Introduction to Programming in Python (1)

Steve Renals
s.renals@ed.ac.uk

ICL — 25 September 2006

- Introduction
  - Overview
  - Running programs
  - Modules
- Basic object types
  - Numbers and variables
  - Strings
  - Lists
  - Dictionaries
- Summary

Python books

- Mark Lutz and David Ascher (2004). *Learning Python*, 2nd Edition, O'Reilly.
- Allen Downey, Jeff Elkner and Chris Meyers (2001), *How to Think Like a Computer Scientist: Learning with Python*, Green Tea Press.
  http://www.greenteapress.com/thinkpython/
- David Beazley (2006), *Python Essential Reference*, 3rd edition, Developer's Library, Sams Publishing.
- Mark Lutz (2002). *Python Pocket Reference*, 2nd Edition, O'Reilly.
- Mark Lutz (2006). *Programming Python*, 3rd Edition, O'Reilly.
- Alex Martelli (2006). *Python in a Nutshell*

Python features

- Free, portable, powerful

Python features

- Free, portable, powerful
- Easy to mix in components from other languages

Python features

- Free, portable, powerful
- Easy to mix in components from other languages
- Object-oriented (including operator overloading, polymorphism, multiple inheritance)

Python features

- Free, portable, powerful
- Easy to mix in components from other languages
- Object-oriented (including operator overloading, polymorphism, multiple inheritance)
- Easy to use, easy to learn, easy to understand

Python features

- Free, portable, powerful
- Easy to mix in components from other languages
- Object-oriented (including operator overloading, polymorphism, multiple inheritance)
- Easy to use, easy to learn, easy to understand
- NLTK-Lite (Natural Language ToolKit) is a Python package that we will use in ICL

(*Learning Python*, chapter 1)

Using Python interactively

The easiest way to use Python initially is interactively:

```
% python
>>> print ’ICL’
ICL
>>> print 3*4
12
>>> print 2**16
65536
>>> myname = ’Steve’
>>> myname
’Steve’
```

(*Learning Python*, chapter 3)

Using Python interactively

The easiest way to use Python initially is interactively:

```
% python
>>> print 'ICL'
ICL
>>> print 3*4
12
>>> print 2**16
65536
>>> myname = 'Steve'
>>> myname
'Steve'
```

(*Learning Python*, chapter 3)
Can also use the **IDLE** environment: `idle`
May editors/IDEs support python: (X)Emacs, Textmate, Komodo, …

Modules

- To save code you need to write it in files
- *Module*: a text file containing Python code

Modules

- To save code you need to write it in files
- *Module*: a text file containing Python code

Example: write the following to file `foo.py`:

```
print 25*3 # multiply by 3
print 'ICL ' + 'lecture 2' # concatenate strings using +
myname = 'Steve'
```

(No leading spaces!)

Modules

- To save code you need to write it in files
- *Module*: a text file containing Python code

Example: write the following to file `foo.py`:

```
print 25*3 # multiply by 3
print ’ICL ’ + ’lecture 2’ # concatenate strings using +
myname = ’Steve’
```

(No leading spaces!) Then run it as follows:

```
% python foo.py
75
ICL lecture 2
%
```

Importing modules

Every file ending in `.py` is a Python module.

Importing modules

Every file ending in `.py` is a Python module.
Modules can contain attributes such as functions,

Importing modules

Every file ending in `.py` is a Python module.
Modules can contain attributes such as functions,
We can *import* this module into Python:

```
% python
>>> import foo
75
ICL lecture 2
>>> foo.myname
'Steve'
```

Executable scripts

On unix/linux can make normal Python text files *executable*:

- **The first line is special** beginning with `#!`
- **File has executable privileges** (`chmod +x file.py`)

Executable scripts

On unix/linux can make normal Python text files *executable*:

- **The first line is special** beginning with `#!`
- **File has executable privileges** (`chmod +x file.py`)

Example: write the following to file `foo.py`:

```
#!/usr/bin/python
print 25*3 # multiply by 3
print 'ICL ' + 'lecture 2' # concatenate strings using +
myname = 'Steve'
```

Executable scripts

On unix/linux can make normal Python text files *executable*:

- **The first line is special** beginning with `#!`
- **File has executable privileges** (`chmod +x file.py`)

Example: write the following to file `foo.py`:

```
#!/usr/bin/python
print 25*3 # multiply by 3
print 'ICL ' + 'lecture 2' # concatenate strings using +
myname = 'Steve'
```

```
% chmod +x foo.py
% foo.py
75
ICLlecture 2
%
```

Module reloads

- Importing is expensive—after the first import of a module, repeated imports of a module have no effect (even if you have edited it)
- Use `reload` for force Python to rerun the file again:

Module reloads

- Importing is expensive—after the first import of a module, repeated imports of a module have no effect (even if you have edited it)
- Use reload for force Python to rerun the file again:

```
>>> import foo
75
ICL lecture 2
```

Module reloads

- Importing is expensive—after the first import of a module, repeated imports of a module have no effect (even if you have edited it)
- Use `reload` for force Python to rerun the file again:

```
>>> import foo
75
ICL lecture 2
```

Re-edit foo.py to print 25*4 and reload

```
>>> reload(foo)
100
ICL lecture 2
<module ’foo’ from ’foo.py’>
```

Module attributes

Let bar.py contain the following:

```
university = ‘Edinburgh‘
school = ‘Informatics‘
```

Module attributes

Let bar.py contain the following:

```
university = ‘Edinburgh‘
school = ‘Informatics‘
```

```
>>> import bar
>>> print bar.school
Informatics
```

Module attributes

Let bar.py contain the following:

```
university = ‘Edinburgh‘
school = ‘Informatics‘
```

```
>>> import bar
>>> print bar.school
Informatics
```

```
>>> from bar import school
>>> print school
Informatics
```

Module attributes

Let `bar.py` contain the following:

```
university = ‘Edinburgh‘
school = ‘Informatics‘
```

```
>>> import bar
>>> print bar.school
Informatics
```

```
>>> from bar import school
>>> print school
Informatics
```

```
>>> from bar import *
>>> print university
Edinburgh
```

Module attributes

Let `bar.py` contain the following:

```
university = ‘Edinburgh‘
school = ‘Informatics‘
```

```
>>> import bar
>>> print bar.school
Informatics
```

```
>>> from bar import school
>>> print school
Informatics
```

```
>>> from bar import *
>>> print university
Edinburgh
```

`from` copies named *attributes* from a module, so they are *variables* in the recipient.

Python program structure

- **Programs** are composed of *modules*

Python program structure

- **Programs** are composed of *modules*
- **Modules** contain *statements*

Python program structure

- **Programs** are composed of *modules*
- **Modules** contain *statements*
- **Statements** contain *expressions*

Python program structure

- **Programs** are composed of *modules*
- **Modules** contain *statements*
- **Statements** contain *expressions*
- **Expressions** create and process *objects*

Python program structure

- **Programs** are composed of *modules*
- **Modules** contain *statements*
- **Statements** contain *expressions*
- **Expressions** create and process *objects*

(Statements include: variable assignment, function calls, control flow, module access, building functions, building objects, print) (*Learning Python*, chapter 4)

Python's built-in objects

1. Numbers: integer, floating point, complex
2. Strings
3. Lists
4. Dictionaries
5. Tuples
6. Files

Numbers (and variables)

- Usual number operators, eg: `+`, `*`, `/`, `**`, `and`, `&`
- Usual operator precedence:
  `A * B + C * D` = `(A * B) + (C * D)`
  (use parens for clarity and to reduce bugs)

Numbers (and variables)

- Usual number operators, eg: `+, *, /, **, and, &`
- Usual operator precedence:
  `A * B + C * D` = `(A * B) + (C * D)`
  (use parens for clarity and to reduce bugs)
- Useful packages: `math`, `random`
- Serious users: `numeric`, `numarray`

Numbers (and variables)

- Usual number operators, eg: `+`, `*`, `/`, `**`, `and`, `&`
- Usual operator precedence:
  `A * B + C * D = (A * B) + (C * D)`
  (use parens for clarity and to reduce bugs)
- Useful packages: `math`, `random`
- Serious users: `numeric`, `numarray`
- Variables
  - created when first assigned a value
  - replaced with their values when used in expressions
  - must be assigned before use
  - no need to declare ahead of time

Strings

- String handling in Python is easy and powerful (unlike C, C++, Java)

(*Learning Python*, chapter 5)

Strings

- String handling in Python is easy and powerful (unlike C, C++, Java)
- Strings may be written using single quotes:
  `'This is a Python string'`
- or double quotes
  `"and so is this"`

(*Learning Python*, chapter 5)

Strings

- String handling in Python is easy and powerful (unlike C, C++, Java)
- Strings may be written using single quotes:
  `'This is a Python string'`
- or double quotes
  `"and so is this"`
- They are the same, it just makes it easy to include single (double) quotes:
  `'He said "what?"'`, `"He's here"`

(*Learning Python*, chapter 5)

Backslash in strings

- Backslash \ can be used to escape (protect) certain non-printing or special characters
- \n is newline, \t is tab

Backslash in strings

- Backslash \ can be used to escape (protect) certain non-printing or special characters
- \n is newline, \t is tab

```
>>> s = 'Name\tAge\nJohn\t21\nBob\t44'
>>> print s
Name    Age
John    21
Bob     44
>>> t = '"Mary\'s"'
>>> print t
"Mary's"
```

Triple quote

Use a triple quote (`"""` or `’’’`) for a string over several lines:

```
>>> s = """this is
... a string
... over 3 lines"""
>>> t = ’’’so
... is
... this’’’
>>> print s
this is
a string
over 3 lines
>>> print t
so
is
this
```

String operations

- Concatenation (+)
- Length (`len`)
- Repetition (*)
- Indexing and slicing (`[]`)

String operations

- Concatenation (+)
- Length (len)
- Repetition (*)
- Indexing and slicing ([])

```
s = ’computational’
t = ’linguistics’
cl = s + ’ ’ + t # ’computational linguistics’
l = len(cl) # 25
u = ’-’ * 6 # ’------’
c = s[3] # p
x = cl[11:16] # ’al li’
y = cl[20:] # ’stics’
z = cl[:-1] # ’computational linguistic’
```

String methods

- Methods are functions applied associated with objects
- String methods allow strings to be processed in a more sophisticated way

String methods

- Methods are functions applied associated with objects
- String methods allow strings to be processed in a more sophisticated way

```
s = ’example’
s = s.capitalize() # ’Example’
t = s.lower() # ’example’
flag = s.isalpha() # True
s = s.replace(’amp’, ’M’) # ’exMle’
i = t.find(’xa’) # 1
n = t.count(’e’) # 2
```

Lists in Python

- *Ordered* collections of arbitrary objects
- Accessed by *indexing* based on offset
- Variable length, heterogenous (can contain any type of object), nestable
- *Mutable* (can change the elements, unlike strings)

Lists in Python

- *Ordered* collections of arbitrary objects
- Accessed by *indexing* based on offset
- Variable length, heterogenous (can contain any type of object), nestable
- *Mutable* (can change the elements, unlike strings)

```
>>> s = ['a', 'b', 'c']
>>> t = [1, 2, 3]
>>> u = s + t # ['a', 'b', 'c', 1, 2, 3]
>>> n = len(u) # 6
>>> for x in s:
...     print x
...
a
b
c
```

Indexing and slicing lists

- Indexing and slicing work like strings
- Indexing returns the object element
- Slicing returns a list
- Can use indexing and slicing to change contents

Indexing and slicing lists

- Indexing and slicing work like strings
- Indexing returns the object element
- Slicing returns a list
- Can use indexing and slicing to change contents

```
l = [’a’, ’b’, ’c’, ’d’]
x = l[2] # ’c’
m = l[1:] # [’b’, ’c’, ’d’]
l[2] = ’z’ # [’a’, ’b’, ’z’, ’d’]
l[0:2] = [’x’, ’y’] # [’x’, ’y’, ’z’, ’d’]
```

(*Learning Python*, chapter 6)

List methods

- Lists also have some useful methods
- `append` adds an item to the list
- `extend` adds multiple items
- `sort` orders a list in place

List methods

- Lists also have some useful methods
- append adds an item to the list
- extend adds multiple items
- sort orders a list in place

```
l = ['x', 'y', 'z', 'd']
l.sort() # ['d', 'x', 'y', 'z']
l.append('q') # ['d', 'x', 'y', 'z', 'q']
l.extend(['r', 's']) # ['d', 'x', 'y', 'z', 'q', 'r', 's']
l.append(['v', 'w']) # ['d', 'x', 'y', 'z', 'q', 'r', 's', ['v', 'w']]
```

Dictionaries

Dictionaries are

- Addressed by *key*, not by offset
- *Unordered* collections of arbitrary objects
- Variable length, heterogenous (can contain any type of object), nestable
- *Mutable* (can change the elements, unlike strings)

Dictionaries

Dictionaries are

- Addressed by *key*, not by offset
- *Unordered* collections of arbitrary objects
- Variable length, heterogenous (can contain any type of object), nestable
- *Mutable* (can change the elements, unlike strings)

- Think of dictionaries as a set of key:value pairs
- Use a key to access its value

(*Learning Python*, chapter 7)

Dictionary example

```
level = {’icl’ : 9, ’nlssd’ : 11, ’inf2b’ : 8}
x = level[’nlssd’] # 11
n = len(level) # 3

flag = level.has_key(’inf2b’) # True
l = level.keys() # [’nlssd’, ’inf2b’, ’icl’]

level[’dil’] = 11 # {’dil’: 11, ’nlssd’: 11, ’inf2b’: 8, ’icl’: 9}
level[’icl’] = 10 # {’dil’: 11, ’nlssd’: 11, ’inf2b’: 8, ’icl’: 10}

l = level.items() # [(’dil’, 11), (’nlssd’, 11), (’inf2b’, 8), (’icl’,
10)]
l = level.values() # [11, 11, 8, 10]
```

Notes on dictionaries

- Sequence operations don't work: dictionaries are *mappings*, not sequences
- Dictionaries have a *set* of keys: only one value per key
- Assigning to a new key adds an entry
- Keys can be any immutable object, not just strings

Notes on dictionaries

- Sequence operations don't work: dictionaries are *mappings*, not sequences
- Dictionaries have a *set* of keys: only one value per key
- Assigning to a new key adds an entry
- Keys can be any immutable object, not just strings
- Dictionaries can be used as "records"
- Dictionaries can be used for sparse matrices

Summary

- Introduction to Python
- Python programs and modules
- Basic objects: numbers, strings, lists, dictionaries