WIKIPEDIA
La enciclopedia libre

Python

Python es un lenguaje de alto nivel de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código, se utiliza para desarrollar aplicaciones de todo tipo, por ejemplo: Instagram, Netflix, Spotify, Panda3D, entre otros.[2] Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma.

Administrado por Python Software Foundation, posee una licencia de código abierto, denominada Python Software Foundation License.[3] Python se clasifica constantemente como uno de los lenguajes de programación más populares.[4]

Historia

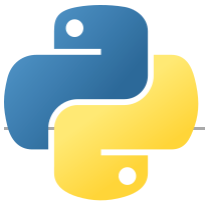
Python (https://myprogrammingschool.com/python-introduction-tutorial/) fue creado a finales de los años ochenta[5] por Guido van Rossum en Stichting Mathematisch Centrum (CWI), en los Países Bajos, como un sucesor del lenguaje de programación ABC, capaz de manejar excepciones e interactuar con el sistema operativo Amoeba.[6]

El nombre del lenguaje proviene de la afición de su creador por los humoristas británicos Monty Python.[7]

Guido van Rossum es el principal autor de Python, y su continuo rol central en decidir la dirección de Python es reconocido, refiriéndose a él como *Benevolente Dictador Vitalicio* (en inglés: *Benevolent Dictator for Life*, BDFL); sin embargo el 12 de julio de 2018 declinó de dicha situación de honor sin dejar un sucesor o sucesora y con una declaración altisonante:[8]

> Entonces, ¿qué van a hacer todos ustedes? ¿Crear una democracia? ¿Anarquía? ¿Una dictadura? ¿Una federación?
>
> Guido van Rossum[9]

Python



```
setOfNumbers = []

print("How many random numbers do you want to generate?")
max = int(input())

for i in range (max):
    setOfNumbers.append(random.randrange(1,101,1))
setOfNumbers.sort()
print(setOfNumbers)

print("Which number do you want to find in the set of random numbers")
searchNumber = int(input())

firstPos = 0
lastPos = max-1
found = False

while (not found and firstPos <= lastPos):
    midPos = int((firstPos + lastPos)/2)

    if (searchNumber == setOfNumbers[midPos]) :
        found = True
    else :
        if (searchNumber < setOfNumbers[midPos]):
            lastPos = midPos - 1
        else :
            firstPos = midPos + 1


if (found) :
    print("Your item is in the list")
else :
    print("Your item is not in the list")
```

Desarrollador(es)

Python Software Foundation

Sitio web oficial (https://www.python.org/)

Información general

El 20 de febrero de 1991, van Rossum publicó el código por primera vez en alt.sources (news:alt.sources), con el número de versión 0.9.0.[10] En esta etapa del desarrollo ya estaban presentes clases con herencia, manejo de excepciones, funciones y los tipos modulares, como: `str`, `list`, `dict`, entre otros. Además en este lanzamiento inicial aparecía un sistema de módulos adoptado de Modula-3; van Rossum describe el módulo como «una de las mayores unidades de programación de Python».[5] El modelo de excepciones en Python es parecido al de Modula-3, con la adición de una cláusula `else`.[6] En el año 1994 se formó comp.lang.python (news:comp.lang.python), el foro de discusión principal de Python, marcando un hito en el crecimiento del grupo de usuarios de este lenguaje.

Python alcanzó la versión 1.0 en enero de 1994. Una característica de este lanzamiento fueron las herramientas de la programación funcional: `lambda`, `reduce`, `filter` y `map`.[11] Van Rossum explicó que «hace 12 años, Python adquirió lambda, reduce(), filter() y map(), cortesía de Amrit Perm, un hacker informático de Lisp que las implementó porque las extrañaba».[12]

La última versión liberada proveniente de CWI fue Python 1.2. En 1995, van Rossum continuó su trabajo en Python en la Corporation for National Research Initiatives (CNRI) en Reston, Virginia, donde lanzó varias versiones del software.

Durante su estancia en CNRI, van Rossum lanzó la iniciativa *Computer Programming for Everybody* (CP4E), con el fin de hacer la programación más accesible a más gente, con un nivel de 'alfabetización' básico en lenguajes de programación, similar a la alfabetización básica en inglés y habilidades matemáticas necesarias por muchos trabajadores. Python tuvo un papel crucial en este proceso: debido a su orientación hacia una sintaxis limpia, ya era idóneo, y las metas de CP4E presentaban similitudes con su predecesor, ABC. El proyecto fue patrocinado por DARPA.[13] Para el año 2007, el proyecto CP4E se encontraba inactivo[14] ; a pesar de ello, Python continúa intentando ser fácil de aprender y no muy arcano en su sintaxis y semántica, con el objetivo de ser entendible incluso para no-programadores.

Extensiones comunes	`.py`, `.pyc`, `.pyd`, `.pyo`, `.pyw`, `.pyz`, `.pyi`
Paradigma	Multiparadigma: orientado a objetos, imperativo, funcional, reflexivo
Apareció en	1991
Diseñado por	Guido van Rossum
Última versión estable	3.12.2[1] (6 de febrero de 2024 (1 mes y 7 días))
Sistema de tipos	Fuertemente tipado, dinámico
Implementaciones	CPython, IronPython, Jython, Python for S60, PyPy, ActivePython, Unladen Swallow
Dialectos	Stackless Python, RPython
Influido por	ABC, ALGOL 68, C, Haskell, Icon, Lisp, Modula-3, Perl, Smalltalk, Java
Ha influido a	Boo, Cobra, D, Falcon, Genie, Groovy, Ruby, JavaScript, Cython, Go Latino
Sistema operativo	Multiplataforma
Licencia	Python Software Foundation License

En el año 2000, el equipo principal de desarrolladores de Python se cambió a BeOpen.com para formar el equipo BeOpen PythonLabs. CNRI pidió que la versión 1.6 fuera pública, continuando su desarrollo hasta que el equipo de desarrollo abandonó CNRI; su programa de lanzamiento y el de la versión 2.0 tenían una significativa cantidad de traslapo.[15] Python 2.0 fue el primer y único lanzamiento de BeOpen.com. Después que Python 2.0 fuera publicado por BeOpen.com, Guido van Rossum y los otros desarrolladores de PythonLabs se unieron en Digital Creations.

Python 2.0 tomó una característica mayor del lenguaje de programación funcional Haskell: listas por comprensión. La sintaxis de Python para esta construcción es muy similar a la de Haskell, salvo por la preferencia de los caracteres de puntuación en Haskell, y la preferencia de Python por palabras claves alfabéticas. Python 2.0 introdujo además un sistema de recolección de basura capaz de recolectar referencias cíclicas.[15]

Guido van Rossum, creador de **Python**, en la convención OSCON 2006

Posterior a este doble lanzamiento, y después que van Rossum dejara CNRI para trabajar con desarrolladores de software comercial, quedó claro que la opción de usar Python con software disponible bajo la GNU GPL era muy deseable. La licencia usada entonces, la Python License, incluía una cláusula estipulando que la licencia estaba gobernada por el estado de Virginia, por lo que, bajo la óptica de los abogados de Free Software Foundation (FSF), se hacía incompatible con GPL. Para las versiones 1.61 y 2.1, CNRI y FSF hicieron compatibles la licencia de Python con GPL, renombrándola como Python Software Foundation License. En el año 2001, van Rossum fue premiado con el FSF Award for the Advancement of Free Software.

Python 2.1 fue un trabajo derivado de las versiones 1.6.1 y 2.0. Es a partir de este momento que Python Software Foundation (PSF) pasa a dirigir el proyecto, organizada como una organización sin ánimo de lucro fundada en el año 2001, tomando como modelo a la Apache Software Foundation.[3] Incluida con este lanzamiento estuvo una implementación del alcance de variables más parecida a las reglas del static scoping originado por Scheme.[16]

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label)
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
   else:
      print '"];'
      children = []
      for n, child in enumerate(ast[1:]):
         children.append(dotwrite(child))
      print '    %s -> {' % nodename,
      for name in children:
         print '%s' % name,
```

Código Python con coloreado de sintaxis

Una innovación mayor en Python 2.2 fue la unificación de los tipos en Python (tipos escritos en C), y clases (tipos escritos en Python) dentro de una jerarquía. Esa unificación logró un modelo de objetos de Python puro y consistente.[17] También fueron agregados los generadores, que fueron inspirados por el lenguaje Icon.[18]

Las adiciones a la biblioteca estándar de Python y las decisiones sintácticas fueron influenciadas fuertemente por Java en algunos casos: el paquete `logging`,[19] introducido en la versión 2.3, está basado en log4j; el parser SAX, introducido en 2.0; el paquete `threading`,[20] cuya clase *Thread* expone un subconjunto de la interfaz de la clase homónima en Java.

Python 2.7.x (última versión de la serie Python 2.x) fue oficialmente descontinuado el 1 de enero de 2020 (paso inicialmente planeado para 2015), por lo que ya no se publicarán parches de seguridad y otras mejoras para él.[21] [22] Con el final del ciclo de vida de Python 2, solo tienen soporte la rama Python 3.6.x[23] y posteriores.

Con Python 3.5 llegaría el soporte incluido para entrada/salida asíncrona a través de la biblioteca `asyncio`, orientada a aplicaciones que requieren alto rendimiento de código concurrente, como servidores web, bibliotecas de conexión de bases de datos y colas de tareas distribuidas.[24]

En la actualidad, Python se aplica en los campos de inteligencia artificial y *machine learning*.[25]

Características y paradigmas

Python es un lenguaje de programación multiparadigma. Esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, programación imperativa y programación funcional. Otros paradigmas están soportados mediante el uso de extensiones.

Python usa tipado dinámico y conteo de referencias para la gestión de memoria.

Una característica importante de Python es la resolución dinámica de nombres; es decir, lo que enlaza un método y un nombre de variable durante la ejecución del programa (también llamado enlace dinámico de métodos).

Otro objetivo del diseño del lenguaje es la facilidad de extensión. Se pueden escribir nuevos módulos fácilmente en C o C++. Python puede incluirse en aplicaciones que necesitan una interfaz programable.[26]

Aunque la programación en Python podría considerarse en algunas situaciones hostil a la programación funcional tradicional expuesta por Lisp, existen bastantes analogías entre Python y los lenguajes minimalistas de la familia Lisp (como Scheme).

Filosofía

Los usuarios de Python se refieren a menudo a la **filosofía de Python**, que es bastante similar a la filosofía de Unix. El código que siga los principios de Python es reconocido como "pythónico". Estos principios fueron descritos por el desarrollador de Python Tim Peters en El Zen de Python:

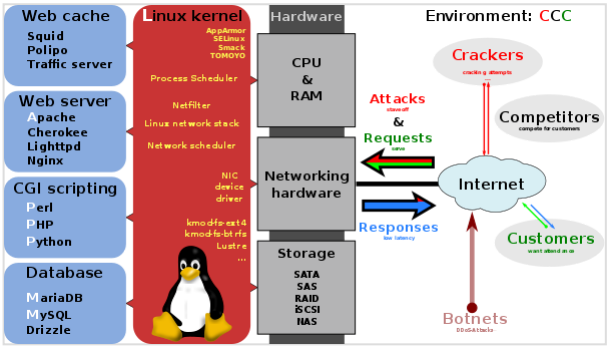
> - Bello es mejor que feo.
> - Explícito es mejor que implícito.
> - Simple es mejor que complejo.
> - Complejo es mejor que complicado.
> - Plano es mejor que anidado.

> - Disperso es mejor que denso.
> - La legibilidad cuenta.
> - Los casos especiales no son tan especiales como para quebrantar las reglas.
> - Lo práctico gana a lo puro.
> - Los errores nunca deberían dejarse pasar silenciosamente.
> - A menos que hayan sido silenciados explícitamente.
> - Frente a la ambigüedad, rechaza la tentación de adivinar.
> - Debería haber una —y preferiblemente solo una— manera obvia de hacerlo.
> - Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.[27]
> - Ahora es mejor que nunca.
> - Aunque *nunca* es a menudo mejor que *ya mismo*.
> - Si la implementación es difícil de explicar, es una mala idea.
> - Si la implementación es fácil de explicar, puede que sea una buena idea.
> - Los espacios de nombres (*namespaces*) son una gran idea. ¡Hagamos más de esas cosas!
>
> Tim Peters, El Zen de Python

Desde la versión 2.1.2, Python incluye estos puntos (en su versión original en inglés) como un huevo de pascua que se muestra al ejecutar `import this`.[28]

Modo interactivo

El intérprete de Python estándar incluye un *modo interactivo* en el cual se escriben las instrucciones en una especie de intérprete de comandos: las expresiones pueden ser introducidas una a una, pudiendo verse el resultado de su evaluación inmediatamente, lo que da la posibilidad de probar porciones de código en el modo interactivo antes de integrarlo como parte de un programa. Esto resulta útil tanto para las personas que se están familiarizando con el lenguaje como para los programadores más avanzados.



La LAMP comprende Python (aquí con Squid)

Existen otros programas, como IDLE, bpython (http://bpython-interpreter.org/) e IPython,[29] que añaden funcionalidades extra al modo interactivo, como completamiento automático de código y coloreado de la sintaxis del lenguaje.

Ejemplo del modo interactivo:

```
>>> 1 + 1
2
>>> a = range(10)
>>> print(list(a))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Elementos del lenguaje y sintaxis

Python está destinado a ser un lenguaje de fácil lectura. Su formato es visualmente ordenado y, a menudo, usa palabras clave en inglés donde otros idiomas usan puntuación. A diferencia de muchos otros lenguajes, no utiliza corchetes para delimitar bloques y se permiten puntos y coma después de las declaraciones, pero rara vez, si es que alguna vez, se utilizan. Tiene menos excepciones sintácticas y casos especiales que C o Pascal.

Diseñado para ser leído con facilidad, una de sus características es el uso de palabras donde otros lenguajes utilizarían símbolos. Por ejemplo, los operadores lógicos `!`, `||` y && en Python se escriben `not`, `or` y `and`, respectivamente.

El contenido de los bloques de código (bucles, funciones, clases, etc.) es delimitado mediante espacios o tabuladores, conocidos como sangrado o indentación, antes de cada línea de órdenes pertenecientes al bloque.[30] Python se diferencia así de otros lenguajes de programación que mantienen como costumbre declarar los bloques mediante un conjunto de caracteres, normalmente entre llaves `{}`.[31] [32] Se pueden utilizar tanto espacios como tabuladores para sangrar el código, pero se recomienda no mezclarlos.[33]

Función factorial en C (sangría opcional)

```c
int factorial(int x)
{
    if (x < 0 || x % 1 != 0) {
        printf("x debe ser un numero entero mayor o igual a 0");
        return -1; // Error
    }
    if (x == 0) {
        return 1;
    }
    return x * factorial(x - 1);
}
```

Función factorial en Python (sangría obligatoria)

```python
def factorial(x):
    assert x >= 0 and x % 1 == 0, "x debe ser un entero mayor o igual a 0."
    if x == 0:
        return 1
    else:
        return x * factorial(x - 1)
```

Debido al significado sintáctico de la sangría, cada instrucción debe estar contenida en una sola línea. No obstante, si por legibilidad se quiere dividir la instrucción en varias líneas, añadiendo una barra invertida \ al final de una línea, se indica que la instrucción continúa en la siguiente.

Estas instrucciones son equivalentes:

```
lista = ['valor 1', 'valor 2', 'valor 3']
cadena = 'Esto es una cadena bastante larga'
```

```
lista = ['valor 1', 'valor 2' \
      , 'valor 3']
cadena = 'Esto es una cadena ' \
       'bastante larga'
```

Comentarios

Los comentarios se pueden poner de dos formas. La primera y más apropiada para comentarios largos es utilizando la notación ''' comentario ''', tres apóstrofos de apertura y tres de cierre. La segunda notación utiliza el símbolo #, que se extiende hasta el final de la línea.

El intérprete no tiene en cuenta los comentarios, lo cual es útil si deseamos poner información adicional en el código. Por ejemplo, una explicación sobre el comportamiento de una sección del programa.

```
'''
Comentario más largo en una línea en Python
'''
print("Hola mundo")  # También es posible añadir un comentario al final de una línea de código
```

Variables

Las variables se definen de forma dinámica, lo que significa que no se tiene que especificar cuál es su tipo de antemano y puede tomar distintos valores en otro momento, incluso de un tipo diferente al que tenía previamente. Se usa el símbolo = para asignar valores.

```
x = 1
x = "texto"  # Esto es posible porque los tipos son asignados dinámicamente
```

Los nombres de variables pueden contener números y letras pero deben comenzar con una letra. Además, existen 35 palabras reservadas:[34] [35]

- `and`
- `as`
- `assert`
- `async`
- `continue`
- `def`
- `del`
- `elif`
- `finally`
- `for`
- `from`
- `global`
- `is`
- `lambda`
- `None`
- `nonlocal`
- `raise`
- `return`
- `True`
- `try`

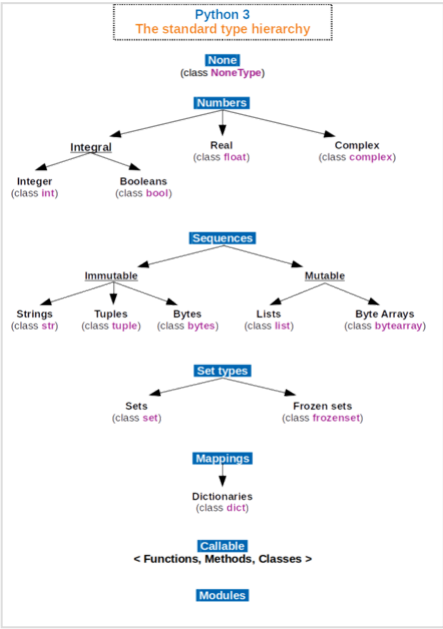
- `await`
- `break`
- `class`
- `else`
- `except`
- `False`
- `if`
- `import`
- `in`
- `not`
- `or`
- `pass`
- `while`
- `with`
- `yield`

A partir de Python 3.10 existen también *soft keywords*, palabras que son reservadas en ciertos contextos, pero que normalmente pueden ser usadas como nombres de variables. Estos identificadores son `match`, `case` y `_`.

Tipos de datos

Los *tipos de datos básicos* se pueden resumir en esta tabla:

Tipo	Clase	Notas	Ejemplo
`str`	Cadena en determinado formato de codificación (UTF-8 por defecto)	Inmutable	`'Cadena'`
`bytes`	Vector o *array* de bytes	Inmutable	`b'Cadena'`
`list`	Secuencia	Mutable, puede contener objetos de diversos tipos	`[4.0, 'Cadena', True]`
`tuple`	Secuencia	Inmutable, puede contener objetos de diversos tipos	`(4.0, 'Cadena', True)`
`set`	Conjunto	Mutable, sin orden, no contiene duplicados	`{4.0, 'Cadena', True}`
`frozenset`	Conjunto	Inmutable, sin orden, no contiene duplicados	`frozenset([4.0, 'Cadena', True])`
`dict`	Diccionario	Grupo de pares clave:valor	`{'key1': 1.0, 'key2': False}`
`int`	Número entero	Precisión arbitraria	`42`
`float`	Número decimal	Coma flotante de doble precisión	`3.1415927`
`complex`	Número complejo	Parte real y parte imaginaria *j*.	`(4.5 + 3j)`
`bool`	Booleano	Valor booleano (verdadero o falso)	`True` o `False`



Jerarquía de los tipos básicos en Python 3.

- Mutable: si su contenido (o dicho valor) puede cambiarse en tiempo de ejecución.

- Inmutable: si su contenido (o dicho valor) no puede cambiarse en tiempo de ejecución.

Condicionales

Una sentencia condicional ejecuta su bloque de código interno solo si se cumple cierta condición. Se define usando la palabra clave `if` seguida de la condición y el bloque de código. Si existen condiciones adicionales, se introducen usando la palabra clave `elif` seguida de la condición y su bloque de código. Las condiciones se evalúan de manera secuencial hasta encontrar la primera que sea verdadera, y su bloque de código asociado es el único que se ejecuta. Opcionalmente, puede haber un bloque final (la palabra clave `else`, seguida de un bloque de código) que se ejecuta solo cuando todas las condiciones anteriores fueron falsas.

```
>>> verdadero = True
>>> if verdadero:  # No es necesario poner "verdadero == True"
...     print("Verdadero")
... else:
...     print("Falso")
...
Verdadero
>>> lenguaje = "Python"
>>> if lenguaje == "C":  # lenguaje no es "C", por lo que este bloque se obviará y evaluará la siguiente condición
...     print("Lenguaje de programación: C")
... elif lenguaje == "Python":  # Se pueden añadir tantos bloques "elif" como se quiera
...     print("Lenguaje de programación: Python")
... else:  # En caso de que ninguna de las anteriores condiciones fuera cierta, se ejecutaría este bloque
...     print("Lenguaje de programación: indefinido")
...
Lenguaje de programación: Python
>>> if verdadero and lenguaje == "Python":  # Uso de "and" para comprobar que ambas condiciones son verdaderas
...     print("Verdadero y Lenguaje de programación: Python")
...
Verdadero y Lenguaje de programación: Python
```

Bucle for

El bucle *for* es similar a *foreach* en otros lenguajes. Recorre un objeto iterable, como una lista, una tupla o un generador, y por cada elemento del iterable ejecuta el bloque de código interno. Se define con la palabra clave `for` seguida de un nombre de variable, seguido de `in`, seguido del iterable, y finalmente el bloque de código interno. En cada iteración, el elemento siguiente del iterable se asigna al nombre de variable especificado:

```
>>> lista = ["a", "b", "c"]
>>> for i in lista:  # Iteramos sobre una lista, que es iterable
...     print(i)
```

```
...
a
b
c
>>> cadena = "abcdef"
>>> for i in cadena:  # Iteramos sobre una cadena, que también es iterable
...     print(i, end=', ')  # Añadiendo end=', ' al final hacemos que no introduzca un salto de línea, sino una coma y un espacio
...
a, b, c, d, e, f,
```

Bucle while

El bucle *while* evalúa una condición y, si es verdadera, ejecuta el bloque de código interno. Continúa evaluando y ejecutando mientras la condición sea verdadera. Se define con la palabra clave `while` seguida de la condición, y a continuación el bloque de código interno:

```
>>> numero = 0
>>> while numero < 10:
...     print(numero, end=" ")
...     numero += 1  # Un buen programador modificará las variables de control al finalizar el ciclo while
...
0 1 2 3 4 5 6 7 8 9
```

Listas y Tuplas

- Para declarar una *lista* se usan los corchetes `[]`, en cambio, para declarar una *tupla* se usan los paréntesis `()`. En ambas los elementos se separan por comas, y en el caso de las *tuplas* es necesario que tengan como mínimo una coma.
- Tanto las *listas* como las *tuplas* pueden contener elementos de diferentes tipos. No obstante, las *listas* suelen usarse para elementos del mismo tipo en cantidad variable mientras que las *tuplas* se reservan para elementos distintos en cantidad fija.
- Para acceder a los elementos de una *lista* o *tupla* se utiliza un índice entero (empezando por "0", no por "1"). Se pueden utilizar índices negativos para acceder elementos a partir del final.
- Las *listas* se caracterizan por ser mutables, es decir, se puede cambiar su contenido en tiempo de ejecución, mientras que las *tuplas* son inmutables ya que no es posible modificar el contenido una vez creadas.

Listas

```
>>> lista = ["abc", 42, 3.1415]
>>> lista[0]  # Acceder a un elemento por su índice
'abc'
>>> lista[-1]  # Acceder a un elemento usando un índice negativo
3.1415
```

```
>>> lista.append(True)  # Añadir un elemento al final de la lista
>>> lista
['abc', 42, 3.1415, True]
>>> del lista[3]  # Borra un elemento de la lista usando un índice (en este caso: True)
>>> lista[0] = "xyz"  # Re-asignar el valor del primer elemento de la lista
>>> lista[0:2]  # Mostrar los elementos de la lista del índice "0" al "2" (sin incluir este último)
['xyz', 42]
>>> lista_anidada = [lista, [True, 42]] # Es posible anidar listas
>>> lista_anidada
[['xyz', 42, 3.1415], [True, 42]]
>>> lista_anidada[1][0]  # Acceder a un elemento de una lista dentro de otra lista (del segundo elemento, mostrar el primer elemento)
True
```

Tuplas

```
>>> tupla = ("abc", 42, 3.1415)
>>> tupla[0]  # Acceder a un elemento por su índice
'abc'
>>> del tupla[0]  # No es posible borrar (ni añadir) un elemento en una tupla, lo que provocará una excepción
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
>>> tupla[0] = "xyz"  # Tampoco es posible re-asignar el valor de un elemento en una tupla, lo que también provocará una excepción
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tupla[0:2]  # Mostrar los elementos de la tupla del índice "0" al "2" (sin incluir este último)
('abc', 42)
>>> tupla_anidada = (tupla, (True, 3.1415))  # También es posible anidar tuplas
>>> 1, 2, 3, "abc"  # Esto también es una tupla, aunque es recomendable ponerla entre paréntesis (recuerde que requiere, al menos, una coma)
(1, 2, 3, 'abc')
>>> (1)  # Aunque se encuentra entre paréntesis, esto no es una tupla, ya que no posee al menos una coma, por lo que únicamente aparecerá el valor
1
>>> (1,)  # En cambio, en este otro caso, sí es una tupla
(1,)
>>> (1, 2)  # Con más de un elemento no es necesaria la coma final
(1, 2)
>>> (1, 2,)  # Aunque agregarla no modifica el resultado
(1, 2)
```

Diccionarios

- Para declarar un diccionario se usan las llaves `{}`. Contienen elementos separados por comas, donde cada elemento está formado por un par `clave:valor` (el símbolo `:` separa la clave de su valor correspondiente).
- Los diccionarios son mutables, es decir, se puede cambiar el contenido de un valor en tiempo de ejecución.

- En cambio, las claves de un diccionario deben ser inmutables. Esto quiere decir, por ejemplo, que no podremos usar ni listas ni diccionarios como claves.
- El valor asociado a una clave puede ser de cualquier tipo de dato, incluso un diccionario.

```
>>> diccionario = {"cadena": "abc", "numero": 42, "lista": [True, 42]}  # Diccionario que tiene diferentes valores por cada clave, incluso una lista
>>> diccionario["cadena"]  # Usando una clave, se accede a su valor
'abc'
>>> diccionario["lista"][0]  # Acceder a un elemento de una lista dentro de un valor (del valor de la clave "lista", mostrar el primer elemento)
True
>>> diccionario["cadena"] = "xyz"  # Re-asignar el valor de una clave
>>> diccionario["cadena"]
'xyz'
>>> diccionario["decimal"] = 3.1415927  # Insertar un nuevo elemento clave:valor
>>> diccionario["decimal"]
3.1415927
>>> diccionario_mixto = {"tupla": (True, 3.1415), "diccionario": diccionario}  # También es posible que un valor sea un diccionario
>>> diccionario_mixto["diccionario"]["lista"][1]  # Acceder a un elemento dentro de una lista, que se encuentra dentro de un diccionario
42
>>> diccionario = {("abc",): 42}  # Sí es posible que una clave sea una tupla, pues es inmutable
>>> diccionario = {["abc"]: 42}  # No es posible que una clave sea una lista, pues es mutable, lo que provocará una excepción
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Sentencia match-case

Python cuenta con la estructura match-case desde la versión 3.10. Esta tiene el nombre de **Structural Pattern Matching (https://www.python.org/dev/peps/pep-0636/)**.

```
match variable:
    case condicion:
        # codigo
    case condicion:
        # codigo
    case condicion:
        # codigo
    case _:
        # codigo
```

Cabe destacar que esta funcionalidad es considerablemente más compleja que el conocido *switch-case* de la mayoría de lenguajes de programación, ya que no solo permite realizar una comparación del valor, sino que también puede comprobar el tipo del objeto, y sus atributos. Además, puede realizar un desempaquetado directo de secuencias de datos, y comprobarlos de forma específica.

En el siguiente ejemplo, se comprueban los atributos de nuestra instancia de `Punto`. Si en estos no se cumple que `x = 10` y `y = 40`, se pasará a la siguiente condición.

Es importante anotar que `Punto(x=10, y=40)` no está construyendo un nuevo objeto, aunque pueda parecerlo.

```
from dataclasses import dataclass

@dataclass
class Punto:
    x: int
    y: int

coordenada = Punto(10, 34)

match coordenada:
    case Punto(x=10, y=40):  # los atributos "x" e "y" tienen el valor specificado
        print("Coordenada 10, 40")
    case Punto():  # si es una instancia de Punto
        print("es un punto")
    case _:  # ninguna condición cumplida (default)
        print("No es un punto")
```

En versiones anteriores, existen diferentes formas de realizar esta operación lógica de forma similar:

Usando if, elif, else

Podemos usar la estructura de la siguiente manera:

```
>>> if condicion1:
...     hacer1
>>> elif condicion2:
...     hacer2
>>> elif condicion3:
...     hacer3
>>> else:
...     hacer
```

En esa estructura se ejecutara controlando la condicion1, si no se cumple pasara a la siguiente y así sucesivamente hasta entrar en el else. Un ejemplo práctico sería:

```
>>> def calculo(op, a, b):
...     if op == 'sum':
...         return a + b
```

```
...     elif op == 'rest':
...          return a - b
...     elif op == 'mult':
...         return a * b
...     elif op == 'div':
...         return a / b
...     else:
...          return None
>>>
>>> print(calculo('sum',3,4))
7
```

Podríamos decir que el lado negativo de la sentencia armada con if, elif y else es que si la lista de posibles operaciones es muy larga, las tiene que recorrer una por una hasta llegar a la correcta.

Usando diccionarios

Podemos usar un diccionario para el mismo ejemplo:

```
>>> def calculo(op, a, b):
...     return {
...         'sum': lambda: a + b,
...         'rest': lambda: a - b,
...         'mult': lambda: a * b,
...         'div': lambda: a/b
...     }.get(op, lambda: None)()
>>>
>>> print(calculo('sum',3,4))
7
```

De esta manera, si las opciones fueran muchas, no recorrería todas; solo iría directamente a la operación buscada en la última línea (`.get(op, lambda: None)()`) y estaríamos dando una opción por defecto. El motivo por el que se usan expresiones lambda dentro del diccionario es para prevenir la ejecución de las instrucciones que contienen a la hora de definir el diccionario. Este únicamente define funciones como valores del diccionario, y posteriormente, al obtener estas mediante `get()`, se llama a la función, ejecutando la expresión que esta contiene.

Conjuntos

- Los conjuntos se construyen mediante la expresión `set(items)`, donde *items* es cualquier objeto iterable, como listas o tuplas. Los conjuntos no mantienen el orden ni contienen elementos duplicados.
- Se suelen utilizar para eliminar duplicados de una secuencia, o para operaciones matemáticas como intersección, unión, diferencia y diferencia simétrica.

```
>>> conjunto_inmutable = frozenset(["a", "b", "a"])  # Se utiliza una lista como objeto iterable
>>> conjunto_inmutable
frozenset(['a', 'b'])
>>> conjunto1 = set(["a", "b", "a"])  # Primer conjunto mutable
>>> conjunto1
set(['a', 'b'])
>>> conjunto2 = set(["a", "b", "c", "d"])  # Segundo conjunto mutable
>>> conjunto2
set(['a', 'c', 'b', 'd'])  # Los conjuntos no mantienen el orden, como los diccionarios
>>> conjunto1 & conjunto2  # Intersección
set(['a', 'b'])
>>> conjunto1 | conjunto2  # Unión
set(['a', 'c', 'b', 'd'])
>>> conjunto1 - conjunto2  # Diferencia (1)
set([])
>>> conjunto2 - conjunto1  # Diferencia (2)
set(['c', 'd'])
>>> conjunto1 ^ conjunto2  # Diferencia simétrica
set(['c', 'd'])
```

Listas por comprensión

Una *lista por comprensión* (en inglés *list comprehension*) es una expresión compacta para definir listas. Al igual que `lambda`, aparece en lenguajes funcionales. Ejemplos:

```
>>> range(5)  # La función range devuelve una lista, empezando en 0 y terminando con el número indicado menos uno
[0, 1, 2, 3, 4]
>>> [i * i for i in range(5)]  # Por cada elemento del rango, lo multiplica por sí mismo y lo agrega al resultado
[0, 1, 4, 9, 16]
>>> lista = [(i, i + 2) for i in range(5)]
>>> lista
[(0, 2), (1, 3), (2, 4), (3, 5), (4, 6)]
```

Funciones

- Las funciones se definen con la palabra clave `def`, seguida del nombre de la función y sus parámetros. Otra forma de escribir funciones, aunque menos utilizada, es con la palabra clave `lambda` (que aparece en lenguajes funcionales como Lisp).
- El valor devuelto en las funciones con `def` será el dado con la instrucción `return`.
- Las funciones pueden recibir parámetros especiales para manejar el exceso de argumentos.
  - El parámetro `*args` recibe como una tupla un número variable de argumentos posicionales.

- El parámetro `**kwargs` recibe como un diccionario un número variable de argumentos por palabras clave.

`def`:

```
>>> def suma(x, y=2):
...     return x + y  # Retornar la suma del valor de la variable "x" y el valor de "y"
...
>>> suma(4)  # La variable "y" no se modifica, siendo su valor: 2
6
>>> suma(4, 10)  # La variable "y" sí se modifica, siendo su nuevo valor: 10
14
```

`*args`:

```
>>> def suma(*args):
...     resultado = 0
...     # Se itera la tupla de argumentos
...     for num in args:
...         resultado += num #  Suma todos los argumentos
...     return resultado  # Retorna el resultado de la suma
...
>>> suma(2, 4)
6
>>> suma(1, 3, 5, 7, 9)  # No importa el número de variables posicionales que se pasen a la función
25
```

`**kwargs`:

```
def suma(**kwargs):
...     resultado = 0
...     # Se itera el diccionario de argumentos
...     for key, value in kwargs.items():
...         resultado += value  # Suma todos los valores de los argumentos
...     return resultado
...
>>>suma(x=1, y=3)
4
>>>suma(x=2, y=4, z=6)  # No importa el número de variables por clave que se pasen a la función
12
```

`lambda`:

```
>>> suma = lambda x, y=2: x + y
>>> suma(4)  # La variable "y" no se modifica, siendo su valor: 2
```

```
6
>>> suma(4, 10)  # La variable "y" sí se modifica, siendo su nuevo valor: 10
14
```

Clases

- Las clases se definen con la palabra clave `class`, seguida del nombre de la clase y, si hereda de otras clases, los nombres de estas.
- En Python 2.x era recomendable que una clase heredase de `object`, en Python 3.x ya no hace falta.
- En una clase, un *método* equivale a una función, y un *atributo* equivale a una variable.[36]
- `__init__` es un método especial que se ejecuta al instanciar la clase, se usa generalmente para inicializar atributos y ejecutar métodos necesarios. Al igual que todos los métodos en Python, debe tener al menos un parámetro (generalmente se utiliza `self`). El resto de parámetros serán los que se indiquen al instanciar la clase.
- Los atributos que se desee que sean accesibles desde fuera de la clase se deben declarar usando `self.` delante del nombre.
- En Python no existe el concepto de encapsulamiento[37] , por lo que el programador debe ser responsable de asignar los valores a los atributos.

```
>>> class Persona():
...     def __init__(self, nombre, edad):
...         self.nombre = nombre  # Un atributo cualquiera
...         self.edad = edad  # Otro atributo cualquiera
...     def mostrar_edad(self):  # Es necesario que, al menos, tenga un parámetro, generalmente self
...         print(self.edad) # mostrando un atributo
...     def modificar_edad(self, edad):  # Modificando edad
...         if 0 > edad < 150:  # Se comprueba que la edad no sea menor que 0 (algo imposible) ni mayor que 150 (algo realmente difícil)
...             return False
...         else:  # Si está en el rango 0-150, entonces se modifica la variable
...             self.edad = edad  # Se modifica la edad
...
>>> p = Persona('Alicia', 20)  # Instanciando la clase. Como se puede ver, no se especifica el valor de self
>>> p.nombre  # La variable "nombre" del objeto sí es accesible desde fuera
'Alicia'
>>> p.nombre = 'Andrea'  # Y por tanto, se puede cambiar su contenido
>>> p.nombre
'Andrea'
>>> p.mostrar_edad()  # Se llama a un método de la clase
20
>>> p.modificar_edad(21)  # Es posible cambiar la edad usando el método específico que hemos hecho para hacerlo de forma controlada
>>> p.mostrar_edad()
21
```

Módulos

Existen muchas propiedades que se pueden agregar al lenguaje importando módulos, conjuntos de funciones y clases para realizar determinadas tareas usualmente escritos también en Python. Un ejemplo es el módulo tkinter[38] , que permite crear interfaces gráficas basadas en la biblioteca Tk. Otro ejemplo es el módulo `os`, que provee acceso a muchas funciones del sistema operativo. Los módulos se agregan al código escribiendo la palabra `import`, seguida del nombre del módulo que queramos usar.[39]

Instalación de módulos (pip)

La instalación de módulos en Python se puede realizar mediante la herramienta de software Pip, que suele estar incluida en las instalaciones de Python. Esta herramienta permite la gestión de los distintos paquetes o módulos instalables para Python, incluyendo así las siguientes características:

- Instalación de paquetes.
  - Instalación de versiones concretas de paquetes.
  - Instalación a partir de un archivo de configuración.
- Desinstalación.
- Actualización.

Interfaz al sistema operativo

El módulo **os** provee funciones para interactuar con el sistema operativo:

```
>>> import os
>>> os.name  # Devuelve el nombre del sistema operativo
'posix'
>>> os.mkdir("/tmp/ejemplo")  # Crea un directorio en la ruta especificada
```

Para tareas de administración de archivos, el módulo **shutil** provee una interfaz de más alto nivel:

```
>>> import shutil
>>> shutil.copyfile('datos.db', 'informacion.db')
'informacion.db'
>>> shutil.move('/build/programas', 'dir_progs')
'dir_progs'
```

Comodines de archivos

El módulo **glob** provee una función para crear listas de archivos a partir de búsquedas con comodines en carpetas:

```
>>> import glob
>>> glob.glob('*.py')
['numeros.py', 'ejemplo.py', 'ejemplo2.py']
```

Argumentos de línea de órdenes

Los argumentos de línea de órdenes se almacenan en el atributo argv del módulo **sys** como una lista.

```
>>> import sys
>>> print(sys.argv)
['demostracion.py', 'uno', 'dos', 'tres']
```

Matemática

El módulo **math** permite acceder a las funciones de matemática de punto flotante:

```
>>> import math
>>> math.cos(math.pi / 3)
0.494888338963
>>> math.log(1024, 2)
10.0
```

El módulo **random** se utiliza para realizar selecciones al azar:

```
>>> import random
>>> random.choice(['durazno', 'manzana', 'frutilla'])
'durazno'
>>> random.sample(range(100), 10)   # Elección sin reemplazo
[30, 23, 17, 24, 8, 81, 41, 80, 28, 13]
>>> random.random()  # Un float al azar en el intervalo [0, 1)
0.23370387692726126
>>> random.randrange(6)  # Un entero al azar en el intervalo [0, 6)
3
```

El módulo **statistics** se utiliza para estadística básica, por ejemplo: media, mediana, varianza, etc.:

```
>>> import statistics
>>> datos = [1.75, 2.75, 1.25, 0.5, 0.25, 1.25, 3.5]
```

```
>>> statistics.mean(datos)
1.6071428571428572
>>> statistics.median(datos)
1.25
>>> statistics.variance(datos)
1.3720238095238095
```

Fechas y horas

Los módulos **time** y **datetime** permiten trabajar con fechas y horas.

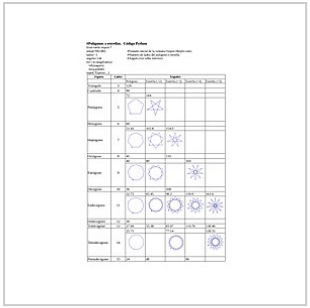
```
>>> from datetime import datetime
>>> import time
>>> datetime.now().isoformat()  # Devuelve la fecha y hora actual
'2010-08-10T18:01:17.900401'
>>> datetime.now().strftime("%Y-%m-%d %H:%M:%S")  # Devuelve la fecha y/u hora actual con el formato especificado
'2010-08-10 18:01:17'
>>> time.strftime("%Y-%m-%d %H:%M:%S")  # Método equivalente
'2010-08-10 18:01:17'
```

Módulo Turtle

El módulo **turtle** permite la implementación de gráficas tortuga:

```
>>> import turtle
>>> turtle.pensize(2)
>>> turtle.left(120)
>>> turtle.forward(100)
```

Polígonos con el módulo Turtle:



Polígonos regulares y estrellas

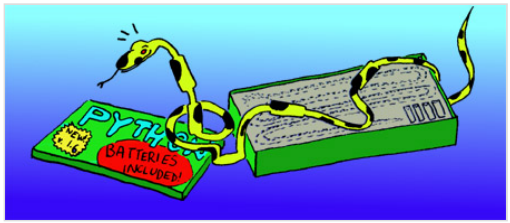
Sistema de objetos

En Python todo es un objeto (incluso las clases). Las clases, al ser objetos, son instancias de una metaclase. Python, además, soporta herencia múltiple y polimorfismo.

```
>>> cadena = "abc"  # Una cadena es un objeto de "str"
>>> cadena.upper()  # Al ser un objeto, posee sus propios métodos
'ABC'
>>> lista = [True, 3.1415]  # Una lista es un objeto de "list"
>>> lista.append(42)  # Una lista (al igual que todo) es un objeto, y también posee sus propios métodos
>>> lista
[True, 3.1415, 42]
```

Biblioteca estándar

Python tiene una gran biblioteca estándar, usada para una diversidad de tareas. Esto viene de la filosofía "pilas incluidas" (*batteries included*) en referencia a los módulos de Python. Los módulos de la biblioteca estándar pueden complementarse con módulos personalizados escritos en C o en Python. Debido a la gran variedad de herramientas incluidas en la biblioteca estándar, combinada con la capacidad de usar lenguajes de bajo nivel como C y C++ (los cuales son capaces de interactuar con otras bibliotecas), Python es un lenguaje que combina su clara sintaxis con el inmenso poder de lenguajes de más bajo nivel.[40]



Python viene con "pilas incluidas"

Implementaciones

Existen diversas implementaciones del lenguaje:

- CPython es la implementación original, disponible para varias plataformas en el sitio oficial de Python.
- IronPython es la implementación para .NET.
- Stackless Python es la variante de CPython que trata de no usar el *stack* de C (www.stackless.com (http://www.stackless.com/)).
- Jython es la implementación hecha en Java.
- Pippy es la implementación realizada para Palm (pippy.sourceforge.net (http://pippy.sourceforge.net/)).
- PyPy es una implementación de Python escrita en Python y optimizada mediante JIT (pypy.org (http://pypy.org/)).
- ActivePython es una implementación privativa de Python con extensiones, para servidores en producción y aplicaciones de misión crítica desarrollado por ActiveState Software.

Incidencias

A lo largo de su historia, Python ha presentado una serie de incidencias, de las cuales las más importantes han sido las siguientes:

- El 13 de febrero de 2009 se lanzó una nueva versión de Python bajo el nombre clave *"Python 3000"* o,[41] abreviado, *"Py3K"*.[42] Esta nueva versión incluye toda una serie de cambios que requieren reescribir el código de versiones anteriores. Para facilitar este proceso, junto con Python 3 se ha publicado una herramienta de traducción automática llamada **2to3**.[43] [44]
- El sistema operativo Windows 10, a partir de su actualización de mayo de 2019, dispone de la característica de preinstalación asistida del lenguaje Python y varias de sus herramientas adicionales.[45]

Véase también

- PyPI, repositorio de paquetes de software de terceros para Python.
- Django, framework de desarrollo web.
- Cython, lenguaje de programación para simplificar la escritura de módulos de extensión para Python en C y C++.
- Flask, framework de desarrollo web.
- CubicWeb, framework de desarrollo web en plataforma semántica.
- Pygame, conjunto de módulos para la creación de videojuegos en dos dimensiones.
- Tkinter, binding de la biblioteca gráfica Tcl/Tk para Python.
- PyGTK, binding de la biblioteca gráfica GTK para Python.
- wxPython, binding de la biblioteca gráfica wxWidgets para Python.
- PyQt y PySide, bindings de la biblioteca gráfica Qt para Python.

- Plone, sistema de gestión de contenidos.
- Biopython, colección de bibliotecas orientadas a la bioinformática para Python.
- NumPy, biblioteca que da soporte al cálculo con matrices y vectores.
- SciPy, biblioteca que permite realizar análisis científico como optimización, álgebra lineal, integración y ecuaciones diferenciales, entre otras operaciones.
- Pandas, biblioteca que permite el análisis de datos a través de series y *dataframes*.
- Pyomo, colección de paquetes de software de Python para formular modelos de optimización
- Scikit-learn, biblioteca que implementa algoritmos de aprendizaje automático.

Referencias

1. «Changelog - Python Documentation» (https://docs.python.org/3.12/whatsnew/changelog.html). python.org. Consultado el 5 de octubre de 2023.
2. «¿Qué es Python?» (https://web.archive.org/web/20200224120525/https://luca-d3.com/es/data-speaks/diccionario-tecnologico/python-lenguaje) (html). *LUCA*. Archivado desde el original (https://luca-d3.com/es/data-speaks/diccionario-tecnologico/python-lenguaje) el 24 de febrero de 2020. Consultado el 24 de febrero de 2020.
3. History and License (https://docs.python.org/3/license.html)
4. «TIOBE Index - TIOBE» (https://www.tiobe.com/tiobe-index). *www.tiobe.com*. Consultado el 1 de mayo de 2023.
5. «artima - The Making of Python» (https://www.artima.com/articles/the-making-of-python). *www.artima.com*. Consultado el 2 de mayo de 2023.
6. «Why was Python created in the first place?» (https://docs.python.org/faq/general#why-was-python-created-in-the-first-place). General Python FAQ.
7. «1. Whetting Your Appetite» (https://docs.python.org/3/tutorial/appetite.html). *Python documentation*. Consultado el 2 de mayo de 2023.
8. Tannhausser (12 de julio de 2018). «Guido van Rossum dimite como líder de Python» (https://web.archive.org/web/20180712235229/https://lamiradadelreplicante.com/2018/07/12/guido-van-rossum-dimite-como-lider-de-python/) (html). *La Mirada del Replicante*. Archivado desde el original (https://lamiradadelreplicante.com/2018/07/12/guido-van-rossum-dimite-como-lider-de-python/) el 12 de julio de 2018. Consultado el 21 de julio de 2018. «Como veis no solo transfiere el poder, sino que evita designar sucesor y deja en manos de los corel developers la tarea de organizar como será la transición, así como el modelo de gobierno en un futuro. »
9. van Rossum, Guido (12 de julio de 2018). «[python-committers] Transfer of power» (https://web.archive.org/web/20180712225051/https://www.mail-archive.com/python-committers@python.org/msg05628.html) (html). *Mail Archive Com* (en inglés). Archivado desde el original (https://www.mail-archive.com/python-committers@python.org/msg05628.html) el 12 de julio de 2018. Consultado el 21 de julio de 2018. «I am not going to appoint a successor. So what are you all going to do? Create a democracy? Anarchy? A dictatorship? A federation? »
10. van Rossum, Guido (20 de enero de 2009). «A Brief Timeline of Python» (http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html). *The History of Python* (en inglés). Consultado el 14 de febrero de 2021.
11. Chacón Sartori, Camilo. *Computación y programación funcional : introducción al cálculo lambda y la programación funcional usando Racket y Python*. [Barcelona]: Marcombo. ISBN 8426732437.
12. The fate of reduce() in Python 3000 (http://www.artima.com/weblogs/viewpost.jsp?thread=98196)
13. Computer Programming for Everybody (https://web.archive.org/web/20090223101648/http://python.org/doc/essays/cp4e.html)

14. Index of /cp4e (https://web.archive.org/web/20070312152257/http://www.python.org/cp4e/)
15. What's New in Python 2.0 (https://web.archive.org/web/20070329043037/http://www.amk.ca/python/2.0/)
16. PEP 227 -- Statically Nested Scopes (https://www.python.org/dev/peps/pep-0227/)
17. PEPs 252 and 253: Type and Class Changes (https://docs.python.org/whatsnew/2.2.html#peps-252-and-253-type-and-class-changes)
18. PEP 255: Simple Generators (https://docs.python.org/whatsnew/2.2.html#pep-255-simple-generators)
19. PEP 282 -- A Logging System (https://www.python.org/dev/peps/pep-0282/)
20. threading — Higher-level threading interface (https://docs.python.org/library/threading.html)
21. «Sunsetting Python 2» (https://www.python.org/doc/sunset-python-2/) (en inglés). *python.org*. 21 de enero de 2020.
22. «PEP 373 -- Python 2.7 Release Schedule» (https://www.python.org/dev/peps/pep-0373/) (en inglés). *python.org*. 21 de enero de 2020.
23. «Python Developer's Guide — Python Developer's Guide» (https://devguide.python.org/#status-of-python-branches) (en inglés). *devguide.python.org*. 21 de enero de 2020.
24. «asyncio — E/S asíncrona» (https://docs.python.org/es/3/library/asyncio.html). *docs.python.org*. Consultado el 19 de marzo de 2023.
25. «Machine Learning (aprendizaje automático) con Python: una introducción práctica» (https://www.edx.org/course/machine-learning-aprendizaje-automatico-con-python). *edX* (en inglés). Consultado el 6 de julio de 2020.
26. Rocky. «Applications for Python» (https://www.codelivly.com/applications-for-python/).
27. "Holandés" hace referencia a Guido van Rossum, el autor del lenguaje de programación Python, que es holandés. También hace referencia a la gran concentración de desarrolladores holandeses conocidos en relación con otras nacionalidades.
28. PEP 20 -- The Zen of Python (https://www.python.org/dev/peps/pep-0020/)
29. «Copia archivada» (https://web.archive.org/web/20180804135112/http://ipython.scipy.org/). Archivado desde el original (http://ipython.scipy.org/) el 4 de agosto de 2018. Consultado el 25 de febrero de 2010.
30. Python Software Foundation. «More control flow options» (https://docs.python.org/2/tutorial/controlflow.html#defining-functions). *Python v2.7.8 Documentation* (en inglés). Consultado el 20 de julio de 2014.
31. Eric Huss. «Function Definition» (https://web.archive.org/web/20150118141700/http://www.acm.uiuc.edu/webmonkeys/book/c_guide/index.html). *The C Library Reference Guide* (en inglés). Archivado desde el original (http://www.acm.uiuc.edu/webmonkeys/book/c_guide/index.html) el 18 de enero de 2015. Consultado el 20 de julio de 2014.
32. Álvarez, Miguel Ángel (2 de noviembre de 2001). «Funciones en Javascript» (http://www.desarrolloweb.com/articulos/583.php). *desarrolloweb.com* (en inglés). Consultado el 20 de julio de 2014.
33. David Goodger. «Code Like a Pythonista: Idiomatic Python» (https://web.archive.org/web/20140527204143/http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#whitespace-1). *Python.net* (en inglés). Archivado desde el original (http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#whitespace-1) el 27 de mayo de 2014. Consultado el 20 de julio de 2014.
34. Downey, Allen; Elkner, Jeffrey (1 de abril de 2002). «Aprenda a Pensar Como un Programador con Python» (https://web.archive.org/web/20171023174011/https://argentinaenpython.com/quiero-aprender-python/aprenda-a-pensar-como-un-programador-con-python.pdf) (pdf). *Argentina Python*. p. 40. Archivado desde el original (https://argentinaenpython.com/quiero-aprender-python/aprenda-a-pensar-como-un-programador-con-python.pdf) el 23 de octubre de 2017. Consultado el 21 de marzo de 2020.
35. «2. Análisis léxico» (https://docs.python.org/3/reference/lexical_analysis.html). *Python documentation*. docs.python.org. Consultado el 19 de marzo de 2023.
36. Recuero de los Santos, Paloma (13 de mayo de 2020). «Python para todos: Diferencia entre método y función» (https://web.archive.org/web/20200514004933/https://empresas.blogthinkbig.com/python-para-todos-metodo-vs-funcion/) (html). Archivado desde el original

(https://empresas.blogthinkbig.com/python-para-todos-metodo-vs-funcion/) el 14 de mayo de 2020. Consultado el 13 de mayo de 2020.
37. Encapsulación en Python (http://www.genbetadev.com/python/cazadores-de-mitos-las-propiedades-privadas-en-python)
38. «Python GUI Programming With Tkinter» (https://www.codelivly.com/python-gui-programming-with-tkinter/). *Codelivly*. 2022.
39. «Pequeño paseo por la Biblioteca Estándar» (https://web.archive.org/web/20170915165232/http://docs.python.org.ar/tutorial/3/stdlib.html). *Tutorial de Python (y Django!) en Español*. Archivado desde el original (http://docs.python.org.ar/tutorial/3/stdlib.html) el 15 de septiembre de 2017. Consultado el 16 de agosto de 2017.
40. «La Biblioteca Estándar de Python» (https://docs.python.org/es/3.9/library/index.html). *docs.python.org*. Consultado el 26 de abril de 2021.
41. Python 3.0.1 (https://www.python.org/downloads/release/python-301/)
42. PEP 3000 -- Python 3000 (https://www.python.org/dev/peps/pep-3000/)
43. 2to3 - Automated Python 2 to 3 code translation (https://docs.python.org/3.1/library/2to3.html)
44. Novedades de Python 3.0 (https://docs.python.org/3.1/whatsnew/3.0.html)
45. Dower, Steve (21 de mayo de 2019). «Who put Python in the Windows 10 May 2019 Update?» (https://devblogs.microsoft.com/python/python-in-the-windows-10-may-2019-update/) (html). *Microsoft Blog* (en inglés). Consultado el 23 de mayo de 2019.

Bibliografía

- Knowlton, Jim (2009). *Python*. tr: Fernández Vélez, María Jesús (1 edición). Anaya Multimedia-Anaya Interactiva. ISBN 978-84-415-2513-9.
- Martelli, Alex (2007). *Python. Guía de referencia*. tr: Gorjón Salvador, Bruno (1 edición). Anaya Multimedia-Anaya Interactiva. ISBN 978-84-415-2317-3.

Enlaces externos

- Wikilibros: Python