Introducción a Python y a sus módulos fundamentales

Autor: Pedro González Rodelas

Fecha de la primera versión: 06/05/2018

Sucesivas revisiones: 06-07/05/2018

Fecha de la última revisión: 16/05/2019

Ficheros de programa en Python

- Usualmente el código Python es almacenado en ficheros de texto plano con la extensión "`.py`":

  `miprograma.py`

- Cada línea de un programa en Python se asume que es una (o parte de una) sentencia del lengaje Python.
- No obstante, también existen los denominados notebooks de IPython, con extensión "`.ipynb`" en los que se puede incluir y combinar código Python en ciertas celdillas interactivas de entrada (numeradas con In[]), junto con sus correspondientes celdillas de salida (indicadas con Out[]), aparte de celdillas de texto explicativo y de estructuración en secciones, subsecciones, etc (usando código Markdown), salidas de gráficos, etc.
- Un ejemplo de este tipo de ficheros (en formato JSON), es este propio notebook de IPython, que se puede ejecutar gracias al servidor de Jupyter (consultar la página web oficial de Jupyter (http://www.jupyter.org)) que permite también ejecutar notebooks con otros muchos lenguajes, como 'R', 'Julia', etc.
- Aparte de todo esto, estos notebooks también permiten el uso de órdenes propias del sistema operativo, así como otras pseudo-órdenes genéricas (o comandos 'mágicos' que a veces suelen ir precedidos de '%' si son independientes del SO o de '!' si pertenecen al SO usado), que se traducirán al SO sobre el que se esté ejecutando el notebook, para poder realizar acciones propias de dicho sistema e interactuar con ficheros y directorios sin necesidad de salir del propio notebook.
- Para consultar una descripción detallada sobre la instalación de este software y sus múltiples posibilidades, consultar el contenido del siguiente https://www.ugr.es/~prodelas/ftp/TallerPython.html (https://www.ugr.es/~prodelas/ftp/TallerPython.html).

Ejemplos:

In [1]:
```
a = 2; b = 3;  # así es como se suele dar valores a las variables
```

In [2]:
```
a+b, a*b       # aquí estamos realizando operaciones de suma y produ
cto
```

Out[2]:
```
(5, 6)
```

In [3]:
```
# varios valores separados por comas conforman lo que se denomina un
a tupla
a,b   # en este caso se trata de una simple pareja de valores
```

Out[3]:
```
(2, 3)
```

In [4]:
```
%pwd   # para mostrar el directoriio actual de trabajo
```

Out[4]:
```
'C:\\Users\\Pedro\\Downloads'
```

In [5]:
```
%%file  hola-en-castellano.py

print("Hola")
```

```
Writing hola-en-castellano.py
```

In [6]:
```
%ls *.py   # para listar los ficheros en el directorio
```

```
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 84A6-5A60

 Directorio de C:\Users\Pedro\Downloads


 Directorio de C:\Users\Pedro\Downloads


 Directorio de C:\Users\Pedro\Downloads


 Directorio de C:\Users\Pedro\Downloads


 Directorio de C:\Users\Pedro\Downloads


 Directorio de C:\Users\Pedro\Downloads


 Directorio de C:\Users\Pedro\Downloads


 Directorio de C:\Users\Pedro\Downloads


 Directorio de C:\Users\Pedro\Downloads


16/05/2019  11:42                15 hola-en-castellano.py
16/05/2019  11:36             5.765 MEFLagrange1D_Dirichlet.py
16/05/2019  11:36             3.757 MEFLagrange1D_Interpol.py
16/05/2019  11:36             5.809 MEFLagrange1D_Neumann.py
16/05/2019  11:41             5.898 version_information.py
               5 archivos         21.244 bytes
               0 dirs     132.448.256 bytes libres
```

In [7]:
```
# !more hola-en-castellano.py
# usar %type en SOs tipo Windows
```

In [8]:
```
run hola-en-castellano.py
```

```
Hola
```

In [9]:
```
# cd ..  para cambiar al directorio padre
```

Codificación de caracteres

La codificación de caracteres estándar es la ASCII (acrónimo inglés de American Standard Code for Information Interchange — Código Estándar Estadounidense para el Intercambio de Información), pero nosotros podríamos usar cualqier otro sistema de codificación, por ejemplo UTF-8, que también nos permitirá incluir caracteres acentuados y la letra 'ñ', tan habituales en el idioma español.

Para especificar que UTF-8 será usado, debemos incluir la siguiente línea especial

```
# -*- coding: UTF-8 -*-
```

al principio del fichero. De esta manera podremos usar también caracteres acentuados o internacionales en nuestro archivo. Aunque esta opción ya se está convirtiendo en la opción por defecto en las nuevas versiones de Python e IPython.

Aparte de estas dos líneas *opcionales* al comienzo de cualquier fichero de código Python, no se requiere código adicional alguno para inicializar un programa Python, salvo el código del programa propiamente dicho.

Notebooks de IPython

Este fichero - que se trata de un notebook de IPython - no sigue este patrón estándar de simple código Python en un fichero de texto. En vez de esto, un notebook de IPython es almacenado como un fichero con formato JSON (http://en.wikipedia.org/wiki/JSON). La principal ventaja es que podremos mezclar texto formateado, código Python, junto con las correspondientes salidas y resultados. Esto requiere que al mismo tiempo se esté ejecutando el correspondiente servidor de notebooks de IPython, y por lo tanto no se trataría ya de un simple programa de Python ejecutándose de manera independiente, como en los casos explicados anteriormente. Aparte de este detalle, no hay diferencias entre el código Python que habría que introducir en un fichero de código Python y el que escribiremos en un notebook IPython como este.

Módulos

Por otra parte, la mayor functionalidad de Python se la proporcionarán los correspondientes *módulos* que cargemos. La biblioteca Estándar de Python es una enorme colección de módulos que proporcionan una implementación independiente de la plataforma o sistema operativo de todos las acciones más comunes, tales como el propio aceso el sistema operativo, operaciones de entrada y salida, trabajo con cadenas de caracteres ("strings" en inglés), comunicaciones, y mucho más.

Referencias

- The Python Language Reference: https://docs.python.org/3.5/library/ (https://docs.python.org/3.5/library/)
- The Python Standard Library: http://docs.python.org/3/library/ (http://docs.python.org/3/library/)
- Página web del proyecto SymPy: http://sympy.org/en/index.html (http://sympy.org/en/index.html)
- Version Online de SymPy para tests y demostraciones: http://live.sympy.org (http://live.sympy.org)
- Tutorial de NumPy: https://scipy.github.io/old-wiki/pages/Tentative_NumPy_Tutorial (https://scipy.github.io/old-wiki/pages/Tentative_NumPy_Tutorial)
- Una guía de Numpy para usuarios de MATLAB: https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html (https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html)
- Documentación oficial del proyecto Scipy: https://scipy.org (https://scipy.org)
- Un tutorial sobre cómo empezar a usar SciPy: http://docs.scipy.org/doc/scipy/reference/tutorial/index.html (http://docs.scipy.org/doc/scipy/reference/tutorial/index.html)
- La página web del proyecto matplotlib: http://www.matplotlib.org (http://www.matplotlib.org)
- Una extensa galeria que muestra varios tipos de gráficos creados con matplotlib: http://matplotlib.org/gallery.html (http://matplotlib.org/gallery.html)
- Un buen turorial de matplotlib: http://www.loria.fr/~rougier/teaching/matplotlib (http://www.loria.fr/~rougier/teaching/matplotlib)
- Otra buena referencia para matplotlib; http://scipy-lectures.github.io/matplotlib/matplotlib.html (http://scipy-lectures.github.io/matplotlib/matplotlib.html)

Para usar alguna función o procedimiento incluido en un módulo en un programa Python éste tiene que ser cargado primero. Estos deberán ser importados usando la orden `import`.

Por ejemplo, para importar por completo alguno de los módulos `NumPy`, `SciPy`, `SymPy`, `MatPlotlib` que contienen nuevas clases de objetos (como los 'arrays'), numerosas funciones matemáticas y numéricas, aparte de poder realizar cálculos simbólicos y obtener representaciones gráficas muy variadas, escribiríamos por ejemplo:

In [10]:
```
from numpy import *
from sympy import *
from scipy import *
from matplotlib import *
```

Estas sentencias cargarían los módulos completos, haciéndo disponible su uso posterior en el programa o notebook.

Este patrón puede resultar conveniente, pero en programas de cierta envergadura, que incluyan numerosos módulos, puede resultar a menudo una buena idea mantener los símbolos y funciones de cada módulo en su propio entorno de nombres ("namespace" en inglés), usando simplemente el patrón `import numpy`. Esto elminaría problemas de confusión, con potenciales y posibles colisiones entre los nombres de funciones entre diferentes módulos. No obstante en este último caso tendremos que anteponer el nombre del módulo antes de cada una de las funciones del módulo a usar:

In [11]:
```
import numpy
```

Consultando el contenido de un módulo, así como su documentación

Una vez importado un módulo, podremos listar los símbolos y funciones que proporciona usando la función o comando `dir`:

In [12]:
```
print(dir(numpy))
```

```
['ALLOW_THREADS', 'BUFSIZE', 'CLIP', 'ComplexWarning', 'DataSource
', 'ERR_CALL', 'ERR_DEFAULT', 'ERR_IGNORE', 'ERR_LOG', 'ERR_PRINT'
, 'ERR_RAISE', 'ERR_WARN', 'FLOATING_POINT_SUPPORT', 'FPE_DIVIDEBY
ZERO', 'FPE_INVALID', 'FPE_OVERFLOW', 'FPE_UNDERFLOW', 'False_', '
Inf', 'Infinity', 'MAXDIMS', 'MAY_SHARE_BOUNDS', 'MAY_SHARE_EXACT'
, 'MachAr', 'ModuleDeprecationWarning', 'NAN', 'NINF', 'NZERO', 'N
aN', 'PINF', 'PZERO', 'PackageLoader', 'RAISE', 'RankWarning', 'SH
IFT_DIVIDEBYZERO', 'SHIFT_INVALID', 'SHIFT_OVERFLOW', 'SHIFT_UNDER
FLOW', 'ScalarType', 'Tester', 'TooHardError', 'True_', 'UFUNC_BUF
SIZE_DEFAULT', 'UFUNC_PYVALS_NAME', 'VisibleDeprecationWarning', '
WRAP', '_NoValue', '__NUMPY_SETUP__', '__all__', '__builtins__', '
__cached__', '__config__', '__doc__', '__file__', '__git_revision_
_', '__loader__', '__mkl_version__', '__name__', '__package__', '_
_path__', '__spec__', '__version__', '_import_tools', '_mat', 'abs
', 'absolute', 'absolute_import', 'add', 'add_docstring', 'add_new
doc', 'add_newdoc_ufunc', 'add_newdocs', 'alen', 'all', 'allclose'
, 'alltrue', 'alterdot', 'amax', 'amin', 'angle', 'any', 'append',
'apply_along_axis', 'apply_over_axes', 'arange', 'arccos', 'arccos
h', 'arcsin', 'arcsinh', 'arctan', 'arctan2', 'arctanh', 'argmax',
'argmin', 'argpartition', 'argsort', 'argwhere', 'around', 'array'
, 'array2string', 'array_equal', 'array_equiv', 'array_repr', 'arr
ay_split', 'array_str', 'asanyarray', 'asarray', 'asarray_chkfinit
e', 'ascontiguousarray', 'asfarray', 'asfortranarray', 'asmatrix',
'asscalar', 'atleast_1d', 'atleast_2d', 'atleast_3d', 'average', '
bartlett', 'base_repr', 'bench', 'binary_repr', 'bincount', 'bitwi
se_and', 'bitwise_not', 'bitwise_or', 'bitwise_xor', 'blackman', '
bmat', 'bool', 'bool8', 'bool_', 'broadcast', 'broadcast_arrays',
'broadcast_to', 'busday_count', 'busday_offset', 'busdaycalendar',
'byte', 'byte_bounds', 'bytes0', 'bytes_', 'c_', 'can_cast', 'cast
', 'cbrt', 'cdouble', 'ceil', 'cfloat', 'char', 'character', 'char
array', 'choose', 'clip', 'clongdouble', 'clongfloat', 'column_sta
ck', 'common_type', 'compare_chararrays', 'compat', 'complex', 'co
mplex128', 'complex64', 'complex_', 'complexfloating', 'compress',
'concatenate', 'conj', 'conjugate', 'convolve', 'copy', 'copysign'
, 'copyto', 'core', 'corrcoef', 'correlate', 'cos', 'cosh', 'count
_nonzero', 'cov', 'cross', 'csingle', 'ctypeslib', 'cumprod', 'cum
product', 'cumsum', 'datetime64', 'datetime_as_string', 'datetime_
data', 'deg2rad', 'degrees', 'delete', 'deprecate', 'deprecate_wit
h_doc', 'diag', 'diag_indices', 'diag_indices_from', 'diagflat', '
diagonal', 'diff', 'digitize', 'disp', 'divide', 'division', 'dot'
, 'double', 'dsplit', 'dstack', 'dtype', 'e', 'ediff1d', 'einsum',
'emath', 'empty', 'empty_like', 'equal', 'errstate', 'euler_gamma'
, 'exp', 'exp2', 'expand_dims', 'expm1', 'extract', 'eye', 'fabs',
'fastCopyAndTranspose', 'fft', 'fill_diagonal', 'find_common_type'
, 'finfo', 'fix', 'flatiter', 'flatnonzero', 'flexible', 'fliplr',
'flipud', 'float', 'float16', 'float32', 'float64', 'float_', 'flo
ating', 'floor', 'floor_divide', 'fmax', 'fmin', 'fmod', 'format_p
arser', 'frexp', 'frombuffer', 'fromfile', 'fromfunction', 'fromit
er', 'frompyfunc', 'fromregex', 'fromstring', 'full', 'full_like',
'fv', 'generic', 'genfromtxt', 'get_array_wrap', 'get_include', 'g
et_printoptions', 'getbufsize', 'geterr', 'geterrcall', 'geterrobj
', 'gradient', 'greater', 'greater_equal', 'half', 'hamming', 'han
ning', 'histogram', 'histogram2d', 'histogramdd', 'hsplit', 'hstac
k', 'hypot', 'i0', 'identity', 'iinfo', 'imag', 'in1d', 'index_exp
', 'indices', 'inexact', 'inf', 'info', 'infty', 'inner', 'insert'
, 'int', 'int0', 'int16', 'int32', 'int64', 'int8', 'int_', 'int_a
sbuffer', 'intc', 'integer', 'interp', 'intersect1d', 'intp', 'inv
ert', 'ipmt', 'irr', 'is_busday', 'isclose', 'iscomplex', 'iscompl
exobj', 'isfinite', 'isfortran', 'isinf', 'isnan', 'isneginf', 'is
posinf', 'isreal', 'isrealobj', 'isscalar', 'issctype', 'issubclas
```

In [13]:
```
numpy.arange(0,10,1)  # fíjese que lo que estaríamos generando es un array que va de 0 a 9
```

Out[13]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Como una tercera alternativa, también podemos importar sólo una selección de símbolos de un determinado módulo, listando explícitamente sólo aquellos que queramos importar, en vez de usar el carácter comodín `*`:

In [14]:
```
# Por ejemplo para operar de manera simbólica con senos y cosenos importaríamos
# las siguientes funciones y órdenes incluidas dentro del módulo 'SymPy'
from sympy import Symbol, simplify, sin, cos, pi
```

In [15]:
```
x = Symbol('x')   # para usar la letra 'x' como variable simbólica
```

In [16]:
```
x?
```

In [17]:
```
simplify(sin(x)**2+cos(x)**2)  # así simplificaríamos la expresión trinonométrica
```

Out[17]:
```
1
```

In [18]:
```
x = cos(2 * pi)

print(x)
```

```
1
```

In [19]:
```
x?
```

In [20]:
```
# Es habitual cargar alguno de estos módulos con un pseudónimo determinado
import numpy as np               # aquí cargamos el módulo numpy con el pseudónimo np
import sympy as sp               # y el módulo sympy de cálculo simbólico como sp
import matplotlib.pyplot as plt # este módulo nos permitirá obtener gráficos
# También se podrían cargar estos módulos sin pseudónimos, pero en ese caso tendríamos
# que anteponer el nombre completo del módulo delante de cada una de las funciones y
# procedimientos incluidos en dicho submódulo. Por ejemplo:
#   numpy.sin(x) en vez de np.sin(x)  para usar y evaluar la función seno numéricamente
#   sympy.sin(x) en vez de sp.sin(x)  si queremos realizar más bien cálculos simbólicos
```

In [21]:
```
import sympy

print(dir(sympy))
```

```
['Abs', 'AccumBounds', 'Add', 'Adjoint', 'AlgebraicField', 'Algebr
aicNumber', 'And', 'AppliedPredicate', 'AssumptionsContext', 'Atom
', 'AtomicExpr', 'BasePolynomialError', 'Basic', 'BlockDiagMatrix'
, 'BlockMatrix', 'C', 'CC', 'CRootOf', 'Catalan', 'Chi', 'Ci', 'Ci
rcle', 'ClassRegistry', 'CoercionFailed', 'Complement', 'ComplexFi
eld', 'ComplexRegion', 'ComplexRootOf', 'ComputationFailed', 'Cond
itionSet', 'Contains', 'CosineTransform', 'Curve', 'DeferredVector
', 'Derivative', 'Determinant', 'DiagonalMatrix', 'DiagonalOf', 'D
ict', 'DiracDelta', 'Domain', 'DomainError', 'Dummy', 'E', 'E1', '
EPath', 'EX', 'Ei', 'Eijk', 'Ellipse', 'EmptySequence', 'EmptySet'
, 'Eq', 'Equality', 'Equivalent', 'EulerGamma', 'EvaluationFailed'
, 'ExactQuotientFailed', 'Expr', 'ExpressionDomain', 'ExtraneousFa
ctors', 'FF', 'FF_gmpy', 'FF_python', 'FU', 'FallingFactorial', 'F
initeField', 'FiniteSet', 'FlagError', 'Float', 'FourierTransform'
, 'FractionField', 'Function', 'FunctionClass', 'FunctionMatrix',
'GF', 'GMPYFiniteField', 'GMPYIntegerRing', 'GMPYRationalField', '
Ge', 'GeneratorsError', 'GeneratorsNeeded', 'GeometryError', 'Gold
enRatio', 'GramSchmidt', 'GreaterThan', 'GroebnerBasis', 'Gt', 'Ha
damardProduct', 'HankelTransform', 'Heaviside', 'HeuristicGCDFaile
d', 'HomomorphismFailed', 'I', 'ITE', 'Id', 'Identity', 'Idx', 'Im
ageSet', 'ImmutableDenseMatrix', 'ImmutableMatrix', 'ImmutableSpar
seMatrix', 'Implies', 'Indexed', 'IndexedBase', 'Integer', 'Intege
rRing', 'Integral', 'Intersection', 'Interval', 'Inverse', 'Invers
eCosineTransform', 'InverseFourierTransform', 'InverseHankelTransf
orm', 'InverseLaplaceTransform', 'InverseMellinTransform', 'Invers
eSineTransform', 'IsomorphismFailed', 'KroneckerDelta', 'LC', 'LM'
, 'LT', 'Lambda', 'LambertW', 'LaplaceTransform', 'Le', 'LessThan'
, 'LeviCivita', 'Li', 'Limit', 'Line', 'Line3D', 'Lt', 'MatAdd', '
MatMul', 'MatPow', 'Matrix', 'MatrixBase', 'MatrixExpr', 'MatrixSl
ice', 'MatrixSymbol', 'Max', 'MellinTransform', 'Min', 'Mod', 'Mon
omial', 'Mul', 'MultivariatePolynomialError', 'MutableDenseMatrix'
, 'MutableMatrix', 'MutableSparseMatrix', 'N', 'Nand', 'Ne', 'NonS
quareMatrixError', 'Nor', 'Not', 'NotAlgebraic', 'NotInvertible',
'NotReversible', 'Number', 'NumberSymbol', 'O', 'OperationNotSuppo
rted', 'OptionError', 'Options', 'Or', 'Order', 'POSform', 'Piecew
ise', 'Plane', 'Point', 'Point2D', 'Point3D', 'PoleError', 'Polifi
cationFailed', 'Poly', 'Polygon', 'PolynomialDivisionFailed', 'Pol
ynomialError', 'PolynomialRing', 'Pow', 'PrecisionExhausted', 'Pre
dicate', 'Product', 'ProductSet', 'PurePoly', 'PythonFiniteField',
'PythonIntegerRing', 'PythonRationalField', 'Q', 'QQ', 'QQ_gmpy',
'QQ_python', 'RR', 'Range', 'Rational', 'RationalField', 'Ray', 'R
ay3D', 'RealField', 'RealNumber', 'RefinementFailed', 'RegularPoly
gon', 'Rel', 'RisingFactorial', 'RootOf', 'RootSum', 'S', 'SOPform
', 'SYMPY_DEBUG', 'Segment', 'Segment3D', 'SeqAdd', 'SeqFormula',
'SeqMul', 'SeqPer', 'Set', 'ShapeError', 'Shi', 'Si', 'Sieve', 'Si
neTransform', 'SparseMatrix', 'StrictGreaterThan', 'StrictLessThan
', 'Subs', 'Sum', 'Symbol', 'SymmetricDifference', 'SympifyError',
'TableForm', 'Trace', 'Transpose', 'Triangle', 'Tuple', 'Unequalit
y', 'UnificationFailed', 'Union', 'UnivariatePolynomialError', 'Wi
ld', 'WildFunction', 'Xor', 'Ynm', 'Ynm_c', 'ZZ', 'ZZ_gmpy', 'ZZ_p
ython', 'ZeroMatrix', 'Znm', '__builtins__', '__cached__', '__doc_
_', '__file__', '__loader__', '__name__', '__package__', '__path__
', '__spec__', '__sympy_debug', '__version__', 'acos', 'acosh', 'a
cot', 'acoth', 'acsc', 'add', 'adjoint', 'airyai', 'airyaiprime',
'airybi', 'airybiprime', 'apart', 'apart_list', 'apply_finite_diff
', 'are_similar', 'arg', 'as_finite_diff', 'asec', 'asech', 'asin'
, 'asinh', 'ask', 'ask_generated', 'assemble_partfrac_list', 'asso
c_laguerre', 'assoc_legendre', 'assume', 'assuming', 'assumptions'
, 'atan', 'atan2', 'atanh', 'basic', 'bell', 'bernoulli', 'besseli
', 'besselj', 'besselk', 'besselsimp', 'bessely', 'beta', 'binomia
```

A su vez, usando la función de ayuda (`help`) también podremos obtener una descripción de cada una de las funciones (ya que casi todas, pero no todas las funciones poseen una adecuada documentación o "docstrings", pero la gran mayoría de ellas sí que estarán documentadas de esta manera).

In [22]:
```
help(math.log)
```

```
Help on built-in function log in module math:

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base
e) of x.
```

In [23]:
```
from math import log
log(10.)
```

Out[23]:
```
2.302585092994046
```

In [24]:
```
log(10, 2)
```

Out[24]:
```
3.3219280948873626
```

Podemos también usar la función `help` directamente para módulos: Prueba por ejemplo

```
help(math)
```

Algunos módulos muy útiles conforman las librerías estándar de Python `os`, `sys`, `math`, `shutil`, `re`, `subprocess`, `multiprocessing`, `threading`.

Una lista completa de los módulos estándar para Python 2 y Python 3 se tienen en http://docs.python.org/2/library/ (http://docs.python.org/2/library/) y http://docs.python.org/3/library/ (http://docs.python.org/3/library/), respectivamente.

Variables y tipos de objetos

Nombres de símbolos

Los nombres de las variable en Python pueden contener caracteres alfanuméricos `a-z`, `A-Z`, `0-9` y alguno de los caracteres especiales como _, pero obligatoriamente deben de comenzar con una letra.

Por convención, los nombres de variable empezarán con una letra minúscula, mientras que los nombres de "Clases" comenzarán con una letra mayúscula.

Además, existe cierto número de palabras clave en Python que no pueden ser usadas como nombres de variables. Estas palabras clave son:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Nota: Tenga cuidado con la palabra clave `lambda`, que podría habitualmente ser un nombre de variable en un programa científico. Pero al tratarse también de una palabra clave, no estaría permitido su uso como variable en este caso.

Sentencias de asignación

El operador de asignación en Python es `=`. Python tiene un tipado de variables dinámico, lo que hace que no tengamos que especificar el tipo de una variable cuando la creamos.

Asignando cierto valor a una nueva variable, estaríamos creando la variable:

In [25]:
```
# asignación de variable
x = 1.0
mi_variable = 12.2
```

Así pues, aunque no haya sido explicitamente especificado, una variable sí que tendría un tipo asociado, derivado por cierto del valor que tenga asignado en ese momento (de ahí el término de tipaje dinámico).

In [26]:
```
type(x)
```

Out[26]:
```
float
```

Así pues, si le asignamos un nuevo valor a la variable, este tipo puede cambiar.

In [27]:
```
x = 1
```

In [28]:
```
type(x)
```

Out[28]:
```
int
```

Si tratamos de usar una variable que aún no ha sido definida obtendríamos un `NameError`:

In [29]:
```
# Habría que descomentar la línea inferior antes de ejecutarla
# print(y)
```

Tipos Fundamentales

In [30]:
```
# enteros
x = 1
type(x)
```

Out[30]:
```
int
```

In [31]:
```
# números en coma flotante
x = 1.0
type(x)
```

Out[31]:
```
float
```

In [32]:
```
# booleanos
b1 = True
b2 = False

type(b1)
```

Out[32]:
```
bool
```

In [33]:
```
# números complejos: nótese el uso de `j` para especificar la unidad imaginaria
x = 1.0 - 1.0j
type(x)
```

Out[33]:
```
complex
```

In [34]:
```
print(x)
```

```
(1-1j)
```

In [35]:
```
print(x.real, x.imag)
```

```
1.0 -1.0
```

Utilidades de tipado ("Type utilities")

El módulo `types` contiene un cierto número de definiciones y funciones relacionadas con tipos de nombres que pueden ser usados para probar si las variables son de un determinado tipo:

In [36]:
```
import types

# imprime todos los tipos definidos en el módulo `types`
print(dir(types))
```

```
['BuiltinFunctionType', 'BuiltinMethodType', 'CodeType', 'Coroutin
eType', 'DynamicClassAttribute', 'FrameType', 'FunctionType', 'Gen
eratorType', 'GetSetDescriptorType', 'LambdaType', 'MappingProxyTy
pe', 'MemberDescriptorType', 'MethodType', 'ModuleType', 'SimpleNa
mespace', 'TracebackType', '_GeneratorWrapper', '__all__', '__buil
tins__', '__cached__', '__doc__', '__file__', '__loader__', '__nam
e__', '__package__', '__spec__', '_calculate_meta', '_collections_
abc', '_functools', 'coroutine', 'new_class', 'prepare_class']
```

In [37]:
```
x = 1.0

# esta sentencia chequea si la variable x es de tipo coma flotante (
'float')
type(x) is float
```

Out[37]: True

In [38]:
```
# y esta si la variable x es de tipo entero ('int')
type(x) is int
```

Out[38]: False

También podemos usar el método `isinstance` para testear los tipos de las variables:

In [39]:
```
isinstance(x, float)
```

Out[39]: True

Repertorio de Tipos ('type' en inglés)

In [40]:
```
x = 1.5

print(x, type(x))
```

```
1.5 <class 'float'>
```

In [41]:
```
x = int(x)

print(x, type(x))
```

```
1 <class 'int'>
```

In [42]:
```
z = complex(x)

print(z, type(z))
```

```
(1+0j) <class 'complex'>
```

In [43]:
```
# x = float(z)
```

Comprobamos que las variables complejas no pueden ser convertidas a tipo real o entero. Necesitaríamos usar más bien `z.real` o `z.imag` para extraer la parte real o imaginaria del número complejo que queramos:

In [44]:
```
y = bool(z.real)

print(z.real, " -> ", y, type(y))

y = bool(z.imag)

print(z.imag, " -> ", y, type(y))
```

```
1.0  ->  True <class 'bool'>
0.0  ->  False <class 'bool'>
```

Operadores y comparaciones

La mayor parte de los operadores y comparaciones que efectuemos en Python funcionarán como se espera:

- Operadores Aritméticos `+`, `-`, `*`, `/`, `//` (división entera), '**' exponenciación, etc.

In [45]:
```
1 + 2, 1 - 2, 1 * 2, 1 / 2    #  ¡atención con la división entera!
```

Out[45]: (3, -1, 2, 0.5)

In [46]:
```
1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0
```

Out[46]: (3.0, -1.0, 2.0, 0.5)

In [47]:
```
# También podemos forzar una división entera de números reales
3.0 // 2.0
```

Out[47]: 1.0

In [48]:
```
# Nótese que el operador exponenciación en python no es ^, sino **
2 ** 2
```

Out[48]: 4

In [49]:
```
# ^ más bien realiza una comparación
# de la representación binaria de ambos operandos
# y el resultado se vuelve a pasar a base 10
1^2, 2^2, 2^3, 1^4
```

Out[49]: (3, 0, 1, 5)

Por otro lado, el operador / siempre realizará una división en coma flotante, siempre que estemos usando una versión de Python 3.x.

Sin embargo este no es el caso en Python 2.x, donde el resultado de `/` es siempre un entero si los operadores son enteros. para ser más específicos, `1/2 = 0.5` (`float`) en Python 3.x, y `1/2 = 0` (`int`) en Python 2.x (pero `1.0/2 = 0.5` en Python 2.x).

- Los operadores booleanos serán `and`, `not`, `or`.

In [50]:
```
True and False
```

Out[50]: False

In [51]:
```
not False
```

Out[51]: True

In [52]:
```
True or False
```

Out[52]: True

- Los operadores de comparación >, <, >= (mayor, menor, mayor o igual), <= (menor o igual), == igualdad, `is` identidad.

In [53]:
```
2 > 1, 2 < 1
```

Out[53]: (True, False)

In [54]:
```
2 > 2, 2 < 2
```

Out[54]: (False, False)

In [55]:
```
2 >= 2, 2 <= 2
```

Out[55]: (True, True)

In [56]:
```
# igualdad
[1,2] == [1,2]
```

Out[56]: True

In [57]:
```
# ¿objetos idénticos?
l1 = l2 = [1,2]

l1 is l2
```

Out[57]: True

Tipos compuestos: Strings, Listas y diccionarios

Strings (o cadenas de carácteres)

Los strings serán el tipo de variable usado para almacenar mensajes de texto.

In [58]:
```
s = "Hola mundo"
type(s)
```

Out[58]:
```
str
```

In [59]:
```
# para obtener la longitud del string: el número de caracteres
len(s)
```

Out[59]:
```
10
```

In [60]:
```
# para reemplazar un substring (subconjunto de caracteres) en un
# string con otra expresión, usaremos la función 'replace'
s2 = s.replace("mundo", "amigo")
print(s2)
```

```
Hola amigo
```

Se puede indexar cada uno de los caracteres en un string usando `[]`, teniendo en cuenta eso sí, que el primer carácter empezará con el índice 0 y no el 1.

In [61]:
```
s[0]
```

Out[61]:
```
'H'
```

Aviso importante pues para usuarios habituales de MATLAB o Mathematica:

¡El indexado siempre empieza por 0!

También podemos extraer una parte concreta del string usando la sintaxis `[inicio:final]`, que extraerá aquellos caracteres de la cadena comprendidos entre el índice `inicio` y `final` -1 (teniendo en cuenta lo ya resaltado anteriormente acerca de que el primer índice siempre será por defecto el 0, y que el carácter con índice `final` no será pues incluido:

In [62]:
```
str1=s[0:4]
print(str1)
```

```
Hola
```

In [63]:
```
str2=s[4:5]
print(str2)
```

In [64]:
```
str3=s[5:10]
print(str3)
```

```
mundo
```

Por otro lado, si omitimos alguno (o ambos) de los límites `inicio` o `final` en `[inicio:final]`, entonces por defecto se tomará el comienzo y/o el final de la cadena, respectivamente:

In [65]:
```
s[:4]
```

Out[65]: `'Hola'`

In [66]:
```
s[5:]
```

Out[66]: `'mundo'`

In [67]:
```
s[:]
```

Out[67]: `'Hola mundo'`

También podemos definir un paso de salto usando la sintaxis `[inicio:final:paso]` (el valor por defecto para `paso` es 1, como ya se ha visto más arriba):

In [68]:
```
s[::1]
```

Out[68]: `'Hola mundo'`

In [69]:
```
s[::2], s[1::2]
```

Out[69]: `('Hl ud', 'oamno')`

Esta técnica se denomina rebanadora (*slicing* en inglés). Puede consultar más acerca de la sintaxis correspondiente en: http://docs.python.org/release/2.7.3/library/functions.html?highlight=slice#slice (http://docs.python.org/release/2.7.3/library/functions.html?highlight=slice#slice)

De hecho Python tiene un conjunto muy rico de funciones para el procesamiento de texto. Consultar por ejemplo http://docs.python.org/2/library/string.html (http://docs.python.org/2/library/string.html) para más información.

Ejemplos de formateo de texto ('strings')

In [70]:
```
print(str1, 1.0, False, -1j)
# De hecho, la sentencia 'print' convierte todos
# los argumentos a strings
```

```
Hola 1.0 False (-0-1j)
```

In [71]:
```
print(str1 + str2 + str3)
# los strings sumados con '+' simplemente son concatenados
```

```
Hola mundo
```

In [72]:
```
# También podríamos simplemente haber escrito
str1+str2+str3
```

Out[72]:
```
'Hola mundo'
```

In [73]:
```
print("value = %f" % 1.0)
# podemos también usar un estilo de formateo tipo lenguaje C
```

```
value = 1.000000
```

In [74]:
```
# así podemos crear un string formateada como queramos
s2 = "valor1 = %.2f. valor2 = %d" % (3.1415, 1.5)

print(s2)
```

```
valor1 = 3.14. valor2 = 1
```

In [75]:
```
# otra alternativa más intuitiva sería
s3 = 'valor1 = {0}, valor2 = {1}'.format(3.1415, 1.5)

print(s3)
```

```
valor1 = 3.1415, valor2 = 1.5
```

Listas

Las listas son muy similares a las cadenas de caracteres, excepto que cada elemento puede ser de cualquier tipo.

La sintaxis para crear listas en Python es usando corchetes `[...]`:

In [76]:
```
l = [1,2,3,4]

print(type(l))
print(l)
```

```
<class 'list'>
[1, 2, 3, 4]
```

También podemos usar la misma técnica de rebanado o "slicing" que usamos con los strings para manipular listas:

In [77]:
```
print(l)

print(l[1:3])

print(l[::2])
```

```
[1, 2, 3, 4]
[2, 3]
[1, 3]
```

Volvemos a recordar a los usuarios habituales de MATLAB o Mathematica ¡que el indexado en Python comienza en 0!

In [78]:
```
l[0]
```

Out[78]: 1

Los elementos en una lista no tienen que ser del mismo tipo:

In [79]:
```
l = [1, 'a', 1.0, 1-1j]

print(l)
```

```
[1, 'a', 1.0, (1-1j)]
```

A su vez, las listas en Python aparte de ser no homogéneas, también pueden anidarse arbitrariamente:

In [80]:
```
lista_anidada = [1, [2, [3, [4, [5]]]]]

lista_anidada
```

Out[80]: [1, [2, [3, [4, [5]]]]]

Las listas a su vez juegan un papel muy importante en Python. Por ejemplo serán usadas en bucles y otras estructuras de control de flujo (discutidas más abajo). También se dispone de un cierto número de convenientes funciones para generar listas de varios tipos, por ejemplo la función `range` :

In [81]:
```
inicio = 10
final  = 30
paso   = 2

range(inicio, final, paso)
```

Out[81]: range(10, 30, 2)

In [82]:
```
# En Python 3 range genera más bien un iterador
# que puede convertirse en una lista usando 'list(...)'.
# Pero esto no tiene ningún efecto en Python 2
list(range(inicio, final, paso))
```

Out[82]: [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

In [83]:
```
list(range(-10, 10))
```

Out[83]:
```
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7,
8, 9]
```

In [84]:
```
s
```

Out[84]:
```
'Hola mundo'
```

In [85]:
```
# Para convertir un string en una lista de caracteres podemos usar:
s2 = list(s)

s2
```

Out[85]:
```
['H', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o']
```

In [86]:
```
# Ordenando listas
s2.sort()

print(s2)
```

```
[' ', 'H', 'a', 'd', 'l', 'm', 'n', 'o', 'o', 'u']
```

Añadiendo, insertando, modificando, y eliminando elementos de listas

In [87]:
```
# Empezamos creando una nueva lista vacía
l = []

# y añadimos elementos usando `append`
l.append("A")
l.append("d")
l.append("d")

print(l)
```

```
['A', 'd', 'd']
```

Ahora podemos modificar listas asignandoles nuevos valores a ciertos elementos de la lista. Hablando en una jerga técnica, las listas son *mutables*.

In [88]:
```
l[1] = "p"
l[2] = "p"

print(l)
```

```
['A', 'p', 'p']
```

In [89]:
```
l[1:3] = ["d", "d"]

print(l)
```

```
['A', 'd', 'd']
```

Podemos insertar un elemento en una posición specifica usando `insert`

In [90]:
```
l.insert(0, "i")
l.insert(1, "n")
l.insert(2, "s")
l.insert(3, "e")
l.insert(4, "r")
l.insert(5, "t")

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'A', 'd', 'd']
```

Para eliminar el primer elemento con un valor específico usaremos 'remove'

In [91]:
```
l.remove("A")

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'd', 'd']
```

mientras que para eliminar el elemento en una posición específica usaremos más bien `del`:

In [92]:
```
del l[7]
del l[6]

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't']
```

Consultar `help(list)` para ver más detalles, o lea la documentación online al respecto

Tuplas ("Tuples" en inglés)

Las tuplas son parecidas a las listas, excepto en el hecho de que no pueden ser modificadas después de ser creadas; esto es son *inmutables*.

En Python, las tuplas se crean usando la sintaxis `(..., ..., ...)`, o incluso sin paréntesis `..., ...`:

In [93]:
```
punto = (10, 20)

print(punto, type(punto))
```

```
(10, 20) <class 'tuple'>
```

In [94]:
```
punto = 10, 20

print(punto, type(punto))
```

```
(10, 20) <class 'tuple'>
```

Podemos desempaquetar una tupla asignándola a una lista de variables separadas por comas:

In [95]:
```
x, y = punto

print("x =", x)
print("y =", y)
```

```
x = 10
y = 20
```

Pero si intentamos asignar un nuevo valor a un cierto elemento de una tupla el intérprete de Python nos devolverá un error o excepción:

In [96]:
```
# point[0] = 20
```

Diccionarios

Los diccionarios también son como las listas, excepto que cada elemento es un par de tipo llave-valor ("key-value" en inglés). Así pues, la sintaxis para los diccionarios es `{llave1 : valor1, ...}`:

In [97]:
```
parametros = {"parametro1" : 1.0,
              "parametro2" : 2.0,
              "parametro3" : 3.0,}

print(type(parametros))
print(parametros)
```

```
<class 'dict'>
{'parametro1': 1.0, 'parametro3': 3.0, 'parametro2': 2.0}
```

In [98]:
```
print("parametro1 = " + str(parametros["parametro1"]))
print("parametro2 = " + str(parametros["parametro2"]))
print("parametro3 = " + str(parametros["parametro3"]))
```

```
parametro1 = 1.0
parametro2 = 2.0
parametro3 = 3.0
```

In [99]:
```
parametros["parametro1"] = "A"
parametros["parametro2"] = "B"

# añadiendo una nueva entrada
parametros["parametro4"] = "D"

print("parametro1 = " + str(parametros["parametro1"]))
print("parametro2 = " + str(parametros["parametro2"]))
print("parametro3 = " + str(parametros["parametro3"]))
print("parametro4 = " + str(parametros["parametro4"]))
```

```
parametro1 = A
parametro2 = B
parametro3 = 3.0
parametro4 = D
```

Control de Flujo

Sentencias condicionales: if, elif, else

La sintaxis de Python para la ejecución de órdenes condicionales de código usa la palabras clave `if`, `elif` (else if), `else`:

In [100]:
```
sentencia1 = False
sentencia2 = False

if sentencia1:
    print("sentencia1 es Verdad")

elif sentencia2:
    print("sentencia2 es Verdad")

else:
    print("sentencia1 y sentencia2 son Falsas")
```

```
sentencia1 y sentencia2 son Falsas
```

Aquí encontramos por primera vez una característica peculiar y algo inusual del lenguaje de programación Python: los bloques de programa vienen definidos por su nivel de indentación.

Compare con el código equivalente en lenguaje C:

```
if (statement1)
{
    printf("statement1 is True\n");
}
else if (statement2)
{
    printf("statement2 is True\n");
}
else
{
    printf("statement1 and statement2 are False\n");
}
```

Vemos que en lenguaje C los bloques de programa vienen delimitados por llaves `{` y `}`. Por otra parte, el nivel de indentación (espacios en blanco antes de las sentencias del código) realmente no importan (siendo completamente opcionales).

Sin embargo, en Python, la extensión y alcance de un bloque de código viene definido por el nivel de indentación (usualmente un tabulado o bien unos cuatro espacios en blanco). Esto significa que debemos ser cuidadosos para indentar correctamente nuestro código, si no queremos tener errores de sintaxis.

Ejemplos:

In [101]:
```
sentencia1 = sentencia2 = True

if sentencia1:
    if sentencia2:
        print("ambas sentencia1 y setencia2 son verdad (True)")
```

```
ambas sentencia1 y setencia2 son verdad (True)
```

In [102]:
```
# ¡Mala indentación!
if sentencia1:
    if sentencia2:
    print("ambas sentencia1 y setencia2 son verdad (True)")
    # esta línea está mal indentada
```

```
  File "<ipython-input-102-f2c23337a17a>", line 4
    print("ambas sentencia1 y setencia2 son verdad (True)")
        ^
IndentationError: expected an indented block
```

In [103]:
```
sentencia1 = False

if sentencia1:
    print("imprime si la sentencia1 es verdad (True)")

    print("todavía dentro del bloque if")
```

In [104]:
```
if sentencia1:
    print("imprime si la sentencia1 es verdad (True)")

print("ya estamos fuera del bloque if")
```

```
ya estamos fuera del bloque if
```

Bucles

En Python, los bucles pueden programarse de diferentes formas, pero la más común es mediante la sentencia `for`, que se usa junto con objetos iterables, como las listas. Su sintaxis básica es:

bucles `for` :

In [105]:
```
for x in [1,2,3]:
    print(x)
```

```
1
2
3
```

El bucle `for` itera sobre los elementos de la lista suministrada, y ejecuta el bloque de código contenido dentro del bucle, una vez para cada elemento. Cualquier tipo de lista puede ser usada en un bucle `for`. Por ejemplo:

In [106]:
```
for x in range(4): # por defecto range empieza en 0
    print(x)
```

```
0
1
2
3
```

Nota: ¡`range(4)` no incluye el 4 !

In [107]:
```
for x in range(-3,3):
    print(x)
```

```
-3
-2
-1
0
1
2
```

In [108]:
```
for palabra in ["cálculo", "científico", "con", "Python"]:
    print(palabra)
```

```
cálculo
científico
con
Python
```

Para iterar sobre pares llave-valor (key-value) de un diccionario:

In [109]:
```
for llave, valor in parametros.items():
    print(llave + " = " + str(valor))
```

```
parametro1 = A
parametro3 = 3.0
parametro4 = D
parametro2 = B
```

A veces es útil tener acceso a los indices de los valores mientras iteramos sobre una lista. Podemos usar la función `enumerate` para esto:

In [110]:
```
for idx, x in enumerate(range(-3,3)):
    print(idx, x)
```

```
0 -3
1 -2
2 -1
3 0
4 1
5 2
```

Compresión de listas:

Una forma conveniente y compacta de inicializar listas es la siguiente:

In [111]:
```
l1 = [x**2 for x in range(0,5)]

print(l1)
```

```
[0, 1, 4, 9, 16]
```

Bucles `while`:

In [112]:
```
i = 1

while i <= 5:
    print(i)

    i = i + 1

print("hecho")
```

```
1
2
3
4
5
hecho
```

Nótese que la sentencia `print("hecho")` no forma parte del cuerpo de la sentencia `while` por la diferente indentación.

Funciones

Una función en Python se define usando la palabra clave `def`, seguida por un nombre de función, con sus correspondientes paréntesis `()`, y los dos puntos `:`. El código siguiente, con un nivel de indentación adicional, es el cuerpo de la función.

In [113]:
```
def func0():
    print("prueba")
```

In [114]:
```
func0()
```

```
prueba
```

Opcionalmente, aunque altamente recomendable, también podemos definir lo que se denomina "docstring", que es una pequeña documentación para describir el propósito y comportamiento de la función que vamos a definir. Esta escueta documentación debería ir directamente después del comando que define a la función, y justo antes del cuerpo que define el código de la misma.

In [115]:
```
def func1(s):
    """
    Imprime un string 's' y nos dice cuantos caracteres tiene
    """

    print(s + " tiene " + str(len(s)) + " caracteres")
```

In [116]:
```
help(func1)
```

```
Help on function func1 in module __main__:

func1(s)
    Imprime un string 's' y nos dice cuantos caracteres tiene
```

In [117]:
```
func1("prueba")
```

```
prueba tiene 6 caracteres
```

Vemos pues que funciones que simplemente imprimen un cierto mensaje o resultado pueden definirse con un `print`, sin embargo si pretendemos que devuelvan un cierto valor deberemos usar la palabra clave `return`:

In [118]:
```
def cuadrado(x):
    """
    Devuelve el cuadrado del número x.
    """
    return x ** 2
```

In [119]:
```
cuadrado(4)
```

Out[119]:
```
16
```

Podemos devolver múltiples valores de una función usando tuplas (ver más arriba):

In [120]:
```
def potencias(x):
    """
    Devuelve unas cuantas potencias de x.
    """
    return x ** 2, x ** 3, x ** 4
```

In [121]:
```
potencias(3)
```

Out[121]:
```
(9, 27, 81)
```

In [122]:
```
# La posibilidad de asignación múltiple
# es otra de las particularidades de Python

x2, x3, x4 = potencias(3)

print(x3)
```

```
27
```

Argumentos por defecto y palabras clave ("keywords") como argumentos

En la propia definición de una función, podemos dar valores por defecto a los argumentos que puede tomar dicha función:

In [123]:
```
def mifuncion(x, p=2, debug=False):
    if debug:
        print("evaluando mifuncion para x = " + str(x)
              + " usando un exponente p = " + str(p))
    return x**p
```

De esta manera, si no proporcionamos ningún valor para el argumento correspondiente a `debug` cuando llamamos a la función `mifuncion` su valor por defecto será el valor indicado en la definición de la función:

In [124]:
```
mifuncion(5)
```

Out[124]: 25

In [125]:
```
mifuncion(5, debug=True)
```

```
evaluando mifuncion para x = 5 usando un exponente p = 2
```

Out[125]: 25

Sin embargo, si proporcionamos de manera explícita el nombre de los argumentos en la correspondiente llamada a la función, entonces ni siquiera es necesario que éstos vengan en el mismo orden en el que se definió la función. Esta propiedad se denomina argumentos por palabras claves (*keyword*), y a menudo es bastante útil para funciones con muchos argumentos opcionales, cuyo orden no tenemos que memorizar en absoluto.

In [126]:
```
mifuncion(p=3, debug=True, x=7)
```

```
evaluando mifuncion para x = 7 usando un exponente p = 3
```

Out[126]: 343

Funciones sin nombre (anónimas, o funciones "lambda")

En Python podemos tamién crear funciones sin nombre, usando la palabra clave `lambda`:

In [127]:
```
f1 = lambda x: x**2

# es equivalente a

def f2(x):
    return x**2
```

In [128]:
```
f1(2), f2(2)
```

Out[128]: (4, 4)

Esta técnica es útil por ejemplo cuando queremos pasarle una simple función como argumento a otra función, como esta:

In [129]:
```
# map es a su vez una función incorporada en python
map(lambda x: x**2, range(-3,4));
```

In [130]:
```
# en Python 3 podemos usar `list(...)` para convertir el iterador
# en una lista explícita
list(map(lambda x: x**2, range(-3,4)))
```

Out[130]:
```
[9, 4, 1, 0, 1, 4, 9]
```

Clases de objetos y POO (Programación Orientada a Objetos)

Las clases son una característica clave en la programación orientada a objetos ("object-oriented programming" en inglés). Una clase es una estructura para representar a un objecto así como las distintas operaciones que pueden realizarse sobre dicho objeto.

En Python una clase puede contener *atributos* (variables) y *métodos* (funciones).

Por otro lado, una clase se define casi de la misma manera que una función, pero usando la palabra clave `class`; además la definición de clase contiene usualmente también un cierto número de definiciones de métodos de clase (que serán funciones en la clase).

- Cada método de la clase deberá tener un argumento `self` como primer argumento de la lista de argumentos. Este objeto es una auto-referencia ("self-reference" en inglés).
- Otros nombres de métodos de la clase también tendrán un significado especial, por ejemplo:
  - `__init__`: Es el nombre del método que es invocado cuando el objeto es creado.
  - `__str__` : Es un método que es invocado cuando sólo se necesita una simple reperesentación alfanumérica de la clase, como por ejemplo al imprimir.
  - Hay otros muchos nombres especiales asociados a las clases, consultar por ejemplo http://docs.python.org/3/reference/datamodel.html#special-method-names (http://docs.python.org/3/reference/datamodel.html#special-method-names)

In [131]:
```
class Point:
    """
    Clase para representar un punto en un sistema de coordenadas
    cartesianas.
    """

    def __init__(self, x, y):
        """
        Crea un nuevo Punto con coordenaddas  x, y.
        """
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """
        Traslada el punto mediante un desplazamiento dx y dy
        en cada dirección.
        """
        self.x += dx
        self.y += dy

    def __str__(self):
        return("Punto en [%f, %f]" % (self.x, self.y))
```

Para crear una nueva instancia de una clase:

In [132]:
```
p1 = Point(0, 0)

# esto invoca el método  __init__ en la clase Point

print(p1)          # esto invoca el método __str__
```

```
Punto en [0.000000, 0.000000]
```

Para invocar un método de la clase en la instancia `p` de dicha clase:

In [133]:
```
p2 = Point(1, 1)

p1.translate(0.25, 1.5)

print(p1)
print(p2)
```

```
Punto en [0.250000, 1.500000]
Punto en [1.000000, 1.000000]
```

Nótese que la llamada a un método de la clase puede modificar de hecho esa particular instancia del objeto en cuestión, pero no tiene efecto sobre cualquier otra instancia de la clase, ni tampoco sobre cualquier otra variable global que se estuviera usando en ese momento.

Esta es una de las convenientes propiedades que tiene el diseño orientado a objetos: programar funciones y variables relacionadas que son agrupadas en entidades separadas e independientes.

Módulos

Uno de los conceptos más importantes en este tipo de programación es la reutilización de código para evitar repeticiones.

La idea es escribir funciones y clases con un propósito y objetivo bien definido, y reutilizarlas siempre que un código similar se necesite en una parte diferente del programa (programación modular). El resultado en general suele ser una gran mejora en el buen mantenimiento e interpretación del código, que en la práctica supone que nuestro programa tendrá menos errores ("bugs" en inglés), y será más fácil de depurar y extender.

Python soporta este tipo de programación modular a diferentes niveles. Funciones y clases son ejemplos de herramientas para una programción modular de bajo nivel. Por otro lado, los módulos de Python son otra construcción de más alto nivel para este tipo de programación. En estos módulos recolectamos variables relacionadas, funciones y clases en un determinado módulo. Así pues un módulo de Python se define en un fichero de python (con la extensión `.py`), de manera que puede ser accesible a otros módulos y programas de Python usando la sentencia `import`.

Considere el ejemplo siguiente: el fichero `mimodulo.py` contiene un simple ejemplo de implementación de una variable, una función y una clase:

In [134]:
```
%%file mimodulo.py
"""
Ejemplo de modulo python. Contiene una variable llamada mi_variable
,
una funcion llamada mi_funcion, y una clase llamada MiClase.
"""

mi_variable = 0

def mi_funcion():
    """
    Ejemplo de funcion
    """
    return mi_variable

class MiClase:
    """
    Ejemplo de clase.
    """

    def __init__(self):
        self.variable = mi_variable

    def set_variable(self, nuevo_valor):
        u"""
        Le da a  self.variable un nuevo valor
        """
        self.variable = nuevo_valor

    def get_variable(self):
        return self.variable
```

```
Writing mimodulo.py
```

Ahora ya podemos importar el módulo `mimodulo` en cualquiera de nuestros programas Python usando `import`:

In [135]:
```
import mimodulo
```

Usamos `help(modulo)` para obtener un resumen de lo que nos proporciona dicho modulo:

In [136]:
```
help(mimodulo)
```

```
Help on module mimodulo:

NAME
    mimodulo

DESCRIPTION
    Ejemplo de modulo python. Contiene una variable llamada mi_var
iable,
    una funcion llamada mi_funcion, y una clase llamada MiClase.

CLASSES
    builtins.object
        MiClase

    class MiClase(builtins.object)
     |  Ejemplo de clase.
     |
     |  Methods defined here:
     |
     |  __init__(self)
     |      Initialize self.  See help(type(self)) for accurate si
gnature.
     |
     |  get_variable(self)
     |
     |  set_variable(self, nuevo_valor)
     |      Le da a  self.variable un nuevo valor
     |
     |  ----------------------------------------------------------
------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    mi_funcion()
        Ejemplo de funcion

DATA
    mi_variable = 0

FILE
    c:\users\pedro\downloads\mimodulo.py
```

In [137]:
```
mimodulo.mi_variable
```

Out[137]: 0

In [138]:
```
mimodulo.mi_funcion()
```

Out[138]: 0

In [139]:
```
mi_clase = mimodulo.MiClase()
mi_clase.set_variable(10)
mi_clase.get_variable()
```

Out[139]: 10

Si en cualquier momento efectuamos cambios en el código de `mimodulo.py`, necesitaremos volver a cargar dicho módulo usando la orden `reload`:

In [140]:
```
reload(mimodulo)  # funciona sólo con python 2
```

Out[140]:
```
<module 'mimodulo' from 'C:\\Users\\Pedro\\Downloads\\mimodulo.py'
>
```

Excepciones

En Python los errores pueden y deberíann ser manejados con un lenguaje especial construido expresamente para tal efecto y denominado "Exceptions". Así pues, cuando ocurre algún error se generan las denominadas excepciones, que pueden interrumpir el flujo normal del programa y llevarnos a otro punto concreto del código donde se han definido convenientemente otras sentencias para tratar estas excepciones.

Para generar una de estas excepciones podemos usar la sentencia `raise`, que toma un argumento que debe ser una instancia de la clase `BaseException` o cierta clase derivada de esta.

In [141]:
```
raise Exception("descripción del error")
```

```
---------------------------------------------------------------
---------
Exception                                 Traceback (most recent c
all last)
<ipython-input-141-ae92a99eabb1> in <module>()
----> 1 raise Exception("descripción del error")

Exception: descripción del error
```

Un uso típico de una de tales excepciones es abortar ciertas funciones cuando ocurre alguna condición de error, por ejemplo:

```
def mi_funcion(argumentos):

    if not verify(argumentos):
        raise Exception("Argumentos invalidos")

    # aquí iría el resto del código
```

Para captar errores generados por ciertas funciones y métodos de clases, o por el propio intérprete de Python, use las órdenes `try` y `except`:

```
try:
    # aquí iría el código normal
except:
    # aquí iría el código para tratar de solucionar el error
    # este código no se ejecutaría si no tiene lugar
    # el error generado más arrriba
```

Por ejemplo:

In [142]:
```
try:
    print("valor de la variable test")
    # genera un error: la variable test no está definida
    print(test)
except:
    print("Una excepción ha sido capturada:")
    print("parece ser que esta variable no está aún asignada")
```

```
valor de la variable test
<function test at 0x000000000833E950>
```

Para obtener información acerca del error, podemos acceder a la instancia e la clase `Exception` que describe la excepción usando por ejemplo:

```
except Exception as e:
```

In [143]:
```
try:
    print("test")
    # genera un error: la variable test no está definida
    print(test)
except Exception as e:
    print("Excepción capturada:  " + str(e))
```

```
test
<function test at 0x000000000833E950>
```

Lecturas adicionales

- http://www.python.org (http://www.python.org) - La página oficial del lenguaje de programación Python.
- http://www.python.org/dev/peps/pep-0008 (http://www.python.org/dev/peps/pep-0008) - Una guía de estilo para la programación con Python. Muy recomendada.
- http://www.greenteapress.com/thinkpython/ (http://www.greenteapress.com/thinkpython/) - Un libro gratis sobre programación con Python.
- Python Essential Reference (http://www.amazon.com/Python-Essential-Reference-4th-Edition/dp/0672329786) - Un buen libro de referencia sobre programación con Python.

Versiones

In [144]:
```
%load_ext version_information

%version_information
```

Out[144]:

Software	Version
Python	3.5.2 64bit [MSC v.1900 64 bit (AMD64)]
IPython	5.1.0
OS	Windows 7 6.1.7601 SP1
Thu May 16 11:43:16 2019 Hora de verano romance	