Introducción a Python

ESTEBAN VÁZQUEZ

Objetivos

- En este módulo el alumno trabajará con los principales elementos del lenguaje Python para el tratamiento de datos en entornos de Data Engineering modernos.
- En estos entornos predomina la creación de Data Pipelines (logística de datos) usando el lenguaje Python, con lo cual el módulo incluirá también conceptos relacionados a esta temática.

Temario

1. Conceptos básicos de programación en Python
2. Bucles y estructuras de decisión en Python
3. Funciones y estructuras de datos en Python
4. Uso de módulos y paquetes en Python
5. Programación orientada a objetos en Python
6. Manipulación de ficheros en Python.
7. Estructura datos complejos.

Conceptos básicos de programación en Python

Tema 1

¿ Qué es un LP ?

Definición: Un lenguaje de programación es un sistema notacional para describir computaciones de una forma legible tanto para la máquina como para el ser humano.

Computación

    Máquina Turing, tesis de Church

Legibilidad por parte de la máquina

Legibilidad por parte del ser humano

Características

Eficiencia
Expresividad
Capacidad de mantenimiento
Legibilidad
Confiabilidad
Seguridad
Simplicidad
Productividad

Abstracciones

Clases: Datos y control
Niveles: básicas, estructuradas y unitarias

Abstracciones de datos:

- Básicas: tipos básicos (enteros, reales, ...)
- Estructuradas: tipos estructurados (arreglos, registros)
- Unitarias: Tipos abstractos de datos (TDAs), paquetes, módulos, clases, componentes

Abstracciones

<u>Abstracciones de control</u>

Básicas: asignación, goto

Estructuradas: condicionales e iteradores

Unitarias: paquetes, módulos, hilos y tareas.

Un lenguaje de programación es completo en Turing siempre que tenga variables enteras y aritméticas, y que ejecute enunciados en forma secuencial, incluyendo enunciados de asignación, selección e iteración.

Paradigmas de programación

Imperativo

modelo de Von Neuman, cuello de botella de Von Neuman

Orientado a Objetos

TDAs, encapsulación, modularidad, reutilización

Funcional

noción abstracta de función, cálculo lambda, recursividad, listas

Lógico

Lógica simbólica, programación declarativa

Definición del lenguaje

Sintaxis (estructura)
- Gramáticas libres de contexto, estructura léxica, tokens

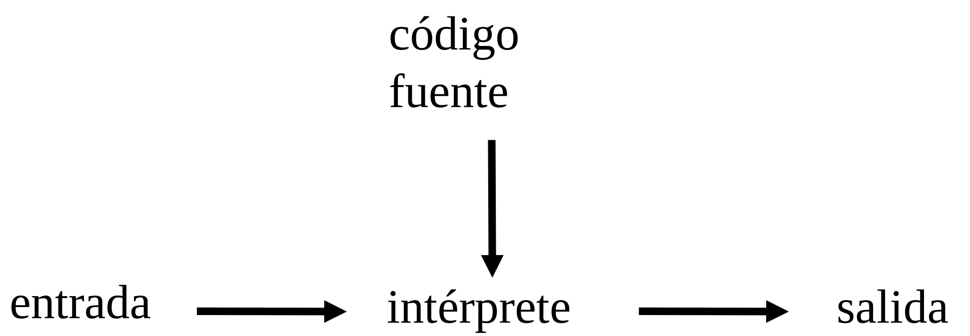
Semántica (significado)
- Lenguaje natural
- Semántica operacional
- Semántica denotacional

Traducción del lenguaje

Traductor es un programa que acepta otros programas escritos en un lenguaje y:

los ejecuta directamente (interprete)

los transforma en una forma adecuada para su ejecución (compilador).



Traducción

código fuente → compilación → código objeto → traducción adicional → código ejecutable

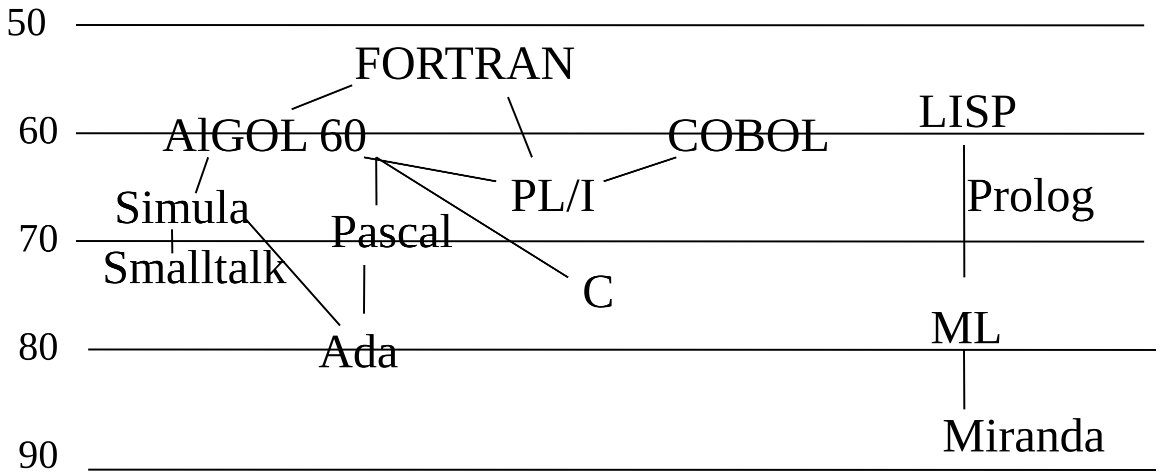
Pseudointérpretes: intermedio entre interprete y compilador: lenguajes intermedios

Operaciones de un traductor: analizador léxico (tokens), analizador sintáctico, analizador semántico, preprocesador

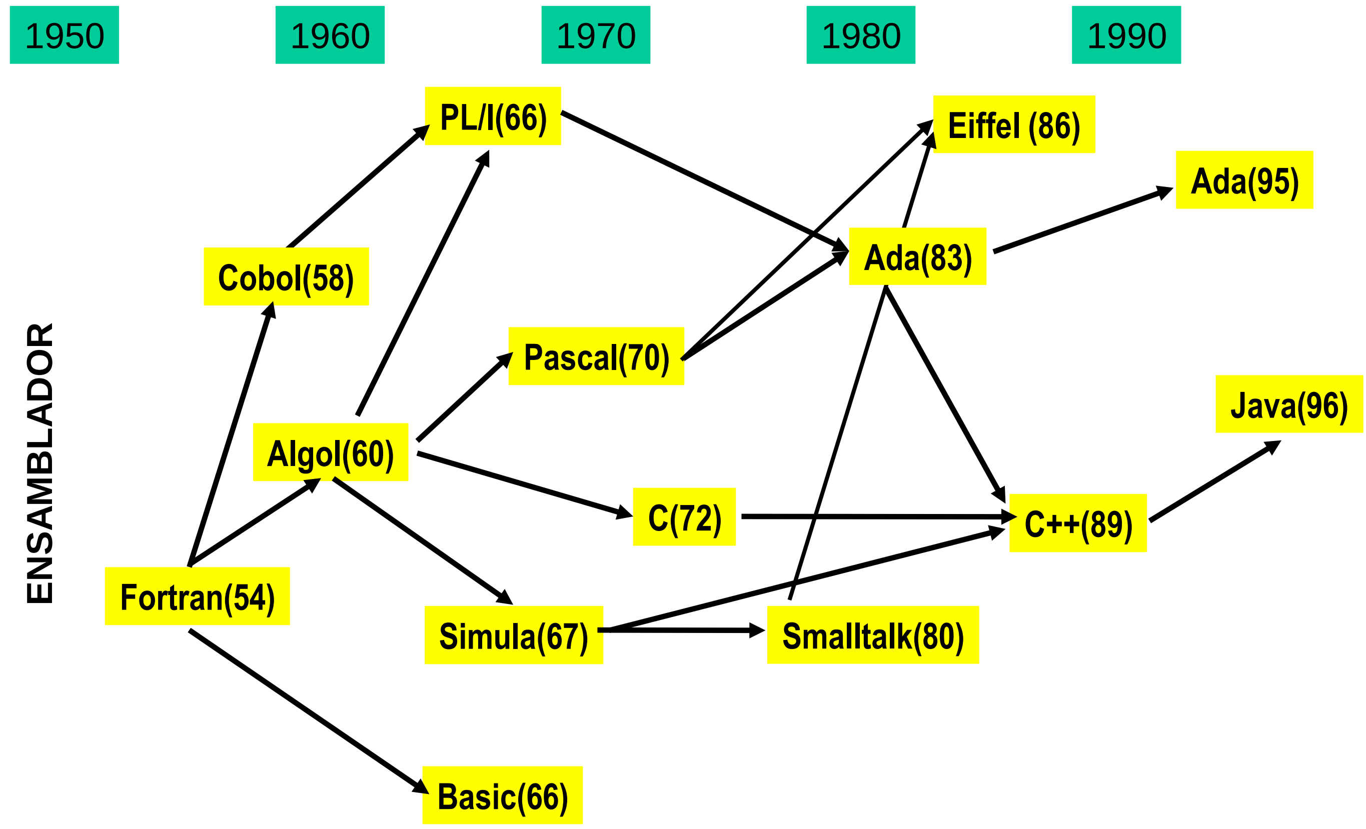
Traducción

Tiempo de compilación y tiempo de ejecución
Propiedades estáticas: tiempo de compilación
Propiedades dinámicas: tiempo de ejecución
Recuperación de errores (compilación y ejecución)
Eficiencia y optimización (compilación o ejecución)

Historia


50
FORTRAN
60
AlGOL 60
COBOL
LISP
PL/I
Simula
Pascal
Prolog
70
Smalltalk
C
80
Ada
ML
90
Miranda

Historia



1950
1960
1970
1980
1990
ENSAMBLADOR
PL/I(66)
Eiffel (86)
Ada(95)
Cobol(58)
Ada(83)
Pascal(70)
Java(96)
Algol(60)
C(72)
C++(89)
Fortran(54)
Simula(67)
Smalltalk(80)
Basic(66)

Preguntas

Clasifica los siguientes lenguajes (C, Ada, C++, Java, LISP, Prolog, Visual Basic, JavaScript, C#, PHP) en base a :

- Paradigma
- tipos de traductor (compilador, interprete, pseudointérprete)
- Propiedades estáticas y dinámicas
- Eficiencia

Extraed conclusiones de esta clasificación

¿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), and Tkinter.
  - Se recomienda incluir python en la variable de entorno PATH
- Sitio oficial de documentación

Instalación de librerías científicas en Python

- Los módulos se instalan con el comando pip

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

Table 1-1. List of Fields of Study and Corresponding Python Modules

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

Distribuciones alternativas de Python

- Existen distribuciones alternativas de Python:
  - IronPython (Python running on .NET)
  - Jython (Python running on the Java Virtual Machine)
  - PyPy (A fast python implementation with a JIT compiler)
  - Stackless Python (Branch of CPython with microthreads)
  - MicroPython (Python running on micro controllers)
  - IPython (provides a rich architecture for interactive computing)

Implementaciones alternativas de Python

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

Tipos de datos

- 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.

Definiciones

- Objetos. Todos los datos en un programa Python se representan por objetos. Cada objeto se caracteriza por su identidad, tipo y valor.
- Referencias a objetos.
- Literales.
- Operadores.
- Identificadores.
- Variables.
- Expresiones.

Palabras reservadas

- Las palabras reservadas no se pueden usar como identificadores.

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

Integers - Enteros

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

values	integers						
typical literals	1234 99 0 1000000						
operations	sign	add	subtract	multiply	floored divide	remainder	power
operators	+ -	+	-	*	//	%	**

Python's int data type

  - 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

- Ejemplo

```
>>> 123 + 456 - 789
-210
>>> 123456789012345678901234567890 + 1
123456789012345678901234567891
>>> 1234567890123456789012345678901234567890 + 1
1234567890123456789012345678901234567891
>>> 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
```

Floating-Point Numbers - Reales

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

values	*real numbers*				
typical literals	3.14159	6.022e23	2.0	1.4142135623730951	
operations	*addition*	*subtraction*	*multiplication*	*division*	*exponentiation*
operators	+	-	*	/	**

Python's `float` *data type*

  – Para obtener el máximo valor entero usar `sys.float_info.max`
  – Tienen una representación IEEE de 64 bits. Típicamente tienen 15-17 dígitos decimales de precisión

Floating-Point Numbers - Reales

- Ejemplo

```
>>> 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 + b j

```
>>> 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

- Ejemplo

```
>>> 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
```

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 y operador asignación

- Python es 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

- 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

```
>>> 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 2 < 13 13 > 2	3 != 2 2 <= 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): …	

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*

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()`
  - 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
151

# 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. The rstrip() strips the right (trailing) whitespaces; while lstrip() strips the left (leading) 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 string x to an integer or conversion of float x to an integer by truncation towards zero*
`float(x)`	*conversion of string or integer x to a 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 caracter?

Python no tiene un tipo de dato dedicado a caracteres. Un caracter 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 potente integrada (lista - list) para arrays dinámicos.
- Una lista es encerrada entre corchetes [ ].
- Puede contener elementos de diferentes tipos.
- Puede crecer y encogerse dinámicamente.
- Los elementos se acceden mediante índice, **empezando por cero**.
- Hay funciones integradas (ej. `len()`, `max()`, `min()`, `sum()`), y operadores.

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[:].

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'] = 'pcmq@sant.com'     # Add new item
>>> dct
{'name': 'Peter', 'age': 22, 'email': 'pcmq@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 = `{`k_1`: `v_1`, `k_2`: `v_2`, ..., `k_n`: `v_n`}`	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 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

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
booksite functions for standard output from our `stdio` *module*	
`stdio.write(x)`	*write* x *to standard output*
`stdio.writeln(x)`	*write* x *to standard output, followed by a newline*
Note 1: Any type of data can be used (and will be automatically converted to `str`*).*	
Note 2: If no argument is specified, x *defaults to the empty string.*	
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

Bucles y estructuras de decisión en Python

Tema 2

Condicionales – if - else

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

```
if test-1:
    block-1
elif test-2:
    block-2
......
elif test-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'
```

Operadores de comparación y lógicos

- Python dispone de operadores de comparación que devuelven un valor booleano True o False:
  - < , <= , == , != , > , >=
  - in, not in: Comprueba si un elemento está|no está en una secuencia (lista, tupla, etc).
  - is, is not: Comprueba si dos variables tienen la misma referencia
- Python dispone de tres operadores lógicos (Boolean):
  - and
  - or
  - not

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
```

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
```

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 stdio
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')]
```

Instrucción break

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

```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

Instrucción continue

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

```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

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.

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)
```

Funciones y estructuras de datos en Python

Tema 3

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 - Ejemplos

```
>>> def my_square(x):
        """Returnthe  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 thegivennumber'
>>> 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"""
    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 thegiven  number.
    Examples (can beused  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
```

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 loas 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 dict.
        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 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))
```

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)
1 a
2 b
3 c
>>>enumerate(['a', 'b', 'c'])
<enumerate object at 0x7ff0c6b75a50>

#Tuple
>>>for i, v in enumerate(('d', 'e', 'f')): print(i, v)
1 d
2 e
3 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)]
```

Uso de módulos y paquetes en Python

Tema 4

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__

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
'greet.py: the greet module with attributes msg and greet()'
>>> 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
    ...doc-string...
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', ...]
```

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>
-----------------------------------------------
Adescription 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

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'
```

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)
```

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 witharg  =', a)
    x = 1  # Definealocal   variable
    def inner(b):    # Definean  inner function
        print('inner begins witharg  =%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')
```

Función lambda

- Las funciones lambda son funcione anónimas o funciones sin nombre. Se usan para definir una función inline. La sintaxis es:

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

```
>>> def f1(a, b, c): return a + b + c # ordinary function
>>> f1(1, 2, 3)
6
>>> type(f1)
<class 'function'>
>>> f2 = lambda a, b, c: a + b + c # Define a Lambda funct
>>> f2(1, 2, 3)  # Invoke function
6
>>> type(f2)
<class 'function'>
```

Las funciones son objetos

- Las funciones son objetos, por tanto:
  - Una función se puede asignar a una variable
  - Una función puede ser pasada en una función como argumento
  - 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
```

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.

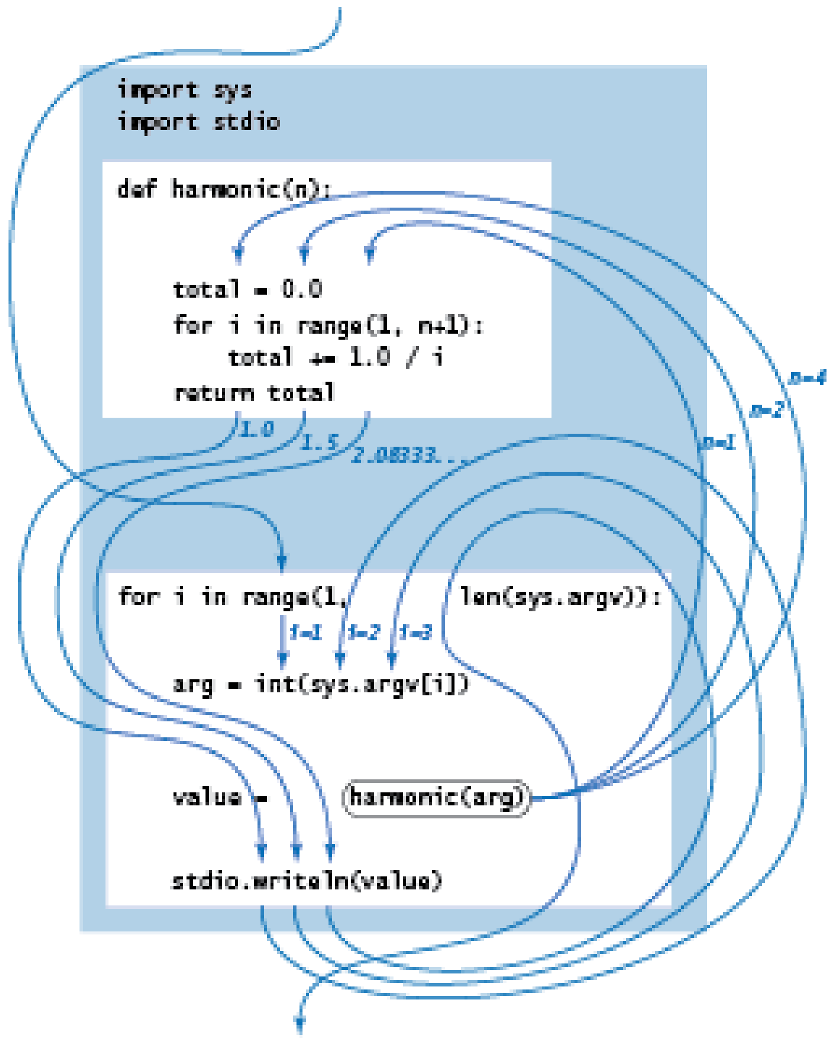
```
x = 'global'          # Global file-scope

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

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

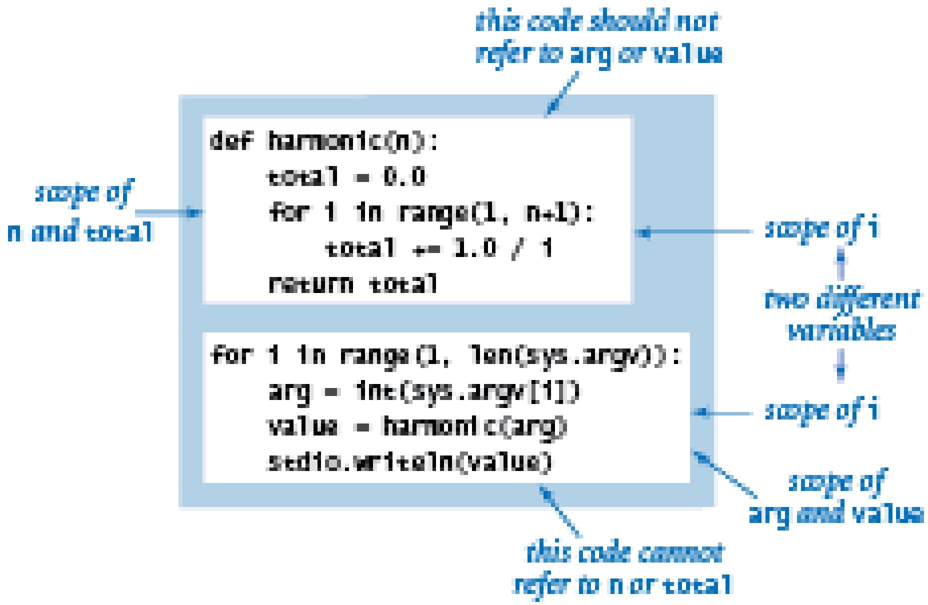
- import
- def
- return



Flow of control for python harmonicf.py 1 2 4

Funciones – alcance

- Las variables son locales en el bloque donde se definen



Scope of local and parameter variables

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)`

Typical code for implementing functions

Funciones - Paso de argumentos

- Los argumentos de tipo integer, float, boolean, o string por valor. El resto de objetos se pasan por referencia.

Funciones – código típico con arrays

mean of an array	`def mean(a):`
	`    total = 0.0`
	`    for v in a:`
	`        total += v`
	`    return total / len(a)`

dot product of two vectors of the same length	`def dot(a, b):`
	`    total = 0`
	`    for i in range(len(a)):`
	`        total += a[i] * b[i]`
	`    return total`

exchange two elements in an array	`def exchange(a, i, j):`
	`    temp = a[i]`
	`    a[i] = a[j]`
	`    a[j] = temp`

write a one-dimensional array (and its length)	`def write1D(a):`
	`    stdio.writeln(len(a))`
	`    for v in a:`
	`        stdio.writeln(v)`

read a two-dimensional array of floats (with dimensions)	`def readFloat2D():`
	`    m = stdio.readInt()`
	`    n = stdio.readInt()`
	`    a = stdarray.create2D(m, n, 0.0)`
	`    for i in range(m):`
	`        for j in range(n):`
	`            a[i][j] = stdio.readFloat()`
	`    return a`

Typical code for implementing functions with arrays

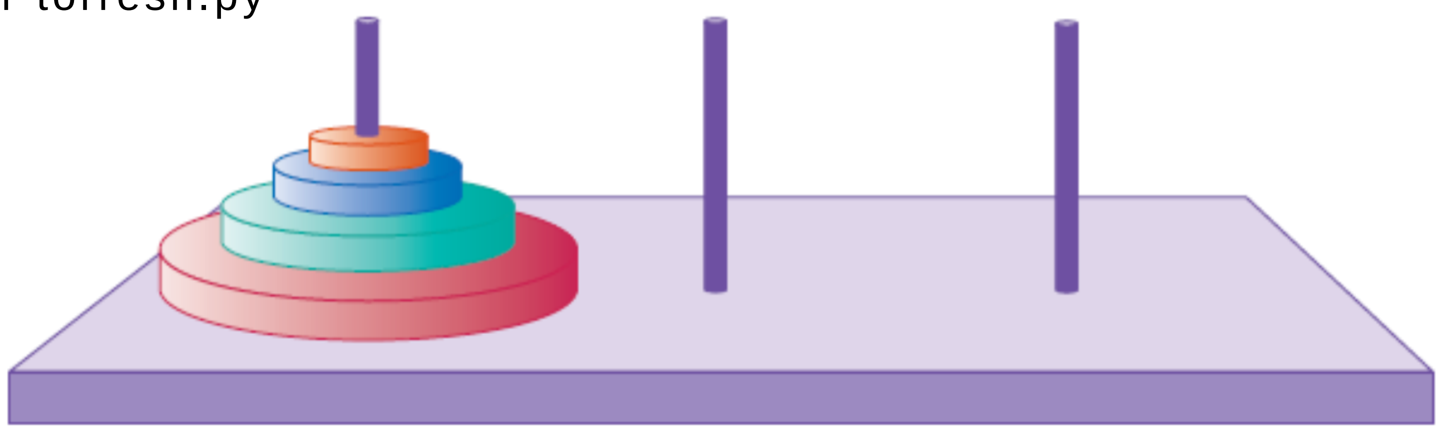
Funciones - recursión

- Técnica de programación utilizada en muchas aplicaciones. Capacidad de invocar una función desde la misma función.

```
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(5, "A", "C", "B")

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



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
```

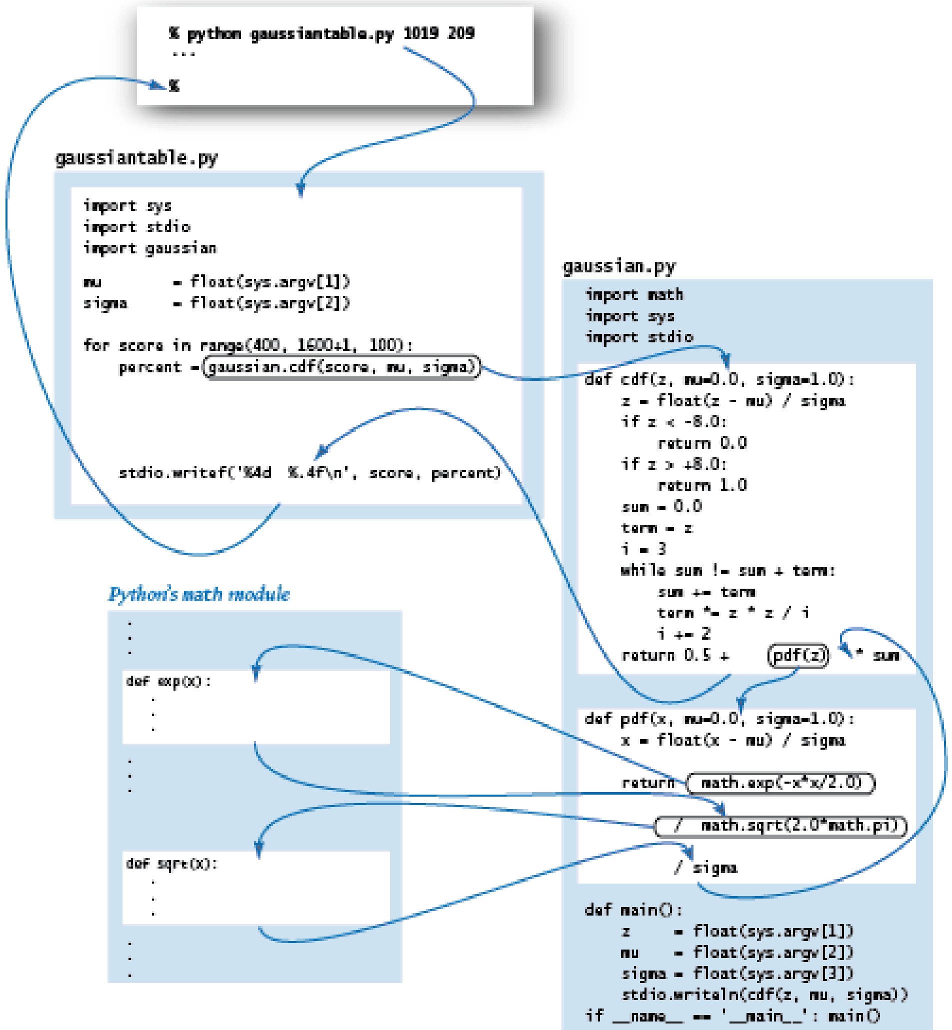
Funciones como objetos

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

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

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

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 el 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()
  - Hacer accesible el módulo para el cliente.

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, Pygame, Matplolib, SciPy, SymPy, Ipython.
- Documentación.

```
>>> import stddraw
>>> help stddraw
```

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 fileat 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)
```

Módulos de librería standard Python de uso común

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

```
>>> import math      # import an external module
>>> dir(math)        # List all attributes
['e', 'pi', 'sin', 'cos', 'tan', 'tan2', ....]
>>> help(math)
......
>>> help(math.atan2)
......
```

Módulos de librería standard Python de uso común

```
>>> 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
```

Módulos math y cmath

- El módulo math proporciona acceso las funciones definidas por el lenguaje C. Los más comunes son:
  - Constantes: pi, e.
  - 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 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
start = time.time()

"codigo que se desea medir el tiempo aqui"

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

Sympy

- Sympy es una librería que permite hacer operaciones simbólicas en lugar de con valores numéricos
- Portal Sympy
- SymPy Tutorial

Scipy

- Librería de funciones matemáticas para cálculo numérico tales como integración y optimización
- Portal
- Tutorial

Programación orientada a objetos en Python

Tema 5

Programación orientada a objetos (OOP) en Python

- Una clase es una plantilla de entidades del mismo tipo. Una instancia es una realización particular de una clase. Python soporta instancias de clases y objetos.
- Un objeto contiene atributos: atributos de datos (o variables) y comportamiento (llamados métodos). Para acceder un atributo se usa el operador punto, ejemplo: nombre_instancia.nombre_atributo
- Para crear una instancia de una clase se invoca el constructor:nombre_instancia = nombre_clase(*args)

Objetos de clase vs Objetos de instancia

- Los objetos de clase sirven como factorías para generar objetos de instancia. Los objetos instanciados son objetos reales creados por una aplicación. Tienen su propio espacio de nombres.
- La instrucción class crea un objeto de clase con el nombre de clase. Dentro de la definición de la clase se puede crear variables de clase y métodos mediante defs que serán compartidos por todas las instancias

Sintaxis de la definición de clase

- La sintaxis es:

```
class class_name(superclass1, ...):
    """Class doc-string"""
    class_var1 = value1  # Class variables
    ......
    def __init__(self, arg1, ...):
        """Constructor"""
        self.instance_var1 = arg1  # inst var by assignment
        ......
    def __str__(self):
        """For printf() and str()"""
        ......
    def __repr__(self):
        """For repr() and interactive prompt"""
        ......
    def method_name(self, *args, **kwargs):
        """Method doc-string"""
        ......
```

Contructor: Self

- El primer parámetro de un constructor debe ser self
- Cuando se invoca el constructor para crear un nuevo objeto, el parámetro self se asigna al objeto que esta siendo inicializado

```
def __init__(self):
    self._itemCount = 0
    self._totalPrice = 0
```

Referencia al objeto inicializado

```
register = CashRegister()
```

Cuando el contructor termina this es la referencia al objeto creado

Ejemplo

- circle.py:

```
from math import pi

class Circle:
"""A Circle instance models a circle with a radius"""
    def __init__(self, radius=1.0):
        """Constructor with default radius of 1.0"""
        self.radius = radius # Create an inst var radius
    def __str__(self):
        """Return string, invoked by print() and str()"""
        return 'circle with radius of %.2f' % self.radius
    def __repr__(self):
        """Return string used to re-create this instance"""
        return 'Circle(radius=%f)' % self.radius
    def get_area(self):
        """Return the area of this Circle instance"""
        return self.radius * self.radius * pi
```

Ejemplo

- circle.py (cont.):

```
# if run under Python interpreter, __name__ is '__main__'. # If imported into another module, __name__ is 'circle'
if __name__ == '__main__':
    c1 = Circle(2.1)             # Construct an instance
    print(c1)                    # Invoke __str__()
    print(c1.get_area())
    c2 = Circle()                # Default radius
    print(c2)
    print(c2.get_area())         # Invoke member method
    c2.color = 'red'   # Create new attribute via assignment
    print(c2.color)
    #print(c1.color)   # Error - c1 has no attribute color
    # Test doc-strings
    print(__doc__)                         # This module
    print(Circle.__doc__)                  # Circle class
    print(Circle.get_area.__doc__)         # get_area() method
    print(isinstance(c1, Circle)) # True
```

Construcción de clase

- Para construir una instancia de una instancia se realiza a través del constructor nombre_clase(…)

```
c1 = Circle(1.2)
c2 = Circle()       # radius default
```

- Python crea un objeto Circle, luego invoca el método __init__(self, radius) con self asignado a la nueva instancia
  - __init__() es un inicializador para crear variables de instancia
  - __init()__ nunca devuelve un valor
  - __init()__ es opcional y se puede omitir si no hay variables de instancia

Clase Point y sobrecarga de operadores

- point.py modela un punto 2D con coordenadas x e y. Se sobrecarga los operadores + y *:

```
""" point.py: point module defines the Point class"""
class Point:
    """A Point  models a 2D point x and y coordinates"""
    def __init__(self, x=0,y=0):
        """Constructor x and y with default of (0,0)"""
        self.x = x
        self.y = y
    def __str__(self):
        """Return a descriptive string for this instance"""
        return '(%.2f, %.2f)' %(self.x, self.y)
    def __add__(self, right):
        """Override the '+' operator"""
        p = Point(self.x + right.x, self.y + right.y)
        return p
```

Clase Point y sobrecarga de operadores

```
    def __mul__(self, factor):
        """Override the '*' operator"""
        self.x *= factor
        self.y *= factor
        return self
# Test
if __name__ == '__main
    _':   p1 = Point()
    print(p1)        # (0.00, 0.00)
    p1.x = 5
    p1.y = 6
    print(p1)        # (5.00, 6.00)
    p2 = Point(3, 4)
    print(p2)        # (3.00, 4.00)
    print(p1 + p2)   # (8.00, 10.00) Same as p1.  add  (p2)
    print(p1)        # (5.00, 6.00) No change
    print(p2 * 3)    # (9.00, 12.00) Same as p1.__mul__(p2)
    print(p2)        # (9.00, 12.00) Changed
```

Herencia

- cylinder.py un cilindro se puede derivar de un circulo

```
"""cylinder.py: which defines the Cylinder class"""
from circle import Circle # Using the Circle class
class Cylinder(Circle):
    """The Cylinder class is a subclass of Circle"""
    def __init__(self, radius = 1.0, height = 1.0):
        """Constructor"""
        super().__init__(radius)  # Invoke superclass
        self.height = height
    def __str__(self):
        """Self Description for print()"""
        return 'Cylinder(radius=%.2f,height=%.2f)'
              %(self.radius,
              self.height)
    def get_volume(self):
        """Return the volume of the cylinder"""
        return self.get_area() * self.height
```

Herencia

- cylinder.py (cont.)

```
if __name__ == '__main__':
    cy1 = Cylinder(1.1, 2.2)
    print(cy1)                    # inherited superclass' method
    print(cy1.get_area())         # Invoke its method
    print(cy1.get_volume())       # Default radius and height
    cy2 = Cylinder()
    print(cy2)
    print(cy2.get_area())
    print(cy2.get_volume())
    print(dir(cy1))
    print(Cylinder.get_area)
    print(Circle.get_area)
    c1 = Circle(3.3)
    print(c1)    # Output: circle with radius of 3.30
    print(issubclass(Cylinder, Circle))   # True
    print(issubclass(Circle, Cylinder))   # False
    print(isinstance(cy1, Cylinder))      # True
```

Métodos mágicos

Magic Method	Invoked Via	Invocation Syntax
__lt__(self, right) __gt__(self, right) __le__(self, right) __ge__(self, right) __eq__(self, right) __ne__(self, right)	**Comparison Operators**	self < right self > right self <= right self >= right self == right self != right
__add__(self, right) __sub__(self, right) __mul__(self, right) __truediv__(self, right) __floordiv__(self, right) __mod__(self, right) __pow__(self, right)	**Arithmetic Operators**	self + right self - right self * right self / right self // right self % right self ** right

Métodos mágicos

<table>
<tr><th>Magic Method</th><th>Invoked Via</th><th>Invocation Syntax</th></tr>
<tr><td>__and__(self, right)
__or__(self, right)
__xor__(self, right)
__invert__(self)
__lshift__(self, n)
__rshift__(self, n)</td><td>Bitwise Operators</td><td>self & right
self | right
self ^ right
~self
self << n
self >> n</td></tr>
<tr><td>__str__(self)
__repr__(self)
__sizeof__(self)</td><td>Function call</td><td>str(self), print(self)
repr(self)
sizeof(se
lf)</td></tr>
<tr><td>__len__(self)
__contains__(self, item)
__iter__(self)
__next__(self)
__getitem__(self, key)
__setitem__(self, key, value)
__delitem__(self, key)</td><td>Sequence Operators & Functions</td><td>len(self)
item in self
iter(self)
next(self)
self[key]
self[key] = value
del self[key]</td></tr>
</table>

Métodos mágicos

Magic Method	Invoked Via	Invocation Syntax
__int__(self) __float__(self) __bool__(self) __oct__(self) __hex__(self)	**Type Conversion Function call**	int(self) float(se lf) bool(se lf) oct(self) hex(self)
__init__(self, *args) __new__(cls, *args)	**Constructor**	x = ClassName(*args)
__del__(self)	**Operator del**	del x
__index__(self)	**Convert this object to an index**	x[self]
__radd__(self, left) __rsub__(self, left) ...	**RHS (Reflected) addition, subtraction, etc.**	left + self left - self ...
__iadd__(self, right) __isub__(self, right) ...	**In-place addition, subtraction, etc**	self += right self -= right ...

Métodos mágicos

Magic Method	Invoked Via	Invocation Syntax
__pos__(self) __neg__(self)	**Unary Positive and Negative operators**	+self -self
__round__(self) __floor__(self) __ceil__(self) __trunc__(self)	**Function Call**	round(self) floor(self) ceil(self) trunc(se lf)
__getattr__(self, name) __setattr__(self, name, value) __delattr__(self, name)	**Object's attributes**	self.name self.name = value del self.name
__call__(self, *args, **kwargs)	**Callable Object**	obj(*args, **kwargs);
__enter__(self),__exit__(self)	Context Manager with-statement	

Números random

- Módulo stdrandom.py

function call	*description*
`uniformInt(lo, hi)`	*uniformly random integer in the range [*`lo, hi`*)*
`uniformFloat(lo, hi)`	*uniformly random float in the range [*`lo, hi`*)*
`bernoulli(p)`	`True` *with probability* `p` *(*`p` *defaults to* `0.5`*)*
`binomial(n, p)`	*number of heads in* `n` *coin flips, each of which is heads with probability* `p` *(*`p` *defaults to* `0.5`*)*
`gaussian(mu, sigma)`	*normal, mean* `mu`*, standard deviation* `sigma` *(*`mu` *defaults to* `0.0`*,* `sigma` *defaults to* `1.0`*)*
`discrete(a)`	`i` *with probability proportional to* `a[i]`
`shuffle(a)`	*randomly shuffle the array* `a[]`

API for our `stdrandom` *module*

Procesado de arrays

- Módulo stdarray.py

function call	*description*
`create1D(n, val)`	*array of length* `n`*, each element initialized to* `val`
`create2D(m, n, val)`	`m`*-by-*`n` *array, each element initialized to* `val`
`readInt1D()`	*array of integers, read from standard input*
`readInt2D()`	*two-dimensional array of integers, read from standard input*
`readFloat1D()`	*array of floats, read from standard input*
`readFloat2D()`	*two-dimensional array of floats, read from standard input*
`readBool1D()`	*array of booleans, read from standard input*
`readBool2D()`	*two-dimensional array of booleans, read from standard input*
`write1D(a)`	*write array* `a[]` *to standard output*
`write2D(a)`	*write two-dimensional array* `a[]` *to standard output*

Note 1: 1D format is an integer `n` *followed by* `n` *elements.*
2D format is two integers `m` *and* `n` *followed by* `m × n` *elements in row-major order.*
Note 2: Booleans are written as `0` *and* `1` *instead of* `False` *and* `True`*.*

API for our `stdarray` *module*

- Módulo stdstats.py

function call	*description*
`mean(a)`	*average of the values in the numeric array* `a[]`
`var(a)`	*sample variance of the values in the numeric array* `a[]`
`stddev(a)`	*sample standard deviation of the values in the numeric array* `a[]`
`median(a)`	*median of the values in the numeric array* `a[]`
`plotPoints(a)`	*point plot of the values in the numeric array* `a[]`
`plotLines(a)`	*line plot of the values in the numeric array* `a[]`
`plotBars(a)`	*bar plot of the values in the numeric array* `a[]`

API for our `stdstats` *module*

Beneficios de la programación modular

- Programas de tamaño razonable.
- Depuración.
- Reusabilidad de código.
- Mantenimiento.

Programación orientada a objetos - Métodos

- Un método es una función asociada a un objeto específico.
- Se invoca utilizando el nombre del objeto seguido del operador punto (.) seguido por el nombre del método y los argumentos del mismo.

	method	*function*
sample call	`x.bit_length()`	`stdio.writeln(bits)`
typically invoked with	variable name	module name
parameters	object reference and argument(s)	argument(s)
primary purpose	manipulate object value	compute return value

Methods versus functions

Programación orientada a objetos – Métodos de la clase str

translate from DNA to mRNA (replace 'T' with 'U')

```
def translate(dna):
    dna = dna.upper()
    rna = dna.replace('T', 'U')
    return rna
```

is the string s a palindrome?

```
def isPalindrome(s):
    n = len(s)
    for i in range(n // 2):
        if s[i] != s[n-1-i]:
            return False
    return True
```

extract file name and extension from a command-line argument

```
s = sys.argv[1]
dot = s.find('.')
base      = s[:dot]
extension = s[dot+1:]
```

write all lines on standard input that contain a string specified as a command-line argument

```
query = sys.argv[1]
while stdio.hasNextLine():
    s = stdio.readLine()
    if query in s:
        stdio.writeln(s)
```

is an array of strings in ascending order?

```
def isSorted(a):
    for i in range(1, len(a)):
        if a[i] < a[i-1]:
            return False
    return True
```

Typical string-processing code

Tipo de dato definido por el usuario

- Se define un tipo Charge para partículas cargadas.
- Se usa la ley de Coulomb para el cálculo del potencial de un punto debido a una carga $V=kq/r$, donde q es el valor de la carga, r es la distancia del punto a la carga y $k=8.99 \times 10^9 \, N \, m^2/C^2$.

Constructor

operation	*description*
`Charge(x0, y0, q0)`	*a new charge centered at* (`x0`, `y0`) *with charge value* `q0`
`c.potentialAt(x, y)`	*electric potential of charge* `c` *at point* (`x`, `y`)
`str(c)`	`'q0 at (x0, y0)'` *(string representation of charge* `c`*)*

API for our user-defined `Charge` *data type*

Convenciones sobre ficheros

- El código que define el tipo de dato definido por el usuario Charge se coloca en un fichero del mismo nombre (sin mayúscula) charge.py
- Un programa cliente que usa el tipo de dato Charge se pone en el cabecero del programa:

```
from charge import Charge
```

Creación de objetos, llamada de métodos y representación de String

```
#-----------------------------
# chargeclient.py
#-----------------------------
import sys
import stdio
from charge import Charge
# Acepta floats x e y como argumentso en la línea de comandos. Crea dos objetos
# Charge con posición y carga. Imprime el potencial en (x, y) en la salida estandard x =
float(sys.argv[1])
y = float(sys.argv[2])
c1 = Charge(.51, .63, 21.3)
c2 = Charge(.13, .94, 81.9)
v1 = c1.potentialAt(x, y)
v2 = c2.potentialAt(x, y)
stdio.writef('potential at (%.2f, %.2f) due to\n', x, y)
stdio.writeln(    '  + str(c1) + '  and')
'                 '  + str(c2))
stdio.writef('is %.2e\n', v1+v2)
#--------------------------------------------------------------
# python chargeclient.py .2 .5
# potential at (0.20, 0.50) due to  #
    21.3 at (0.51, 0.63)  and
#   81.9 at (0.13, 0.94)
# is 2.22e+12from charge import Charge
```

Elementos básicos de un tipo de dato

- API

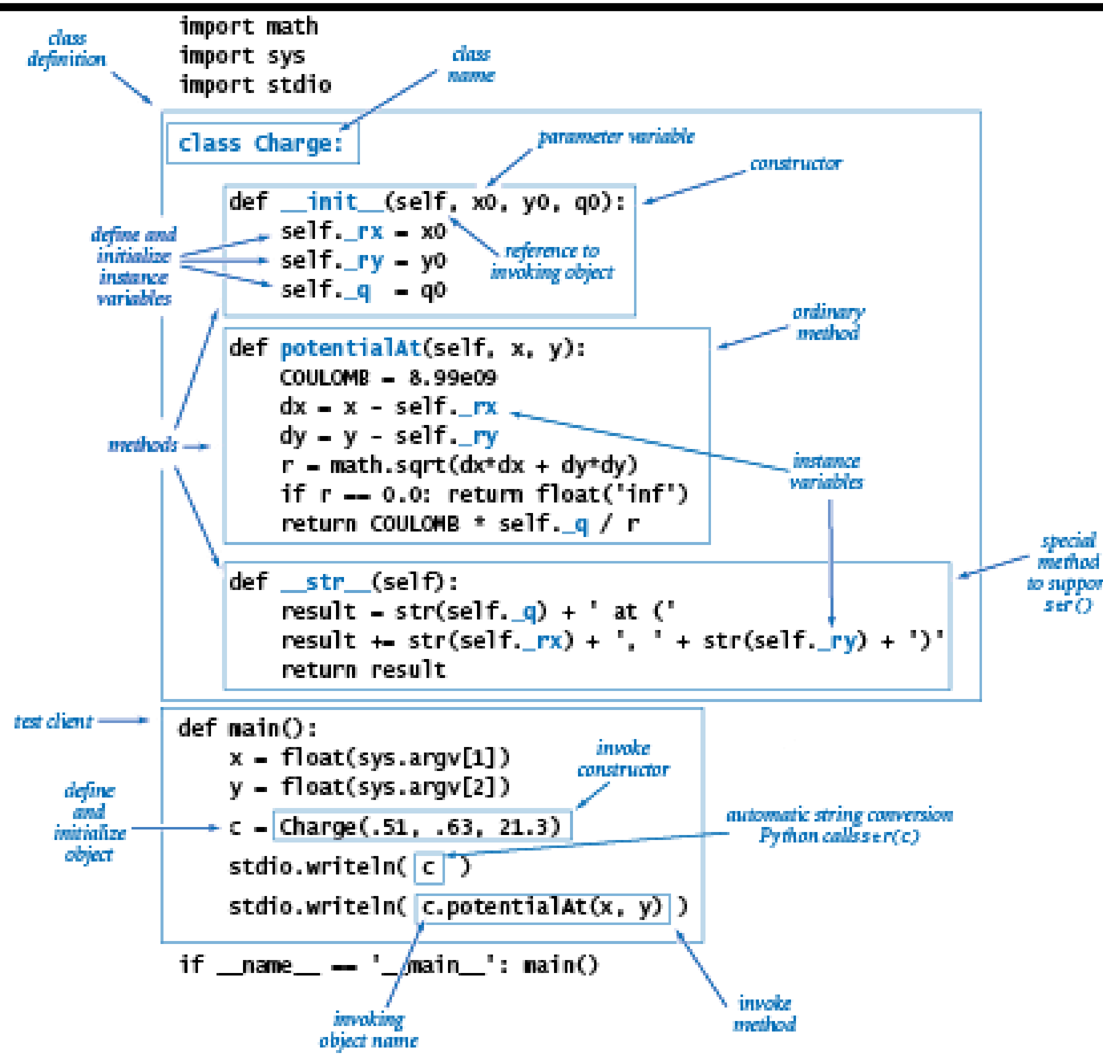
operation	description
`Charge(x0, y0, q0)`	*a new charge centered at* (x_0, y_0) *with charge value* q_0
`c.potentialAt(x, y)`	*electric potential of charge* `c` *at point* `(x, y)`
`str(c)`	`'q0 at (x0, y0)'` *(string representation of charge* `c`*)*

API for our user-defined `Charge` *data type*

- Clase. Fichero charge.py. Palabra reservada class
- Constructor. Método especial __init__(), self
- Variable de instancia. _nombrevar
- Métodos. Variable de instancia self
- Funciones intrínsecas. __str()__
- Privacidad

Implementación de Charge

- En charge.py



Anatomy of a class (data-type) definition

Clases Stopwatch, Histogram, Turtle

- En stopwatch.py

operation	*description*
`Stopwatch()`	*a new stopwatch (running at the start)*
`watch.elapsedTime()`	*elapsed time since* `watch` *was created, in seconds*

API for our user-defined `Stopwatch` *data type*

- En histogram.py

operation	*description*
`Histogram(n)`	*a new histogram from the integer values in* 0, 1, …, n − 1
`h.addDataPoint(i)`	*add an occurrence of integer* `i` *to the histogram* `h`
`h.draw()`	*draw* `h` *to standard drawing*

API for our user-defined `Histogram` *data type*

- En turtle.py

operation	*description*
`Turtle(x0, y0, a0)`	*a new turtle at (x0, y0) facing* `a0` *degrees from the x-axis*
`t.turnLeft(delta)`	*instruct* `t` *to turn left (counterclockwise) by* `delta` *degrees*
`t.goForward(step)`	*instruct* `t` *to move forward distance* `step`*, drawing a line*

API for our user-defined `Turtle` *data type*

Clase Complex

- Métodos especiales. En Python la expresión a + b se reemplaza con la llamada del método a.__mul__(b)

client operation	*special method*	*description*
`Complex(x, y)`	`__init__(self, re, im)`	*new* `Complex` *object with value* `x+yi`
`a.re()`		*real part of* `a`
`a.im()`		*imaginary part of* `a`
`a + b`	`__add__(self, other)`	*sum of* `a` *and* `b`
`a * b`	`__mul__(self, other)`	*product of* `a` *and* `b`
`abs(a)`	`__abs__(self)`	*magnitude of* `a`
`str(a)`	`__str__(self)`	`'x + yi'` *(string representation of* `a`*)*

API for a user-defined `Complex` *data type*

Métodos especiales

client operation	*special method*	*description*
`x + y`	`__add__(self, other)`	*sum of x and y*
`x - y`	`__sub__(self, other)`	*difference of x and y*
`x * y`	`__mul__(self, other)`	*product of x and y*
`x ** y`	`__pow__(self, other)`	*x to the yth power*
`x / y`	`__truediv__(self, other)`	*quotient of x and y*
`x // y`	`__floordiv__(self, other)`	*floored quotient of x and y*
`x % y`	`__mod__(self, other)`	*remainder when dividing x by y*
`+x`	`__pos__(self)`	*x*
`-x`	`__neg__(self)`	*arithmetic negation of x*

Note: Python 2 uses `__div__` *instead of* `__truediv__`.

Special methods for arithmetic operators

Ficheros

Tema 6

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.

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

```
withopen('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()
```

Estructura datos complejos

Tema 7

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 y diccionario

- 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 lista y 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'}
```

Ciclos – patrones

write first n+1 powers of 2	`power = 1`
	`for i in range(n+1):`
	`    stdio.writeln(str(i) + ' ' + str(power))`
	`    power *= 2`

write largest power of 2 less than or equal to n	`power = 1`
	`while 2*power <= n:`
	`    power *= 2`
	`stdio.writeln(power)`

write a sum *(1 + 2 + . . . + n)*	`total = 0`
	`for i in range(1, n+1):`
	`    total += i`
	`stdio.writeln(total)`

write a product *(n! = 1 × 2 × . . . × n)*	`product = 1`
	`for i in range(1, n+1):`
	`    product *= i`
	`stdio.writeln(product)`

write a table of n+1 function values	`for i in range(n+1):`
	`    stdio.write(str(i) + ' ')`
	`    stdio.writeln(2.0 * math.pi * i / n)`

write the ruler function (see Program 1.2.1)	`ruler = '1'`
	`stdio.writeln(ruler)`
	`for i in range(2, n+1):`
	`    ruler = ruler + ' ' + str(i) + ' ' + ruler`
	`    stdio.writeln(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.

Listas

- Es una estructura de datos que almacena una secuencia de objetos, normalmente del mismo tipo.
- El acceso a los elementos de la lista se basa en índices encerrados por corchetes. En una lista bidimensional se realiza con un par de índices.
- El índice del primer elemento es 0
- Las formas de procesar arrays en Python son:
  - Tipo de dato implícito Python.
  - Uso del módulo Python numpy.
  - Uso del módulo stdarray.

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

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.

Matriz con Listas - lectura

```
def lee_matriz(M):
#Dato de la dimensión de la matriz,
    print('Lectura Matriz')
    m = int(input('Numero de filas '))
    n = int(input('Numero de columnas '))
#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)))
```

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('')
```

Example Problem: Simple Statistics

Many programs deal with large collections of similar information.

- Words in a document
- Students in a course
- Data from an experiment
- Customers of a business
- Graphics objects drawn on the screen
- Cards in a deck

Sample Problem: Simple Statistics

Let’s review some code we wrote in chapter 8:

```
# average4.py
#    A program to average a set of numbers
#    Illustrates sentinel loop using empty string as sentinel

def main():
    sum = 0.0
    count = 0
    xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    print "\nThe average of the numbers is", sum / count

main()
```

Sample Problem: Simple Statistics

This program allows the user to enter a sequence of numbers, but the program itself doesn't keep track of the numbers that were entered – it only keeps a running total.

Suppose we want to extend the program to compute not only the mean, but also the median and standard deviation.

Sample Problem: Simple Statistics

The *median* is the data value that splits the data into equal-sized parts.

For the data 2, 4, 6, 9, 13, the median is 6, since there are two values greater than 6 and two values that are smaller.

One way to determine the median is to store all the numbers, sort them, and identify the middle value.

Sample Problem: Simple Statistics

The *standard deviation* is a measure of how spread out the data is relative to the mean.

If the data is tightly clustered around the mean, then the standard deviation is small. If the data is more spread out, the standard deviation is larger.

The standard deviation is a yardstick to measure/express how exceptional the data is.

Sample Problem: Simple Statistics

The standard deviation is

$$s = \sqrt{\frac{\sum(\bar{x} - x_i)^2}{n-1}}$$

Here $\bar{x}$ is the mean, x_i represents the i^{th} data value and n is the number of data values.

The expression $(\bar{x} - x_i)^2$ is the square of the "deviation" of an individual item from the mean.

Sample Problem: Simple Statistics

The numerator is the sum of these squared "deviations" across all the data.

Suppose our data was 2, 4, 6, 9, and 13.

The mean is 6.8

The numerator of the standard deviation is

$$(6.8-2)^2+(6.8-4)^2+(6.8-6)^2+(6.8-9)^2+(6.8-13)^2=149.6$$

$$s=\sqrt{\frac{149.6}{5-1}}=\sqrt{37.4}=6.11$$

Sample Problem: Simple Statistics

As you can see, calculating the standard deviation not only requires the mean (which can't be calculated until all the data is entered), but also each individual data element!

We need some way to remember these values as they are entered.

Applying Lists

We need a way to store and manipulate an entire collection of numbers.
We can’t just use a bunch of variables, because we don’t know many numbers there will be.
What do we need? Some way of combining an entire collection of values into one object.

Lists and Arrays

Python lists are ordered sequences of items. For instance, a sequence of *n* numbers might be called *S*:

$S = s_0, s_1, s_2, s_3, \ldots, s_{n-1}$

Specific values in the sequence can be referenced using subscripts. By using numbers as subscripts, mathematicians can succinctly summarize computations over items in a sequence using subscript variables.

$$\sum_{i=0}^{n-1} s_i$$

Lists and Arrays

Suppose the sequence is stored in a variable `s`. We could write a loop to calculate the sum of the items in the sequence like this:

```
sum = 0
for i in range(n):
    sum = sum + s[i]
```

Almost all computer languages have a sequence structure like this, sometimes called an *array*.

Lists and Arrays

A list or array is a sequence of items where the entire sequence is referred to by a single name (i.e. `s`) and individual items can be selected by indexing (i.e. `s[i]`).

In other programming languages, arrays are generally a fixed size, meaning that when you create the array, you have to specify how many items it can hold.

Arrays are generally also *homogeneous*, meaning they can hold only one data type.

Lists and Arrays

Python lists are dynamic. They can grow and shrink on demand.

Python lists are also *heterogeneous*, a single list can hold arbitrary data types.

Python lists are mutable sequences of arbitrary objects.

List Operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

List Operations

Except for the membership check, we've used these operations before on strings.
The membership operation can be used to see if a certain value appears anywhere in a sequence.

```
>>> lst = [1,2,3,4]
>>> 3 in lst
True
```

List Operations

The summing example from earlier can be written like this:

```
sum = 0
for x in s:
    sum = sum + x
```

Unlike strings, lists are mutable:

```
>>> lst = [1,2,3,4]
>>> lst[3]
4
>>> lst[3] = "Hello‟
>>> lst
[1, 2, 3, 'Hello']
>>> lst[2] = 7
>>> lst
[1, 2, 7, 'Hello']
```

List Operations

A list of identical items can be created using the repetition operator. This command produces a list containing 50 zeroes:

```
zeroes = [0] * 50
```

List Operations

Lists are often built up one piece at a time using append.

```
nums = []
x = input('Enter a number: ')
while x >= 0:
    nums.append(x)
    x = input('Enter a number: ')
```

Here, `nums` is being used as an accumulator, starting out empty, and each time through the loop a new value is tacked on.

List Operations

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

List Operations

```
>>> lst = [3, 1, 4, 1, 5, 9]
>>> lst.append(2)
>>> lst
[3, 1, 4, 1, 5, 9, 2]
>>> lst.sort()
>>> lst
[1, 1, 2, 3, 4, 5, 9]
>>> lst.reverse()
>>> lst
[9, 5, 4, 3, 2, 1, 1]
>>> lst.index(4)
2
>>> lst.insert(4, "Hello")
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1, 1]
>>> lst.count(1)s
2
>>> lst.remove(1)
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1]
>>> lst.pop(3)
3
>>> lst
[9, 5, 4, 'Hello', 2, 1]
```

List Operations

Most of these methods don't return a value – they change the contents of the list in some way.
Lists can grow by appending new items, and shrink when items are deleted. Individual items or entire slices can be removed from a list using the `del` operator.

List Operations

```
>>> myList=[34, 26, 0, 10]
>>> del myList[1]
>>> myList
[34, 0, 10]
>>> del myList[1:3]
>>> myList
[34]
```

`del` isn’t a list method, but a built-in operation that can be used on list items.

List Operations

Basic list principles

- A list is a sequence of items stored as a single object.
- Items in a list can be accessed by indexing, and sublists can be accessed by slicing.
- Lists are mutable; individual items or entire slices can be replaced through assignment statements.

List Operations

Lists support a number of convenient and frequently used methods.
Lists will grow and shrink as needed.

Statistics with Lists

One way we can solve our statistics problem is to store the data in lists.
We could then write a series of functions that take a list of numbers and calculates the mean, standard deviation, and median.
Let's rewrite our earlier program to use lists to find the mean.

Statistics with Lists

Let’s write a function called `getNumbers` that gets numbers from the user.

- We’ll implement the sentinel loop to get the numbers.
- An initially empty list is used as an accumulator to collect the numbers.
- The list is returned once all values have been entered.

Statistics with Lists

```
def getNumbers():
    nums = []     # start with an empty list

    # sentinel loop to get numbers
    xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        nums.append(x)   # add this value to the list
        xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    return nums
```

Using this code, we can get a list of numbers from the user with a single line of code:

```
data = getNumbers()
```

Statistics with Lists

Now we need a function that will calculate the mean of the numbers in a list.

Input: a list of numbers

Output: the mean of the input list

```
def mean(nums):
    sum = 0.0
    for num in nums:
        sum = sum + num
    return sum / len(nums)
```

Statistics with Lists

The next function to tackle is the standard deviation.

In order to determine the standard deviation, we need to know the mean.

- Should we recalculate the mean inside of `stdDev`?
- Should the mean be passed as a parameter to `stdDev`?

Statistics with Lists

Recalculating the mean inside of `stdDev` is inefficient if the data set is large.

Since our program is outputting both the mean and the standard deviation, let's compute the mean and pass it to `stdDev` as a parameter.

Statistics with Lists

```
def stdDev(nums, xbar):
    sumDevSq = 0.0
    for num in nums:
        dev = xbar - num
        sumDevSq = sumDevSq + dev * dev
    return sqrt(sumDevSq/(len(nums)-1))
```

The summation from the formula is accomplished with a loop and accumulator.

`sumDevSq` stores the running sum of the squares of the deviations.

Statistics with Lists

We don’t have a formula to calculate the median. We’ll need to come up with an algorithm to pick out the middle value.

First, we need to arrange the numbers in ascending order.

Second, the middle value in the list is the median.

If the list has an even length, the median is the average of the middle two values.

Statistics with Lists

Pseudocode -

```
sort the numbers into ascending order
if the size of the data is odd:
    median = the middle value
else:
    median = the average of the two middle values
return median
```

Statistics with Lists

```
def median(nums):
    nums.sort()
    size = len(nums)
    midPos = size / 2
    if size % 2 == 0:
        median = (nums[midPos] + nums[midPos-1]) / 2.0
    else:
        median = nums[midPos]
    return median
```

Statistics with Lists

With these functions, the main program is pretty simple!

```
def main():
    print 'This program computes mean, median and standard deviation.'

    data = getNumbers()
    xbar = mean(data)
    std = stdDev(data, xbar)
    med = median(data)

    print '\nThe mean is', xbar
    print 'The standard deviation is', std
    print 'The median is', med
```

Statistics with Lists

Statistical analysis routines might come in handy some time, so let's add the capability to use this code as a module by adding:

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

Lists of Objects

All of the list examples we've looked at so far have involved simple data types like numbers and strings.

We can also use lists to store more complex data types, like our student information from chapter ten.

Lists of Objects

Our grade processing program read through a file of student grade information and then printed out information about the student with the highest GPA.
A common operation on data like this is to sort it, perhaps alphabetically, perhaps by credit-hours, or even by GPA.

Lists of Objects

Let's write a program that sorts students according to GPA using our `Sutdent` class from the last chapter.

```
Get the name of the input file from the user
Read student information into a list
Sort the list by GPA
Get the name of the output file from the user
Write the student information from the list into a file
```

Lists of Objects

Let’s begin with the file processing. The following code reads through the data file and creates a list of students.

```
def readStudents(filename):
    infile = open(filename, 'r')
    students = []
    for line in infile:
        students.append(makeStudent(line))
    infile.close()
    return students
```

We’re using the `makeStudent` from the `gpa` program, so we’ll need to remember to import it.

Lists of Objects

Let’s also write a function to write the list of students back to a file.

Each line should contain three pieces of information, separated by tabs: name, credit hours, and quality points.

```
def writeStudents(students, filename):
    # students is a list of Student objects
    outfile = open(filename, 'w')
    for s in students:
        outfile.write("%s\t%f\t%f\n" %
                (s.getName(),s.getHours(),s.getQPoints()))
    outfile.close()
```

Lists of Objects

Using the functions `readStudents` and `writeStudents`, we can convert our data file into a list of students and then write them back to a file. All we need to do now is sort the records by GPA.

In the statistics program, we used the `sort` method to sort a list of numbers. How does Python sort lists of objects?

Lists of Objects

Python compares items in a list using the built-in function `cmp`.

`cmp` takes two parameters and returns `-1` if the first comes before the second, `0` if they're equal, and `1` if the first comes after the second.

Lists of objects

```
>>> cmp(1,2)
-1
>>> cmp(2,1)
1
>>> cmp("a","b")
-1
>>> cmp(1,1.0)
0
>>> cmp("a",5)
1
```

For types that aren't directly comparable, the standard ordering uses rules like "numbers always comes before strings."

Lists of Objects

To make sorting work with our objects, we need to tell `sort` how the objects should be compared.

We do this by writing our own custom `cmp`-like function and then tell `sort` to use it when sorting.

To sort by GPA, we need a routine that will take two students as parameters and then returns `-1`, `0`, or `1`.

Lists of Objects

We can use the built-in `cmp` function.

```
def cmpGPA(s1, s2):
    return cmp(s1.gpa(), s2.gpa()
```

We can now sort the data by calling sort with the appropriate comparison function (`cmpGPA`) as a parameter.

```
data.sort(cmpGPA)
```

Lists of Objects

```
data.sort(cmpGPA)
```

Notice that we didn't put `()`'s after the function call `cmpGPA`.

This is because we don't want to *call* `cmpGPA`, but rather, we want to send `cmpGPA` to the sort method to use as a comparator.

Lists of Objects

```
# gpasort.py
#    A program to sort student information into GPA order.


from gpa import Student, makeStudent

def readStudents(filename):
   infile = open(filename, 'r')
   students = []
   for line in infile:
      students.append(makeStudent(line))
   infile.close()
   return students

def writeStudents(students, filename):
   outfile = open(filename, 'w')
   for s in students:
      outfile.write("%s\t%f\t%f\n" %
                 (s.getName(), s.getHours(),
                    s.getQPoints()))
   outfile.close()
```

```
def cmpGPA(s1, s2):
   return cmp(s1.gpa(), s2.gpa())

def main():
   print "This program sorts student grade information by
GPA"
   filename = raw_input("Enter the name of the data file: ")
   data = readStudents(filename)
   data.sort(cmpGPA)
   filename = raw_input("Enter a name for the output file: ")
   writeStudents(data, filename)
   print "The data has been written to", filename

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

Designing with Lists and Classes

In the `dieView` class from chapter ten, each object keeps track of seven circles representing the position of pips on the face of the die.

Previously, we used specific instance variables to keep track of each, `pip1`, `pip2`, `pip3`, …

What happens if we try to store the circle objects using a list?

In the previous program, the pips were created like this:

```
self.pip1 = self.__makePip(cx, cy)
```

`__makePip` is a local method of the `DieView` class that creates a circle centered at the position given by its parameters.

Designing with Lists and Classes

One approach is to start with an empty list of pips and build up the list one pip at a time.

```
pips = []
pips.append(self.__makePip(cx-offset,cy-offset)
pips.append(self.__makePip(cx-offset,cy)
…
self.pips = pips
```

An even more straightforward approach is to create the list directly.

```
self.pips = [self.__makePip(cx-offset,cy-offset),
             self.__makePip(cx-offset,cy),
             …
             self.__makePip(cx+offset,cy+offset)
             ]
```

Python is smart enough to know that this object is continued over a number of lines, and waits for the ‘]’. Listing objects like this, one per line, makes it much easier to read.

Designing with Lists and Classes

Putting our pips into a list makes many actions simpler to perform.

To blank out the die by setting all the pips to the background color:

```
for pip in self.pips:
      pip.setFill(self.background)
```

This cut our previous code from seven lines to two!

Designing with Lists and Classes

We can turn the pips back on using the pips list. Our original code looked like this:

```
self.pip1.setFill(self.foreground)
self.pip4.setFill(self.foreground)
self.pip7.setFill(self.foreground)
```

Into this:

```
self.pips[0].setFill(self.foreground)
self.pips[3].setFill(self.foreground)
self.pips[6].setFill(self.foreground)
```

Designing with Lists and Classes

Here's an even easier way to access the same methods:

```
for i in [0,3,6]:
   self.pips[i].setFill(self.foreground)
```

We can take advantage of this approach by keeping a list of which pips to activate!

```
Loop through pips and turn them all off
Determine the list of pip indexes to turn on
Loop through the list of indexes - turn on those pips
```

Designing with Lists and Classes

```
for pip in self.pips:
self.pip.setFill(self.background)
if value == 1:
on = [3]
elif value == 2:
on = [0,6]
elif value == 3:
on = [0,3,6]
elif value == 4:
on = [0,2,4,6]
elif value == 5:
on = [0,2,3,4,6]
else:
on = [0,1,2,3,4,5,6]
for i in on:
self.pips[i].setFill(self.foreground)
```

Designing with Lists and Classes

We can do even better!

The correct set of pips is determined by `value`. We can make this process *table-driven* instead.

We can use a list where each item on the list is itself a list of pip indexes.

For example, the item in position 3 should be the list `[0,3,6]` since these are the pips that must be turned on to show a value of 3.

Designing with Lists and Classes

Here’s the table-driven code:

```
onTable = [ [], [3], [2,4], [2,3,4], [0,2,4,6],
            [0,2,3,4,6], [0,1,2,4,5,6] ]

for pip in self.pips:
   self.pip.setFill(self.background)

on = onTable[value]
for i in on:
   self.pips[i].setFill(self.foreground)
```

Designing with Lists and Classes

```
onTable = [ [], [3], [2,4], [2,3,4], [0,2,4,6], [0,2,3,4,6], [0,1,2,4,5,6] ]

for pip in self.pips:
   self.pip.setFill(self.background)

on = onTable[value]
for i in on:
   self.pips[i].setFill(self.foreground)
```

The table is padded with ‘[]’ in the 0 position, since it shouldn’t ever be used.

The `onTable` will remain unchanged through the life of a `dieView`, so it would make sense to store this table in the constructor and save it in an instance variable.

Designing with Lists and Classes

```
# dieview2.py
#     A widget for displaying the value of a die.
#     This version uses lists to simplify keeping track of
pips.

class DieView:
    """ DieView is a widget that displays a graphical
representation
    of a standard six-sided die."""

    def __init__(self, win, center, size):
        """Create a view of a die, e.g.:
           d1 = GDie(myWin, Point(40,50), 20)
        creates a die centered at (40,50) having sides
        of length 20."""

        # first define some standard values
        self.win = win
        self.background = "white" # color of die face
        self.foreground = "black" # color of the pips
        self.psize = 0.1 * size   # radius of each pip
        hsize = size / 2.0        # half of size
        offset = 0.6 * hsize      # distance from center to
outer pips

        # create a square for the face
        cx, cy = center.getX(), center.getY()
        p1 = Point(cx-hsize, cy-hsize)
        p2 = Point(cx+hsize, cy+hsize)
        rect = Rectangle(p1,p2)
        rect.draw(win)
        rect.setFill(self.background)

# Create 7 circles for standard pip locations
        self.pips = [ self.__makePip(cx-offset, cy-offset),
                      self.__makePip(cx-offset, cy),
                      self.__makePip(cx-offset, cy+offset),
                      self.__makePip(cx, cy),
                      self.__makePip(cx+offset, cy-offset),
                      self.__makePip(cx+offset, cy),
                      self.__makePip(cx+offset, cy+offset) ]

        # Create a table for which pips are on for each value
        self.onTable = [ [], [3], [2,4], [2,3,4],
            [0,2,4,6], [0,2,3,4,6], [0,1,2,4,5,6] ]

        self.setValue(1)

    def __makePip(self, x, y):
        """Internal helper method to draw a pip at (x,y)"""
        pip = Circle(Point(x,y), self.psize)
        pip.setFill(self.background)
        pip.setOutline(self.background)
        pip.draw(self.win)
        return pip

    def setValue(self, value):
        """ Set this die to display value."""
        # Turn all the pips off
        for pip in self.pips:
            pip.setFill(self.background)

        # Turn the appropriate pips back on
        for i in self.onTable[value]:
            self.pips[i].setFill(self.foreground)
```

Designing with Lists and Classes

Lastly, this example showcases the advantages of encapsulation.

- We have improved the implementation of the `dieView` class, but we have not changed the set of methods it supports.
- We can substitute this **new** version of the class without having to modify any other code!
- Encapsulation allows us to build complex software systems as a set of “pluggable modules.”

Introducción a Python

ESTEBAN VAZQUEZ
ESTEBAN.VAZQUEZ@TESLATECHNOLOGIES.COM