	a or b	res = True or False # res es True
No se cumple a	not a	res = not True # res es False
¿a igual a b?	a == b	res = 5 == 5 # res es True
¿a distinto a b?	a != b	res = 5 != 5 # res es False

Operadores booleanos

¿a menor que b?	a < b	res = 5 < 5 # res es False
¿a menor o igual que b?	a <= b	res = 5 <= 5 # res es True
¿a mayor que b?	a > b	res = 6 > 5 # res es True
¿a mayor o igual que b?	a >= b	res = 5 >= 6 # res es False

Operadores booleanos

¿A está incluído en B?

res = “apple” in “Pineapple”
# res es True

res = 3 in [2, 3, 4, 5, 6]
# res es True

res = “title” in {“title” : “Wonderwall”, “artist” : “Oasis”}
# res es True

Operadores de cadenas

Concatenación	“Michael” + “Jordan”	name = “S.” + “Jobs” # name es “S.Jobs”
Multiplicación	“Michael” * 5	food = “monja” * 3 # food es “monjamonjamonja”

Operadores a nivel de bit

and	&	bits = 3 & 2 # bits es 2
or	|	bits = 3 | 2 # bits es 3
xor	^	bits = 3 ^ 2 # bits es 1
not	~	bits = ~3 # bits es -4
Desplazamiento izq	<<	bits = 3 << 1 # bits es 6
Desplazamiento der	>>	bits = 3 >> 1 # bits es 1

Control de flujo

if

```
if name == “Luke Skywalker”:
    print “I’m a Jedi”
```

En Python no hay llaves, lo que hay dentro de un if se indica con la sangría del texto (una tabulación o espacios).

if…else

if name == “Luke Skywalker”**:**

→ print “I’m a Jedi”

else:

→ print “I’m not Luke”

if…elif…elif…else

if name == “Luke Skywalker”**:**

→ print “I’m a Jedi”

elif name == “Chewbacca”**:**

→ print “Aaaaaaaaaaaahhhhhhh”

else:

→ print “I’m not Luke”

x if y else z

name = “Luke” **if** is_luke_skywalker **else** “Anakin”

while

while darth_vader_is_alive**:**

→ print “Luke, I’m your father”

print “Darth Vader is gone”

for…in

fruits = [“Apple”, “Peach”, “Watermelon”, “Orange”]

for fruit **in** fruits**:**

→ print “I’m eating”, fruit

print “I’m finished!”

Funciones

Definición

```
def fight( fighter_one, fighter_two ):
    """Esta funcion enfrenta a dos luchadores y devuelve el ganador. ¡Vamos Rocky!"""
    if fighter_one == “Rocky Balboa”:
        return fighter_one
    else:
        return fighter_two
```

Llamada o ejecución

```
winner = fight(“Rocky”, “Apollo”)

winner = fight(“Tyson”, “Holyfield”)
```

Parámetros por defecto

```
def  fight (fighter_one, fighter_two, mode="boxing"):
    ...

winner = fight("Rocky", "Apollo")

winner = fight("Riu", "Ken", "karate")
```

Cambiando el orden de los parámetros

```
def  fight (fighter_one, fighter_two, mode="boxing", rounds=1):
      ...

winner = fight("Rocky", "Apollo", rounds=12)

winner = fight("Riu", "Ken", rounds=3, mode="karate")
```

Parámetros variables (*args)

```
def  fight_club (*fighters):
        """Esta función hace ganar a Chuck Norris"""
for fighter in fighters:
        if fighter == “Chuck Norris”:
                return fighter
return None
```

Parámetros variables (*args)

fight_club (“Tyler Durden”, “John Doe”)

fight_club (“Bud Spencer”, “Chuck Norris”, “Van Damme”)

fight_club (“Jackie Chan”, “Bruce Lee”, “Bruce Willis”, “Jet Li”)

Parámetros variables como diccionarios (**kwargs)

```
def  fight_club (**kwargs):

    """Esta función hace ganar a Chuck Norris"""

    if “fighters” in kwargs:

        …

    if “mode” in kwargs:

        ….
```

Parámetros variables como diccionarios (**kwargs)

```
fight_club (fighters=[“Riu”, “Ken”], mode=“street”, rounds=3)

old_school = [“Bud Spencer”, “Chuck Norris”, “Van Damme”]

fight_club (fighters= old_school, mode=“freestyle”, rounds=0)

kung_fu = [“Jackie Chan”, “Bruce Lee”, “Bruce Willis”, “Jet Li”]

fight_club (fighters= old_school, mode=“kung_fu”, blood=True)
```

¿Referencia o valor?

- En Python, los parámetros de las funciones siempre se pasan por referencia.

- Es decir, que si modificas su valor dentro de la función, queda modificado también fuera.

- Aunque, como en Python hay objetos inmutables, como las tuplas, algunos no pueden ser modificados.

- En general se ven modificados: listas, diccionarios y objetos.

Funciones como parámetros

```
def fight_club (fighters, engine):
    for fighter1 in fighters:
        for fighter2 in fighters:
            winner = engine(fighter1, fighter2)
```

Funciones como parámetros

fight_club (fighters=[“Riu”, “Ken”], **engine=street_fighter**)

old_school = [“Bud Spencer”, “Chuck Norris”, “Van Damme”]

fight_club (fighters=old_school, **engine=strongest_survive**)

Funciones anónimas

- Se utiliza el operador lambda
- Sólo pueden tener una sola expresión
- No necesita return
- Se utiliza mucho con la función filter
- Sintaxis

  lambda <param1>, <param2>: <instructions>

Funciones anónimas

```
exp = lambda n, m: n**m

print exp(2, 2) # devuelve 4

print exp(3, 2) # devuelve 9

print (lambda n: n**2)(2) # devuelve 4

print (lambda n: n**2)(3) # devuelve 9
```

Decoradores

Un decorador es una función que recibe una función como parámetro y devuelve otra función como resultado.

Decoradores

```
def logger(function):

    def wrapper(*args, **kwargs):

        print "Arguments: %s, %s" % (args, kwargs)

        return function(*args, **kwargs)

    return wrapper
```

Decoradores

decorated_fn = logger(fight_club)

decorated_fn(“Bud Spencer”, “Chuck Norris”)

>> Arguments: (‘Bud Spencer’, ‘Chuck Norris’)

>> Chuck Norris

Decoradores

```
@logger

def fight_club(*figthters):
    ...

fight_club(“Bud Spencer”, “Chuck Norris”)

>> Arguments: (‘Bud Spencer’, ‘Chuck Norris’)

>> Chuck Norris
```

Decoradores

@winner_email_notification

@logger

def fight_club(*figthters):

    ...

Orientación a Objetos

Definición de una clase

```
class Fighter(object):

    """Finally, we have a fighter class"""
```

Definición de una clase

```
class Fighter:

    """This is an old-style Python class"""
```

Definición de métodos

```
class Fighter:

    """Finally, we have a fighter class"""

    def __init__(self, name):
        self.name = name
        …

def warmup(self):

        …

def fight(self, other_fighter):

        …
```

Herencia

```
class Champion(Fighter):

    """Champion extends figther"""
```

Herencia Multiple

```
class KungFuChampion(Champion, KungFuFighter):

    """A class that extends Champion and
KungFuFighter"""
```

Métodos públicos y privados

```
class Fighter(object):

    ...

    def __private_method(self):

        ...

    def public_method(self):

        ...

    def _protected_method(self): # realmente es público

        ...
```

Métodos especiales

def __init__(self, args):

Método llamado automáticamente tras crear el objeto para inicializar atributos.

def __new__(cls, args):

Es un método estático que se ejecuta antes de __init__. Se encarga de construir y devolver el objeto.

def __del__(self):

Destructor, se ejecuta cuando un elemento va a ser eliminado.

Métodos especiales

def __str__(self, args):

Para crear una cadena de texto que represente al obeto. Se llama cuando usamos str() o print.

def __cmp__(self, other):

Método llamado cuando se utilizan métodos de comparación.
Debe devolver un número negativo si nuestro objeto es menor, cero si son iguales, y un número positivo si nuestro objeto es mayor.

def __len__(self):

Método llamado para comprobar la longitud del objeto usando la función len().

Métodos estáticos

```
class Fighter(object):

    ...

    @staticmethod

    def a_static_method(args):

        ...
```

Métodos de clase

```
class Fighter(object):

    ...

    @classmethod

    def a_class_method(cls, args):

        ...
```

Atributos de clase

class Fighter(object):

styles = [“Freestyle”, “Boxing”, Kung-fu”]

. . .

Clases abstractas

- Python no tiene clases abstractas como tal.
- La única aproximación es una clase con métodos que simplemente lanzan una excepción NotImplemented.

Clases abstractas

```
class Fighter(object):

    def warmup(self):

        raise NotImplementedError
```

Instanaciación

fighter = **Fighter(“Bud Spencer”)**

kung_fu_fighter = **KungFuFighter(“Bruce Lee”)**

Llamadas a métodos

fighter**.warmup()**

kung_fu_fighter**.warmup()**

fighter**.fight(kung_fu_fighter)**

Acceso a atributos

fighter**.name**

king_fu_fighter**.style**

Getters y setters

```
class Fighter(object):

    ...

    def __getattr__(self, name):

        return self.style if name == “style” else None

    def __setattr__(self, name, value):

        self.style = value if name == “style” else self.style
```

Llamadas a super

```
class KungFuFighter(Fighter):

    def __init__(self):
        super(KungFuFighter, self).__init__()

    def fight(self, other_fighter):
        super(KungFuFighter, self).fight(other_fighter)
```

Excepciones

Excepciones

try:

    ...

except:

    print “An exception happened”

Excepciones

```
try:
    ...
except NameError:
    print “A NameError exception happened”
except ValueError:
    print “A ValueError exception happened”
except:
    print “An unknown exception happened”
```

Excepciones

```
try:
    ...
except NameError as err:
    print “A NameError exception happened: ”, err
except ValueError as err:
    print “A ValueError exception happened: ”, err
except:
    print “An unknown exception happened”
```

Excepciones

```
try:
    ...
except (NameError, ValueError):
    print “A NameError or ValueError exception happened”
except:
    print “An unknown exception happened”
```

Excepciones

```
try:
    ...
except:
    print “An exception happened”
else:
    print “Everything it’s ok”
```

Excepciones

```
try:
    ...
except:
    print “An exception happened”
finally:
    print “Always happens”
```

Lanzando Excepciones

raise Exception(“Exception raised by me”)

Tipos de excepciones

BaseException: Clase de la que heredan todas las excepciones.

Exception(BaseException): Super clase de todas las excepciones que no sean de salida.

GeneratorExit(Exception): Se pide que se salga de un generador.

StandardError(Exception): Clase base para todas las excepciones que no tengan que ver con salir del intérprete.

ArithmeticError(StandardError): Clase base para los errores aritméticos.

Tipos de excepciones

FloatingPointError(ArithmeticError): Error en una operación de coma flotante.

OverflowError(ArithmeticError): Resultado demasiado grande para poder representarse.

ZeroDivisionError(ArithmeticError): Lanzada cuando el segundo argumento de una operación de división o módulo era 0.

AssertionError(StandardError): Falló la condición de un estamento assert.

AttributeError(StandardError): No se encontró el atributo.

Tipos de excepciones

EOFError(StandardError): Se intentó leer más allá del final de fichero.

EnvironmentError(StandardError): Clase padre de los errores relacionados con la entrada/salida.

IOError(EnvironmentError): Error en una operación de entrada/salida.

OSError(EnvironmentError): Error en una llamada a sistema.

WindowsError(OSError): Error en una llamada a sistema en Windows

Tipos de excepciones

ImportError(StandardError): No se encuentra el módulo o el elemento del módulo que se quería importar.

LookupError(StandardError): Clase padre de los errores de acceso.

IndexError(LookupError): El índice de la secuencia está fuera delrango posible.

KeyError(LookupError): La clave no existe.

MemoryError(StandardError): No queda memoria suficiente.

NameError(StandardError): No se encontró ningún elemento con ese nombre

Tipos de excepciones

UnboundLocalError(NameError): El nombre no está asociado a ninguna variable.

ReferenceError(StandardError): El objeto no tiene ninguna referencia fuerte apuntando hacia él.

RuntimeError(StandardError): Error en tiempo de ejecución no especificado.

NotImplementedError(RuntimeError): Ese método o función no está implementado.

SyntaxError(StandardError): Clase padre para los errores sintácticos.

Tipos de excepciones

IndentationError(SyntaxError): Error en la indentación del archivo.

TabError(IndentationError): Error debido a la mezcla de espacios y tabuladores.

SystemError(StandardError): Error interno del intérprete.

TypeError(StandardError): Tipo de argumento no apropiado.

ValueError(StandardError): Valor del argumento no apropiado.

Tipos de excepciones

UnicodeError(ValueError): Clase padre para los errores relacionados con unicode.

UnicodeDecodeError(UnicodeError): Error de decodificación unicode.

UnicodeEncodeError(UnicodeError): Error de codificación unicode.

UnicodeTranslateError(UnicodeError): Error de traducción unicode.

Tipos de excepciones

StopIteration(Exception): Se utiliza para indicar el final del iterador.

Warning(Exception): Clase padre para los avisos.

DeprecationWarning(Warning): Clase padre para avisos sobre características obsoletas.

FutureWarning(Warning): Aviso. La semántica de la construcción cam- biará en un futuro.

ImportWarning(Warning): Aviso sobre posibles errores a la hora de importar.

PendingDeprecationWarning(Warning): Aviso sobre características que se marcarán como obsoletas en un futuro próximo.

RuntimeWarning(Warning): Aviso sobre comportmaientos dudosos en tiempo de ejecución.

Tipos de excepciones

SyntaxWarning(Warning): Aviso sobre sintaxis dudosa.

UnicodeWarning(Warning): Aviso sobre problemas relacionados con

Unicode, sobre todo con problemas de conversión.

UserWarning(Warning): Clase padre para avisos creados por el progra- mador.

KeyboardInterrupt(BaseException): El programa fué interrumpido por el usuario.

SystemExit(BaseException): Petición del intérprete para terminar la ejecución.

Módulos

Módulos

- En Python, los programas pueden dividirse en módulos

- Cada archivo de Python, es un módulo

Importando módulos

import <modulename>

Importando módulos

```
# fight_club.py

figthters = [“Tyler Durden”, “Jonh Doe”]

def fight(fighter_one, fighter_two):

    ...
```

Importando módulos

import fight_club

fighter_one = **fight_club.fighters**[0]

fighter_two = **fight_club.fighters**[1]

winner = **fight_club.fight**(fighter_one, fighter_two)

Importando módulos

from fight_club import fighters, fight

fighter_one = **fighters**[0]

fighter_two = **fighters**[1]

winner = **fight**(fighter_one, fighter_two)

Importando módulos

from fight_club import fighters as luchadores

from fight_club import fight as lucha

fighter_one = **luchadores**[0]

fighter_two = **luchadores**[1]

winner = **lucha**(fighter_one, fighter_two)

Paquetes

Paquetes

- Los paquetes sirven para organizar los módulos
- Los paquetes son directorios que tienen un archivo __init__.py
- Este archivo puede estar vacío (lo habitual) o usarlo para definir “constantes”.

Entornos virtuales

Entornos virtuales

- Los entornos virtuales nos permiten tener diferentes entornos para desarrollar diferentes proyectos sin que se mezclen versiones de librerías

- Por así decirlo, nos permite tener una especia de “máquina virtual” para desarrollar independiente a las demás.

Instalación UNIX

```
# Linux, Mac OS X
$ sudo pip install virtualenv

# Fedora
$ sudo yum install python-virtualenv

# Debian, Ubuntu
$ sudo apt-get install python-virtualenv
```

Activación

```
user$ /path/to/virtualenv/bin/activate

(virtualenv)user$
```

Desactivación

```
(virtualenv)user$ deactivate

user$
```

Instalación de paquetes

(virtualenv)user$ pip install <package_name>

(virtualenv)user$ pip install -r requirements.txt

Ver qué paquetes hay instalados

```
(virtualenv)user$ pip freeze
gevent==1.0
greenlet==0.4.1
requests==1.2.3
wsgiref==0.1.2
```

Generación de archivo de requisitos

```
(virtualenv)user$ pip freeze > requirements.txt
```


python
TM