Introducción a la programación en Python

Pedro Corcuera

Dpto. Matemática Aplicada y Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

Objetivos

- Revisión de la programación en Python
- Funciones y Módulos

Índice

- Introducción
- Tipos de datos
- Condicionales y ciclos
- Arrays
- Entrada y Salida
- Funciones
- Módulos
- Packages

¿Qué es Python?

- Python es un lenguaje de programación *interpretado* de alto nivel y multiplataforma (Windows, MacOS, Linux). Creado por Guido van Rossum (1991).
- Es sencillo de aprender y de entender.
- Los archivos de python tienen la extensión **.py**
  - Archivos de texto que son interpretados por el compilador. Para ejecutar programas en Python es necesario el *intérprete de python*, y el código a ejecutar.
- Python dispone de un entorno interactivo y muchos módulos para todo tipo de aplicaciones.

Instalación de Python

- La última versión de Python es la 3.
- Sitio oficial de descargas.
  - Con ello se instala el intérprete Python, IDLE (**I**ntegrated **D**evelopment and **L**earning **E**nvironment)
  - Se recomienda incluir Python en la variable de entorno PATH
- Sitio oficial de documentación

Instalación de librerías científicas y gráficas en Python

- Los módulos se instalan con el comando pip.
  - En una ventana de comando (cmd. símbolo del sistema):

```
> python -m pip install --user numpy scipy matplotlib ipython jupyter sympy
```

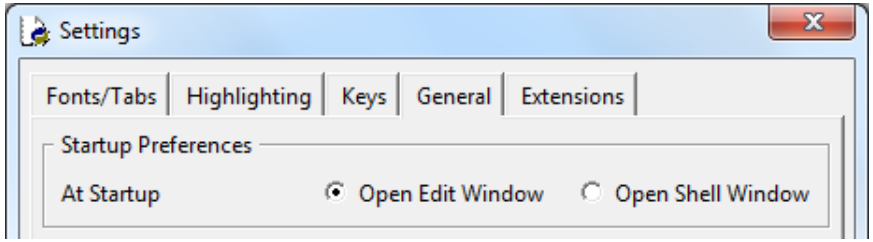
```
En Anaconda: usar comando conda, ejemplo: conda install -c anaconda numpy
```

List of Fields of Study and Corresponding Python Modules

Field of Study	Name of Python Module	Field of Study	Name of Python Module
Scientific Computation	`scipy, numpy, sympy`	Image Processing	`scikit-image`
Statistics	`pandas`	Plotting	`Matplotlib`
Networking	`networkx`	Database	`SQLAlchemy`
Cryptography	`pyOpenSSL`	HTML and XML parsing	`BeautifulSoup`
Game Development	`PyGame`	Natural Language Processing	`nltk`
Graphic User Interface	`pyQT`	Testing	`nose`
Machine Learning	`scikit-learn, tensorflow`		

Configuración editor IDLE y verificación

- Configuración de IDLE (Integrated DeveLopment Environment for Python):
  - Ejecutar IDLE de Python y en Options→Configure IDLE → General →Open Edit Window. Click en Ok y cerrar.



- Escribir con IDLE el fichero holamundo.py

```
# Escribe 'Hola, Mundo' en salida standard
print('Hola, Mundo')
```

- En una ventana de comandos ejecutar con:

```
>python holamundo.py
```

Implementaciones alternativas de Python

- Hay paquetes que incluyen librerías especializadas:
  - Anaconda (for data management, analysis and visualization of large data sets)
  - Enthought Canopy (for scientific computing)
  - ActiveState ActivePython (scientific computing modules)
  - pythonxy (Scientific-oriented Python Distribution)
  - winpython (scientific Python distribution for Windows)
  - Conceptive Python SDK (business, desktop and database)
  - PyIMSL Studio (for numerical analysis)
  - eGenix PyRun (portable Python runtime)
- Versión cloud:
  - PythonAnywhere (run Python in the browser)

Paquetes Python para computación científica y numérica

- **Numpy** - paquete fundamental para computación científica y numérica con Python.
- **SciPy** – librería para computación científica y técnica. Contiene módulos para optimización, álgebra lineal, integración, interpolación, funciones especiales, FFT, procesado de señal e imagen, solucionadores de EDO y otros temas comunes en ciencias e ingeniería.
- **Matplotlib** – librería para gráficos 2D.
- **Pandas** – librería para análisis de datos.
- **SimPy** – librería para cálculos simbólicos.

Anaconda - instalación

- Anaconda es un paquete de distribución que incluye un compilador Python, paquetes Python y el editor Spyder, todo en un solo paquete.
- Anaconda incluye Python, Jupyter Notebook y otros paquetes de uso común en computación científica y ciencia de datos. Por ejemplo, con Jupyter notebook se puede trabajar con Python de forma interactiva.
- Descargar anaconda para el SO y versión Python deseado.
- Se encuentra virtualizado en el portal Porticada UC

Editores Python

Un editor es un programa donde se puede crear, ejecutar, probar y depurar código. Ejemplos de editores Python son:

- Python IDLE
- Spyder
- Jupyter Notebook
- Visual Studio Code
- Visual Studio
- PyCharm

idle

Python 3.7.2 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: F:\users\pedro\discoD\Cursos\CAI\Python\F2C.py ==========
Enter temperature in degrees Fahrenheit: 60
Temperature in degrees Centigrade is 15.555555555555555 degrees C
>>>
```

Ln: 7 Col: 4

F2C.py - F:\users\pedro\discoD\Cursos\CAI\Python\F2C.py (3.7.2)

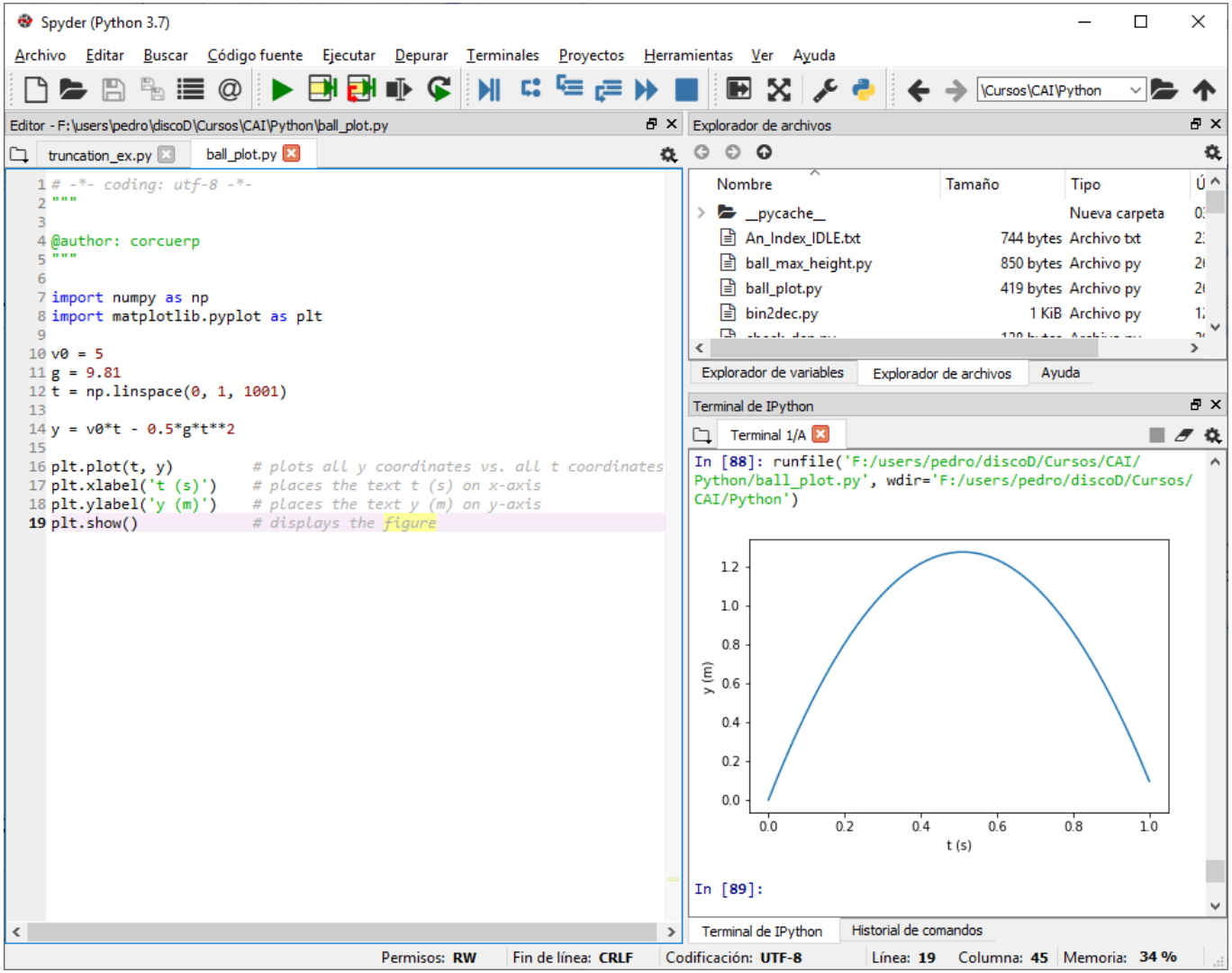
File Edit Format Run Options Window Help

```
# A function to convert degrees Fahrenheit to degrees Centigrade.
# See Exercise 1(a).
# Save file as F2C.py.
# Run the Module (or type F5).
def F2C():
    F = int(input('Enter temperature in degrees Fahrenheit: '))
    C = (F - 32) * 5 / 9
    print('Temperature in degrees Centigrade is {} degrees C'.format(C))

F2C()
```

Ln: 11 Col: 0

Spyder


Spyder (Python 3.7)
Archivo Editar Buscar Código fuente Ejecutar Depurar Terminales Proyectos Herramientas Ver Ayuda
\Cursos\CAI\Python
Editor - F:\users\pedro\discoD\Cursos\CAI\Python\ball_plot.py
truncation_ex.py
ball_plot.py
-*- coding: utf-8 -*-
"""
@author: corcuerp
"""
import numpy as np
import matplotlib.pyplot as plt
v0 = 5
g = 9.81
t = np.linspace(0, 1, 1001)
y = v0*t - 0.5*g*t**2
plt.plot(t, y) # plots all y coordinates vs. all t coordinates
plt.xlabel('t (s)') # places the text t (s) on x-axis
plt.ylabel('y (m)') # places the text y (m) on y-axis
plt.show() # displays the figure
Explorador de archivos
Nombre Tamaño Tipo
__pycache__ Nueva carpeta
An_Index_IDLE.txt 744 bytes Archivo txt
ball_max_height.py 850 bytes Archivo py
ball_plot.py 419 bytes Archivo py
bin2dec.py 1 KiB Archivo py
Explorador de variables Explorador de archivos Ayuda
Terminal de IPython
Terminal 1/A
In [88]: runfile('F:/users/pedro/discoD/Cursos/CAI/Python/ball_plot.py', wdir='F:/users/pedro/discoD/Cursos/CAI/Python')
y (m)
t (s)
In [89]:
Terminal de IPython Historial de comandos
Permisos: RW Fin de línea: CRLF Codificación: UTF-8 Línea: 19 Columna: 45 Memoria: 34 %

Tutoriales

- La página de documentación de Python ofrece una serie de tutoriales según la versión.
- Una alternativa que ofrece una visión del lenguaje, algoritmos y estructuras de datos es la página del libro Introduction to Programming in Python (IPP) de Sedgewick y Wayne.
- Tutorial numpy
- Tutorial matplotlib

Comentarios

- Para escribir un comentario en Python, se coloca el carácter # (hash) antes del comentario.
- Python ignora todo lo que hay después de la marca hash hasta el final de la línea. Se puede colocar en cualquier parte del código.
- Para conseguir comentarios multilínea se usa:
  - el carácter # en cada línea del comentario
  - cadenas de caracteres multilínea encerrados con """ … """

Documentación de código con docstrings

- La documentación del código Python se basa en docstrings, que son cadenas de caracteres (encerradas con `'''`) que se visualizan con la función `help()`.

```
>>> help(str)
```

- Colocadas debajo del objeto que se quiere documentar, asigna el valor `__doc__` automáticamente

Documentación de código con docstrings

```
def get_spreadsheet_cols(file_loc, print_cols=False):
    """Gets and prints the spreadsheet's header columns
    Parameters
    ----------
    file_loc : str
        The file location of the spreadsheet
    print_cols : bool, optional
        A flag used to print the columns to the console (default is False)
    Returns
    -------
    list
        a list of strings used that are the header columns
    """
    file_data = pd.read_excel(file_loc)
    col_headers = list(file_data.columns.values)
    if print_cols:
        print("\n".join(col_headers))
    return col_headers
```

Variables

- Una variable es un nombre que se refiere a un valor (obtenido de constantes, expresiones o funciones), definida con el operador de asignación ("=").
- Se les asigna un tipo dinámicamente, es decir, no se declara su tipo y su tipo puede cambiar.
- Los nombres de variables deben autodocumentarse, y pueden ser de cualquier longitud. Se forma por letras, números (nunca al inicio del nombre) y el carácter "_". Por convención se usan minúsculas.
- No pueden coincidir con las palabras reservadas.

Palabras reservadas

- Las palabras reservadas no se pueden usar como identificadores de variables.

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

- Se puede conocer el tipo de dato que contiene una variable con la instrucción `type(var)`

Valores y Tipos de datos

- Un valor es uno de los elementos básicos con los que trabaja un programa, como una letra o número.
- Un tipo de dato es el conjunto de valores y el conjunto de operaciones definidas en esos valores.
- Python tiene un gran número de tipos de datos incorporados tales como Números (Integer, Float, Boolean, Complex Number), String, List, Tuple, Set, Dictionary and File.
- Otros tipos de datos de alto nivel, tales como Decimal y Fraction, están soportados por módulos externos.

Integers - Enteros

- Tipo de dato (int) para representar enteros o números naturales.

valores	*integers*						
literales típicos	1234 99 0 1000000						
operaciones	*sign*	*add*	*subtract*	*multiply*	*floored divide*	*remainder*	*power*
operadores	+ -	+	-	*	//	%	**

Tipo de dato int Python

  - Se puede expresar enteros en hexadecimal con el prefijo 0x (o 0X); en octal con el prefijo 0o (o 0O); y en binario con prefijo 0b (o 0B). Ejemplo: 0x1abc, 0X1ABC, 0o1776, 0b11000011.
  - A diferencia de otros lenguajes, los enteros en Python son de tamaño ilimitado.

Integers - Enteros

- Ejemplos

```
>>> 123 + 456 - 789
-210
>>> 12345678901234567890123456789O + 1
12345678901234567890123456789O1
>>> 12345678901234567890123456789O12345678901234567890 + 1
12345678901234567890123456789O12345678901234567891
>>> 2 ** 888      # Raise 2 to the power of 888
......
>>> len(str(2 ** 888))  # Convert integer to string and get its length
268                     # 2 to the power of 888 has 268 digits
>>> type(123)    # Get the type
<class 'int'>
>>> help(int)    # Show the help menu for type int
......
>>> float(1234)    # type casting
1234.0
```

Floating-Point Numbers - Reales

- Tipo de dato (float) para representar números en punto flotantes, para uso en aplicaciones científicas o comerciales

valores	*real numbers*				
literales típicos	3.14159	6.022e23	2.0	1.4142135623730951	
operaciones	*addition*	*subtraction*	*multiplication*	*division*	*exponentiation*
operadores	+	-	*	/	**

Tipo de dato float Python

  - Para obtener el máximo valor entero usar (`import sys`) `sys.float_info.max`
  - Tienen una representación IEEE de 64 bits. Típicamente tienen 15 dígitos decimales de precisión

Floating-Point Numbers - Reales

- Ejemplos

```
>>> 1.23 * -4e5
-492000.0
>>> type(1.2)          # Get the type
<class 'float'>
>>> import math        # Using the math module
>>> math.pi
3.141592653589793
>>> import random      # Using the random module
>>> random.random()    # Generate a random number in [0, 1)
0.890839384187198
```

Números complejos

- Tipo de dato (complex) para representar números complejos de la forma `a + bj`

```
>>> x = 1 + 2j  # Assign var x to a complex number
>>> x           # Display x
(1+2j)
>>> x.real      # Get the real part
1.0
>>> x.imag      # Get the imaginary part
2.0
>>> type(x)     # Get type
<class 'complex'>
>>> x * (3 + 4j)  # Multiply two complex numbers
(-5+10j)
>>> z = complex(2, -3) # Assign a complex number
```

Booleans

- Tipo de dato (bool) que tiene dos valores: `True` y `False`
- El entero 0, un valor vacío (como una cadena vacía '', "", lista vacía [ ], tuple vacía ( ), diccionario vacío { }), y None es tratado como `False`; todo lo demás es tratado como `True` .
- Los Booleans se comportan como enteros en operaciones aritméticas con 1 para True y 0 para False.

Booleans

- Ejemplos

```
>>> 8 == 8       # Compare
True
>>> 8 == 9
False
>>> type(True)  # Get type
<class 'bool'>
>>> bool(0)
False
>>> bool(1)
True
>>> True + 3
4
>>> False + 1
1
>>> 0.1+0.2-0.3 == 0
False
```

Otros tipos

- Otros tipos de números son proporcionados por módulos externos, como decimal y fraction

```
# floats are imprecise
>>> 0.1 * 3
0.30000000000000004

# Decimal are precise
>>> import decimal  # Using the decimal module
>>> x = decimal.Decimal('0.1')  # Construct a Decimal object
>>> x * 3    # Multiply  with overloaded * operator
Decimal('0.3')
>>> type(x)  # Get type
<class 'decimal.Decimal'>
```

El valor None

- Python proporciona un valor especial llamado None que puede ser usado para inicializar un objeto (en OOP)

```
>>> x = None
>>> type(x)   # Get type
<class 'NoneType'>
>>> print(x)
None

# Use 'is' and 'is not' to check for 'None' value.
>>> print(x is None)
True
>>> print(x is not None)
False
```

Tipado dinámico

- Python tiene tipado dinámico, esto es, asocia tipos con objetos en lugar de variables. Así, una variable no tiene un tipo fijo y se le puede asignar un objeto de cualquier tipo. Una variable solo proporciona una referencia a un objeto.
- No es necesario declarar una variable. Una variable se crea automáticamente cuando un valor es asignado la primera vez, que enlaza el objeto a la variable. Se puede usar la función implícita type(nombre_var) para obtener el tipo de objeto referenciado por una variable.

Tipado dinámico y operador asignación

- Ejemplo:

```
>>> x = 1          # Assign an int value to create variable x
>>> x              # Display x
1
>>> type(x)        # Get the type of x
<class 'int'>
>>> x = 1.0        # Re-assign x to a float
>>> x
1.0
>>> type(x)        # Show the type
<class 'float'>
>>> x = 'hello'    # Re-assign x to a string
>>> x
'hello'
>>> type(x)        # Show the type
<class 'str'>
>>> x = '123'      # Re-assign x to a string (of digits)
>>> x
'123'
>>> type(x)        # Show the type
<class 'str'>
```

Conversión de tipo (casting)

- Se puede convertir tipos mediante las funciones integradas int( ), float( ), str( ), bool( ), etc.

```
>>> x = '123'
>>> type(x)
<class 'str'>
>>> x = int(x)    # Parse str to int, and assign back to x
>>> x
123
>>> type(x)
<class 'int'>
>>> x = float(x)  # Convert x from int to float, and assign back to
x
>>> x
123.0
>>> type(x)
<class 'float'>
```

Conversión de tipo (casting)

```
>>> x = str(x)  # Convert x from float to str, and assign back to x
>>> x
'123.0'
>>> type(x)
<class 'str'>
>>> len(x)        # Get the length of the string
5
>>> x = bool(x) # Convert x from str to boolean, and assign back to x
>>> x             # Non-empty string is converted to True
True
>>> type(x)
<class 'bool'>
>>> x = str(x)    # Convert x from bool to str
>>> x
'True'
```

El operador asignación (=)

- En Python no es necesario *declarar* las variables antes de usarlas. La asignación inicial crea la variable y enlaza el valor a la variable

```
>>> x = 8          # Create a variable x by assigning a value
>>> x = 'Hello'    # Re-assign a value (of a different type) to x

>>> y              # Cannot access undefined (unassigned) variable
NameError: name 'y' is not defined
```

del

- Se puede usar la instrucción `del` para eliminar una variable

```
>>> x = 8      # Create variable x via assignment
>>> x
8
>>> del x      # Delete variable x
>>> x
NameError: name 'x' is not defined
```

Asignación por pares y en cadena

- La asignación es asociativa por la derecha

```
>>> a = 1  # Ordinary assignment
>>> a
1
>>> b, c, d = 123, 4.5, 'Hello' # assignment of 3 variables pares
>>> b
123
>>> c
4.5
>>> d
'Hello'
>>> e = f = g = 123  # Chain assignment
>>> e
123
>>> f
123
>>> g
123
```

Operadores aritméticos

Operador	Descripción	Ejemplos
+	Addition	
-	Subtraction	
*	Multiplication	
/	Float Division (returns a float)	1 / 2 ⇒ 0.5 -1 / 2 ⇒ -0.5
//	Integer Division (returns the floor integer)	1 // 2 ⇒ 0 -1 // 2 ⇒ -1 8.9 // 2.5 ⇒ 3.0 -8.9 // 2.5 ⇒ -4.0 -8.9 // -2.5 ⇒ 3.0
**	Exponentiation	2 ** 5 ⇒ 32 1.2 ** 3.4 ⇒ 1.858729691979481
%	Modulus (Remainder)	9 % 2 ⇒ 1 -9 % 2 ⇒ 1 9 % -2 ⇒ -1 -9 % -2 ⇒ -1 9.9 % 2.1 ⇒ 1.5 -9.9 % 2.1 ⇒ 0.6000000000000001

Operadores de comparación

- Los operadores de comparación se aplican a enteros y flotantes y producen un resultado booleano

Operador	Descripción	Ejemplo
<, <=, >, >=, ==, !=	Comparison	2 == 3 3 != 2 2 < 13 2 <= 2 13 > 2 3 >= 3
in, not in	x in y comprueba si x está contenido en la secuencia y	lis = [1, 4, 3, 2, 5] if 4 in lis: …. if 4 not in lis: …

is, is not	x is y es True si x y y hacen referencia al mismo objeto	x = 5
		if (type(x) is int): …
		x = 5.2
		if (type(x) is not int): …

Comparación encadenada

- Python permite comparación encadenada de la forma n1 < x < n2

```
>>> x = 8
>>> 1 < x < 10
True
>>> 1 < x and x < 10  # Same as above
True
>>> 10 < x < 20
False
>>> 10 > x > 1
True
>>> not (10 < x < 20)
True
```

Operadores lógicos

- Se aplican a booleans. No hay exclusive-or (xor)

Operador	Descripción
and	Logical AND
or	Logical OR
not	Logical NOT

a	not a
False	True
True	False

a	b	a and b	a or b
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Truth-table definitions of `bool` *operations*

Comparación de secuencias

- Los operadores de comparación están sobrecargados para aceptar secuencias (string, listas, tuplas)

```
>>> 'a' < 'b'
True
>>> 'ab' < 'aa'
False
>>> 'a' < 'b' < 'c'
True
>>> (1, 2, 3) < (1, 2, 4)
True
>>> [1, 2, 3] <= [1, 2, 3]
True
```

Operadores de bits

- Permiten operaciones a nivel de bits

Operador	Descripción	Ejemplo x=0b10000001 y=0b10001111
&	bitwise AND	x & y ⇒ 0b10000001
|	bitwise OR	x | y ⇒ 0b10001111
~	bitwise NOT (or negate)	~x ⇒ -0b10000010
^	bitwise XOR	x ^ y ⇒ 0b00001110
<<	bitwise Left-Shift (padded with zeros)	x << 2 ⇒ 0b1000000100
>>	bitwise Right-Shift (padded with zeros)	x >> 2 ⇒ 0b100000

Operadores de asignación

Operador	Ejemplo	Equivalente a
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

```
b, c, d = 123, 4.5, 'Hello' # asignación multiple
```

Funciones integradas

- Python contiene funciones integradas para manipular números:
  - Matemáticas: `round()`,`pow()`,`abs()`,`max()`,`min()`
  - Conversión de tipos: `int()`, `float()`, `str()`, `bool()`, `type()`
  - Conversión de base: `hex()`, `bin()`, `oct()`

```
>>> x = 1.23456 # Test built-in function round()
>>> type(x)
<type 'float'>
>>> round(x)     # Round to the nearest integer
1
>>> type(round(x))
<class 'int'>
```

Funciones integradas

```
>>> round(x, 1)  # Round to 1 decimal place
1.2
>>> round(x, 2)  # Round to 2 decimal places
1.23
>>> round(x, 8)  # No change - not for formatting
1.23456
>>> pow(2, 5) # Test other built-in functions
32
>>> abs(-4.1)
4.1
# Base radix conversion
>>> hex(1234)
'0x4d2'
>>> bin(254)
'0b11111110'
>>> oct(1234)
'0o2322'
>>> 0xABCD  # Shown in decimal by default
43981
```

Funciones integradas

```
# List built-in functions
>>> dir(__builtins__)
['type', 'round', 'abs', 'int', 'float', 'str', 'bool', 'hex',
'bin', 'oct',......]

# Show number of built-in functions
>>> len(dir(__builtins__))  # Python 3
154

# Show documentation of __builtins__ module
>>> help(__builtins__)
```

Cadenas de caracteres - Strings

- Tipo de dato (str) para representar cadenas de caracteres, para uso en procesado de textos.
  - Se delimitan por ('...'), ("..."), ('''...'''), o ("""...""")
  - Python 3 usa el conjunto de caracteres Unicode
  - Para especificar caracteres especiales se usan "secuencias de escape". Ejemplo: \t, \n, \r
  - Los String son immutables, es decir, su contenido no se puede modificar
  - Para convertir números en strings se usa la función `str()`
  - Para convertir strings a números se usa `int()` o `float()`

Ejemplo Strings

```
>>> s1 = 'apple'
>>> s1
'apple'
>>> s2 = "orange"
>>> s2
'orange'
>>> s3 = "'orange'"   # Escape sequence not required
>>> s3
"'orange'"
>>> s3 ="\"orange\""  # Escape sequence needed
>>> s3
'"orange"'

# A triple-single/double-quoted string can span multiple lines
>>> s4 = """testing
testing"""
>>> s4
'testing\ntesting'
```

Funciones y operadores para cadenas de caracteres

Función/Operador	Descripción	Ejemplos `s = 'Hello'`
len()	Length	len(s) ⇒ 5
in	Contain?	'ell' in s ⇒ True 'he' in s ⇒ False
+	Concatenation	s + '!' ⇒ 'Hello!'
*	Repetition	s * 2 ⇒ 'HelloHello'
[i], [-i]	Indexing to get a character. The front index begins at 0; back index begins at - 1 (=len()-1).	s[1] ⇒ 'e' s[-4] ⇒ 'e'
[m:n], [m:], [:n], [m:n:step]	Slicing to get a substring. From index m (included) to n (excluded) with an optional step size. The default m=0, n=-1, step=1.	s[1:3] ⇒ 'el' s[1:-2] ⇒ 'el' s[3:] ⇒ 'lo' s[:-2] ⇒ 'Hel' s[:] ⇒ 'Hello' s[0:5:2] ⇒ 'Hlo'

Ejemplo de funciones/operadores Strings

```
>>> s = "Hello, world"   # Assign a string literal to the variable
s
>>> type(s)              # Get data type of s
<class 'str'>
>>> len(s)       # Length
12
>>> 'ello' in s  # The in operator
True
# Indexing
>>> s[0]       # Get character at index 0; index begins at 0
'H'
>>> s[1]
'e'
>>> s[-1]      # Get Last character, same as s[len(s) - 1]
'd'
>>> s[-2]      # 2nd last character
'l'
```

Ejemplo de funciones/operadores Strings

```
# Slicing
>>> s[1:3]     # Substring from index 1 (included) to 3 (excluded)
'el'
>>> s[1:-1]
'ello, worl'
>>> s[:4]      # Same as s[0:4], from the beginning
'Hell'
>>> s[4:]      # Same as s[4:-1], till the end
'o, world'
>>> s[:]       # Entire string; same as s[0:len(s)]
'Hello, world'
# Concatenation (+) and Repetition (*)
>>> s = s + " again"  # Concatenate two strings
>>> s
'Hello, world again'
>>> s * 3             # Repeat 3 times
'Hello, world againHello, world againHello, world again'
>>> s[0] = 'a'  # String is immutable
TypeError: 'str' object does not support item assignment
```

Funciones específicas para cadenas de caracteres

- La clase str proporciona varias funciones miembro. Suponiendo que s es un objeto str:
  - `s.strip(), s.rstrip(), s.lstrip()`: the strip() strips the leading and trailing whitespaces. rstrip() strips the right whitespaces and lstrip() the left whitespaces.
  - `s.upper(), s.lower(), s.isupper(), s.islower()`
  - `s.find(s), s.index(s)`
  - `s.startswith(s)`
  - `s.endswith(s)`
  - `s.split(delimiter-str), delimiter-str.join(list-of-strings)`

Conversión de tipos

- Explícita: uso de funciones `int()`, `float()`, `str()`, y `round()`

function call	*description*
`str(x)`	*conversion of object x to a string*
`int(x)`	*conversion of object x to a integer or conversion of float x to an integer by truncation towards zero*
`float(x)`	*conversion of string or integer x to float*
`round(x)`	*nearest integer to number x*

APIs for some built-in type conversion functions

- Implícita: Python convierte automáticamente enteros y flotantes convenientemente.

¿Tipo carácter?

Python no tiene un tipo de dato dedicado a caracteres. Un carácter es un string de longitud 1. Las funciones integradas `ord()` y `char()` operan sobre string 1

```
>>> ord('A') # ord(c) returns the integer ordinal (Unicode)
65
>>> ord('水')
27700
# chr(i) returns a one-character string with Unicode ordinal I
# 0 <= i <= 0x10ffff.
>>> chr(65)
'A'
>>> chr(27700)
'水'
```

Formato de Strings

Python 3 usa la función `format()` y `{}`

```
# Replace format fields {} by arguments in format() in the same
order
>>> '|{}|{}|more|'.format('Hello', 'world')
'|Hello|world|more|'

# You can use positional index in the form of {0}, {1}, ...
>>> '|{0}|{1}|more|'.format('Hello', 'world')
'|Hello|world|more|'
>>> '|{1}|{0}|more|'.format('Hello', 'world')
'|world|Hello|more|'

# You can use keyword inside {}
>>> '|{greeting}|{name}|'.format(greeting='Hello', name='Peter')
'|Hello|Peter|'
```

Formato de Strings

```
# specify field width and alignment in the form of i:n or key:n,
# where i positional index, key keyword, and n field width.
>>> '|{1:8}|{0:7}|'.format('Hello', 'Peter')
'|Peter   |Hello  |'      # Default left-aligned
# > (right align), < (left align), -< (fill char)
>>> '|{1:8}|{0:>7}|{2:-<10}|'.format('Hello', 'Peter', 'again')
'|Peter   |  Hello|again-----|'
>>> '|{greeting:8}|{name:7}|'.format(name='Peter', greeting='Hi')
'|Hi      |Peter  |'
# Format int using 'd' or 'nd', Format float using 'f' or 'n.mf'
>>> '|{0:.3f}|{1:6.2f}|{2:4d}|'.format(1.2, 3.456, 78)
'|1.200|  3.46|  78|'
# With keywords
>>> '|{a:.3f}|{b:6.2f}|{c:4d}|'.format(a=1.2, b=3.456, c=78)
'|1.200|  3.46|  78|'
```

Formato de Strings

Se pueden usar las funciones string `str.rjust(n)`, `str.ljust(n)`, `str.center(n)`, `str.zfill(n)` donde n es el ancho de campo

```
>>> '123'.rjust(5) # Setting field width and alignment
'  123'
>>> '123'.ljust(5)
'123  '
>>> '123'.center(5)
' 123 '
>>> '123'.zfill(5)  # Pad with leading zeros
'00123'
>>> '1.2'.rjust(5) # Floats
'  1.2'
>>> '-1.2'.zfill(6)
'-001.2'
```

Listas

- Python dispone de una estructura de datos integrada (lista - list) para arrays dinámicos.
- Es una estructura de datos que almacena una secuencia de objetos, normalmente del mismo tipo, que puede crecer y encogerse dinámicamente.
- Una lista se encierra entre *corchetes* [ ] y los elementos se acceden mediante índice, **empezando por cero**.
- Hay funciones integradas (ej. `len()`, `max()`, `min()`, `sum()`), y operadores.

Listas - ejemplo

```
x = [0.30, 0.60, 0.10]
y = [0.50, 0.10, 0.40]
total = 0.0
for i in range(len(x)):
    total += x[i]*y[i]
```

i	*x[i]*	*y[i]*	*x[i]*y[i]*	*total*
				0.00
0	0.30	0.50	0.15	0.15
1	0.60	0.10	0.06	0.21
2	0.10	0.40	0.04	0.25

Trace of dot product computation

Operadores para listas

Operador	Descripción	Ejemplos `lst = [8, 9, 6, 2]`
in	Contain?	9 in lst ⇒ True 5 in lst ⇒ False
+	Concatenation	lst + [5, 2] ⇒ [8, 9, 6, 2, 5, 2]
*	Repetition	lst * 2 ⇒ [8, 9, 6, 2, 8, 9, 6, 2]
[i], [-i]	Indexing to get an item. Front index begins at 0; back index begins at -1 (or len-1).	lst[1] ⇒ 9 lst[-2] ⇒ 6 lst[1] = 99 ⇒ modify an existing item
[m:n], [m:], [:n], [m:n:step]	Slicing to get a sublist. From index m (included) to n (excluded) with an optional step size. The default m is 0, n is len-1.	lst[1:3] ⇒ [9, 6] lst[1:-2] ⇒ [9] lst[3:] ⇒ [2] lst[:-2] ⇒ [8, 9] lst[:] ⇒ [8, 9, 6, 2] lst[0:4:2] ⇒ [8, 6] newlst = lst[:] ⇒ copy the list lst[4:] = [1, 2] ⇒ modify a sub-list
del	Delete one or more items (for mutable sequences only)	del lst[1] ⇒ lst is [8, 6, 2] del lst[1:] ⇒ lst is [8] del lst[:] ⇒ lst is [] (clear all items)

Funciones para listas

Función	Descripción	Ejemplos `lst = [8, 9, 6, 2]`
len()	Length	len(lst) ⇒ 4
max(), min()	Maximum and minimum value (for list of numbers only)	max(lst) ⇒ 9 min(lst) ⇒ 2
sum()	Sum (for list of numbers only)	sum(lst) ⇒ 16

- Suponiendo que lst es un objeto list:
  - lst.append(item): append the given item behind the lst and return None; same as lst[len(lst):] = [item].
  - lst.extend(lst2): append the given list lst2 behind the lst and return None; same as lst[len(lst):] = lst2.
  - lst.insert(index, item): insert the given item before the index and return None. Hence, lst.insert(0, item) inserts before the first item of the lst; lst.insert(len(lst), item) inserts at the end of the lst which is the same as lst.append(item).
  - lst.index(item): return the index of the first occurrence of item; or error.
  - lst.remove(item): remove the first occurrence of item from the lst and return None; or error.
  - lst.pop(): remove and return the last item of the lst.
  - lst.pop(index): remove and return the indexed item of the lst.
  - lst.clear(): remove all the items from the lst and return None; same as del lst[:].
  - lst.count(item): return the occurrences of item.
  - lst.reverse(): reverse the lst in place and return None.
  - lst.sort(): sort the lst in place and return None.
  - lst.copy(): return a copy of lst; same as lst[:].

Operaciones y funciones comunes con Listas

Common List Functions and Operators	
Operation	**Description**
`[]` `[`*elem*$_1$, *elem*$_2$, ..., *elem*$_n$`]`	Creates a new empty list or a list that contains the initial elements provided.
`len(`*l*`)`	Returns the number of elements in list *l*.
`list(`*sequence*`)`	Creates a new list containing all elements of the sequence.
`values * num`	Creates a new list by replicating the elements in the `values` list `num` times.
`values + moreValues`	Creates a new list by concatenating elements in both lists.

Operaciones y funciones comunes con Listas

Common List Functions and Operators

Operation	Description
`l[from : to]`	Creates a sublist from a subsequence of elements in list *l* starting at position `from` and going through but not including the element at position `to`. Both `from` and `to` are optional. (See Special Topic 6.2.)
`sum(l)`	Computes the sum of the values in list *l*.
`min(l)` `max(l)`	Returns the minimum or maximum value in list *l*.
l_1 == l_2	Tests whether two lists have the same elements, in the same order.

Métodos comunes con Listas

Common List Methods	
Method	**Description**
l.pop() *l*.pop(*position*)	Removes the last element from the list or from the given position. All elements following the given position are moved up one place.
l.insert(*position*, *element*)	Inserts the element at the given position in the list. All elements at and following the given position are moved down.
l.append(*element*)	Appends the element to the end of the list.
l.index(*element*)	Returns the position of the given element in the list. The element must be in the list.
l.remove(*element*)	Removes the given element from the list and moves all elements following it up one position.
l.sort()	Sorts the elements in the list from smallest to largest.

Tuplas

- Es similar a las listas excepto que es inmutable (como los string).
- Consiste en una serie de elementos separados por comas, encerrados entre *paréntesis*.
- Se puede convertir a listas mediante list(tupla).
- Se opera sobre tuplas (tup) con:
  - funciones integradas len(tup), para tuplas de números max(tup), min(tup), sum(tup)
  - operadores como in, + y *
  - funciones de tupla tup.count(item), tup.index(item), etc

Diccionarios

- Soportan pares llave-valor (mappings). Es mutable.
- Un diccionario se encierra entre llaves { }. La llave y el valor se separa por : con el formato `{k1:v1, k2:v2, ...}`
- A diferencia de las listas y tuplas que usan un índice entero para acceder a los elementos, los diccionarios se pueden indexar usando cualquier tipo llave (número, cadena, otros tipos).

Ejemplo - Diccionarios

```
>>> dct = {'name':'Peter', 'gender':'male', 'age':21}
>>> dct
{'age': 21, 'name': 'Peter', 'gender': 'male'}
>>> dct['name']       # Get value via key
'Peter'
>>> dct['age'] = 22   # Re-assign a value
>>> dct
{'age': 22, 'name': 'Peter', 'gender': 'male'}
>>> len(dct)
3
>>> dct['email'] = 'abcd@sant.com'   # Add new item
>>> dct
{'name': 'Peter', 'age': 22, 'email': 'abcd@sant.com', 'gender':
'male'}
>>> type(dct)
<class 'dict'>
```

Funciones para diccionarios

- Las más comunes son: (dct es un objeto dict)
  - dct.has_key()
  - dct.items(), dct.keys(), dct.values()
  - dct.clear()
  - dct.copy()
  - dct.get()
  - dct.update(dct2): merge the given dictionary dct2 into dct. Override the value if key exists, else, add new key-value.
  - dct.pop()

Operaciones comunes con diccionarios

Common Dictionary Operations	
Operation	**Returns**
`d = dict()` `d = dict(c)`	Creates a new empty dictionary or a duplicate copy of dictionary *c*.
`d = {}` `d = {k1: v1, k2: v2, ..., kn: vn}`	Creates a new empty dictionary or a dictionary that contains the initial items provided. Each item consists of a key (*k*) and a value (*v*) separated by a colon.
`len(d)`	Returns the number of items in dictionary *d*.
key `in` *d* *key* `not in` *d*	Determines if the key is in the dictionary.
`d[key] = value`	Adds a new *key*/*value* item to the dictionary if the *key* does not exist. If the key does exist, it modifies the value associated with the key.
`x = d[key]`	Returns the value associated with the given key. The key must exist or an exception is raised.

Operaciones comunes con diccionarios

Common Dictionary Operations	
d.get(*key*, *default*)	Returns the value associated with the given key, or the default value if the key is not present.
d.pop(*key*)	Removes the key and its associated value from the dictionary that contains the given key or raises an exception if the key is not present.
d.values()	Returns a sequence containing all values of the dictionary.

Conjuntos - set

- Es una colección de objetos sin ordenar y no duplicados. Es una colección mutable, se puede usar add() para añadir elementos.
- Un set se especifica encerrando los elementos entre entre *llaves*.
- Se puede pensar que un set es un dict de llaves sin valor asociado.
- Python tiene operadores set: & (intersection), | (union), - (difference), ^ (exclusive-or) y in (pertenencia).

Operaciones comunes con conjuntos

Common Set Operations	
Operation	**Description**

`s = set()`
`s = set(seq)`
`s = {e1, e2, ..., en}`
	Creates a new set that is either empty, a duplicate copy of sequence *seq*, or that contains the initial elements provided.

`len(s)`	Returns the number of elements in set *s*.
element `in s` *element* `not in s`	Determines if *element* is in the set.
`s.add(`*element*`)`	Adds a new element to the set. If the element is already in the set, no action is taken.
`s.discard(`*element*`)` `s.remove(`*element*`)`	Removes an element from the set. If the element is not a member of the set, `discard` has no effect, but `remove` will raise an exception.
`s.clear()`	Removes all elements from a set.
`s.issubset(t)`	Returns a Boolean indicating whether set *s* is a subset of set *t*.

Operaciones comunes con conjuntos

Common Set Operations	
s == *t* *s* != *t*	Returns a Boolean indicating whether set *s* is equal to set *t*.
`s.union(t)`	Returns a new set that contains all elements in set *s* and set *t*.
`s.intersection(t)`	Returns a new set that contains elements that are in *both* set *s* and set *t*.
`s.difference(t)`	Returns a new set that contains elements in *s* that are not in set *t*.

Nota: `union`, `intersection` y `difference` devuelve nuevos conjuntos, no modifican el conjunto al que se aplica

Estructuras complejas

- Los contenedores son muy útiles para almacenar colecciones de valores. Las listas y diccionarios pueden contener cualquier dato incluyendo otros contenedores.
- Así se puede crear un diccionario de conjuntos o diccionario de listas

Condicionales – if - else

- Se usa cuando se requiere realizar diferentes acciones para diferentes condiciones.
- Sintaxis general:

```
if condition-1:
    block-1
elif condition-2:
    block-2
......
elif condition-n:
    block-n
else:
    else-block
```

Ejemplo:

```
if score >= 90:
     letter = 'A'
elif score >= 80:
     letter = 'B'
elif score >= 70:
     letter = 'C'
elif score >= 60:
     letter = 'D'
else:
     letter = 'F'
```

Condicionales – if - else

```
import math
#ej1.py
radio = float(input('Ingresa radio '))
cod = int(input('Ingresa codigo '))
if cod == 1 :
     per = 2*math.pi*radio
     print("Perimetro de la circunferencia = ",per)
elif cod == 2 :
     sup = math.pi*radio**2
     print('Superficie del circulo = ',sup)
elif cod == 3 :
     sup = 4*math.pi*radio**2
     print('Superficie de la esfera = ',sup)
elif cod == 4 :
     vol = 4./3.*math.pi*radio**3
     print('Volumen de la esfera = ',vol)
else:
     print('Codigo fuera de rango')
```

Condicionales – patrones

absolute value	if x < 0: x = -x

put x and y into sorted order	if x > y:
	  temp = x
	  x = y
	  y = temp

maximum of x and y	if x > y:
	  maximun = x
	else:
	  maximun = y

error check for remainder operation	if den == 0:
	  print("Division by zero")
	else:
	  print("Remainder = " , num % den)

error check for quadratic formula	discriminant = b*b – 4.0*a*c
	if discriminant < 0.0:
	  print("No real roots")
	else:
	  d = math.sqrt(discriminant)
	  print((-b + d)/2.0)
	  print((-b - d)/2.0)

Forma corta de if - else

- Sintaxis:

```
expr-1 if test else expr-2
# Evalua expr-1 si test es True; sino, evalua expr-2
```

```
>>> x = 0
>>> print('zero' if x == 0 else 'not zero')
zero

>>> x = -8
>>> abs_x = x if x > 0 else -x
>>> abs_x
8
```

Ciclo while

- Instrucción que permite cálculos repetitivos sujetos a una condición. Sintaxis general:

```
while test:
    true-block

# while loop has an optional else block
while test:
    true-block
else:  # Run only if no break encountered
    else-block
```

- El bloque else es opcional. Se ejecuta si se sale del ciclo sin encontrar una instrucción break.

Ciclo while - Ejemplo

```
# Sum from 1 to the given upperbound
n = int(input('Enter the upperbound: '))
i = 1
sum = 0
while (i <= n):
    sum += i
    i += 1
print(sum)
```

Ciclo while - Ejemplo

```
import sys
# Filename: powersoftwo.py. Accept positive integer n as a
# command-line argument. Write to standard output a table
# showing the first n powers of two.
n = int(sys.argv[1])
power = 1
i = 0
while i <= n:
    # Write the ith power of 2.
    print(str(i) + ' ' + str(power))
    power = 2 * power
    i = i + 1
# python powersoftwo.py 1
# 0 1
# 1 2
```

Ciclos - for

- Sintaxis general del ciclo for - in:

```
# sequence:string,list,tuple,dictionary,set
for item in sequence:
    true-block
# for-in loop with a else block
for item in sequence:
    true-block
else:    # Run only if no break encountered
    else-block
```

  - Se interpreta como “para cada ítem en la secuencia...”. El bloque else se ejecuta si el ciclo termina normalmente sin encontrar la instrucción break.

Ciclos - for

- Ejemplos de iteraciones sobre una secuencia.

```
# String: iterating through each character
>>> for char in 'hello': print(char)
h
e
l
l
o
# List: iterating through each item
>>> for item in [123, 4.5, 'hello']: print(item)
123
4.5
Hello
# Tuple: iterating through each item
>>> for item in (123, 4.5, 'hello'): print(item)
123
4.5
hello
```

Ciclos - for

```
# Dictionary: iterating through each key
>>> dct = {'a': 1, 2: 'b', 'c': 'cc'}
>>> for key in dct: print(key, ':', dct[key])
a : 1
c : cc
2 : b

# Set: iterating through each item
>>> for item in {'apple', 1, 2, 'apple'}: print(item)
1
2
apple

# File: iterating through each line
>>> f = open('test.txt', 'r')
>>> for line in f: print(line)
...Each line of the file...
>>> f.close()
```

Ciclos - for

- Iteraciones sobre una secuencia de secuencias.

```
# A list of 2-item tuples
>>> lst = [(1,'a'), (2,'b'), (3,'c')]
# Iterating thru the each of the 2-item tuples
>>> for i1, i2 in lst: print(i1, i2)
...
1 a
2 b
3 c

# A list of 3-item lists
>>> lst = [[1, 2, 3], ['a', 'b', 'c']]
>>> for i1, i2, i3 in lst: print(i1, i2, i3)
...
1 2 3
a b c
```

Ciclos - for

- Iteraciones sobre un diccionario.

```
>>> dct = {'name':'Peter', 'gender':'male', 'age':21}

# Iterate through the keys (as in the above example)
>>> for key in dct: print(key, ':', dct[key])
age : 21
name : Peter
gender : male

# Iterate through the key-value pairs
>>> for key, value in dct.items(): print(key, ':', value)
age : 21
name : Peter
gender : male

>>> dct.items()  # Return a list of key-value (2-item) tuples
[('gender', 'male'), ('age', 21), ('name', 'Peter')]
```

Ciclos – patrones

write first n+1 powers of 2	power = 1; n = 10;
	for i in range(n+1):
	  print(str(i) + ' ' + str(power))
	  power *= 2

write largest power of 2 less than or equal to n	power = 1; n = 10;
	while 2*power <= n:
	  power *= 2
	print(power)

write a sum
(1 + 2 + 3 + … +n)
	total = 0; n = 10;
	for i in range(1, n+1):
	  total += i
	print(total)

write a product
(n! = 1 x 2 x 3 x … x n)
	product = 1; n = 10;
	for i in range(1, n+1):
	  product*= i
	print(product)

write a table of n+1 function values	import math
	n = 10
	for i in range(n+1):
	  print(str(i) + '\t' , end='')
	  print(2.0 * math.pi * i / n , end='\n')

write a ruler	ruler = '1'; n = 10; print(ruler)
	for i in range(2, n+1):
	  ruler = ruler + ' ' + str(i) + ' ' + ruler
	  print(ruler)

Ciclos anidados

Nested Loop Examples

Nested Loops	Output	Explanation

`for i in range(3) :`
`  for j in range(4) :`
`  print("*", end="")`
`  print()`
	****
	****
	****
		Prints 3 rows of 4 asterisks each.

`for i in range(4) :`
`  for j in range(3) :`
`  print("*", end="")`
`  print()`
	***
	***
	***
	***
		Prints 4 rows of 3 asterisks each.

`for i in range(4) :`
`  for j in range(i + 1) :`
`  print("*", end="")`
`  print()`
	*
	**
	***
	****
		Prints 4 rows of lengths 1, 2, 3, and 4.

Ciclos anidados

Nested Loop Examples

```
for i in range(3) :
   for j in range(5) :
      if j % 2 == 1 :
         print("*", end="")
      else :
         print("-", end="")
   print()
```

```
-*-*-
-*-*-
-*-*-
```

Prints alternating dashes and asterisks.

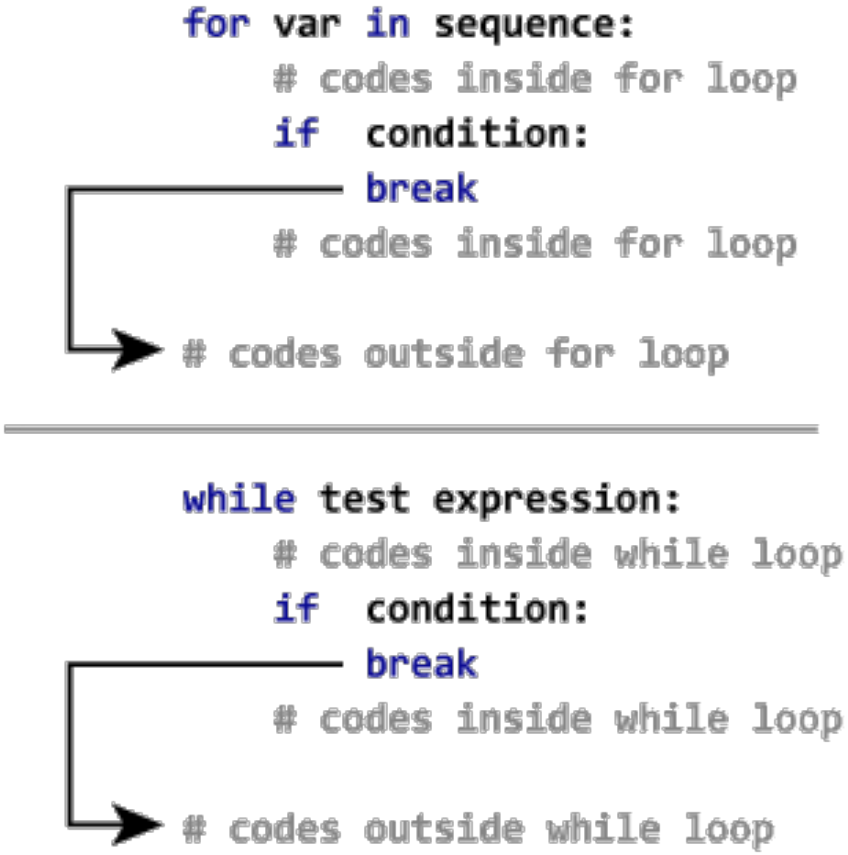
```
for i in range(3) :
   for j in range(5) :
      if i % 2 == j % 2 :
         print("*", end="")
      else :
         print(" ", end="")
   print()
```

```
* * *
 * *
* * *
```

Prints a checkerboard pattern.

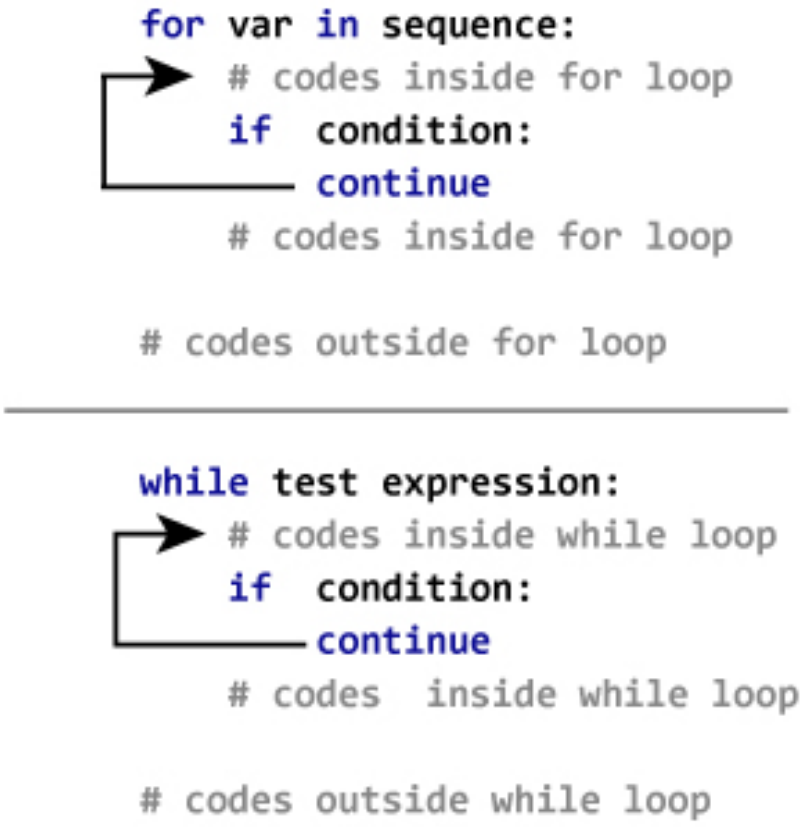
Instrucción break

- `break` termina el ciclo y sigue en la instrucción que sigue al ciclo.



Instrucción continue

- continue se usa para saltar el resto del código del ciclo y continuar con la siguiente iteración.



Instrucciones pass, loop - else

- `pass` no hace nada. Sirve como marcador de una instrucción vacía o bloque vacía.
- `loop - else` se ejecuta si del ciclo se sale normalmente sin encontrar la instrucción break.

Funciones iter() y next()

- La función iter(iterable) devuelve un objeto iterator de iterable y con next(iterator) para iterar a través de los items.

```
>>> i = iter([11, 22, 33])
>>> next(i)
11
>>> next(i)
22
>>> next(i)
33
>>> next(i)  # Raise StopIteration exception if no more item
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> type(i)
<class 'list_iterator'>
```

Función range()

- La función range produce una secuencia de enteros. Formato:
  - range(n) produce enteros desde 0 a n-1;
  - range(m, n) produce enteros desde m a n-1;
  - range(m, n, s) produce enteros desde m a n-1 en paso de s.

```
for num in range(1,5):
    print(num)
# Result
1 2 3 4
```

Función range()

```
# Sum from 1 to the given upperbound
upperbound = int(input('Enter the upperbound: '))
sum = 0
for number in range(1, upperbound+1):  # list of 1 to n
    sum += number
print("The sum is: %d" % sum)
# Sum a given list
lst = [9, 8, 4, 5]
sum = 0
for index in range(len(lst)):  # list of 0 to len-1
    sum += lst[index]
print(sum)
# Better alternative of the above
lst = [9, 8, 4, 5]
sum = 0
for item in lst:  # Each item of lst
    sum += item
print(sum)
# Use built-in function
del sum # Need to remove the sum variable before using builtin function sum
print(sum(lst))
```

Ciclos – for else

- Ejemplo de cláusula else en for

```
# List all primes between 2 and 100
for number in range(2, 101):
    for factor in range(2, number//2+1): # Look for factor
        if number % factor == 0:  # break if a factor found
            print('%d is NOT a prime' % number)
            break
    else:  # Only if no break encountered
        print('%d is a prime' % number)
```

Función enumerate()

- Se puede usar la función integrada enumerate() para obtener los índices posicionales cuando se recorre a través de una secuencia.

```
# List
>>> for i, v in enumerate(['a', 'b', 'c']): print(i, v)
0 a
1 b
2 c
>>> enumerate(['a', 'b', 'c'])
<enumerate object at 0x7ff0c6b75a50>

# Tuple
>>> for i, v in enumerate(('d', 'e', 'f')): print(i, v)
0 d
1 e
2 f
```

Función reversed()

- Se usa para iterar una secuencia en orden inverso.

```
>>> lst = [11, 22, 33]
>>> for item in reversed(lst): print(item, end=' ')
33 22 11
>>> reversed(lst)
<list_reverseiterator object at 0x7fc4707f3828>

>>> str = "hello"
>>> for c in reversed(str): print(c, end='')
olleh
```

Secuencias múltiples y función zip()

- Para iterar sobre dos o más secuencias de forma concurrente y emparejadas se usa la función zip.

```
>>> lst1 = ['a', 'b', 'c']
>>> lst2 = [11, 22, 33]
>>> for i1, i2 in zip(lst1, lst2): print(i1, i2)
a 11
b 22
c 33
>>> zip(lst1, lst2)   # Return a list of tuples
[('a', 11), ('b', 22), ('c', 33)]

# zip() for more than 2 sequences
>>> tuple3 = (44, 55)
>>> zip(lst1, lst2, tuple3)
[('a', 11, 44), ('b', 22, 55)]
```

Creación de lista y diccionario

- Existe una forma concisa para generar una lista (comprehension). Sintaxis:

```
result_list = [expression_with_item for item in in_list]
# with an optional test
result_list = [expression_with_item for item in in_list if test]
# Same as
result_list = []
for item in in_list:
    if test:
        result_list.append(item)
```

Creación de lista

- Ejemplos listas:

```
>>> sq = [item * item for item in range(1,11)]
>>> sq
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> x = [3, 4, 1, 5]
>>> sq_x = [item * item for item in x]  # no test, all items
>>> sq_x
[9, 16, 1, 25]
>>> sq_odd = [item * item for item in x if item % 2 != 0]
>>> sq_odd
[9, 1, 25]

# Nested for
>>> [(x, y) for x in range(1,3) for y in range(1,4) if x != y]
[(1, 2), (1, 3), (2, 1), (2, 3)]
```

Creación de diccionario

- Ejemplos diccionarios:

```
# Dictionary {k1:v1, k2:v2,...}
>>> d = {x:x**2 for x in range(1, 5)}  # Use braces for dictionary
>>> d
{1: 1, 2: 4, 3: 9, 4: 16}

# Set {v1, v2,...}
>>> s = {i for i in 'hello' if i not in 'aeiou'}  # Use braces
>>> s
{'h', 'l'}
```

Conversión de tipos

- En operaciones aritméticos con números de diferentes tipos, se convierten a un tipo común antes de llevar a cabo la operación.
- Las conversiones de tipos también se pueden hacer con las siguientes funciones:

`int(a)`	`Convierte a en entero`
`float(a)`	`Convierte a en real`
`complex(a)`	`Convierte en complejo a + 0j`
`complex(a,b)`	`Convierte en complejo a + bj`

- Estas funciones también sirven para convertir cadenas de caracteres a números.

Entrada - Salida

- Python dispone de funciones intrínsecas para lectura y escritura interactiva. Las más usadas para la entrada estándar son: `input()` y `print()`
- Desde la línea de comando se usa la lista `sys.argv`.

```
>python program.py -v input.dat
```

```
argv[0]: "program.py"
argv[1]: "-v"
argv[2]: "input.dat"
```

input

`input("mensaje")`

– Muestra el mensaje (opcional) y lee una línea de la entrada que se convierte en un *string*. Para convertir el string en un valor numérico se puede aplicar las funciones de conversión de tipos o `eval(string)`

```
aString = input("Escribe tu edad: ") # Mensaje de entrada
age = int(aString)                   # Conversion a int
age = int(input("Escribe tu edad: "))  # compacto
peso = float(input("Escribe tu peso: "))  # compacto

a = eval(input(prompt)) # otro metodo para leer un número
                        # como resultado de una expresión
```

Output con print

`print(objeto1, objeto2, …)`

- La función print convierte *objeto1, objeto2,…*a cadenas y los imprime en la misma línea separados por espacios, al final introduce un cambio de línea. Se puede usar el carácter nueva línea `'\n'` para forzar una nueva línea

```
>>> a = 1234.56789
>>> b = [2, 4, 6, 8]
>>> print(a,b)
1234.56789 [2, 4, 6, 8]
>>> print('a =',a, '\nb =',b)
a = 1234.56789
b = [2, 4, 6, 8]
```

print con formato

`print(objeto1, objeto2, …,end=' ')`

- Para reemplazar el carácter nueva línea `'\n'` con otro

`print('{:fmt1}{:fmt2}…'.format(arg1,arg2,…))`

- `fmt1..` son especificaciones de formato para `arg1..`

Las especificaciones de formato típicas son:

w`d`	Integer
w.*d*`f`	Floating point notation
w.*d*`e`	Exponential notation

donde *w* es el ancho del campo y *d* es el número de dígitos después del punto decimal. La salida se justifica a la derecha en los campos especificados y rellenos con blancos.

print con formato

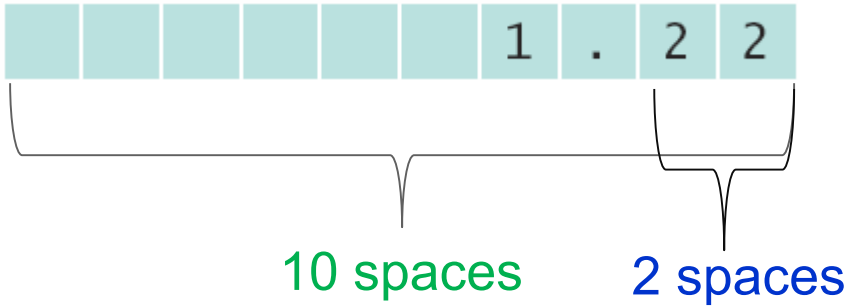
- Ejemplos:

```
>>> a = 1234.56789
>>> n = 9876
>>> print('{:7.2f} '.format(a))
1234.57
>>> print('n = {:6d}'.format(n)) # Pad with spaces
n = 9876
>>> print('n = {:06d}'.format(n)) # Pad with zeros
n =009876
>>> print('{:12.4e} {:6d}'.format(a,n))
1.2346e+03 9876
```

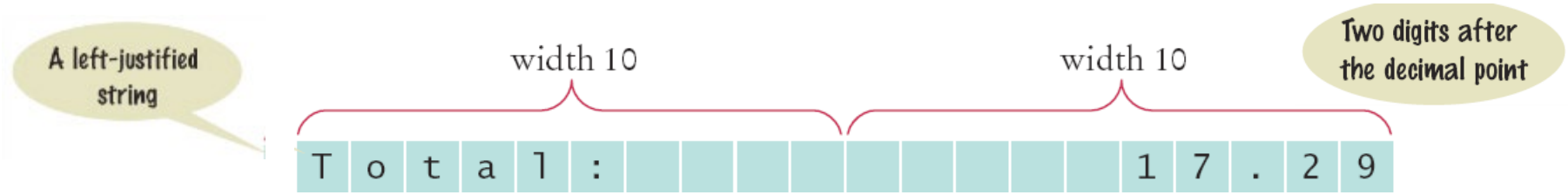
print con formato

- Otra forma de especificar el formato es:

```
print("Precio por litro %.2f"  %(price))  # dos decimales
# %10.2f especificador de formato
print(" Precio por litro %10.2f"  %(price))
```




```
print("%-10s%10.2f" %("Total: ", price))
```



print con formato

- Símbolos de formato:

```
print("%s = %f"  % ("pi", 3.14159))

%c character
%s string
%d signed integer
%x hexadecimal integer
%e exponential notation
%f floating point number
%g the shorter of %f and %e
```

Ejemplos de especificado de formato

Format Specifier Examples		
Format String	Sample Output	Comments
"%d"	24	Use d with an integer.
"%5d"	␣␣␣24	Spaces are added so that the field width is 5.
"%05d"	00024	If you add 0 before the field width, zeroes are added instead of spaces.
"Quantity:%5d"	Quantity:␣␣␣24	Characters inside a format string but outside a format specifier appear in the output.
"%f"	1.219997	Use f with a floating-point number.
"%.2f"	1.22	Prints two digits after the decimal point.
"%7.2f"	␣␣␣1.22	Spaces are added so that the field width is 7.
"%s"	Hello	Use s with a string.
"%d %.2f"	24 1.22	You can format multiple values at once.
"%9s"	␣␣␣␣Hello	Strings are right-justified by default.
"%-9s"	Hello␣␣␣␣	Use a negative field width to left-justify.
"%d%%"	24%	To add a percent sign to the output, use %%.

Forma especial de print

- Python proporciona una forma especial de la función print sin salto de línea al final de los argumentos a mostrar: incluir `end=""` como último argumento de la función print
- Se usa para imprimir valores en la misma línea usando varias instrucciones print

```
print("00",end="")
print(3+4)
# Salida
# 007
```

print con formateo literal de cadenas (f-strings)

- Los f-Strings son un método sencillo para dar formato a cadenas de texto.
- En Python, una cadena de texto se escribe entre comillas dobles ("") o comillas simples ('). Para crear f-strings, se agrega la letra f o F mayúscula antes de las comillas. Ejm: f"esto es un ejemplo de f-string"

print con formateo literal de cadenas (f-strings)

- Para imprimir variables usando f-strings en Python solo hay que especificar el nombre de las variables entre llaves {}. Al ejecutar el código, todos los nombres de las variables serán remplazados con sus respectivos valores. En caso de tener múltiples variables en la cadena de texto, cada variable necesita llaves propias {}
- Ejemplo:

```
num1 = 83
num2 = 9
print(f"El producto de {num1} y {num2} es {num1 * num2}.")
```

Matriz con Listas - lectura

```
def lee_matriz():
    #Dato de la dimensión de la matriz
    print('Lectura Matriz')
    m, n = [int(x) for x in input("Ingresa numero de filas y
columnas separado por blancos:  ").split()]
    #Creacion matriz nula en invocacion
    M = []
    for i in range(m):
        M.append([0]* n)
    #lectura de elementos
    for i in range(m):
        for j in range(n):
            M[i][j] = float(input('Ingresa elemento\
              ({0},{1}): '.format(i,j)))
     return M
```

Matriz con Listas - output

```
def imp_matriz(M):
#imprime matriz
    print ('\nMatriz')
    m = len(M)
    n = len(M[0])
    for i in range(m):
        for j in range(n):
            print(M[i][j],end='\t')
        print('')

# Prueba
M = lee_matriz()
imp_matriz(M)
```

Redirección de entrada/salida

- Redirección de la salida standard a un fichero

```
> python randomseq.py 1000 > data.txt
```

Redirecting standard output to a file

- Redirección de la entrada standard desde un fichero

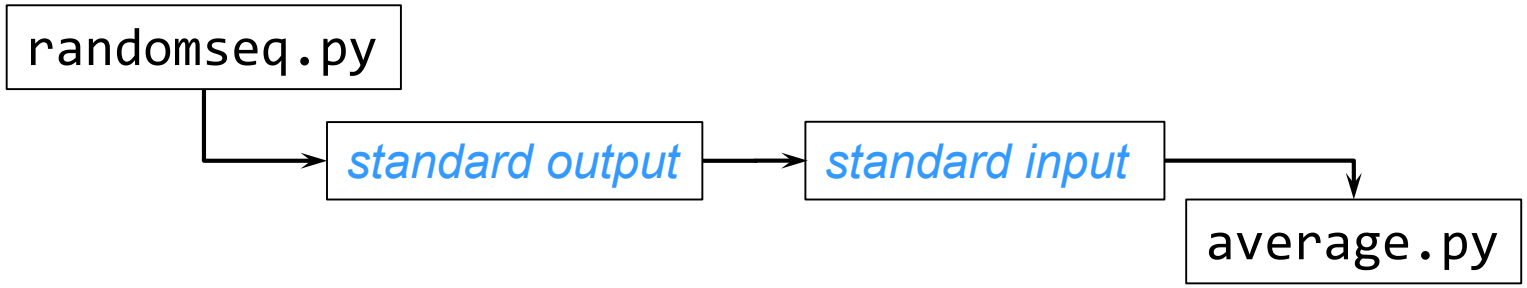
```
> python average.py < data.txt
```

Redirecting from a file to standard input

Piping

- Conectando dos programas

```
> python randomseq.py 1000 | python average.py
```



Piping the of one program to the input of another

```
> python randomseq.py 1000 > data.txt
> python average.py < data.txt
```

- Filtros

```
> python randomseq.py 9 | sort
> python randomseq.py 1000 | more
```

Ficheros

- Python dispone de funciones integradas para gestionar la entrada/salida desde ficheros:
  - open(filename_str, mode): retorna un objeto fichero. Los valores válidos de mode son: 'r' (read-only, default), 'w' (write - erase all contents for existing file), 'a' (append), 'r+' (read and write). También se puede usar 'rb', 'wb', 'ab', 'rb+' para operaciones modo binario (raw bytes).
  - file.close(): Cierra el objeto file.
  - file.readline(): lee una línea (up to a newline and including the newline). Retorna una cadena vacía después end-of-file (EOF).

Ficheros

- file.read(): lee el fichero entero. Retorna una cadena vacía después de end-of-file (EOF).
- file.write(str): escribe la cadena dada en el fichero.
- file.tell(): retorna la “posición en curso”. La “posición en curso" es el número de bytes desde el inicio del fichero en modo binario, y un número opaco en modo texto.
- file.seek(offset): asigna la “posición en curso” a offset desde el inicio del fichero.

Apertura de ficheros

- Para acceder a un fichero de datos (almacenado en disco), se debe crear un *objeto fichero* con

```
file_object = open(filename, action)
```

donde *`filename`* es una cadena que especifica el nombre del fichero (con el path si es necesario) y *`action`* es una de las siguientes cadenas:

`'r'`	Read from an existing file.
`'w'`	Write to a file. If *filename* does not exist, it is created.
`'a'`	Append to the end of the file.
`'r+'`	Read to and write from an existing file.
`'w+'`	Same as `'r+'`, but *filename* is created if it does not exist.
`'a+'`	Same as `'w+'`, but data is appended to the end of the file.

Cierre de ficheros

- Una buena práctica de programación es cerrar un fichero cuando ya no es necesario su acceso con:

```
file_object.close()
```

Lectura de datos desde ficheros

- Hay tres métodos para leer datos desde un fichero:
- El método `file_object.read(n)` lee *n* caracteres y los retorna como una cadena. Si *n* se omite, se leen todos los caracteres en el fichero.
- Con `file_object.readline(n)` se lee *n* caracteres de la línea en curso y los retorna en una cadena que termina en nueva línea \n. Si *n* se omite, se lee la línea entera.
- Con `file_object.readlines()` se leen todas las líneas del fichero.

Lectura de datos desde ficheros

- Una forma conveniente de extraer todas las líneas una a una es usar un ciclo

```
for line in file_object:
    do something with line
```

- Por ejemplo, el código para leer un fichero (sunspots.txt) con formato year/month/date/intensity:

```
x = []
data = open('sunspots.txt', 'r')
for line in data:
   x.append(eval(line.split()[3])) # intensity
data.close()
```

Escritura de datos en ficheros

- El método `file_object.write(string)` escribe una cadena en un fichero
- Con `file_object.writelines(list_strings)` se usa para escribir una lista de cadenas
- Por ejemplo, para escribir una tabla formateada de k y k^2 en el rango 101 a 110 en el fichero testfile:

```
f = open('testfile','w')
for k in range(101,111):
    f.write('{:4d} {:6d}'.format(k,k**2))
    f.write('\n')
f.close()
```

Ficheros - Ejemplos

```
>>> f = open('test.txt', 'w')  # Create (open) a file for
write
>>> f.write('apple\n')         # Write given string to file
>>> f.write('orange\n')
>>> f.close()                  # Close the file

>>> f = open('test.txt', 'r')  # Create (open) a file for
read (default)
>>> f.readline()               # Read till newline
'apple\n'
>>> f.readline()
'orange\n'
>>> f.readline()               # Return empty string after
end-of-file
''
>>> f.close()
```

Ficheros - Ejemplos

```
>>> f = open('test.txt', 'r')
>>> f.read()                        # Read entire file
'apple\norange\n'
>>> f.close()
>>> f = open('test.txt', 'r') # Test tell() and seek()
>>> f.tell()
0
>>> f.read()
'apple\norange\n'
>>> f.tell()
13
>>> f.read()
''
>>> f.seek(0)  # Rewind
0
>>> f.read()
'apple\norange\n'
>>> f.close()
```

Iterando a través de ficheros

- Se puede procesar un fichero texto línea a línea mediante un for-in-loop

```
with open('test.txt') as f: # Auto close the file upon exit
    for line in f:
        line = line.rstrip() # Strip trail spaces and newl
        print(line)

# Same as above
f = open('test.txt', 'r')
for line in f:
    print(line.rstrip())
f.close()
```

Iterando a través de ficheros

- Cada línea incluye un newline

```
>>> f = open('temp.txt', 'w')
>>> f.write('apple\n')
6
>>> f.write('orange\n')
7
>>> f.close()

>>> f = open('temp.txt', 'r')
# line includes a newlin, disable print()'s default newln
>>> for line in f: print(line, end='')
apple
orange
>>> f.close()
```

Control de errores

- Cuando ocurre un error durante la ejecución de un programa se produce una `exception` y el programa aborta. Las excepciones se pueden capturar con instrucciones `try except:`

```
try:
    do something
except error:
    do something else
```

donde *`error`* es el nombre de la excepción interna de Python. Si no hay excepción se ejecuta el bloque `try`, sino se ejecuta el bloque `except`

Control de errores - ejemplo

```
import math

def RaizCuadrada(numero):
    try:
        return math.sqrt(numero)
    except ValueError:
        return 'Error: raíz negativa de ' + str(numero)

print(RaizCuadrada(4))
print(RaizCuadrada(-13))
print(RaizCuadrada(100))
print(RaizCuadrada(-1))
print(RaizCuadrada(25))
```

- Lista de excepciones internas

Assertion and Exception Handling - assert

- Instrucción assert. Se usa para probar una aserción.
  - Sintaxis:

  assert test, error-message

```
>>> x = 0
>>> assert x == 0, 'x is not zero?!'  # Assertion true, no
output

>>> x = 1
# Assertion false, raise AssertionError with the message
>>> assert x == 0, 'x is not zero?!'
......
AssertionError: x is not zero?!
```

Assertion and Exception Handling - Exceptions

- Los errores detectados durante la ejecución se llaman excepciones. Cuando se produce el programa termina abruptamente.

```
>>> 1/0          # Divide by 0
......
ZeroDivisionError: division by zero
>>> zzz          # Variable not defined
......
NameError: name 'zzz' is not defined
>>> '1' + 1      # Cannot concatenate string and int
......
TypeError: Can't convert 'int' object to str implicitly
```

Assertion and Exception Handling - Exceptions

```
>>> lst = [0, 1, 2]
>>> lst[3]         # Index out of range
......
IndexError: list index out of range
>>> lst.index(8)  # Item is not in the list
......
ValueError: 8 is not in list

>>> int('abc')    # Cannot parse this string into int
......
ValueError: invalid literal for int() with base 10: 'abc'

>>> tup = (1, 2, 3)
>>> tup[0] = 11    # Tuple is immutable
......
TypeError: 'tuple' object does not support item assignment
```

Assertion and Exception Handling – try-except-else-finally

- Sintaxis:

```
try:
    statements
except exception-1:                    # Catch one exception
    statements
except (exception-2, exception-3): # Catch multiple except.
    statements
except exception-4 as var_name:  # Retrieve the excep. inst
    statements
except:          # For (other) exceptions
    statements
else:
    statements   # Run if no exception raised
finally:
    statements   # Always run regardless of whether
exception raised
```

Assertion and Exception Handling – try-except-else-finally

- Ejemplo 1: Gestión de índice fuera de rango en acceso a lista: ejem1_excep.py
- Ejemplo2: Validación de entrada.

```
>>> while True:
        try:
            x = int(input('Enter an integer: '))
            break
        except ValueError:
            print('Wrong input! Try again...')    # Repeat

Enter an integer: abc
Wrong input! Try again...
Enter an integer: 11.22
Wrong input! Try again...
Enter an integer: 123
```

Instrucción with-as y gestores de contexto

- La sintaxis de with-as es:

```
with ... as ...:
    statements

# More than one items
with ... as ..., ... as ..., ...:
    statements
```

- Ejemplos:

```
# automatically close the file at the end of with
with open('test.log', 'r') as infile:
    for line in infile:
        print(line)
```

Instrucción with-as y gestores de contexto

- Ejemplos:

```
# automatically close the file at the end of with
with open('test.log', 'r') as infile:
    for line in infile:
        print(line)


# Copy a file
with open('in.txt', 'r') as infile, open('out.txt', 'w') as
outfile:
    for line in infile:
        outfile.write(line)
```

Funciones

- Se definen con la palabra clave `def` seguida por el nombre de la función, la lista de parámetros, las cadenas de documentación y el cuerpo de la función.
- Dentro del cuerpo de la función se puede usar la instrucción `return` para devolver un valor.
- Sintaxis:

```
def function_name(arg1, arg2, ...):
    """Function doc-string"""
    # Can be retrieved via function_name.__doc__
    statements
    return return-value
```

Funciones - terminología

concept	*Python construct*	*description*
function	function	mapping
input value	argument	input to function
output value	return value	output of function
formula	function body	function definition
independent variable	parameter variable	symbolic placeholder for input value

Funciones – control de flujo

- `import`
- `def`
- `return`

```
import sys
import stdio

def harmonic(n):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / i
    return total

for i in range(1,      len(sys.argv)):
    arg = int(sys.argv[i])
    value =    harmonic(arg)
    stdio.writeln(value)
```

Flow of control for `python harmonicf.py 1 2 4`

Funciones – código típico

primality test	def isPrime(n):
	  if n < 2: return False
	  i = 2
	  while i*i <= n:
	  if n % i == 0: return False
	  i += 1
	  return True

hypotenuse of a right triangle	def hypot(a, b) return math.sqrt(a*a + b*b)

generalized harmonic number	def harmonic(n, r=1):
	  total = 0.0
	  for i in range(1, n+1):
	  total += 1.0 / (i ** r)
	  return total

draw a triangle	def drawTriangle(x0, y0, x1, y1, x2, y2):
	  stddraw.line(x0, y0, x1, y1)
	  stddraw.line(x1, y1, x2, y2)
	  stddraw.line(x2, y2, x0, y0)

Funciones - Ejemplos

```
>>> def my_square(x):
        """Return the square of the given number"""
        return x * x

# Invoke the function defined earlier
>>> my_square(8)
64
>>> my_square(1.8)
3.24
>>> my_square('hello')
TypeError: can't multiply sequence by non-int of type
'str'
>>> my_square
<function my_square at 0x7fa57ec54bf8>
>>> type(my_square)
<class 'function'>
```

Funciones - Ejemplos

```
>>> my_square.__doc__   # Show function doc-string
'Return the square of the given number'
>>> help(my_square)     # Show documentaion
my_square(x)
    Return the square of the given number
>>> dir(my_square)      # Show attributes
......
```

Funciones - Ejemplos

```
def fibon(n):
    """Print the first n Fibonacci numbers, where
       f(n)=f(n-1)+f(n-2) and f(1)=f(2)=1
       memoization method
    """
    a, b = 1, 1
    for count in range(n):
        print(a, end=' ')  # print a space
        a, b = b, a+b
    print()   # print a newline

fibon(20)
```

Funciones - Ejemplos

```
def my_cube(x):
    """(number) -> (number)
    Return the cube of the given number.
    Examples (can be used by doctest):
    >>> my_cube(5)
    125
    >>> my_cube(-5)
    -125
    >>> my_cube(0)
    0
    """
    return x*x*x
# Test the function
print(my_cube(8))    # 512
print(my_cube(-8))   # -512
print(my_cube(0))    # 0
```

Funciones - Ejemplos

```
def derivatives(f,x,h=0.0001): # h has a default value
    """function that computes the first two derivatives
     of f (x) by finite differences"""
    df =(f(x+h) - f(x-h))/(2.0*h)
    ddf =(f(x+h) - 2.0*f(x) + f(x-h))/h**2
    return df,ddf
# Test the function
from math import atan
df,ddf = derivatives(atan,0.5) # Uses default value of h
print('First derivative =',df)
print('Second derivative =',ddf)
```

Funciones - Ejemplos

```
def squares(a):
    """a list is passed to a function where it is modified
    """
    for i in range(len(a)):
        a[i] = a[i]**2
# Test the function
a = [1, 2, 3, 4]
squares(a)
print(a) # 'a' now contains 'a**2'
```

Funciones y APIs

- Tipos de funciones: integrada (`int()`, `float()`, `str()`), standard o librería (`math.sqrt()`) requiere importar el módulo donde se encuentra.
- API: application programming interface

function call	*description*
built-in functions	
`abs(x)`	*absolute value of x*
`max(a, b)`	*maximum value of a and b*
`min(a, b)`	*minimum value of a and b*
standard functions from Python's math module	
`math.sin(x)`	*sine of x (expressed in radians)*
`math.cos(x)`	*cosine of x (expressed in radians)*
`math.tan(x)`	*tangent of x (expressed in radians)*
`math.atan2(y, x)`	*polar angle of the point (x, y)*
`math.hypot(x, y)`	*Euclidean distance between the origin and (x, y)*
`math.radians(x)`	*conversion of x (expressed in degrees) to radians*
`math.degrees(x)`	*conversion of x (expressed in radians) to degrees*
`math.exp(x)`	*exponential function of x* (e^x)
`math.log(x, b)`	*base-b logarithm of x* ($\log_b x$) *(the base b defaults to e—the natural logarithm)*
`math.sqrt(x)`	*square root of x*
`math.erf(x)`	*error function of x*
`math.gamma(x)`	*gamma function of x*
Note: The `math` module also includes the inverse functions `asin()`, `acos()`, and `atan()` and the constant variables `e` (`2.718281828459045`) and `pi` (`3.141592653589793`).	
standard functions from Python's `random` module	
`random.random()`	*a random `float` in the interval [0, 1)*
`random.randrange(x, y)`	*a random `int` in [x, y) where x and y are `int`s*

APIs for some commonly used Python functions

Parámetros de funciones

- Los argumentos inmutables (enteros, floats, strings, tuplas) se pasan por *valor*. Es decir, se clona una copia y se pasa a la función, y el original no se puede modificar dentro de la función.
- Los argumentos mutables (listas, diccionarios, sets e instancias de clases) se pasan por *referencia*. Es decir, se pueden modificar dentro de la función.

Parámetros de funciones con valores por defecto

- Se puede asignar un valor por defecto a los parámetros de funciones.

```
>>> def my_sum(n1, n2 = 4, n3 = 5): # n1 required, n2, n3 optional
        """Return the sum of all the arguments"""
        return n1 + n2 + n3

>>> print(my_sum(1, 2, 3))
6
>>> print(my_sum(1, 2))    # n3 defaults
8
>>> print(my_sum(1))       # n2 and n3 default
10
>>> print(my_sum())
TypeError: my_sum() takes at least 1 argument (0 given)
>>> print(my_sum(1, 2, 3, 4))
TypeError: my_sum() takes at most 3 arguments (4 given)
```

Argumentos posicionales y nominales

- Las funciones en Python permiten argumentos posicionales y nombrados.
- Normalmente se pasan los argumentos por posición de izquierda a derecha (posicional).

```
def my_sum(n1, n2 = 4, n3 = 5):
    """Return the sum of all the arguments"""
    return n1 + n2 + n3

print(my_sum(n2 = 2, n1 = 1, n3 = 3))
# Keyword arguments need not follow their positional order
print(my_sum(n2 = 2, n1 = 1))          # n3 defaults
print(my_sum(n1 = 1))                  # n2 and n3 default
print(my_sum(1, n3 = 3))               # n2 default
#print(my_sum(n2 = 2))              # TypeError, n1 missing
```

Número de argumentos posicionales variables

- Python ofrece un número variable (arbitrario) de argumentos. En la definición de función se puede usar * para indicar los restantes argumentos.

```
def my_sum(a, *args): # one posit.arg. & arbit.numb.of args
    """Return the sum of all the arguments (one or more)"""
    sum = a
    for item in args:  # args is a tuple
        sum += item
    return sum

print(my_sum(1))           # args is ()
print(my_sum(1, 2))        # args is (2,)
print(my_sum(1, 2, 3))     # args is (2, 3)
print(my_sum(1, 2, 3, 4))  # args is (2, 3, 4)
```

Número de argumentos posicionales variables

- Python permite poner *args en medio de la lista de parámetros. En ese caso todos los argumentos después de *args deben pasarse por nombre clave.

```
def my_sum(a, *args, b):
    sum = a
    for item in args:
        sum += item
    sum += b
    return sum

print(my_sum(1, 2, 3, 4))
#TypeError: my_sum() missing 1 required keyword-only argument: 'b'
print(my_sum(1, 2, 3, 4, b=5))
```

Número de argumentos posicionales variables

- De forma inversa cuando los argumentos ya están en una lista/tupla, se puede usar * para desempacar la lista/tupla como argumentos posicionales separados.

```
>>> def my_sum(a, b, c): return a+b+c

>>> lst1 = [11, 22, 33]
# my_sum() expects 3 arguments, NOT a 3-item list
>>> my_sum(*lst1) # unpack the list into separate posit. args
66

>>> lst2 = [44, 55]
>>> my_sum(*lst2)
TypeError:my_sum() missing 1 required positional argument: 'c'
```

Argumentos con palabra clave **kwargs

- Para indicar parámetros con palabras claves se puede usar ** para empaquetarlos en un diccionario.

```
def my_print_kwargs(**kwargs):
# Accept variable number of keyword arguments
    """Print all the keyword arguments"""
    for key, value in kwargs.items():  # kwargs is a dicti.
        print('%s: %s' % (key, value))

my_print_kwargs(name='Peter', age=24)

# use ** to unpack a dict.into individual keyword arguments
dict = {'k1':'v1', 'k2':'v2'}
my_print_kwargs(**dict)
# Use ** to unpack dict.into separate keyword args k1=v1, k2=v2
```

Argumentos variables *args y **kwargs

- Se puede usar ambos *args y **kwargs en la definición de una función poniendo *args pimero.

```
def my_print_all_args(*args, **kwargs):
# Place *args before **kwargs
    """Print all positional and keyword arguments"""
    for item in args:  # args is a tuple
        print(item)
    for key, value in kwargs.items(): #kwargs is dictionary
        print('%s: %s' % (key, value))

my_print_all_args('a', 'b', 'c', name='Peter', age=24)
# Place the positional arguments before the keyword
# arguments during invocation
```

Valores retornados por una función

- Se puede retornar valores múltiples desde una función Python. En realidad retorna una tupla.

```
>>> def my_fun():
        return 1, 'a', 'hello'

>>> x, y, z = my_fun()
>>> z
'hello'
>>> my_fun()
(1, 'a', 'hello')
```

Funciones generadoras mediante yield

- Una función generadora que ejecuta `yield`, devuelve el control a quién la llamó, pero la función es pausada y el estado (valor de las variables) es guardado. Esto permite que su ejecución pueda ser reanudada más adelante.
- Los valores generados se pueden obtener iterando con `next`.

Funciones generadoras mediante yield

- En la siguiente función se incrementa una variable n en 1, y después retorna con `yield`. El generador genera tres valores obtenidos usando `next`()

```
def generador():
    n = 1
    yield n
    n += 1
    yield n
    n += 1
    yield n
g = generador()
print(next(g))
print(next(g))
print(next(g))
# Salida: 1\n 2\n 3
```

Variables locales y globales

- Los nombres creados dentro de una función son locales a la función y están disponibles dentro de la función solamente.
- Los nombres creados fuera de las funciones son globales en el módulo y están disponibles dentro de todas las funciones definidas en el módulo.

Variables locales y globales - ejemplo

```
x = 'global'     # x is a global variable for this module

def myfun(arg):  # arg is a local variable for this function
    y = 'local'  # y is also a local variable
    # Function can access both local and global variables
    print(x)
    print(y)
    print(arg)

myfun('abc')
print(x)
#print(y)   # locals are not visible outside the function
#print(arg)
```

- Se puede acceder a variables globales con `global`

Variables función

- A una variable se le puede asignar un valor, una función o un objeto.

```
>>> def square(n): return n * n
>>> square(5)
25
>>> sq = square   # Assign a function to a variable
>>> sq(5)
25
>>> type(square)
<class 'function'>
>>> type(sq)
<class 'function'>
>>> square
<function square at 0x7f0ba7040f28>
>>> sq
<function square at 0x7f0ba7040f28> # same reference square
```

Variables función

- Se puede asignar una invocación específica de una función a una variable.

```
>>> def square(n): return n * n

>>> sq5 = square(5)   # A specific function invocation
>>> sq5
25
>>> type(sq5)
<class 'int'>
```

Funciones anidadas

- Se puede anidar funciones. Definir una función dentro de una función

```
def outer(a):       # Outer function
    print('outer begins with arg =', a)
    x = 1  # Define a local variable
    def inner(b):   # Define an inner function
        print('inner begins with arg = %s' % b)
        y = 2
        print('a = %s, x = %d, y = %d' % (a, x, y))
        print('inner ends')
    # Call inner function defined earlier
    inner('bbb')
    print('outer ends')
# Call outer funct, which in turn calls the inner function
outer('aaa')
```

Las funciones son objetos

- Las funciones son objetos, por tanto:
  - Una función se puede asignar a una variable
  - Una función se puede pasar como argumento en una función
  - Una función puede ser el valor retornado de una función

Paso de una función como argumento de una función

- El nombre de una función es el nombre de una variable que se puede pasar en otra función como argumento.

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

def my_sub(x, y):
    return x - y

def my_apply(func, x, y): # takes a function as first arg
    return func(x, y) # Invoke the function received

print(my_apply(my_add, 3, 2))  # Output: 5
print(my_apply(my_sub, 3, 2))  # Output: 1

# We can also pass an anonymous function as argument
print(my_apply(lambda x, y: x * y, 3, 2))  # Output: 6
```

Funciones como objetos

- En Python cada elemento es un objeto, incluyendo funciones.

```
# Fichero integ.py
# Calcula la integral de Riemann de una function f
def integrate(f, a, b, n=1000):
    total = 0.0
    dt = 1.0 * (b - a) / n
    for i in range(n):
        total += dt * f(a + (i + 0.5) * dt)
    return total
```

Funciones como objetos

```
# Fichero intdrive.py
import integ as fa
def square(x):
    return x*x

def main():
    print(fa.integrate(square,0, 10)

if __name__ == '__main__':
    main()
```

Función lambda

- Las funciones lambda son funciones que tienen la forma de una expresión. Se consideran anónimas o funciones sin nombre. Se usan para definir una función inline. La sintaxis es:

```
func_name = lambda arg1, arg2, ...: return-expression
```

```
>>> f2 = lambda a, b, c: a + b + c # Define a Lambda funct
>>> f2(1, 2, 3)  # Invoke function
6
>>> type(f2)
<class 'function'>
>>> c = lambda x,y : x**2 + y**2
>>> print(c(3,4))
25
```

Funciones - recursión

- Técnica de programación utilizada en muchas aplicaciones. Capacidad de invocarse una función a sí misma.

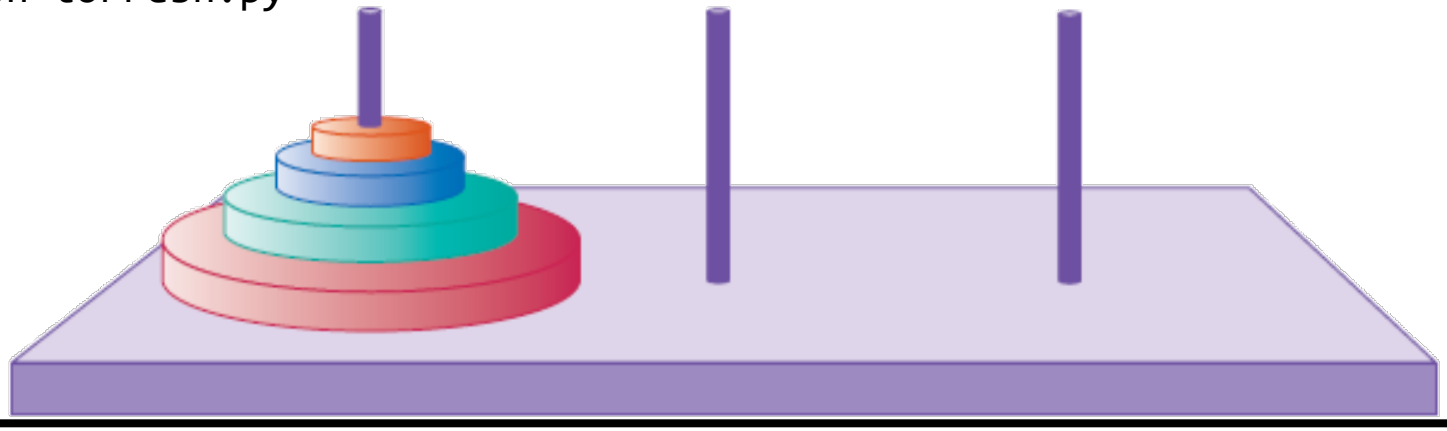
```
import sys
# Return n!
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)
def main():
    n = int(sys.argv[1])
    fact = factorial(n)
    print(fact)
if __name__ == '__main__':
    main()
# python factorial.py 3
# 6
```

Funciones - recursión

```
# Imprime los movimientos para resolver las torres de hanoi
# parametros: numero discos, torre partida, torre final, torre auxiliar
def mover(discos, detorre, atorre, auxtorre) :
  if discos >= 1 :
     mover(discos - 1, detorre, auxtorre, atorre)
     print("Mover disco ", discos, " de ", detorre, " a ", atorre)
     mover(discos - 1, auxtorre, atorre, detorre)

def main() :
   mover(4, "A", "C", "B")

if __name__ == '__main__':
    main()
# python torresh.py
```



Módulos

- Un módulo Python es un fichero que contiene código Python, incluyendo instrucciones, variables, funciones y clases.
- Debe guardarse con la extensión .py
- El nombre del módulo es el nombre del fichero:

<nombre_modulo>.py

- Típicamente un módulo comienza con una cadena de documentación (triple comilla) que se invoca con <nombre_modulo>.__doc__

Módulos

- Un *módulo* contiene funciones que están disponibles para su uso en otros programas.
- Un *cliente* es un programa que hace uso de una función en un módulo.
- Pasos:
  - En el cliente: `import` módulo.
  - En el cliente: hacer llamada a la función.
  - En el módulo: colocar una prueba de cliente main().
  - En el módulo: eliminar código global. Usar

    ```
    if __name__ == '__main__': main()
    ```
  - Accesibilidad del módulo por el cliente (mismo directorio)

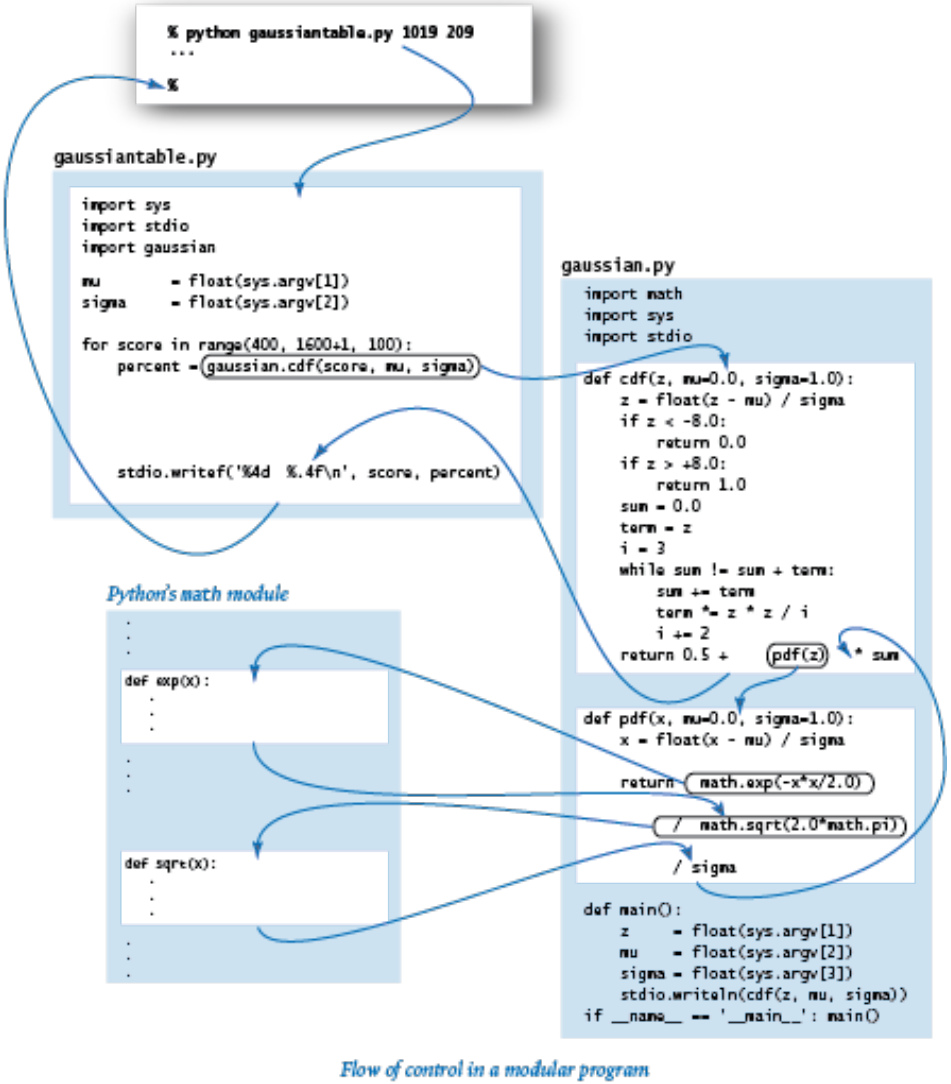
__name__

- Cuando un módulo o paquete de Python es importado, __name__ es asignado al nombre del módulo. Normalmente, este es el nombre del archivo de Python sin la extensión .py
- Si el archivo es parte de un paquete, __name__ también incluirá la ruta del paquete padre.

'__main__'

- Si el módulo es ejecutado en el entorno de *código de máximo nivel*, su __name__ se le asigna el valor del string '__main__'
- *Código de máximo nivel* es el primer módulo de Python especificado por el usuario que empieza a ejecutarse. Es de máximo nivel porque importa todos los demás módulos que necesita el programa.

Módulos



Flow of control in a modular program

Programación modular

- Implementaciones.
- Clientes.
- Application programming interfaces (APIs).

function call	*description*
`gaussian.pdf(x, mu, sigma)`	*Gaussian probability density function* $\phi(x, \mu, \sigma)$
`gaussian.cdf(z, mu, sigma)`	*Gaussian cumulative distribution function* $\Phi(x, \mu, \sigma)$

Note: The default value for `mu` *is* `0.0` *and for* `sigma` *is* `1.0`.

API for our `gaussian` *module*

- Funciones privadas:
  - Funciones que solo se usan en los módulos y que no se ofrecen a los clientes. Por convención se usa un guión bajo como primer caracter del nombre de la función.

Programación modular

- Librerías:
  - Colección de módulos relacionados. Ejemplo: NumPy, Matplolib, SciPy, SymPy, Ipython, Pygame.
- Documentación.

```
>>> import nombre_modulo
>>> help (nombre_modulo)
```

Instrucción import

- Para usar un módulo en un programa se utiliza la instrucción `import`
- Una vez importado, se referencia los atributos del módulo como <nombre_modulo>.<nombre_atributo>
- Se usa import-as para asignar un nuevo nombre al módulo para evitar conflicto de nombres en el módulo
- Se puede agrupar en el siguiente orden:
  - Librería standard
  - Librerías de terceros
  - Librerías de aplicación local

Ejemplo módulo e import

- Ejemplo: fichero greet.py

```
"""
greet
-----
This module contains the greeting message 'msg' and
greeting function 'greet()'.
"""

msg = 'Hello'       # Global Variable

def greet(name):    # Function
    print('{}, {}'.format(msg, name))
```

Ejemplo módulo e import

```
>>> import greet
>>> greet.greet('Peter')  # <module_name>.<function_name>
Hello, Peter
>>> print(greet.msg)      # <module_name>.<var_name>
Hello

>>> greet.__doc__         # module's doc-string
"\ngreet\n-----\nThis module contains the greeting message
'msg' and greeting function 'greet()'.\n"

>>> greet.__name__        # module's name
'greet'

>>> dir(greet)  # List all attributes defined in the module
['__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__',
'greet', 'msg']
```

Ejemplo módulo e import

```
>>> help(greet)  # Show module's name, functions, data, ...
Help on module greet:
NAME
    greet
DESCRIPTION
 greet
    -----
    This module contains the greeting message 'msg' and greeting
function 'greet()'.
FUNCTIONS
    greet(name)
DATA
    msg = 'Hello'
FILE
    /path/to/greet.py

>>> import greet as gr  # Refer. the 'greet' module as 'gr'
>>> gr.greet('Paul')
Hello, Paul
```

Instrucción from - import

- La sintaxis es:

```
from <module_name> import <attr_name>  # import one attribute
from <module_name> import <attr_name_1>, <attr_name_2>, ...
# import selected attributes
from <module_name> import *  #import ALL attributes (NOT recomm.)
from <module_name> import <attr_name> as <name>
# import attribute as the given name
```

- Con from – import se referencia los atributos importados usando <attr_name> directamente.

```
>>> from greet import greet, msg as message
>>> greet('Peter')  # Reference without the 'module_name'
Hello, Peter
>>> message
'Hello'
>>> msg
NameError: name 'msg' is not defined
```

Variable de entorno sys.path y PYTHONPATH

- El camino de búsqueda de módulos es mantenida por la variable Python path del módulo sys, sys.path
- sys.path es inicializada a partir de la variable de entorno PYTHONPATH. Por defecto incluye el directorio de trabajo en curso.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python3.5', '/usr/local/lib/python3.5/dist-packages',
'/usr/lib/python3.5/dist-packages', ...]
```

Módulos de librería standard Python de uso común

- Python dispone de un conjunto de librerías standard.
- Para usarlas se usa 'import <nombre_modulo>' para importar la librería completa o 'from <nombre_modulo> import <nombre_atributo>' para importar el atributo seleccionado.
- También se puede poner un alias

  'import <nombre_modulo> as <alias>'

Módulos de librería standard Python de uso común

```
>>> import math   # import an external module
>>> dir(math)     # List all attributes
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>> help(math)
Help on built-in module math:

NAME
    math
......
```

Módulos de librería standard Python de uso común

```
>>> help(math.atan2)
Help on built-in function atan2 in module math:
atan2(y, x, /)
......
>>> math.atan2(3, 0)
1.5707963267948966
>>> math.sin(math.pi / 2)
1.0
>>> math.cos(math.pi / 2)
6.123233995736766e-17

>>> from math import pi
>>> pi
3.141592653589793
>>> import math as m
>>> print(m.log(m.sin(0.5)))
-0.735166686385
```

Módulo math

- El módulo math proporciona acceso las funciones definidas por el lenguaje C. Los más comunes son:
  - Constantes: pi, e, tau, inf, nan.
  - Potencia y exponente: pow(x,y), sqrt(x), exp(x), log(x), log2(x), log10(x)
  - Conversión float a int: ceil(x), floor(x), trunc(x)
  - Operaciones float: fabs(), fmod()
  - hypot(x,y) (=sqrt(x*x + y*y))
  - Conversión entre grados y radianes: degrees(x), radians(x)
  - Funciones trigonométricas: sin(x), cos(x), tan(x), acos(x), asin(x), atan(x), atan2(x,y)
  - Funciones hiperbólicas: sinh(x), cosh(x), tanh(x), asinh(x), acosh(x), atanh(x)

Módulo math

- Lista de funciones math:

```
['__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'copysign', 'cos', 'cosh', 'degrees',
'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2',
'modf', 'nan', 'pi', 'pow', 'radians',
'remainder', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'tau', 'trunc']
```

Módulo cmath

- El módulo cmath proporciona acceso a las funciones matemáticas para números complejos:

```
['__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atanh', 'cos',
'cosh', 'e', 'exp', 'inf', 'infj', 'isclose',
'isfinite', 'isinf', 'isnan', 'log', 'log10',
'nan', 'nanj', 'phase', 'pi', 'polar', 'rect',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau']
```

Módulo cmath - ejemplo

```
>>> from cmath import sin
>>> x = 3.0 -4.5j
>>> y = 1.2 + 0.8j
>>> z = 0.8
>>> print(x/y)
(-2.56205313375e-016-3.75j)
>>> print(sin(x))
(6.35239299817+44.5526433649j)
>>> print(sin(z))
(0.7173560909+0j)
```

Módulo statistics

- El módulo statistics calcula las propiedades estadísticas básicas.

```
>>> import statistics
>>> dir(statistics)
['mean', 'median', 'median_grouped', 'median_high',
'median_low', 'mode', 'pstdev', 'pvariance', 'stdev',
'variance', ...]
>>> help(statistics)
......
>>> help(statistics.pstdev)
......

>>> data = [5, 7, 8, 3, 5, 6, 1, 3]
>>> statistics.mean(data)
4.75
```

Módulo statistics

```
>>> statistics.median(data)
5.0
>>> statistics.stdev(data)
2.3145502494313788
>>> statistics.variance(data)
5.357142857142857
>>> statistics.mode(data)
statistics.StatisticsError: no unique mode; found 2 equally
common values
```

Módulo random

- El módulo random se usa para generar números pseudo random.

```
>>> import random
>>> dir(random)
......
>>> help(random)
......
>>> help(random.random)
......

>>> random.random()        # float in [0,1)
0.7259532743815786
>>> random.random()
0.9282534690123855
```

Módulo random

- El módulo random se usa para generar números pseudo random.

```
>>> random.randint(1, 6)  # int in [1,6]
3
>>> random.randrange(6)   # From range(6), i.e., 0 to 5
0
>>> random.choice(['apple', 'orange', 'banana'])
'apple'
```

Módulo sys

- El módulo sys (de system) proporciona parámetros y funciones específicos de sistema. Los más usados:
  - sys.exit([exit-status=0]): salir del programa.
  - sys.path: Lista de rutas de búsqueda. Initializado con la variable de entorno PYTHONPATH.
  - sys.stdin, sys.stdout, sys.stderr: entrada, salida y error estándard.
  - sys.argv: Lista de argumentos en la línea de comandos

Módulo sys

- Script test_argv.py

```
import sys
print(sys.argv)        # Print command-line argument list
print(len(sys.argv))   # Print length of list
```

- Ejecución del script

```
$ python test_argv.py
['test_argv.py']
1

$ python test_argv.py hello 1 2 3 apple orange
['test_argv.py', 'hello', '1', '2', '3', 'apple', 'orange']
# list of strings
7
```

Módulo os

- El módulo os proporciona una interfaz con el sistema operativo. Los atributos más usados son:
  - `os.mkdir(path, mode=0777)`: Crea un directorio
  - `os.mkdirs(path, mode=0777])`: Similar a mkdir
  - `os.getcwd()`: devuelve el directorio en curso
  - `os.chdir(path)`: Cambia el directorio en curso
  - `os.system(command)`: ejecuta un commando shell.
  - `os.getenv(varname, value=None)`: devuelve la variable de entorno si existe
  - `os.putenv(varname, value)`: asigna la variable de entorno al valor
  - `os.unsetenv(varname)`: elimina la variable de entorno

Módulo os

- Ejemplo:

```
>>> import os
>>> dir(os)          # List all attributes
......
>>> help(os)         # Show man page
......
>>> help(os.getcwd)  # Show man page for specific function
......

>>> os.getcwd()           # Get current working directory
...current working directory...
>>> os.listdir('.') # List contents of the current direct
...contents of current directory...
>>> os.chdir('test-python')       # Change directory
>>> exec(open('hello.py').read()) # Run a Python script
>>> os.system('ls -l')            # Run shell command
```

Módulo os

- Ejemplo:

```
>>> import os
>>> os.name                              # Name of OS
'posix'
>>> os.makedirs(dir)                     # Create sub-directory
>>> os.remove(file)                      # Remove file
>>> os.rename(oldFile, newFile)          # Rename file
>>> os.listdir('.')
    # Return a list of entries in the given directory
>>> for f in sorted(os.listdir('.')):
        print(f)
```

Módulo date

- Proporciona clases para la manipulación de fechas y tiempos

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', 'date', 'datetime', 'datetime_CAPI',
'time', 'timedelta', 'timezone', 'tzinfo', ...]
>>> dir(datetime.date)
['today', ...]

>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2016, 6, 17)
```

Módulo date

- Proporciona clases para la manipulación de fechas y tiempos

```
>>> import datetime
>>> aday = date(2016, 5, 1)  # Construct a datetime.date i
>>> aday
datetime.date(2016, 5, 1)
>>> diff = today - aday    # Find the diff between 2 dates
>>> diff
datetime.timedelta(47)
>>> dir(datetime.timedelta)
['days', 'max', 'microseconds', 'min', 'resolution',
'seconds', 'total_seconds', ...]
>>> diff.days
47
```

Módulo time

- Se puede usar para medir el tiempo de ejecución de un script

```
>>> import time
>>> dir(time)
['_STRUCT_TM_ITEMS','altzone','asctime','clock',...,
'sleep',..., 'time','time_ns','timezone','tzname']
```

- La función `time()` devuelve el número de segundos transcurridos desde epoch (en sistemas Unix: January 1, 1970, 00:00:00)

```
import time
start = time.time()

"codigo que se desea medir el tiempo aqui"

end = time.time()
print(end - start)
```

Packages

- Un módulo contiene atributos (variables, funciones y clases). Los módulos relevantes (mantenidos en el mismo directorio) se pueden agrupar en un package.
- Python también soporta sub-packages (en sub-directorios).
- Los packages y sub-packages son una forma de organizar el espacio de nombres en la forma:

<pack_name>.<sub_pack_name>.<sub_sub_pack_name>.<module_name>.<attr_name>

Plantilla de módulo individual

```
"""
<package_name>.<module_name>
---------------------------
A description to explain functionality of this module.
Class/Function however should not be documented here.
:author: <author-name>
:version: x.y.z (verion.release.modification)
:copyright: ......
:license: ......
"""
import <standard_library_modules>
import <third_party_library_modules>
import <application_modules>
# Define global variables
......
# Define helper functions
......
# Define the entry 'main' function
def main():
    """The main function doc-string"""
    .......
# Run the main function
if __name__ == '__main__':
    main()
```

Packages

- Para crear un package:
  - Crear un directorio y nombrarlo con el nombre del package
  - Poner los módulos en el directorio
  - Crear un fichero '__init__.py' en el directorio para marcar el directorio como un package
- Módulos en Python Tutorial

Ejemplo package

```
myapp/                    # This directory is in the 'sys.path'
    |
    + mypack1/            # A directory of relevant modules
    |     |
    |     + __init__.py   # Mark as a package called 'mypack1'
    |     + mymod1_1.py   # Reference as 'mypack1.mymod1_1'
    |     + mymod1_2.py   # Reference as 'mypack1.mymod1_2'
    |
    + mypack2/            # A directory of relevant modules
          |
          + __init__.py   # Mark as a package called 'mypack2'
          + mymod2_1.py   # Reference as 'mypack2.mymod2_1'
          + mymod2_2.py   # Reference as 'mypack2.mymod2_2'
```

Ejemplo package

- Si 'myapp' está en 'sys.path' se puede importar 'mymod1_1' como:

```
import mypack1.mymod1_1
# Reference 'attr1_1_1' as 'mypack1.mymod1_1.attr1_1_1'
from mypack1 import mymod1_1
# Reference 'attr1_1_1' as 'mymod1_1.attr1_1_1'
```

Nombres, Espacio de nombres (Namespace) y ámbito

- Un nombre se aplica a casi todo incluyendo una variable, función, clase/instancia, módulo/package
- Los nombre definidos dentro de una función son locales a ella. Los nombres definidos fuera de todas las funciones son globales al módulo y son accesibles por todas las funciones dentro del módulo.
- Un espacio de nombres (namespace) es una colección de nombres.
- El ámbito se refiere a la porción del programa a partir de la cual un nombre se puede acceder sin prefijo.

Cada módulo tiene un Espacio de nombres Global

- Un módulo es un fichero que contiene atributos (variables, funciones y clases) y tiene su propio espacio de nombres globales.
  - Por ello no se puede definir dos funciones o clases con el mismo nombre dentro de un módulo, pero sí en diferentes módulos.
- Cuando se ejecuta el Shell interactivo, Python crea un módulo llamado __main__, con su namespace global asociado.

Cada módulo tiene un Espacio de nombres Global

- Cuando se importa un módulo con 'import <module_name>':
  - En caso de Shell interactivo, se añade <module_name> al namespace de __main__
  - Dentro de otro módulo se añade el nombre al namespace del módulo donde se ha importado.
- Si se importa un atributo con 'from <module_name> import <attr_name>' el <attr_name> se añade al namespace de __main__, y se puede acceder al <attr_name> directamente.

Funciones globals(), locals() y dir()

- Se puede listar los nombres del ámbito en curso con las funciones integradas:
  - globals(): devuelve un diccionario con las variables globales en curso
  - locals(): devuelve un diccionario con las variables locales.
  - dir(): devuelve una lista de los nombres locales, que es equivalente a locals().keys()

Modificación de variables globales dentro de una función

- Para modificar una variable global dentro de una función se usa la instrucción global.

```
x = 'global'       # Global file-scope

def myfun():
    global x   # Declare x global to modify global variable
    x = 'change'
    print(x)

myfun()
print(x)           # Global changes
```

Package NumPy

- NumPy (Numeric Python) es un package externo de Python que proporciona estructuras de datos potentes, tales como los *objetos arrays*, y funciones matemáticas de ejecución muy rápida.
- Se puede instalar desde una ventana de comandos:

```
> pip install numpy
```

- En el terminal de Anaconda:

```
> conda install -c anaconda numpy
```

- Portal tutorial de NumPy
- Lista de rutinas incluídas en NumPy

NumPy – creación de un array

- Los arrays se pueden crear de varias formas. Una de ellas es usar la función array para convertir una lista en un array:

`array(`*`list`*`,`*`type`*`)`

- Ejemplo:

```
>>> from numpy import array
>>> a = array([[2.0, -1.0],[-1.0, 3.0]])
>>> print(a)
[[ 2. -1.]
[-1. 3.]]
>>> b = array([[2, -1],[-1, 3]],float)
>>> print(b)
[[ 2. -1.]
[-1. 3.]]
```

NumPy – creación de un array

- Otras funciones para crear arrays:

```
zeros((dim1,dim2),type)
ones((dim1,dim2),type)
arange(from,to,increment)
```

- Ejemplos:

```
>>> from numpy import *
>>> print(arange(2,10,2))
[2 4 6 8]
>>> print(arange(2.0,10.0,2.0))
[ 2. 4. 6. 8.]
>>> print(zeros(3))
[ 0. 0. 0.]
>>> print(zeros((3),int))
[0 0 0]
>>> print(ones((2,2)))
[[ 1. 1.]
[ 1. 1.]]
```

NumPy – acceso y cambio de elementos array

- Para acceder a los elementos de un array se usan corchetes: `a[i] a[i,j]`
- Ejemplo:

```
>>> from numpy import *
>>> a = zeros((3,3),int)
>>> print(a)
[[0 0 0]
[0 0 0]
[0 0 0]]
>>> a[0] = [2,3,2]     # Change a row
>>> a[1,1] = 5         # Change an element
>>> a[2,0:2] = [8,-3] # Change part of a row
>>> print(a)
[[ 2 3 2]
[ 0 5 0]
[ 8 -3 0]]
```

NumPy – operaciones en arrays

- Los operadores aritméticos se extienden a todos los elementos del array (operaciones vectorizadas).
- Ejemplos:

```
>>> from numpy import array
>>> a = array([0.0, 4.0, 9.0, 16.0])
>>> print(a/16.0)
[ 0. 0.25 0.5625 1. ]
>>> print(a - 4.0)
[ -4. 0. 5. 12.]
>>> from numpy import array,sqrt,sin
>>> a = array([1.0, 4.0, 9.0, 16.0])
>>> print(sqrt(a))
[ 1. 2. 3. 4.]
>>> print(sin(a))
[ 0.84147098 -0.7568025 0.41211849 -0.28790332]
```

NumPy – funciones sobre arrays

- Hay numerosas funciones en numpy para operaciones sobre arrays y otras tareas útiles.
- Ejemplos:

```
>>> from numpy import *
>>> A = array([[4,-2,1],[-2,4,-2],[1,-2,3]],float)
>>> b = array([1,4,3],float)
>>> print(diagonal(A))      # Principal diagonal
[ 4. 4. 3.]
>>> print(diagonal(A,1))    # First subdiagonal
[-2. -2.]
>>> print(trace(A))         # Sum of diagonal elements
11.0
>>> print(argmax(b))        # Index of largest element
1
>>> print(argmin(A,axis=0)) # Indices of smallest col. elem
[1 0 1]
```

NumPy – funciones sobre arrays

- Ejemplos (continuación):

```
>>> print(identity(3)) # Identity matrix
[[ 1. 0. 0.]
[ 0. 1. 0.]
[ 0. 0. 1.]]
```

NumPy – funciones sobre arrays

- Ejemplos de productos con array:

```
from numpy import *
x = array([7,3])
y = array([2,1])
A = array([[1,2],[3,2]])
B = array([[1,1],[2,2]])
# Dot product
print("dot(x,y) =\n",dot(x,y)) # {x}.{y}
print("dot(A,x) =\n",dot(A,x)) # [A]{x}
print("dot(A,B) =\n",dot(A,B)) # [A][B]
# Inner product
print("inner(x,y) =\n",inner(x,y)) # {x}.{y}
print("inner(A,x) =\n",inner(A,x)) # [A]{x}
print("inner(A,B) =\n",inner(A,B)) # [A][B_transpose]
# Outer product
print("outer(x,y) =\n",outer(x,y))
print("outer(A,x) =\n",outer(A,x))
print("outer(A,B) =\n",outer(A,B))
```

NumPy – módulo de álgebra lineal

- numpy incluye un módulo de álgebra lineal llamado lingalg que contiene funciones para invertir matrices y para resolver ecuaciones simultáneas. Ejemplo:

```
>>> from numpy import array
>>> from numpy.linalg import inv,solve
>>> A = array([[ 4.0, -2.0, 1.0], \
               [-2.0, 4.0, -2.0], \
               [ 1.0, -2.0, 3.0]])
>>> b = array([1.0, 4.0, 2.0])
>>> print(inv(A)) # Matrix inverse
[[ 0.33333333 0.16666667 0. ]
[ 0.16666667 0.45833333 0.25 ]
[ 0. 0.25 0.5 ]]
>>> print(solve(A,b)) # Solve [A]{x} = {b}
[ 1. , 2.5, 2. ]
```

NumPy – copia de arrays

- Si `a` es un objeto mutable, como una lista, la instrucción `b = a` no resulta en un nuevo objeto `b`, se crea una nueva referencia al array `a`. Para conseguir una copia independiente del array `a` se usa el método `copy` del módulo `numpy`

```
b = a.copy()
```

NumPy – vectorizando algoritmos

- Las propiedades de propagación de las funciones matemáticas en el módulo `numpy` se puede usar para reemplazar ciclos en el código, conocido como vectorización. Por ejemplo, para evaluar:

$$s = \sum_{i=0}^{100} \sqrt{\frac{i\pi}{100}} \sin \frac{i\pi}{100}$$

- Código directo:

```
from math import sqrt,sin,pi
x = 0.0; s = 0.0
for i in range(101):
    s = s + sqrt(x)*sin(x)
    x = x + 0.01*pi
print(s)
```

Código vectorizado:

```
from numpy import sqrt,sin,arange
from math import pi
x = arange(0.0, 1.001*pi, 0.01*pi)
print(sum(sqrt(x)*sin(x)))
```

Lectura de matriz con NumPy

```
import numpy as np
def lee_matriz():
#lectura de elementos float
    print('Lectura Matriz')   #Datos de la dimensión de la matriz
    m = int(input('Numero de filas '))
    n = int(input('Numero de columnas '))
    M = np.zeros([m, n])  #Creacion matriz de ceros en invocacion
    for i in range(m):
        for j in range(n):
            M[i][j] = float(input(\
                    'Ingresa elemento ({0},{1}): '.format(i,j)))
    return M

M = lee_matriz1()
imprime_matriz(M)
```

Escritura de matriz con NumPy

```
import numpy as np
def imprime_matriz(M):
    ''' Funcion para imprimir matrices '''
    print('Matriz')
    m = M.shape[0]
    n = M.shape[1]
    for i in range(m):
        for j in range(n):
            print(M[i][j],end='\t')
        print('')
```

Visualización con Matplotlib

- Matplotlib es una librería para crear visualizaciones estáticas, animadas e interactivas en Python
- Tutorial
- Ejemplos
- El módulo matplotlib.pyplot es una colección de funciones gráficas similar a Matlab. Requiere instalarse con

```
> pip install matplotlib
```

o

```
> conda install matplotlib
```

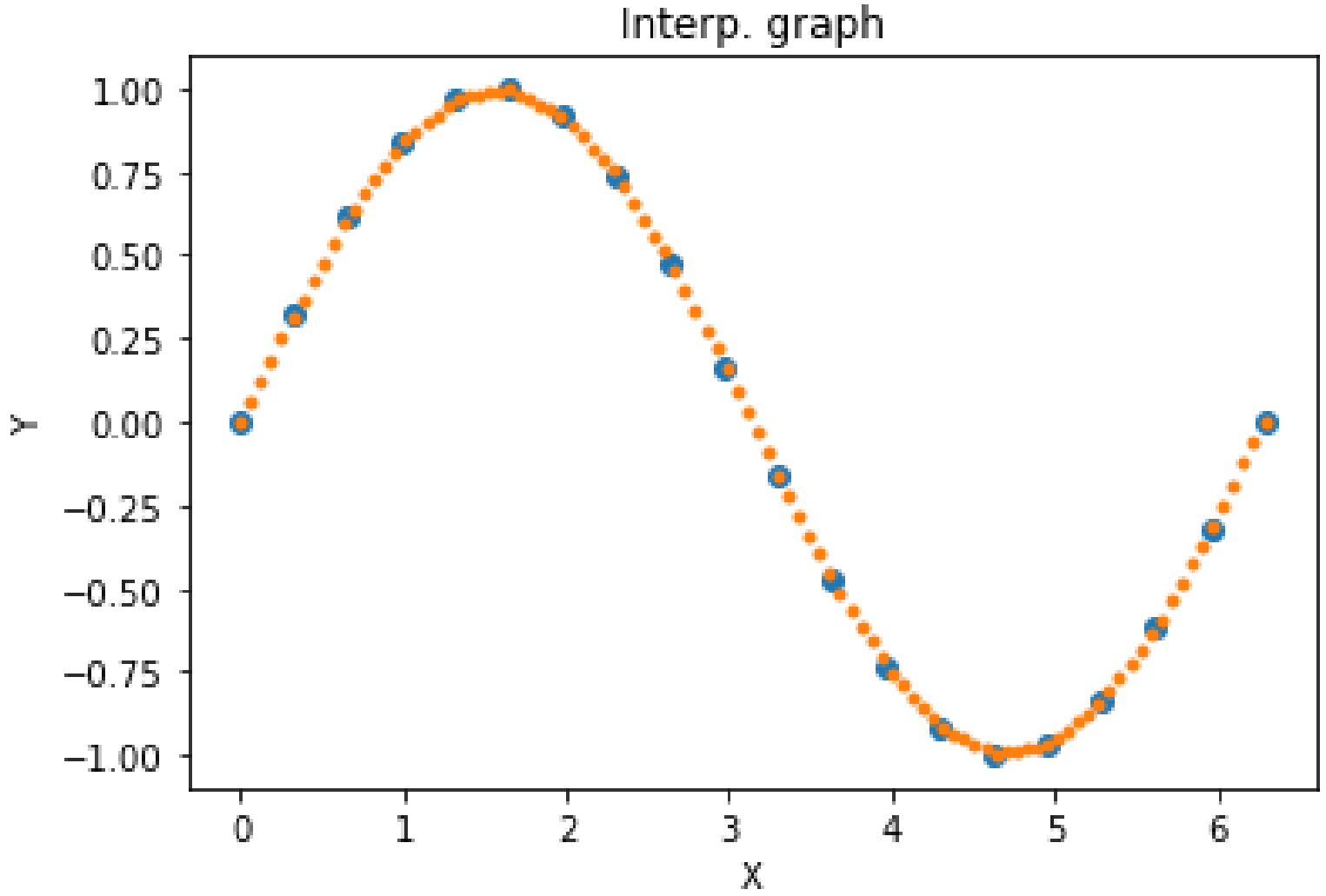
Ejemplo Matplotlib

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 20)
y = np.sin(x)
yp = None
xi = np.linspace(x[0], x[-1], 100)
yi = np.interp(xi, x, y, yp)

fig, ax = plt.subplots()
ax.plot(x, y, 'o', xi, yi, '.')
ax.set(xlabel='X', ylabel='Y', title='Interp. graph')
plt.show()
```

Ejemplo Matplotlib



Gráficas con `matplotlib.pyplot`

- Algunos estilos de líneas y marcadores son:

`'-'`	Solid line
`'--'`	Dashed line
`'-.'`	Dash-dot line
`':'`	Dotted line
`'o'`	Circle marker
`'^'`	Triangle marker
`'s'`	Square marker
`'h'`	Hexagon marker
`'x'`	x marker

- Los códigos para localizar la leyenda son:

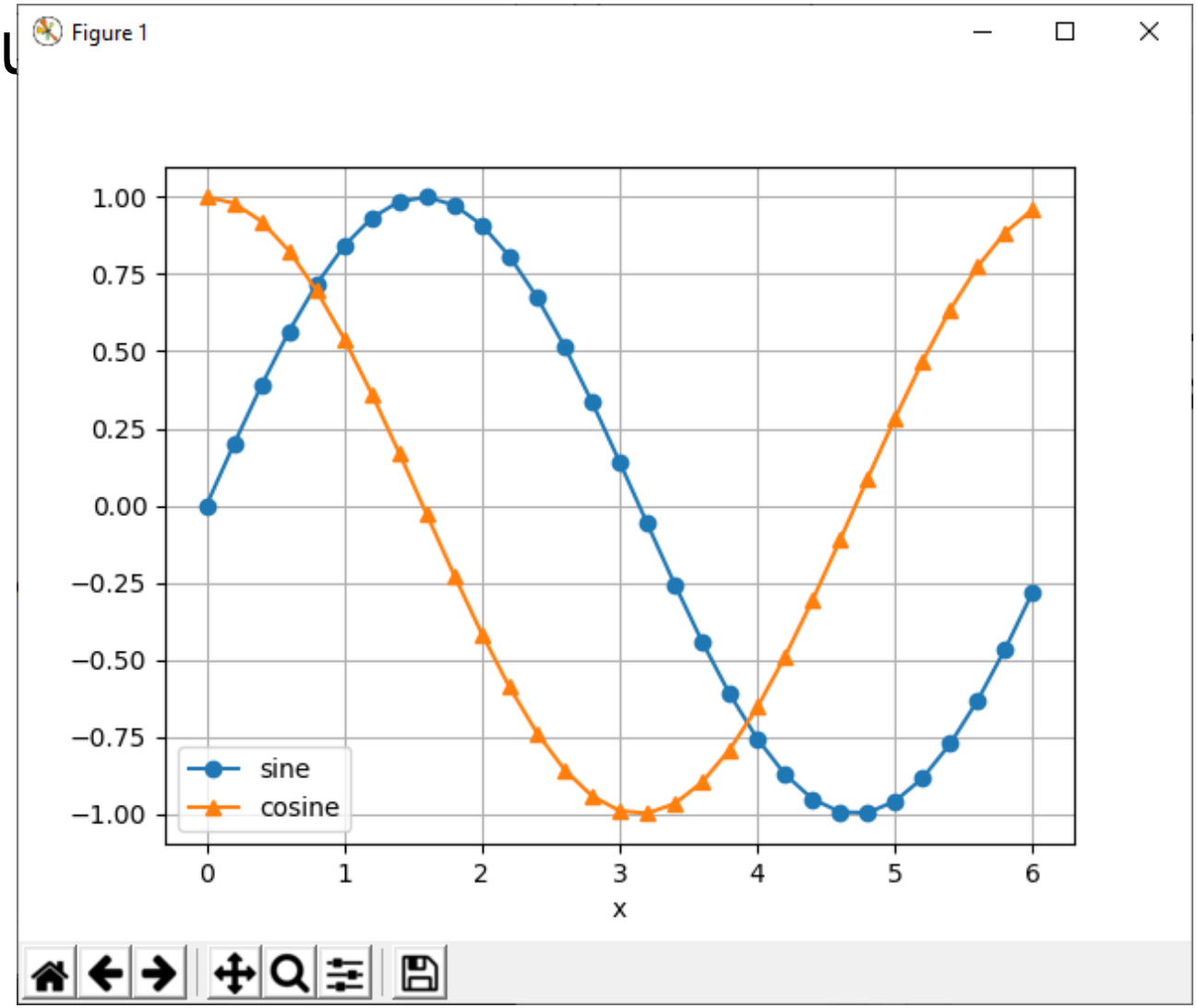
0	"Best" location
1	Upper right
2	Upper left
3	Lower left
4	Lower right

Gráficas con `matplotlib.pyplot`

```
import matplotlib.pyplot as plt
from numpy import arange,sin,cos
x = arange(0.0,6.2,0.2)
plt.plot(x,sin(x),'o-',x,cos(x),'^-')    # Plot with specified
                                         # line and marker style
plt.xlabel('x')                          # Add label to x-axis
plt.legend(('sine','cosine'),loc = 0)    # Add legend in loc. 3
plt.grid(True)                           # Add coordinate grid
plt.savefig('testplot.png',format='png') # Save plot in png
                                         # format for future use
plt.show()                               # Show plot on screen
input("\nPress return to exit")
```

Gráficas con `matplotlib.pyplot`

- Resu



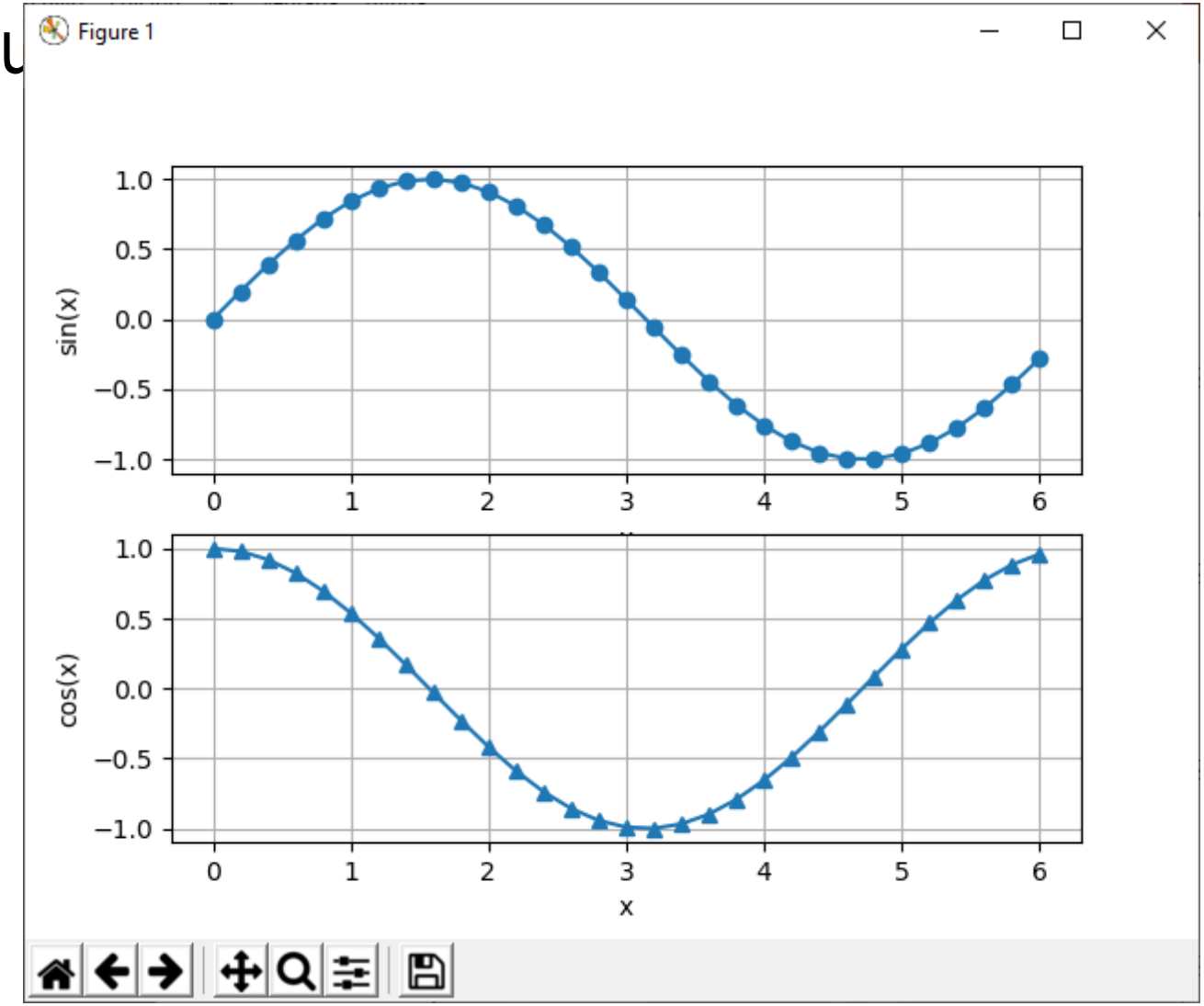
Gráficas con `matplotlib.pyplot`

- Es posible hacer varias gráficas en una figura con la instrucción subplot(*rows,cols,plot number*)

```
import matplotlib.pyplot as plt
from numpy import arange,sin,cos
x = arange(0.0,6.2,0.2)
plt.subplot(2,1,1)
plt.plot(x,sin(x),'o-')
plt.xlabel('x');plt.ylabel('sin(x)')
plt.grid(True)
plt.subplot(2,1,2)
plt.plot(x,cos(x),'^-')
plt.xlabel('x');plt.ylabel('cos(x)')
plt.grid(True)
plt.show()
input("\nPress return to exit")
```

Gráficas con `matplotlib.pyplot`

- Resu



Scipy

- Librería de funciones de algoritmos fundamentales para la computación científica. Incluye funciones matemáticas para cálculo numérico, tales como integración y optimización
- Portal Tutorial
- Ejemplo: integrar método CG, $I = \int_0^{4.5} J_{2.5}(x)dx$

```
import scipy.integrate as integrate
import scipy.special as special
result = integrate.quad(lambda x: special.jv(2.5,x),0,4.5)
print(result)
```

Sympy

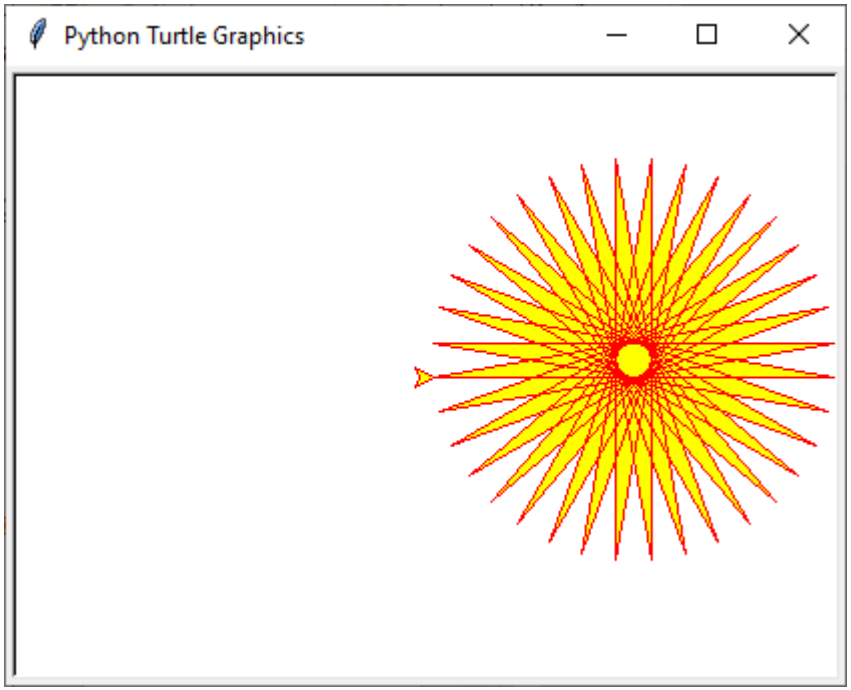
- Sympy es una librería Python que permite hacer matemáticas simbólica
- Portal Sympy
- SymPy Tutorial
- Ejemplo: calcular la integral $I = \int \cos(x) * \exp(x)\, dx$

```
from sympy import *
x = symbols('x')
a = Integral(cos(x)*exp(x), x)
print(Eq(a, a.doit()))
```

Módulo turtle

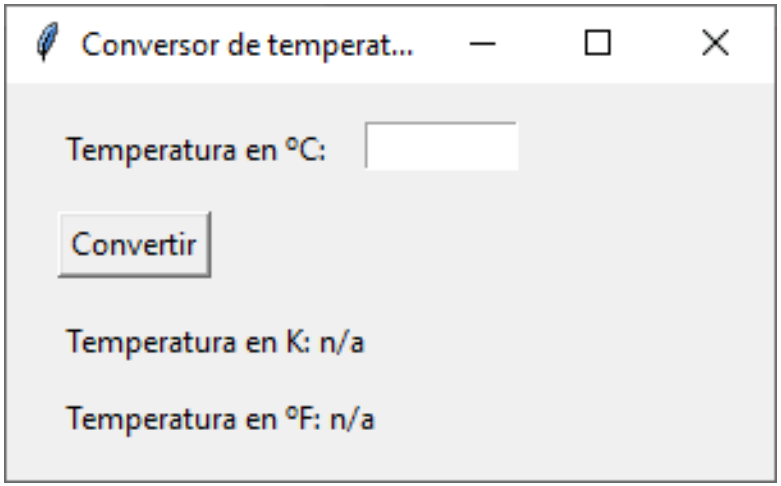
- El módulo turtle es un módulo interno de Python que proporciona primitivas gráficas tipo tortuga. Se usa para introducir la programación a niños.
- Turtle graphics
- Ejemplo:

```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```



Package tkinter

- El package tkinter sirve para el desarrollo de interfaces de usuario (GUI). Ofrece una serie de objetos gráficos (widgets)
- Documentación Tutorial
- Ejemplo:

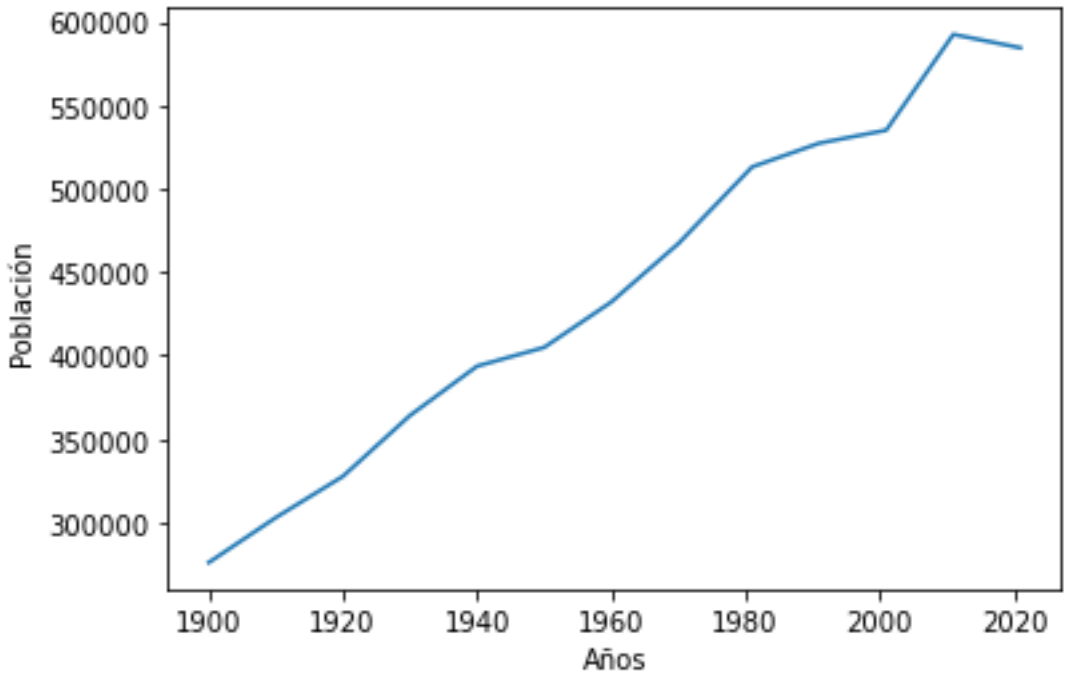


Ejemplo tkinter

```
import tkinter as tk
def convertir_temp():
    temp_celsius = float(caja_temp_celsius.get())
    temp_kelvin = temp_celsius + 273.15
    temp_fahrenheit = temp_celsius*1.8 + 32
    etiqueta_temp_kelvin.config(text=f"Temperatura en K: {temp_kelvin}")
    etiqueta_temp_fahrenheit.config(
        text=f"Temperatura en ºF: {temp_fahrenheit}")
ventana = tk.Tk()
ventana.title("Conversor de temperatura")
ventana.config(width=300, height=155)
etiqueta_temp_celsius = tk.Label(text="Temperatura en ºC:")
etiqueta_temp_celsius.place(x=20, y=15)
caja_temp_celsius = tk.Entry()
caja_temp_celsius.place(x=140, y=15, width=60)
boton_convertir = tk.Button(text="Convertir", command=convertir_temp)
boton_convertir.place(x=20, y=50)
etiqueta_temp_kelvin = tk.Label(text="Temperatura en K: n/a")
etiqueta_temp_kelvin.place(x=20, y=90)
etiqueta_temp_fahrenheit = tk.Label(text="Temperatura en ºF: n/a")
etiqueta_temp_fahrenheit.place(x=20, y=120)
ventana.mainloop()
```

Package pandas

- El package pandas es una herramiento potente para el análisis y manipulación de datos
- Documentación Tutorial
- Ejemplo:



Package pandas

- Ejemplo:

```
import pandas as pd
import matplotlib.pyplot as plt
url = 'https://en.wikipedia.org/wiki/Cantabria'
dfs = pd.read_html(url, match='Historical population')
year = dfs[0]['Year']
population = dfs[0]['Pop.']
df1 = pd.DataFrame({'Year': year, 'Population': population})
df1.drop(df1.tail(1).index,inplace = True)
year = df1['Year'].to_numpy().astype(float)
population = df1['Population'].to_numpy().astype(float)
plt.plot(year,population)
plt.xlabel("Años")
plt.ylabel("Población")
```