Introducción a Python

Javier Béjar

ECSDI 2017/2018

Facultat d'Informàtica de Barcelona, UPC

Extraido de material de Nguyen Duc Minh Khoi y Nowel Strite

Introducción

Python

- Lenguaje de alto nivel
- Énfasis en su legibilidad
- La indentación marca los bloques
- Es multiparadigma: OO, imperativo, funcional, procedural, reflexivo
- Es de tipado dinámico (sin declaraciones de variables) y tiene gestión de memoria automática (sin apuntadores)
- Lenguaje de scripting / interpretado
- Multiplataforma

Python

- Énfasis en: Calidad del software, productividad, portabilidad
- Amplia librería estándar y muchas librerías de soporte (amplia base de desarrolladores)
- Integración con otros lenguajes (C, C++, java, ...)
- Desventajas: no siempre tan rápido como lenguajes compilados
- Lenguaje de propósito general: Sistema, GUI, Bases de datos, Prototipado Rápido, Web, Computación Numérica, Juegos, Inteligencia Artificial, ...
- Dos sabores: Python 2, Python 3

How Python program runs?

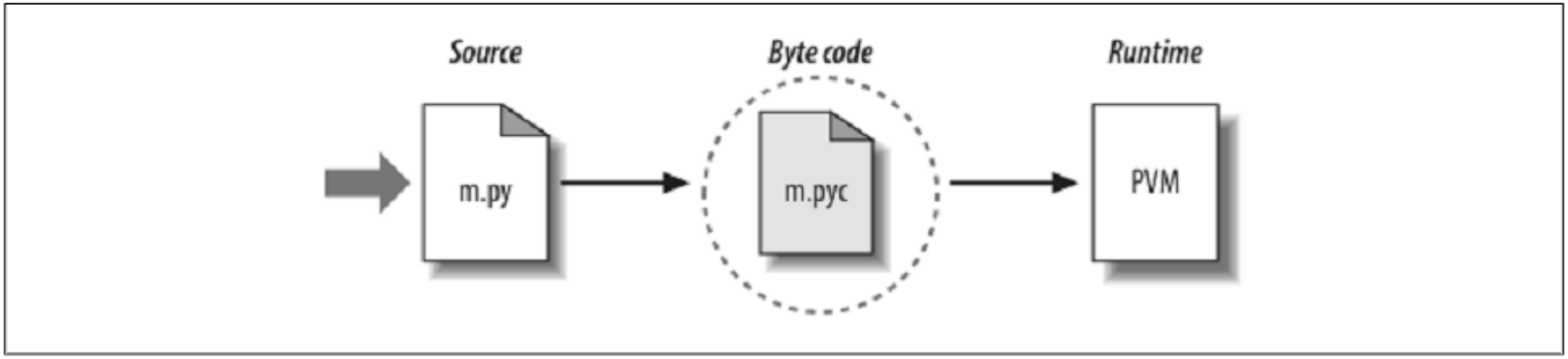

Figure 2-2. Python's traditional runtime execution model: source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.

Notice: pure Python code runs at speeds somewhere between those of a traditional compiled language and a traditional interpreted language

Hello World

```
#!/usr/bin/env python
print "Hello World!"
```

Python: Indentación

- Los lenguajes no suelen preocuparse por la indentación
- Las personas sí, agrupamos cosas similares

```
/* Bogus C code */
if (foo)
    if (bar)
        baz(foo, bar);
else
    qux();
```

```
# Python code
if foo:
    if bar:
        baz(foo, bar)
    else:
        qux()
```

Python/Jupyter Notebooks

- Python permite trabajar con un interprete a través de una página de web (Notebooks)
- Una ventaja es que podemos generar documentos interactivo
- Tenéis un conjunto de notebooks en el repositorio de código de la asignatura, lo podéis bajar haciendo:

```
git clone https://github.com/bejar/ECSDI2017.git
```

- Colocaos en el directorio `Examples/Python`
- Ejecutad `jupyter notebook`

Tipos y operaciones

Tipos de datos: Numéricos

- Enteros y reales
- Complejos
- Decimales con precisión fija
- Racionales
- Booleanos
- Enteros con precisión arbitraria
- Librerías con otros tipos numéricos

- Operaciones aritméticas y comparaciones habituales (ver notebook)

Tipos de datos: Booleanos

- `True`, `False`
- Todo objeto python puede convertirse en booleano
- El `0` y cualquier estructura vacía es equivalente a`False`
- Todo lo demás es `True`

- El valor nulo se representa con `None`

- Podemos diferenciar entre igualdad (==, mismo valor) e identidad (`is`, objetos iguales)

- Operaciones booleanas habituales (ver notebook)

Estructuras de datos: Tuplas

- Colección ordenada de objetos arbitrarios
- Indexable (como un vector)
- Son objetos inmutables
- Tamaño fijo, contenido arbitrario, anidable
- Son objetos iterables
- Representado como un vector de referencias

- Operaciones predefinidas (ver notebook)

Estructuras de datos: Strings

- Simples o dobles comillas, Tres simples comillas permiten hacer string multílinea
- Son objetos inmutables (como las tuplas)
- Usan +, +=, indexación `[]`
- Tienen las operaciones típicas: `split`, `find`, `replace`, ...
- Operaciones predefinidas (ver notebook)

Estructuras de datos: Listas

- Colecciones ordenadas de objetos arbitrarios
- Indexables, mutables
- Longitud arbitraria, anidables
- Son objetos iterables
- Representado como un vector de referencias
- Tienen las operaciones típicas

- Operaciones predefinidas (ver notebook)

Estructuras de datos: Diccionarios

- Accesibles por clave (no por posicion)
- Cualquier objeto no mutable puede ser índice (numeros, strings, tuplas)
- Longitud variable, heterogéneos, anidables
- Son objetos iterables (recorre las claves en un orden arbitrario)
- Implementados como tablas de hash

- Operaciones predefinidas (ver notebook)

Files – common operations

Operation	Interpretation
`output = open(r'C:\spam', 'w')`	Create output file (`'w'` means write)
`input = open('data', 'r')`	Create input file (`'r'` means read)
`input = open('data')`	Same as prior line (`'r'` is the default)
`aString = input.read()`	Read entire file into a single string
`aString = input.read(N)`	Read up to next N characters (or bytes) into a string
`aString = input.readline()`	Read next line (including `\n` newline) into a string
`aList = input.readlines()`	Read entire file into list of line strings (with `\n`)
`output.write(aString)`	Write a string of characters (or bytes) into file
`output.writelines(aList)`	Write all line strings in a list into file
`output.close()`	Manual close (done for you when file is collected)
`output.flush()`	Flush output buffer to disk without closing

Objetos mutables y variables

- Los objetos mutables son referencias
- Las variables son referencias a objetos
- Eso quiere decir que tenemos que tener en cuenta lo mismo que cuando se trabaja con apuntadores
- Variables que tienen asignado el mismo objeto son afectadas cuando se cambia algún elemento a través de cualquiera

```
>>> a = [1, 2, 3]
>>> b = a
>>> a[1] = 22
>>> print b
[1, 22, 3]
```

Objetos mutables y copias

- Para evitar estos efectos la mayoría de los tipos tienen operaciones de copia
- Listas: `[:]` copia toda la lista (solo el primer nivel)
- Diccionarios: operación `copy` (solo el primer nivel y no los valores)
- El módulo `copy` provee dos operaciones `copy` (solo un nivel) `deepcopy` (copia recursiva de todos los objetos de la estructura)

Sintaxis y sentencias

Estructura

- Un programa python se compone de módulos
- Cada módulo está en un fichero
- Los módulos se pueden organizar jerárquicamente siguendo la estructura de directorios (como en java)
- Cada módulo se compone de sentencias (definiciones y expresiones)

Asignación

- La asignación (=) crea referencias a objetos (variables)
- Las variables se crean al hacer la asignación (si no existen)
- Para poder referenciarse han de haberse asignado primero
- La asignación es algo más flexible

```
a = b # normal
a, b = 1, 2          # con tupla
a, b = [1, 2]        # con lista
a,b,c,d = 'abcd'     # con string
a = b = 3            # multiple
```

Sentencias: Condicional IF

- Condicional IF (atención a la indentación)

```
if <condicion>:
    <sentencias>
elif <condicion>:
    <sentencias>
else:
    <sentencias>
```

- IF ternario como el de C

```
a = X if A else Y
```

- Ver notebook

Sentencias: Bucle FOR

- Bucle FOR

```
for <vars> in <iterable>
    <sentencias>
```

- Incluye la posibilidad de salir incondicionalmente (`break`), saltar a la siguiente iteración (`continue`)
- La función `range(i,f,p)` permite generar una secuencia de numeros
- La función `zip` permite fusionar n listas e iterarlas a la vez
- Lo podemos usar para cualquier objeto que sea iterable

Sentencias: Bucle WHILE

- Bucle WHILE

```
while <condicion>:
    <sentencias>
```

- Es más lento que el `for` si iteramos usando un contador o un objeto iterable

List comprehensions

- La sentencia `for` permite generar listas a partir de objetos iterables
- Son más rápidas que hacer un bucle `for` y crear la lista a mano

```
# List comprehension
l = [x+x for x in range(100)]


# Bucle for
l = []
for x in range(100):
    l += [x+x]
```

Funciones

Scopes – the LEGB rules

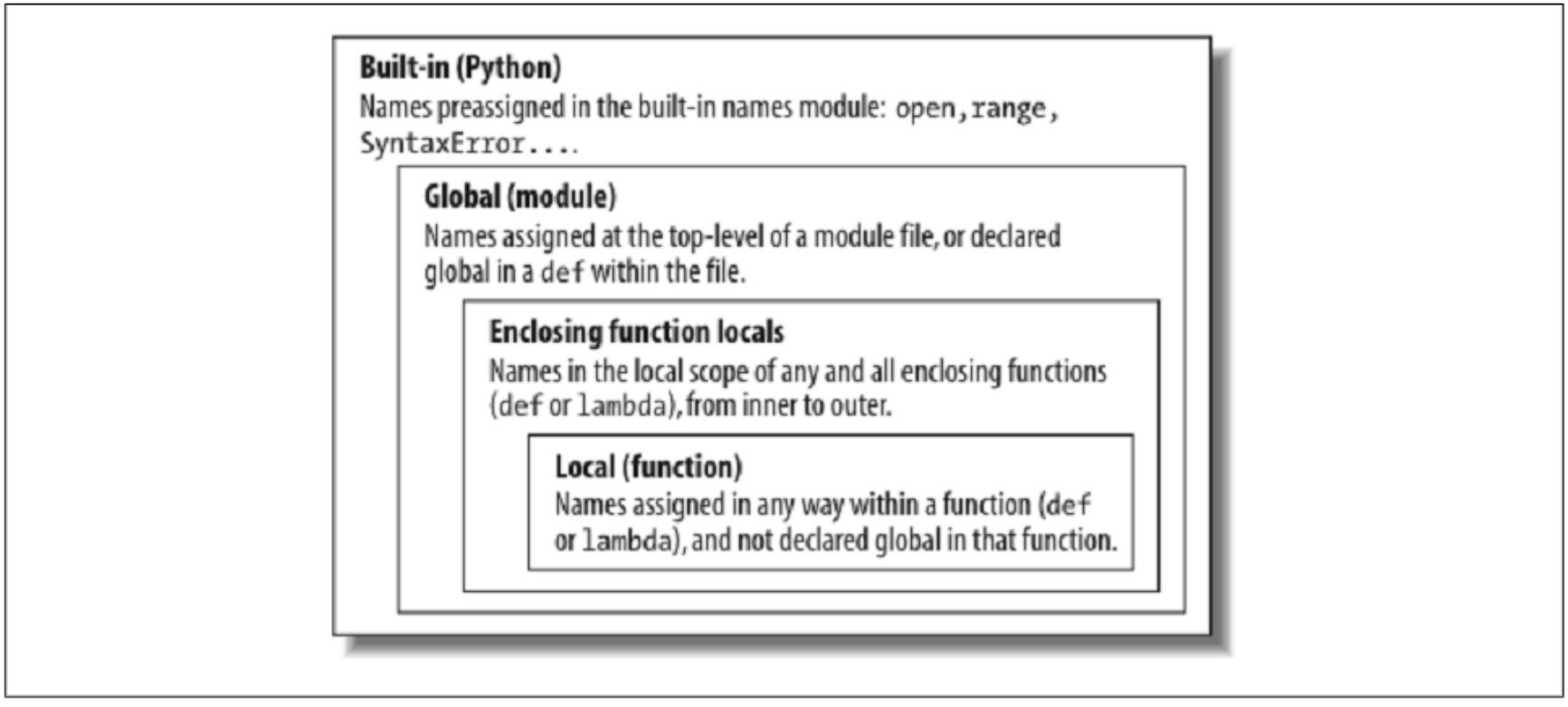

Figure 17-1. The LEGB scope lookup rule. When a variable is referenced, Python searches for it in this order: in the local scope, in any enclosing functions' local scopes, in the global scope, and finally in the built-in scope. The first occurrence wins. The place in your code where a variable is assigned usually determines its scope. In Python 3, nonlocal declarations can also force names to be mapped to enclosing function scopes, whether assigned or not.

Arguments – Matching Modes

Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name
`func(*sequence)`	Caller	Pass all objects in sequence as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in `dict` as individual keyword arguments
`def func(name)`	Function	Normal argument: matches any passed value by position or name
`def func(name=value)`	Function	Default argument value, if not passed in the call
`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary
`def func(*args, name)`	Function	Arguments that must be passed by keyword only in calls (3.0)
`def func(*, name=value)`		

- Keyword-only arguments: arguments that must be passed by keyword only and will never be filled in by a positional argument.

Funciones

```
def nomfuncion (parfijos, parnombre=valdefecto):
    """
    Documentacion de la funcion
    """
    <sentencias>
    return <valores>
```

- Ver notebook

Módulos

Módulos

- `import` *`modulo`*: acceder a todas las definiciones del módulo (usamos el nombre del modulo para acceder a los nombres)
- `import` *`modulo`* `as` *`abrev`*: usar el modulo con un nuevo nombre (usualmente mas corto)
- `from` *`modulo`* `import` *`name`*: importar el nombre para usarlo sin tener que referenciar el módulo
- `from` *`modulo`* `import` *`name`* `as <altname>`: importar el nombre renombrándolo
- `from` *`modulo`* `import *`: importar todos los nombres del módulo

Orientación a objetos

Orientación a objetos

- Herencia Múltiple
- Todo es público
- Variable de autoreferencia `self`
- Los atributos privados empiezan (pero no acaban) en doble subrayado (aún podemos acceder a ellos)
- Los métodos especiales de clase empiezan y acaban en doble subrayado:
  - `__init__`: constructor
  - `__doc__`: documentacion de la clase
  - `__str__`: representación de un objeto como string
  - Todos los métodos sobrecargables que corresponden a sintaxis python (ver transparencias finales)

Clases

```
class nclase(cpadre):
    pass

# Todos los valores asignados en declaracion
# deben ser de tipo innmutable
class nclase(cpadre):
    attrib1 = None
    attrib2 = 1234
```

Clases

```
class nclase(cpadre):
    attrib1 = False

    # Constructor (siempre hay unicamente uno)
    def __init__(self, val, parop=False):
        # Creamos atributo al crear objetos
        self.attrib2 = val
        self.attrib3 = parop

    # funcion de clase
    def classfunction(self, param)
        return self.atrib1 or param
```

Instanciación

```
# Creamos una instancia con el constructor
a = clase(param1, parop=True)

# Acceso directo a los atributos
print a.attrib1

# Acceso directo a los metodos
b = a.classfunction(True)

# Creacion dinamica de atributos
a.nuevoatrib = 33
```

Class Coding Details – operator overloading

- Common operator overloading method:

Method	Implements	Called for
`__init__`	Constructor	Object creation: `X = Class(args)`
`__del__`	Destructor	Object reclamation of `X`
`__add__`	Operator +	`X + Y`, `X += Y` if no `__iadd__`
`__or__`	Operator | (bitwise OR)	`X | Y`, `X |= Y` if no `__ior__`
`__repr__`, `__str__`	Printing, conversions	`print(X)`, `repr(X)`, `str(X)`
`__call__`	Function calls	`X(*args, **kargs)`
`__getattr__`	Attribute fetch	`X.undefined`
`__setattr__`	Attribute assignment	`X.any = value`
`__delattr__`	Attribute deletion	`del X.any`
`__getattribute__`	Attribute fetch	`X.any`
`__getitem__`	Indexing, slicing, iteration	`X[key]`, `X[i:j]`, for loops and other iterations if no `__iter__`
`__setitem__`	Index and slice assignment	`X[key] = value`, `X[i:j] = sequence`
`__delitem__`	Index and slice deletion	`del X[key]`, `del X[i:j]`

Class Coding Details – operator overloading ©

Method	Implements	Called for
`__len__`	Length	`len(X)`, truth tests if no `__bool__`
`__bool__`	Boolean tests	`bool(X)`, truth tests (named `__nonzero__` in 2.6)
`__lt__, __gt__, __le__, __ge__, __eq__, __ne__`	Comparisons	`X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y` (or else `__cmp__` in 2.6 only)
`__radd__`	Right-side operators	`Other + X`
`__iadd__`	In-place augmented operators	`X += Y` (or else `__add__`)
`__iter__, __next__`	Iteration contexts	`I=iter(X)`, `next(I)`; `for` loops, `in` if no `__contains__`, all comprehensions, `map(F,X)`, others (`__next__` is named `next` in 2.6)
`__contains__`	Membership test	`item in X` (any iterable)
`__index__`	Integer value	`hex(X)`, `bin(X)`, `oct(X)`, `O[X]`, `O[X:]` (replaces Python 2 `__oct__`, `__hex__`)
`__enter__, __exit__`	Context manager (Chapter 33)	`with obj as var:`
`__get__, __set__, __delete__`	Descriptor attributes (Chapter 37)	`X.attr, X.attr = value, del X.attr`
`__new__`	Creation (Chapter 39)	Object creation, before `__init__`