Lezione 3

Introduzione alla programmazione con Python

Mauro Ceccanti[‡] and Alberto Paoluzzi[†]

[†]Dip. Informatica e Automazione – Università "Roma Tre"
[‡]Dip. Medicina Clinica – Università "La Sapienza"

Contents

Reference sources

Main references

- Campbell et al. [2009]
- Schuerer et al. [2008]
- Schuerer and Letondal [2008]

Useful readings

- Chapman [2003]
- van Rossum [2002]
- van Rossum [1997]

Contents

Why Python ?

- It is free and well documented
- It runs everywhere
- It has a clean syntax
- It is relevant. Thousands of companies and academic research groups use it every day;
- It is well supported by tools

What is Python? Executive Summary

Extracted from [van Rossum, 2002]

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics

- high-level data structures, with dynamic typing, make it very attractive for Rapid Application Development
- simple, easy to learn syntax emphasizes readability
- supports modules and packages, which encourages program modularity and code reuse
- available free for all major platforms

What is Python? increased productivity

Extracted from [van Rossum, 2002]

- Since there is no compilation step, the edit-test-debug cycle is incredibly fast
- Debugging Python programs is easy: a bug or bad input will never cause a segmentation fault
- Instead, when the interpreter discovers an error, it raises an exception
- When the program doesn't catch the exception, the interpreter prints a stack trace
- A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on
- The debugger is written in Python itself, testifying to Python's introspective power
- On the other hand, often the quickest way to debug a program is to add a few print statements to the source: the fast edit-test-debug cycle makes this simple approach very effective

Comparing Python to Other Languages

Extracted from [van Rossum, 1997]

- [Campbell et al., 2009]
- see Campbell et al. [2009]

Installing

on Mac OS X and Windows

- The suggested book [Campbell et al., 2009] on Python programming is

  Practical Programming:

  An Introduction to Computer Science Using Python

- Basic install
  (Python + NumPy + Wing IDE 101)

  http://www.cdf.toronto.edu/∼csc108h/fall/python.shtml

Contents

Numerical Python

NumPy is the fundamental package needed for scientific computing with Python

It contains:

- a powerful N-dimensional array object
- sophisticated broadcasting functions
- basic linear algebra functions
- basic Fourier transforms
- sophisticated random number capabilities
- tools for integrating Fortran code.
- tools for integrating C/C++ code.

NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined.

This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Scientific Python

SciPy: Scientific Library for Python

- open-source software for mathematics, science, and engineering
- It is also the name of a popular conference on scientific programming with Python
- The SciPy library depends on NumPy
- The SciPy library provides many user-friendly and efficient numerical routines

SciPy – Download

Scientific Library for Python

- Official source and binary releases of NumPy and SciPy
- A better alternative: SciPy Superpack for Python
- Biology packages
- Cookbook: this page hosts "recipes", or worked examples of commonly-done tasks.

BioPython

Python tools for computational molecular biology

- Biopython is a set of freely available tools for biological computation written in Python
- It is a distributed collaborative effort to develop Python libraries and applications
- Biopython aims to address the needs of current and future work in bioinformatics

Useful step-by-step instructions are in Biopython Installation

Contents

Python comments

Comments are to clarify code and are not interpreted by Python

- Comments start with the *hash* character, #, and *extend to the end* of the line
- A comment may appear at the start of a line or following whitespace or code, but *not within a string* literal[1]

```
# this is the first comment
SPAM = 1       # and this is the second comment
               # ... and now a third!
STRING = "# This is not a comment."
```

[1] *Literal* ≡ according with the letter of the scriptures; expression that returns itself by evaluation.

Using Python as a calculator

including comments

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

Contents

Variables and assignment

- 3.4. Declaring variables[2]

[2] from: "DIVE INTO PYTHON – Python from novice to pro", http://www.diveintopython.org/index.html

Using Python as a Calculator

Numbers

- The interpreter acts as a simple calculator: you can type an expression at it and it will write the value
- Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages
- parentheses can be used for grouping

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

Using Python as a Calculator

Numbers

- The equal sign ('=') is used to assign a value to a variable
- Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Using Python as a Calculator

Numbers

- A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

Using Python as a Calculator

Numbers

- Variables must be “defined” (assigned a value) before they can be used, or an error will occur:

```
>>> # try to access an undefined variable
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Using Python as a Calculator

Numbers

- There is full support for floating point
- operators with mixed type operands convert the integer operand to floating point

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Using Python as a Calculator

Numbers

- Complex numbers are also supported
- imaginary numbers are written with a suffix of j or J
- Complex numbers with a nonzero real component are written as (real+imagj), or can be created with the complex(real, imag) function.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Using Python as a Calculator

Numbers

- Complex numbers are always represented as two floating point numbers, the real and imaginary part
- To extract these parts from a complex number z, use z.real and z.imag.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Using Python as a Calculator

Numbers

- The conversion functions to floating point and integer (float(), int() and long()) don't work for complex numbers
- there is no one correct way to convert a complex number to a real number
- Use abs(z) to get its magnitude (as a float) or z.real to get its real part.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

Using Python as a Calculator

Numbers

- In interactive mode, the last printed expression is assigned to the variable _
- This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
>>>
```

- This variable should be treated as read-only by the user
- Don't explicitly assign a value to it
- you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

Contents

Strings

- Besides numbers, Python can also manipulate strings, which can be expressed in several ways
- They can be enclosed in single quotes or double quotes:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Strings

- String literals can span multiple lines in several ways
- Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
 significant."

print hello
```

- newlines still need to be embedded in the string using \n
- the newline following the trailing backslash is discarded
- This example would print the following:

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

Strings

- strings can be surrounded in a pair of matching triple-quotes: """ or '''
- End of lines do not need to be escaped when using triple-quotes, but they will be included in the string

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

- produces the following output:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Strings

- If we make the string literal a “raw” string, sequences are not converted to newlines, but the backslash at the end of the line, and the newline character in the source, are both included in the string as data.
- Thus, the example:

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print hello
```

- would print:

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

Strings

- Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

- Two string literals next to each other are automatically concatenated
- the first line above could also have been written word = ’Help’ ’A’
- this only works with two literals, not with arbitrary string expressions

Strings

Strings can be subscripted (indexed)

- the first character has index 0
- there is no separate character type
- a character is simply a string of size one
- substrings can be specified with the slice notation: two indices separated by a colon.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Strings

- Slice indices have useful defaults
- an omitted first index defaults to zero
- an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]     # The first two characters
'He'
>>> word[2:]     # Everything except the first two characters
'lpA'
```

Strings

- Unlike a C string
- Python strings cannot be changed
- Assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support slice assignment
```

Strings

- However, creating a new string with the combined content is easy and efficient:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

- Here's a useful invariant of slice operations: s[:i] + s[i:] equals s.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Strings

- Degenerate slice indices are handled gracefully:
- an index that is too large is replaced by the string size
- an upper bound smaller than the lower bound returns an empty string.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Strings

- Indices may be negative numbers, to start counting from the right:

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # Everything except the last two characters
'Hel'
```

- But note that -0 is really the same as 0, so it does not count from the right!

```
>>> word[-0]     # (since -0 equals 0)
'H'
```

Strings

- think of the indices as pointing between characters
- with the left edge of the first character numbered 0
- Then the right edge of the last character of a string of n characters has index n
- The slice from i to j consists of all characters between the edges labeled i and j

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Strings

- For non-negative indices, the length of a slice is the difference of the indices
- if both are within bounds
- For example the length of word[1:3] is 2.

The built-in function len() returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Sequence Types

str, unicode, list, tuple, buffer, xrange

strings String literals are written in single or double quotes: 'xyzzy', "frobozz".

Unicode strings specified using a preceding 'u' character: u'abc', u"def"

lists constructed with square brackets, separating items with commas: [a, b, c]

tuples Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, such as a, b, c or (). A single item tuple must have a trailing comma, such as (d,).

buffers created by calling the builtin function buffer(). They don't support concatenation or repetition

xrange objects. Created by calling the builtin function buffer(). They don't support concatenation or repetition

Sequence Types

str, unicode, list, tuple, buffer, xrange

For other containers see the built-in

dict class

set class

collections module.

Contents

User input and formatted printing

- http://docs.python.org/tutorial/inputoutput.html

User input and formatted printing

EXAMPLE

- file input/output
  - bioinf/sw/viewer/wireframe.py
  - bioinf/sw/viewer/backbone.py
  - bioinf/sw/viewer/pdb.py
  - bioinf/sw/viewer/basic.py
  - bioinf/sw/viewer/3ETA.pdb
  - bioinf/sw/viewer/2ACY.pdb
  - bioinf/sw/viewer/1AQU.pdb
  - bioinf/sw/viewer/FL06.py

Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson. *Practical Programming: An Introduction to Computer Science Using Python*. The Pragmatic Bookshelf, Raleigh, North Carolina, USA, 2009.

Brad Chapman. Biopython and why you should love it. http://www.biopython.org/DIST/docs/presentations/biopython.pdf, 2003.

Katja Schuerer and Catherine Letondal. python course in bioinformatics. Technical report, Pasteur Institute, 2008.

Katja Schuerer, Corinne Maufrais, Catherine Letondal, Eric Deveaud, and Marie-Agnes Petit. introduction to programming using python,programming course for biologists at the pasteur institute. Technical report, Pasteur Institute, 2008.

Guido van Rossum. Comparing Python to Other Languages. http://www.python.org/doc/essays/comparisons/, 1997.

Guido van Rossum. What is Python? Executive Summary. http://www.python.org/doc/essays/blurb/, 2002.