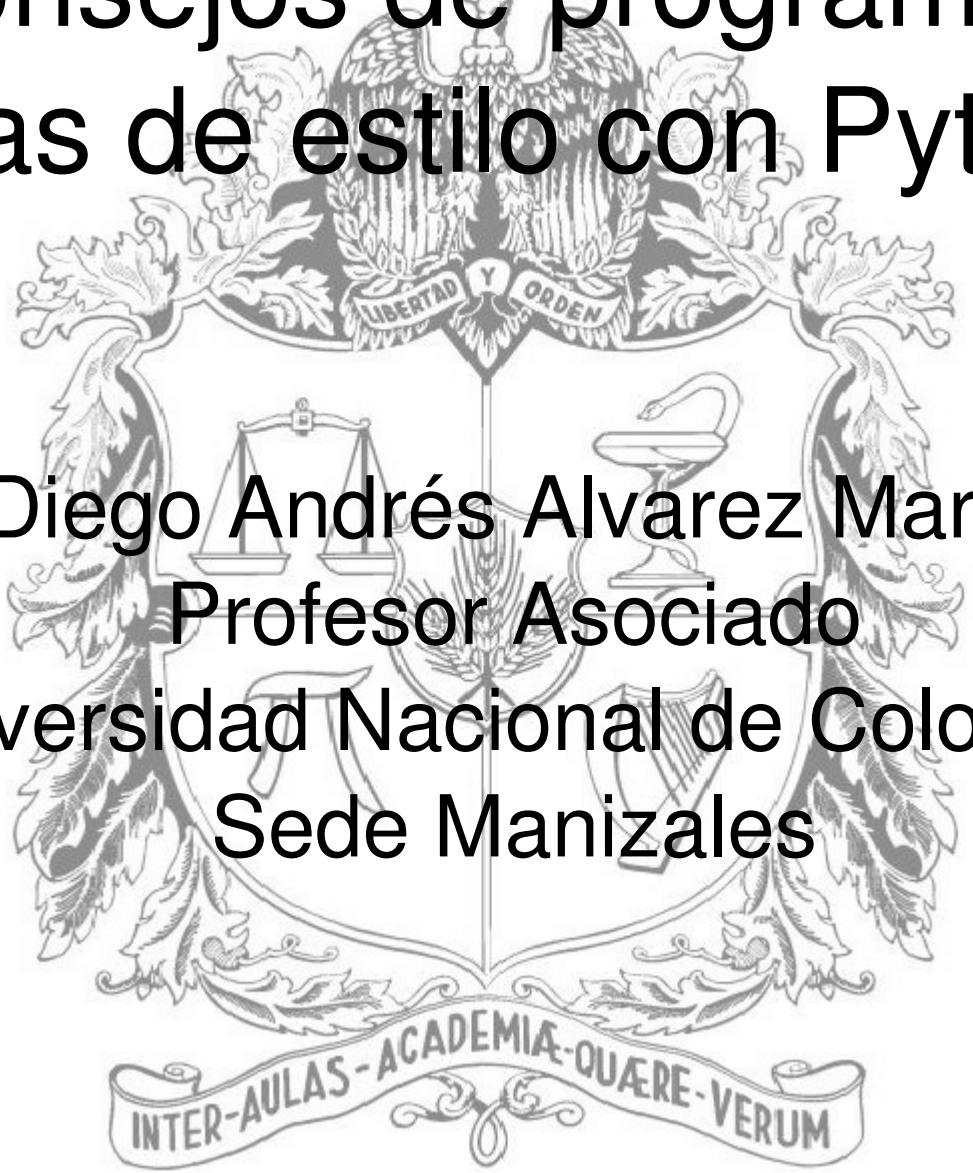
07 - Consejos de programación y normas de estilo con Python 3

Diego Andrés Alvarez Marín
Profesor Asociado
Universidad Nacional de Colombia
Sede Manizales



True == True

Fíjate en la línea 8 de este programa:

es_primo.py

```
1  número = 7
2
3  creo_que_es_primo = True
4  for divisor in range(2, número):
5      if número % divisor == 0:
6          creo_que_es_primo = False
7
8  if creo_que_es_primo:
9      print('El número {0} es primo.'.format(número))
10 else:
11     print('El número {0} no es primo.'.format(número))
```

La condición del if es muy extraña, ¿no? No hay comparación alguna. ¿Qué condición es esa? Muchos estudiantes optan por esta fórmula alternativa para las líneas 8 y similares:

es_primo.py

```
1  número = 7
2
3  creo_que_es_primo = True
4  for divisor in range(2, número):
5      if número % divisor == 0:
6          creo_que_es_primo = False
7
8  if creo_que_es_primo == True:
9      print('El número {0} es primo.'.format(número))
10 else:
11     print('El número {0} no es primo.'.format(número))
```

Les parece más natural porque de ese modo se compara el valor de *creo_que_es_primo* con algo. Pero, si lo piensas bien, esa comparación es superflua: a fin de cuentas, el resultado de la comparación *creo_que_es_primo* == True es True, precisamente lo que ya vale *creo_que_es_primo*.

No es que esté mal efectuar esa comparación extra, sino que no aporta nada y resta legibilidad. Evítala si puedes.

Índice de bucle for-in: ¡prohibido asignar!

Hemos aprendido que el bucle **for-in** utiliza una variable índice a la que se van asignando los diferentes valores del rango. En muchos ejemplos se utiliza la variable *i*, pero solo porque también en matemáticas los sumatorios y productorios suelen utilizar la letra *i* para indicar el nombre de su variable índice. Puedes usar cualquier nombre de variable válido.

Pero que el índice sea una variable cualquiera no te da libertad absoluta para hacer con ella lo que quieras. En un bucle **for-in**, las variables de índice solo deben usarse para consultar su valor, nunca para asignarles uno nuevo. Por ejemplo, este fragmento de programa es incorrecto:

```
1 for i in range(0, 5):
2     i += 2
```

Y ahora que sabes que los bucles pueden anidarse, también has de tener mucho cuidado con sus índices. Un error frecuente entre primerizos de la programación es utilizar el mismo índice para dos bucles anidados. Por ejemplo, estos bucles anidados están mal:

```
1 for i in range(0, 5):
2     for i in range(0, 3):
3         print(i)
```

En el fondo, este problema es una variante del anterior, pues de algún modo se está asignando nuevos valores a la variable *i* en el bucle interior, pero *i* es la variable del bucle exterior y asignarle cualquier valor está prohibido.

Recuerda: *nunca debes asignar un valor a un índice de bucle **for-in** ni usar la misma variable índice en bucles anidados.*

Distribución recomendada del código en un programa

- Se hace un comentario inicial (docstring) que describa brevemente la finalidad del programa.
- Se llaman las librerías (import)
- Se definen las constantes globales **que sean llamadas desde las funciones**
- Se reserva el espacio en memoria para las variables globales que lo requieran y **que sean llamadas desde las funciones**
- Se crean las funciones
- Finalmente se hace el procedimiento principal
- Las variables locales a un procedimiento se inicializan tan pronto como se necesiten.

Importaciones, definiciones de función y programa principal

Los programas que diseñes a partir de ahora tendrán tres «tipos de línea»: importación de módulos (o funciones y variables de módulos), definición de funciones y sentencias del programa principal. En principio puedes alternar líneas de los tres tipos. Mira este programa, por ejemplo,

```
1  def cuadrado(x):
2      return x**2
3
4  mivector = []
5  for i in range(3):
6      mivector.append(float(input('Dame un número: ')))
7
8  def suma_cuadrados(vector):
9      suma = 0
10     for elemento in vector:
11         suma += cuadrado(elemento)
12     return suma
13
14 s = suma_cuadrados(mivector)
15
16 from math import sqrt
17 print('Distancia al origen:', sqrt(s))
```

En él se alternan definiciones de función, importaciones de funciones y sentencias del programa principal, así que resulta difícil hacerse una idea clara de qué hace el programa. No diseñes así tus programas.

Importaciones, definiciones de función y programa principal (y II)

Esta otra versión del programa anterior pone en primer lugar las importaciones, a continuación, las funciones y, al final, de un tirón, las sentencias que conforman el programa principal:

```
1  from math import sqrt
2
3  def cuadrado(x):
4      return x**2
5
6  def suma_cuadrados(vector):
7      suma = 0
8      for elemento in vector:
9          suma += cuadrado(elemento)
10     return suma
11
12 # Programa principal
13 mivector = []
14 for i in range(3):
15     mivector.append(float(input('Dame un número: ')))
16 s = suma_cuadrados(mivector)
17 print('Distancia al origen:', sqrt(s))
```

Es mucho más legible. Te recomendamos que sigas siempre esta organización en tus programas. Recuerda que la legibilidad de los programas es uno de los objetivos del programador.

Parámetros o teclado

Un error frecuente al diseñar funciones consiste en tratar de obtener la información directamente de teclado. No es que esté prohibido, pero es ciertamente excepcional que una función obtenga la información de ese modo. Cuando te pidan diseñar una función que recibe uno o más datos, se sobreentiende que debes suministrarlos como argumentos en la llamada, no leerlos de teclado. Cuando queramos que la función lea algo de teclado, lo diremos *explícitamente*.

Insistimos, y esta vez ilustrando el error con un ejemplo. Imagina que te piden que diseñes una función que diga si un número es par devolviendo True si es así y False en caso contrario. Te piden una función como esta:

```
1 def es_par(n):
2     return n % 2 == 0
```

Muchos programadores novatos escriben *erróneamente* una función como esta otra:

```
1 def es_par():
2     n = int(input('Dame un número: '))
3     return n % 2 == 0
```

Está mal. Escribir esa función así demuestra, cuando menos, falta de soltura en el diseño de funciones. Si hubiésemos querido una función como esa, te hubiésemos pedido una función que lea de teclado un número entero y devuelva True si es par y False en caso contrario.

Condicionales que trabajan directamente con valores lógicos

Ciertas funciones devuelven directamente un valor lógico. Considera, por ejemplo, esta función, que nos dice si un número es o no es par:

```
def es_par(n):
    return n % 2 == 0
```

Si una sentencia condicional toma una decisión en función de si un número es par o no, puedes codificar así la condición:

```
if es_par(n):
    ...
```

Observa que no hemos usado comparador alguno en la condición del if. ¿Por qué? Porque la función *es_par*(*n*) devuelve True o False directamente. Los programadores primerizos tienen tendencia a codificar la misma condición así:

```
if es_par(n) == True:
    ...
```

Es decir, comparan el valor devuelto por *es_par* con el valor True, pues les da la sensación de que un if sin comparación no está completo. No pasa nada si usas la comparación, pero es innecesaria. Es más, si no usas la comparación, el programa es más legible: la sentencia condicional se lee directamente como «si *n* es par» en lugar de «si *n* es par es cierto», que es un extraño circunloquio.

Si deseas comprobar que el número es impar, puedes hacerlo así:

```
if not es_par(n):
    ...
```

Es muy legible: «si no es par *n*». Los programadores que están empezando escriben:

```
if es_par(n) == False:
    ...
```

que se lee como «si *n* es par es falso». Peor, ¿no?

Acostúmbrate a usar la versión que no usa operador de comparación. Es más legible.

Valor de retorno o pantalla

Te hemos mostrado de momento que es posible imprimir información directamente por pantalla desde una función (o procedimiento). Ojo: solo lo hacemos cuando el propósito de la función es mostrar esa información. Muchos aprendices que no han comprendido bien el significado de la sentencia **return**, la sustituyen por una sentencia *print*. Mal. Cuando te piden que diseñes una función que *devuelva* un valor, te piden que lo haga con la sentencia **return**, que es la única forma válida (que conoces) de devolver un valor. Mostrar algo por pantalla no es devolver ese algo. Cuando quieran que muestres algo por pantalla, te lo dirán explícitamente.

Supón que te piden que diseñes una función que reciba un entero y devuelva su última cifra. Te piden esto:

```
def última_cifra(número):
    return número % 10
```

No te piden esto otro:

```
def última_cifra(número):
    print(número % 10)
```

Fíjate en que la segunda definición hace que la función no pueda usarse en expresiones como esta:

```
a = última_cifra(10293) + 1
```

Como *última_cifra* no *devuelve* nada, ¿qué valor se está sumando a 1 y guardando en *a*?

¡Ah! Aún se puede hace peor. Hay quien define la función así:

```
def última_cifra():
    número = int(input('Dame un número: '))
    print(número % 10)
```

No solo demuestra no entender qué es el valor de retorno; además, demuestra que no tiene ni idea de lo que es el paso de parámetros. Evita dar esa impresión: lee bien lo que se pide y usa parámetros y valor de retorno a menos que se te diga explícitamente lo contrario. Lo normal es que la mayor parte de las funciones produzcan datos (devueltos con **return**) a partir de otros datos (obtenidos con parámetros) y que el programa principal o funciones muy específicas lean de teclado y muestren por pantalla.

Normas de estilo

Unas normas de estilo en programación, son tan importantes que todas las empresas dedicadas a programación imponen a sus empleados una mínima uniformidad, para facilitar el intercambio de programas y la modificación por cualquier empleado, sea o no el programador inicial. Por supuesto, cada programa debe ir acompañado de una documentación adicional, que aclare detalladamente cada módulo del programa, objetivos, algoritmos usados, ficheros... (ver por ejemplo https://code.google.com/p/google-styleguide/)

No existen un conjunto de reglas fijas para programar con legibilidad, ya que cada programador tiene su modo y sus manías y le gusta escribir de una forma determinada. Lo que sí existen son un conjunto de reglas generales, que aplicándolas, en mayor o menor medida, se consiguen programas bastante legibles. Aquí intentaremos resumir estas reglas.

Normas de estilo en la programación

- Lo principal es mejorar tanto como se pueda la claridad en el código. El código debe ser leíble por usted mismo (en el futuro) y por otras personas sin dificultad.
- Cuando se escriben los programas se debe priorizar la legibilidad del código por encima de todo, incluso sobre la velocidad de ejecución (excepto en aplicaciones muy específicas).
- Hay que entender que el código normalmente se escribe una vez, pero se lee decenas de veces.
- Sea consistente con su estilo de programación.

Líneas

- Una línea no debe sobrepasar la columna 80.
- Use líneas en blanco para separar el código en párrafos o bloques.
- Coloque una instrucción por línea; sin embargo, si el colocar dos o más sentencias en una misma línea mejora la claridad del programa, entonces hágalo así.
- Evite líneas muy largas. En vez de esto utilice varias líneas.

Example of implicit line continuation:

```
a = some_function(
    '1' + '2' + '3' + '4')
```

Example of explicit line continuation:

```
a = '1' + '2' + \
    '3' + '4'
```

```
# Recommended
total = (first_variable
         + second_variable
         - third_variable)
```

```
# Not Recommended
total = (first_variable +
         second_variable -
         third_variable)
```

Espacios

- Coloque espacios antes y después de cada operador binario y de cada signo =, así como los espacios separan las palabras en un texto.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Use los espacios para resaltar la prioridad de los operadores:

```
# Recommended
y = x**2 + 5
z = (x+y) * (x-y)

# Not Recommended
y = x ** 2 + 5
z = (x + y) * (x - y)
```

```
# Recommended
if x>5 and x%2==0:
    print('x is larger than 5 and divisible by 2!')
```

```
# Not recommended
if x > 5 and x % 2 == 0:
    print('x is larger than 5 and divisible by 2!')
```

Espacios e identaciones

- Use sangrados de 4 espacios (configure su editor). Evite usar tabuladores (tecla TAB).
- En una sentencia que consiste de dos o más líneas, cada línea excepto la primera debe identarse un nivel extra para indicar que es una continuación de la primera.

Ejemplos de identación adecuada

```
def function(arg_one, arg_two,
             arg_three, arg_four):
    return arg_one
```

```
x = 5
if (x > 3 and
        x < 10):
    print(x)
```

```
# Not Recommended
var = function(arg_one, arg_two,
    arg_three, arg_four)
```

```
# Not Recommended
def function(
    arg_one, arg_two,
    arg_three, arg_four):
    return arg_one
```

Utilice mejor una doble identación:

```
def function(
        arg_one, arg_two,
        arg_three, arg_four):
    return arg_one
```

Nombres de variables

- Use nombres de variables simples y descriptivos
- Use el guión bajo _ para mejorar la claridad de los nombres de las variables. Ejemplo: ancho_viga

```
>>> # No recomendado
>>> x = "Pepito Pérez"
>>> y, z = x.split()
>>> print(z, y, sep=', ')
Pérez, Pepito
>>>
>>> # Recomendado
>>> nombre = "Pepito Pérez"
>>> primer_nombre, apellido = nombre.split()
>>> print(apellido, primer_nombre, sep=", ")
Pérez, Pepito
```

Nombres de variables

- Las constantes se escriben en MAYÚSCULAS.
- No use la i maýuscula, la, L minúscula o la O mayúscula como nombres de variables, ya que pueden confundirse con el 1, el 1 y el 0 respectivamente. Ejemplo: “120 vs l2O”, “`120 vs l20`”, “`120 vs 12O`”
- Los nombres cortos tales como `x` o `y` son aceptables cuando su significado es claro y cuando un nombre largo no añadiría información o claridad adicional.
- i,j,k se utilizan exclusivamente como contadores en los ciclos for/while/do-while.

Variables

- Evite en lo posible sobreescribir las variables globales. Utilice en lo posible variables estáticas o en su defecto pase las variables globales como argumentos a la función.
- Declare las variables globales tan cerca al lugar donde las va a utilizar como sea posible.

Comentarios

- Comente el código mientras programa, no después de terminar de programar.
- Después de declarar una variable, escriba un comentario que explique para que sirve esa variable.
- Incluya comentarios antes de cada bloque de código. Estos describirán los propósitos del bloque en el cumplimiento de una tarea cohesiva. El uso de nombres significativos en variables y funciones minimizan la necesidad de comentarios.
- Procure alinear los comentarios que van en líneas seguidas

Comentarios

- Antes de definir una función se debe poner un comentario que diga en una línea o dos qué es lo que hacen dichas funciones y se expliquen los parámetros de la misma. Estos comentarios son para que una persona que esté explorando el código pueda entender qué hace una función sin tener que estudiarla en profundidad. Este breve comentario en el encabezado de la función debe apoyarse a su vez en un nombre de función descriptivo.

Planee la estructura de su código

- Antes de comenzar a programar, lo mejor es planear el código que se va a escribir en una hoja de papel.
- Distribuya su código en secciones: por ejemplo: sección de entrada de datos, sección de cálculos numéricos, sección de presentación de resultados

Evite hacer Hardcoding

http://en.wikipedia.org/wiki/Hard_coding

El Hard-coding hace referencia a una mala práctica en el desarrollo de programas que que consiste en incrustar datos directamente en el código fuente del programa, en lugar de obtener esos datos de una variable externa, parámetros de la línea de comandos, o un archivo.

Se considera como una mala costumbre de programación ya que requiere la modificación del código fuente cada vez que cambian los datos, cuando lo conveniente sería que el usuario final pudiera cambiar estos detalles fuera del código fuente del programa.

```
# Es mejor escribir:
docenas = 12
manzanas =  3*docenas
# que
manzanas = 36
```

```
# Es mejor escribir:
nombre = 'Jorge'
print('Hola', nombre)
# que
print('Hola Jorge')
```

Líneas en blanco

- Utilice las líneas en blanco para separar funciones, clases, y bloques de código dentro de las funciones. La finalidad de esto es dividir el código en bloques que hacen una función en específico de forma similar a como las líneas en blanco separan los párrafos de un texto.
- Escriba un comentario que indique que hace cada “párrafo” al principio de este.

Utilice aquellas construcciones que hagan el código más legible

```
# Not recommended
if word[:3] == 'cat':
    print('The word starts with "cat"')
```

However, this is not as readable as using `.startswith()`:

Python

```
# Recommended
if word.startswith('cat'):
    print('The word starts with "cat"')
```

PEP 0008 -- Style Guide for Python Code
El estilo recomendado para hacer programas en Python

https://www.python.org/dev/peps/pep-0008/

- Use 4-space indentation, and no tabs. Four spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.
- Wrap lines so that they don't exceed 79 characters. This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
- When possible, put comments on a line of their own.
- Use docstrings.

El estilo recomendado para hacer programas en Python

https://www.python.org/dev/peps/pep-0008/ (lealo!!)

- Use docstrings.
- Name your classes and functions consistently; the convention is to use CamelCase for classes and lower_case_with_underscores for functions and methods. Always use self as the name for the first method argument (see A First Look at Classes for more on classes and methods).
- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.
- Likewise, don't use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

El estilo recomendado para hacer los docstrings

https://www.python.org/dev/peps/pep-0257/ (lealo!!)

- Recuerde que si uno tiene buenos docstrings, a partir de ellos se puede generar, directamente del código, la documentación de su programa.

```
def quadratic(a, b, c, x):
    """Solve quadratic equation via the quadratic formula.

    A quadratic equation has the following form:
    ax**2 + bx + c = 0

    There always two solutions to a quadratic equation: x_1 & x_2.
    """
    x_1 = (- b+(b**2-4*a*c)**(1/2)) / (2*a)
    x_2 = (- b-(b**2-4*a*c)**(1/2)) / (2*a)

    return x_1, x_2
```

Use un “Linter” (removedor de motas)

Es una herramienta que analiza el código fuente con el objeto de señalar errores de programación, problemas estilísticos, construcciones sospechosas, etc.

Ejemplos:

- Nombres de variables mal escritos.
- Paréntesis que no se cerraron.
- Espaciado incorrecto en Python.
- Llamado a una función con un número incorrecto de parámetros.

Linters para Python

- Los que vienen con su IDE favorito.
  - PyCharm: tiene un linter excelente!
- Otros (los que están en rojo son los que tengo instalados en mi computador):
  - Flake8 (combina PyFlakes, pycodestyle, Mccabe)
  - Pylama (combina pycodestyle, pep8 y pydocstyle, pep257, PyFlakes,Mccabe, Pylint, Radon, gjslint)

El Zen de Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

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.
- Aunque lo práctico gana a la pureza.
- 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 sólo una- manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.[15]
- 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*

Lecturas obligatorias

- https://www.genbetadev.com/trabajar-como-desarrollador/diez-consejos-para-mejorar-tus-comentarios-de-codigo-fuente
- http://www.variablenotfound.com/2007/12/13-consejos-para-comentar-tu-cdigo.html
- https://www.smashingmagazine.com/2012/10/why-coding-style-matters/
- https://www.python.org/dev/peps/pep-0008/

Referencias

- Wikipedia
- http://www.inventwithpython.com/
- http://www.diveintopython3.net/
- https://realpython.com/python-pep8/
- Documentación de Python:
  - https://docs.python.org/3/tutorial/index.html
  - https://docs.python.org/3/
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: http://dx.doi.org/10.6035/Sapientia93