Tema 1: Introducción a Python

Carmen Graciani Díaz
José Luis Ruiz Reina

Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Razonamiento asistido por computador

Introducción a PYTHON

- Creado a principios de los 90 por Guido van Rossum
- El nombre procede del programa de la BBC “Monty Python’s Flying Circus”
- Todas las distribuciones son de código abierto
- Es un “poderoso” lenguaje de programación “fácil” de aprender
- Cuenta con una amplia biblioteca estándar
- Web oficial de Python: *http://www.python.org*
- Lenguaje interpretado de alto nivel que puede extenderse con C o C++
  - Programación orientada a objetos
  - Programación imperativa
  - Programación funcional

Trabajar con PYTHON

- Está disponible en multitud de plataformas (UNIX, Solaris, Linux, DOS, Windows, OS/2, Mac OS, etc.)
- Modo interactivo
  - Arrancar un intérprete: `python3`
  - Escribir una expresión en el intérprete
  - Evaluar la expresión
- Creación de guiones
  - Escribir el código en un `fichero.py`
  - Añadir al inicio `#!/usr/bin/python3`
  - Darle permiso de ejecución

Interacción

- Intérprete

```
$ python3
Python 3.5.2 |Anaconda 4.0.0 (64-bit)|
(default, Jul  2 2016, 11:16:01)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license"
for more information.

>>> print('¡Hola! y ¡Adiós!')
¡Hola! y ¡Adiós!
>>> exit()
$
```

Entornos de programación

- Entorno de programación:
  - Editar programas
  - Interactuar con el intérprete
  - Desarrollo de proyectos
  - Depuración
- Algunos de los más conocidos:
  - Idle
  - Spyder
  - IPython+Jupyter
  - Pydev
  - PyCharm
  - Emacs+Python-mode

Guiones

- Guión (*script*):

```
#!/usr/bin/python3
# Escribe por pantalla el mensaje: ¡Hola! y ¡Adiós!
print('¡Hola! y ¡Adiós!')
```

- Ejecución:

```
$ ./ejemplo.py
¡Hola! y ¡Adiós!
$
```

- Comentarios en Python: `#` y triples comillas

Tipos de datos numéricos

- Números (son inmutables)

```
>>> 2+2
4
>>> (50-5*6)/4
5.0
>>> a = (1+2j)/(1+1j)
>>> a.real
1.5
>>> ancho = 20
>>> alto = 5*9
>>> area = ancho * alto
>>> area
900
>>> area *= 2
>>> area
1800
```

- Mutable *vs* inmutable. Asignaciones *aumentadas*
- Las variables en Python siempre *son referencias a objetos*

Booleanos

- Dos datos booleanos: `True` y `False`
- Operadores lógicos `and`, `or`, `not` y comparación con `==`

```
>>> 2 == 2
True
>>> 2 == 3
False
>>> True and 2 == 3
False
>>> False or 2 == 2
True
>>> not False
True
>>> True == 1
True
>>> False == 0
True
```

Cadenas

- Secuencia de caracteres, entre comillas simples o dobles (inmutables)
- Algunas operaciones con cadenas:

```
>>> c1="Buenas"
>>> c2=" tardes"
>>> frase = c1+c2
>>> frase
'Buenas tardes'
>>> frase[3:8]
'nas t'
>>> frase[3:8:2]
'nst'
>>> frase[-1]
's'
>>> frase[3:8:-1]
''
>>> frase[3:8]
'nas t'
>>> frase[8:3:-1]
'at sa'
>>> frase[::-1]
'sedrat saneuB'
>>> frase * 4
'Buenas tardesBuenas tardesBuenas tardesBuenas tardes'
```

Cadenas

- Algunos métodos sobre cadenas

```
>>> cad="En un lugar de La Mancha"
>>> cad.index("Mancha")
18
>>> cad.index("plancha")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> cad.find("Mancha")
18
>>> cad.find("plancha")
-1
>>> cad.upper()
'EN UN LUGAR DE LA MANCHA'
>>> cad.count("u")
2
```

Cadenas

- Más operaciones con cadenas:

```
>>> " ".join(["Rojo", "blanco", "negro"])
'Rojo blanco negro'
>>> " y ".join(["Rojo", "blanco", "negro"])
'Rojo y blanco y negro'
>>> "Rojo blanco negro".split(" ")
['Rojo', 'blanco', 'negro']
>>> "Rojo y blanco y negro".split(" ")
['Rojo', 'y', 'blanco', 'y', 'negro']
>>> "Rojo y blanco y negro".split(" y ")
['Rojo', 'blanco', 'negro']
```

Cadenas

- Escritura por pantalla (`print` y `format`)

```
>>> print("Inteligencia","Artificial")
Inteligencia Artificial
>>> c="{0} por {1} es {2}"
>>> x,y,u,z = 2,3,4,5
>>> print(c.format(x,y,x*y))
2 por 3 es 6
>>> print(c.format(u,z,u*z))
4 por 5 es 20
```

Tuplas

- Secuencias ordenadas de elementos, separados por comas y usualmente entre paréntesis

```
>>> 1,2,3,4
(1, 2, 3, 4)
>>> ()
()
>>> 1,
(1,)
>>> a=2
>>> b=3
>>> (a,b,a+b,a-b,a*b,a/b)
(2, 3, 5, -1, 6, 0.6666666666666666)
```

Tuplas

- Se le pueden aplicar operaciones similares a las de cadenas

```
>>> a=("Uno","Dos","Tres","Cuatro")
>>> a[2]
'Tres'
>>> a[1:3]
('Dos', 'Tres')
>>> a[::]
('Uno', 'Dos', 'Tres', 'Cuatro')
>>> a[::-1]
('Cuatro', 'Tres', 'Dos', 'Uno')
>>> a+a[2::-1]
('Uno','Dos','Tres','Cuatro','Tres','Dos','Uno')
>>> "Dos" in a
True
```

Tuplas

- Inmutables:

```
>>> a=("Madrid","Paris","Roma","Berlin","Londres")
>>> b=a
>>> b
('Madrid', 'Paris', 'Roma', 'Berlin', 'Londres')
>>> a[3]="Atenas"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support assignment
>>> a+="Atenas",
>>> a
('Madrid','Paris','Roma','Berlin','Londres','Atenas')
>>> b
('Madrid', 'Paris', 'Roma', 'Berlin', 'Londres')
```

Tuplas

- Tuplas en lado izquierdo de asignaciones:

```
>>> a,b,c=(1,2,3)
>>> a
1
>>> b
2
>>> c
3
>>> a,b=b,a # ¡¡intercambio en una sola instrucción!!
>>> a
2
>>> b
1
```

- Desempaquetado

```
>>> a,*b =(1,2,3,4)
>>> a
1
>>> b
[2, 3, 4]
>>> *a,b = [1,2,3,4]
>>> a
[1, 2, 3]
>>> b
4
```

Listas

- Secuencias ordenadas de elementos, separados por comas y entre corchetes

```
>>> ["a","b","c","d"]
['a', 'b', 'c', 'd']
>>> [2]
[2]
>>> []
[]
```

- Operaciones similares a las de tuplas y cadenas

```
>>> bocadillo = ['pan', 'jamon', 'pan']
>>> 2*bocadillo[:2] + ['huevo'] + [bocadillo[-1]]
['pan', 'jamon', 'pan', 'jamon', 'huevo', 'pan']
>>> triple = 2*bocadillo + ["tomate",'pan']
>>> triple
['pan', 'jamon', 'pan', 'pan', 'jamon', 'pan',
 'tomate', 'pan']
>>> "tomate" in triple
True
>>> len(triple)
8
```

Listas

- Las listas son mutables:

```
>>> l=["hola","adios","hasta pronto"]
>>> m=l
>>> m
['hola','adios','hasta pronto']
>>> l[2]="see you"
>>> l
['hola','adios','see you']
>>> m
['hola','adios','see you']
>>> p=[l,m]
>>> p
[['hola','adios','see you'],['hola','adios','see you']]
>>> m[0]="hello"
>>> p
[['hello','adios','see you'], ['hello','adios','see you']]
>>> p[0][1:2]=[]
>>> p
[['hello', 'see you'], ['hello', 'see you']]
>>> l
['hello', 'see you']
>>> m
['hello', 'see you']
```

Listas

- Algunos métodos sobre listas

```
>>> r=["a",1,"b",2,"c","3"]
>>> r.append("d")
>>> r
['a', 1, 'b', 2, 'c', '3', 'd']
>>> r.extend([4,"e"])
>>> r
['a', 1, 'b', 2, 'c', '3', 'd', 4, 'e']
>>> r.pop()
'e'
>>> r
['a', 1, 'b', 2, 'c', '3', 'd', 4]
>>> r.pop(0)
'a'
>>> r
[1, 'b', 2, 'c', '3', 'd', 4]
>>> r.insert(3,"x")
>>> r
[1, 'b', 2, 'x', 'c', '3', 'd', 4]
```

Definiciones por comprensión

- Las listas se pueden definir sin necesidad de explicitar sus elementos

```
>>> [a for a in range(6)]
[0, 1, 2, 3, 4, 5]
>>> [a for a in range(6) if a % 2==0]
[0, 2, 4]
>>> [a*a for a in range(6) if a % 2==0]
[0, 4, 16]
>>> [(x,y) for x in [1,2,3] for y in ["a","b","c"]]
[(1, 'a'),(1, 'b'),(1, 'c'),(2, 'a'),(2, 'b'),(2, 'c'),
 (3, 'a'),(3, 'b'),(3, 'c')]
>>> (a*a for a in range(6) if a % 2==0)
<generator object <genexpr> at 0x7fbb85aa8a00>
```

- Aplicable también a otros tipos datos de colección:

```
>>> tuple(a*a for a in range(6) if a % 2==0)
(0, 4, 16)
>>> tuple(a%3 for a in range(9))
(0, 1, 2, 0, 1, 2, 0, 1, 2)
>>> {a%3 for a in range(9)} #tipo conjunto,lo vemos a continuació
{0, 1, 2}
```

Conjuntos

- Colecciones de datos, sin orden y sin duplicados, representados entre llaves y separados por comas
- Los elementos de los conjuntos deben ser *hashables*
  - En particular todos los tipos de datos inmutables son hashables
- Los conjuntos son mutables
- Ejemplos:

```
>>> cesta = {"peras", "manzanas", "peras", "manzanas"}
>>> cesta
{'peras', 'manzanas'}
>>> "melocotones" in cesta
False
>>> a = {x for x in "abracadabra" if x not in "abc"}
>>> a
{'r', 'd'}
```

Conjuntos

- Algunos métodos sobre conjuntos

```
>>> s={1,3,5,7,9}
>>> s.add(10)
>>> s
{1, 3, 5, 7, 9, 10}
>>> s.add(10)
>>> s
{1, 3, 5, 7, 9, 10}
>>> s | {1,2,4}
{1, 2, 3, 4, 5, 7, 9, 10}
>>> s
{1, 3, 5, 7, 9, 10}
>>> s & {4,7,15}
{7}
>>> s
{1, 3, 5, 7, 9, 10}
>>> s <= {1,3,5,7,9,10,11,12}
True
>>> s |= {2,4}
>>> s
{1, 2, 3, 4, 5, 7, 9, 10}
```

Diccionarios

- Colección no ordenada de pares *clave*:*valor* (es un tipo de dato mutable)

```
>>> tel = {"juan": 4098, "ana": 4139}
>>> tel["ana"]
4139
>>> "ana" in tel
True
>>> tel["pedro"]=4118
>>> tel
{'juan': 4098, 'pedro': 4118, 'ana': 4139}
>>> tel.keys()
dict_keys(['juan', 'pedro', 'ana'])
>>> del tel['ana']
>>> [(n,t) for (n,t) in tel.items()]
[('juan', 4098), ('pedro', 4118)]
>>> tel["olga"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'olga'
>>> None == tel.get("olga")
True
```

Estructuras de control (instrucción `if`)

- Programa `signo.py`:

```
x = int(input('Escribe un entero: '))

if x < 0:
    print('Negativo:', x)
elif x == 0:
    print('Cero')
else:
    print('{0} es positivo:'.format(x))
```

- El papel de la indentación en Python
- Ejecución:

```
Escribe un entero:34
34 es positivo
```

Estructuras de control (instrucción `for`)

- Programa `media.py`:

```
l, suma, n = [1,5,8,12,3,7], 0, 0
for e in l:
    suma += e
    n +=1
print(suma/n)
```

- Programa `primos.py`:

```
primos = []
for n in range(1, 20, 2):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'es', x, '*', n//x)
            break
    else:
        primos.append(n)
```

  - Salida:

```
9 es 3 * 3
15 es 3 * 5
```

  - Variable `primos`: `[1, 3, 5, 7, 11, 13, 17, 19]`

Estructuras de control (instrucción `while`)

- Programa `fibonacci.py`:

```
a, b = 0, 1
while b < 10:
    print(b,end=',')
    a,b = b,a+b
```

- Programa `busca-indice.py`

```
ind =0
busco = "premio"
lst = ["nada","pierdo","premio","sigue"]
while ind < len(lst):
    if lst[ind] == busco:
       break
    ind += 1
else: ind = -1
```

- Las instrucciones `pass`, `continue`, `break` y `return`

Algunos patrones de iteración

```
>>> notas = {'Juan Gómez': 'notable', 'Pedro Pérez': 'aprobado'}
>>> for k, v in notas.items(): print(k, v)
Pedro Pérez aprobado
Juan Gómez notable

>>> for i, col in enumerate(['rojo', 'azul', 'amarillo']):
...     print(i, col)
0 rojo
1 azul
2 amarillo

>>> preguntas = ['nombre', 'apellido', 'color favorito']
>>> respuestas = ['Juan', 'Pérez', 'rojo']
>>> for p, r in zip(preguntas, respuestas):
...     print('Mi {0} es {1}.'.format(p, r))
Mi nombre es Juan.
Mi apellido es Pérez.
Mi color favorito es rojo.

>>> for i in reversed(range(1, 10, 2)): print(i,end="-")
...
9-7-5-3-1-
```

Tipos iterables e iteradores

- Tipos iterables: aquellos para los que tiene sentido *recorrerlos* y una cierta noción de *siguiente*
  - Cadenas, tuplas, listas, conjuntos, diccionarios, ...
  - Se usan en los bucles: `for` *item* `in` *iterable: ...*
  - Cuando un iterable se usa en un bucle, a partir de él se obtiene automáticamente un *iterador*, que irá generando sus elementos secuencialmente, uno en cada iteración
- Generadores: expresiones como iteradores
  - Por ejemplo: funciones `range`, `enumerate`, `zip`, `reversed`,...:

```
>>> range(1,10,2)
range(1, 10, 2)
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
```

  - Generadores por comprensión:

```
>>> (x * x for x in range(1,10,3))
<generator object <genexpr> at 0x7f1415de9a50>
>>> list(x * x for x in range(1,10,3))
[1, 16, 49]
```

Definición de funciones

- Definición de funciones:

```
def fib(n):
    """Imprime la sucesión de Fibonacci hasta n
       y devuelve el último calculado """
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
    return b
```

- Usando la función:

```
>>> fib(30)
0 1 1 2 3 5 8 13 21
55
>>> x=fib(30)
0 1 1 2 3 5 8 13 21
>>> x
55
```

- Diferencia entre efecto colateral y valor devuelto (`return`)
  - Cuando una función no hace un `return` explícito, devuelve `None`

Argumentos de funciones

- Argumentos con clave

```
>>> def g(x,y): return(x**y)
>>> g(2,3)
8
>>> g(y=3,x=2)
8
>>> g(2,x=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: g() got multiple values
           for keyword argument 'x'
```

Argumentos de funciones

- Argumentos con valor por defecto

```
>>> def j(x,y,z=0): return(x**y + z)
>>> j(2,3)
8
>>> j(2,3,4)
12
```

  - ¡Ojo!, los argumentos por defecto se evalúan una sólo vez, al definir la función:

```
>>> i = [5]
>>> def f(x=i): return x
>>> f()
>>> [5]
>>> i.append(8)
>>> f()
>>> [5,8]
>>> i = []
>>> f()
>>> [5,8]
```

Argumentos de funciones

- ¡Ojo!, las listas se pueden modificar ya que son referencias

```
>>> i = [5]
>>> def f(x=i): return x
>>> f()
>>> [5]
>>> i.append(8)
>>> f()
>>> [5,8]
>>> i = []
>>> f()
>>> [5,8]
```

Argumentos de funciones

- Número arbitario de argumentos

```
>>> def h(x,*y): print(x,y)
>>> h(3,2,5,7,2,5)
3 (2, 5, 7, 2, 5)
>>> h("a","b","c")
a ('b', 'c')
>>> h(10)
10 ()
>>> def d(**y): print(y)
>>> d(a=2,b=3,c=4)
{'a': 2, 'c': 4, 'b': 3}
```

- Llamadas a funciones con desempaquetado

```
>>> def d(x,y,z): return(x**y + z)
...
>>> l=[3,2,4]
>>> d(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: d() takes exactly 3 positional arguments (1 given)
>>> d(*l)
13
```

Errores y gestión de errores

- Excepciones:

```
>>> def devuelve_doble():
        x= int(input("Introduzca un número: "))
        return 2*x
>>> devuelve_doble()
Introduzca un número: a
...
ValueError: invalid literal for int() with base 10: 'a'
```

- Manejo de excepciones con `try...except`

```
>>> def devuelve_doble():
        while True:
            try:
                x= int(input("Introduzca un número: "))
                return 2*x
            except ValueError:
                print("No es un número, inténtelo de nuevo.")

>>> devuelve_doble()
Introduzca un número: a
No es un número, inténtelo de nuevo.
Introduzca un número: d
No es un número, inténtelo de nuevo.
Introduzca un número: 3
6
```

Módulos

- Un módulo es un fichero con definiciones e instrucciones Python. Por ejemplo, `operaciones-es.py`:

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

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

def multiplicacion(x,y): return x*y

def division(x,y): return x/y
```

- Usando módulos con `import`:

```
>>> import operaciones
>>> suma(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'suma' is not defined
>>> operaciones.suma(2,3)
5
>>> operaciones.division(8,5)
1.6
```

Importación de módulos

- Otra manera de importar módulos:

```
>>> import operaciones as ops
>>> ops.resta(4,2)
2
>>> operaciones.resta(4,2)
....
NameError: name 'operaciones' is not defined
```

- Otra más:

```
>>> from operaciones import *
>>> resta(3,2)
1
>>> division(4,6)
0.666666666666666
```

- Y otra:

```
>>> from operaciones import suma,resta
>>> suma(2,3)
5
>>> resta(4,1)
3
>>> multiplicacion(2,3)
..
NameError: name 'multiplicacion' is not defined
```

Segundo orden

- Expresiones `lambda`:

```
>>> lambda x,y: x+y*3
<function <lambda> at 0x7f1415e022f8>
>>> (lambda x,y: x+y*3)(2,3)
11
>>> (lambda x,y: x+y*3)("a","b")
'abbb'
```

- Funciones que devuelven funciones:

```
>>> def incremento(n): return lambda x: x+n
>>> f2 = incremento(2)
>>> f2(5)
7
```

- Funciones que reciben funciones como argumento:

```
>>> def aplica (f,l): return [f(x) for x in l]
>>> aplica (incremento(5),[1,2,3])
[6, 7, 8]
```

Entrada y salida en ficheros

- Apertura de ficheros con `open`:

```
>>> f=open("fichero.txt","r")
>>> f
<_io.TextIOWrapper name='fichero.txt' mode='r' encoding='UTF-8'>
```

  - Se crea un *objeto fichero* con una serie de métodos asociados para realizar las operaciones de entrada/salida

- Modos de apertura (ficheros de texto):
  - `open('fichero.txt','r')`: apertura para lectura (por defecto, puede omitirse)
  - `open('fichero.txt','w')`: apertura para escritura, sobreescribiendo lo existente
  - `open('fichero.txt','a')`: apertura para escritura, añadiendo al final de lo existente
  - `open('fichero.txt','r+')`: apertura para lectura y escritura

- Métodos más usados con ficheros: `f.read()`, `f.readline()`, `f.write()`, `f.close()`

Entrada y salida en ficheros (ejemplos)

- Supongamos el archivo `fichero.txt` con el siguiente texto:

```
Esta es la primera línea
Vamos por la segunda
La tercera ya llegó
Y finalizamos con la cuarta
```

- Lectura con `read`:

```
>>> f=open("fichero.txt")
>>> s=f.read()
>>> s
'Esta es la primera línea\nVamos por la segunda\nLa
tercera ya llegó\nY finalizamos con la cuarta\n\n'
>>> f.close()
```

- Uso del bloque `with` para cierre al finalizar

```
>>> with open('fichero.txt') as f: primera = f.readline()
...
>>> primera
'Esta es la primera línea\n'
```

Entrada y salida en ficheros (ejemplos)

- Lectura con `readline`:

```
>>> f=open("fichero.txt")
>>> s1=f.readline()
>>> s1
'Esta es la primera línea\n'
>>> s2=f.readline()
>>> s2
'Vamos por la segunda\n'
>>> s3=f.readline()
>>> s3
'La tercera ya llegó\n'
>>> s4=f.readline()
>>> s4
'Y finalizamos con la cuarta\n'
>>> f.close()
```

- Iterando sobre las líneas de un fichero de texto:

```
>>> for line in open("fichero.txt"): print(line, end='')

Esta es la primera línea
Vamos por la segunda
La tercera ya llegó
Y finalizamos con la cuarta
```

Entrada y salida en ficheros (ejemplos)

- Escritura con `write` en modo `'a'`:

```
>>> with open('fichero.txt','a') as f:
        f.write("La quinta la escribo yo\n")
24
```

  - Contenido de `fichero.txt`:

```
Esta es la primera línea
Vamos por la segunda
La tercera ya llegó
Y finalizamos con la cuarta
La quinta la escribo yo
```

- Escritura con `write` en modo `'w'`:

```
>>> f=open("fichero.txt","w")
>>> f.write("Reescribo la primera línea\n")
27
>>> f.write("Y también la segunda\n")
21
>>> f.close()
```

  - Contenido de `fichero.txt`:

```
Reescribo la primera línea
Y también la segunda
```

Clases

```
import math

class Punto(object):

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distancia_al_origen(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, otro):
        return self.x == otro.x and self.y == otro.y

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Clases

```
>>> p1=Punto()
>>> p2=Punto(3,4)
>>> p1
<__main__.Punto object at 0x75e510>
>>> str(p1)
'(0, 0)'
>>> str(p2)
'(3, 4)'
>>> p1.x
0
>>> p2.y
4
>>> p1 == p2
False
>>> p2.distancia_al_origen()
5.0
>>> p1.x=3
>>> str(p1)
'(3, 0)'
>>> p1.y=1
>>> p1.distancia_al_origen()
3.162277601683795
```

Clases: observaciones importantes

- Clases *vs* objetos (instancias)
- Atributos: datos y métodos
- La clase `object`
- El parámetro `self`
- El constructor `__init__`
- Métodos especiales: `__init__`, `__str__`, `__eq__`,...

Herencia en clases

```
class Circulo(Punto):

    def __init__(self, radio, x=0, y=0):
        super().__init__(x, y)
        self.radio = radio

    def distancia_del_borde_al_origen(self):
        return abs(self.distancia_al_origen() - self.radio)

    def area(self):
        return math.pi * (self.radio**2)

    def circunferencia(self):
        return 2 * math.pi * self.radio

    def __eq__(self, otro):
        return self.radio == (otro.radio and
                              super().__eq__(otro,self))

    def __str__(self):
        return "Círculo({0.radio!r}, {0.x!r}, {0.y!r})".format(self)
```

Herencia en clases

- `Circulo` es una *especialización* de `Punto`:
  - Hereda los atributos de datos `x` e `y` y el método `distancia_al_origen`.
  - Reimplementa `__init__`, `__str__`, `__eq__`
  - Introduce el nuevo atributo de dato `radio` y los métodos `distancia_del_borde_al_origen`, `area` y `circunferencia`
- Sesión:

```
>>> p=Punto(3, 4)
>>> c=Circulo(1,3,4)
>>> str(p)
'(3, 4)'
>>> str(c)
'Circulo(1, 3, 4)'
>>> p.distancia_al_origen()
5.0
>>> c.distancia_al_origen()
5.0
>>> c.distancia_del_borde_al_origen()
4.0
```

Y mucho más ...

- Más métodos y operaciones
- Otros tipos de datos: decimales, tuplas con nombre, conjuntos inmutables, ...
- Decoraciones
- Generadores definidos por el usuario
- Paquetes y espacios de nombres
- Documentación, pruebas, depurado de programas

Biblioteca estándar (*batteries included*)

- Interacción con el sistema operativo, medidas de eficiencia
- Comodines para los nombres ficheros, compresión de datos
- Argumentos a través de línea de comando, fecha y hora
- Manejo de errores, de cadenas, control de calidad
- Operaciones matemáticas
- Programación en Internet, XML
- ...

Estilo (I)

- Sigue la guía de estilo de Python al escribir tus programas: *Style Guide for Python Code*
- Utilizar 4 espacios para el sangrado
- Una línea no debe contener más de 79 caracteres
- Separar definiciones de funciones, clases y bloques de código con líneas en blanco
- Líneas de comentario independientes
- Incluir documentación en las definiciones

Estilo (II)

- Incluir entre espacios los operadores, ponerlos tras las comas, pero no con los paréntesis: `a = f(2, 3) + g(6)`
- Utiliza `CamelCase` para nombrar las clases y `minúsculas_y_guiones_bajos` para las funciones y métodos. Usa `self` para el primer argumento del método de una clase
- Utiliza texto plano (`ASCII`) o, si es estrictamente necesario, `utf-8`
- Utiliza sólo caracteres `ASCII` para los identificadores

Bibliografía (ver las referencias en la web)

- *The Python tutorial*
- *The Python Language Reference*
- *The Python Standard Library*
- *Python programming: An Introduction to Computer Science (2nd ed.)* J. Zelle (Franklin, Beedle & Ass. Inc.)
- *Programación en Python 3*. M. Summerfield (Anaya Multimedia, 2009)
- Y muchas más...