Introduction to Python

Instructor: Yoann Pitarch (pitarch@irit.fr)

Overview

1. Introduction
2. Basics and Data Types
3. Control Structures
4. Functions
5. Others

1. Introduction

Brief History of Python

- Invented early 90's by Guido van Rossum
- Open sourced from the beginning
- A scripting language, but is much more
- Scalable, object oriented and functional from the beginning
- Used by Google
- **Increasingly popular in the data science world**
- Complementary to R

Some Important Statements

1. This teaching is about Python 2.7 (Python2.7 and 3.x are not compatible)
2. No object-oriented programming will be taught (only procedural)
3. There are 3 ways to learn a new programming language:
    1. The bad way: only reading these slides
    2. The good ways:
        1. Reading pieces of code again and again
        2. Programming, programming and again programming

Installing

- Python is pre-installed on most Unix systems, including Linux and MAC OS X
- The pre-installed version may not be the most recent one (2.7.13 and 3.5 as of Jan 16)
- Download from http://python.org/download/
- Python comes with a large library of standard modules
- There are several options for developing in Python
  - Using the command line
  - Using an IDE (Integrated Development Environment)
    - IDLE – works well with Windows
    - Spyder
    - Emacs with python-mode or your favorite text editor
    - Eclipse with Pydev (http://pydev.sourceforge.net/)
    - I personally use PyCharm

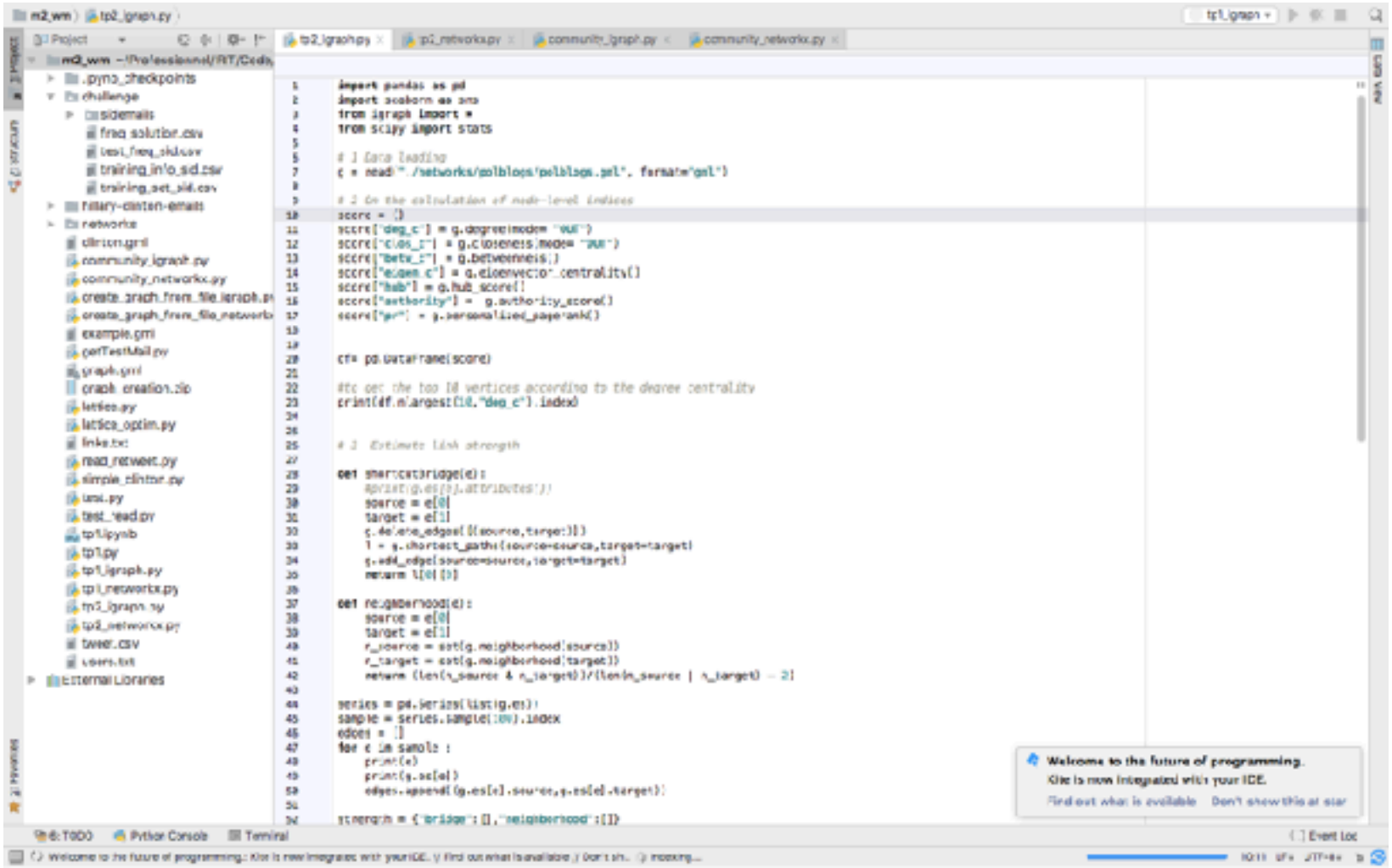
The Python Interpreter

- Python implementations offer both an interpreter and compiler
- Interactive interface to Python

```
503 yoann:~$ python
Python 2.7.13 (default, Dec 18 2016, 07:03:39)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> s = "This is the firstr string of the course"
>>> s
'This is the firstr string of the course'
>>> a = 4
>>> b = a * 2
>>> a,b
(4, 8)
>>> s[:10]
'This is th'
>>> s[-1]
'e'
>>> t = [1,"a", 3, "test"]
>>> t
[1, 'a', 3, 'test']
>>> print t[0]
1
>>> _
```

Example of an IDE interface

PyCharm



Don’t try to understand the code displayed above for now ;-)

Want More Information?

- http://python.org
  - Documentation, tutorials, beginner guide, core distribution, …
- Books
  - *Learning Python* by Mark Lutz
  - *Python Essential Reference* by David Beazley
  - *Python Cookbook*, ed. by Martelli, Ravenscroft and Ascher
  - (online at http://code.activestate.com/recipes/langs/python/)
  - http://wiki.python.org/moin/PythonBooks

Want More Information?

- Online
  - There are many many excellent free tutorials about Python
  - Have a look to
    - http://noeticforce.com/best-free-tutorials-to-learn-python-pdfs-ebooks-online-interactive
  - I personally recommend **Code Academy** (https://www.codecademy.com/learn/learn-python) if you are a beginner

Running Interactively on the Python interpreter

```
Last login: Fri Sep  1 06:45:38 on console
[501 yoann:~$ python
Python 2.7.13 (default, Dec 18 2016, 07:03:39)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> 3+3
6
[>>> a = 3 + 3
[>>> a
6
>>> _
```

- Python prompts with '>>>'.
- To exit Python (not Idle):
  - In Unix, type CONTROL-D
  - In Windows, type CONTROL-Z + <Enter>
  - Type exit()

Running Programs on UNIX

1. Write a program and save it with the extension **.py**

exemple1.py

```
1    a = 4
2    b = 2
3    c = a + b
4    print str(a) + "+" + str(b) + "=" + str(c)
```

2. Open a terminal and type `python` and the path of your Python program and press `<enter>`

```
[508 yoann:~$ python /Users/yoann/Professionnel/IRIT/Enseignements/Python-FOAD/code/exemple1.py
4+2=6
509 yoann:~$
```

3. The output/errors of your program should appear

Overview

Enough to Understand the Code

- Indentation matters to code meaning
  - Block structure indicated by indentation
- First assignment to a variable creates it
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.
- Assignment is = and comparison is ==
- For numbers + - * / % are as expected
  - Special use of + for string concatenation and % for string formatting (as in C's printf)
- Logical operators are words (and, or, not) not symbols
- The basic printing command is print

/!\ See https://www.tutorialspoint.com/python/python_basic_operators.htm for basics operators /!\

Whitespace

- Whitespace is meaningful in Python: especially indentation and placement of newlines
- Use a newline to end a line of code
  - Use `\` when must go to next line prematurely
- No braces `{}` to mark blocks of code, use ***consistent*** indentation instead
  - First line with ***less*** indentation is outside of the block
  - First line with ***more*** indentation starts a nested block
- Colons start of a new block in many constructs, e.g. function definitions, then clauses

Comments

- Start comments with #, rest of line is ignored
- Can include a "documentation string" as the first line of a new function or class you define
- Development environments, debugger, and other tools use it: it's good style to include one

```
def fact(n):
    """fact(n) assumes n is a positive
integer and returns factorial of n."""
assert(n>0)
    return 1 if n==1 else n*fact(n-1)
```

Assignment

- Binding a variable in Python means setting a name to hold a reference to some object
  - Assignment creates references, not copies
- Names in Python do not have an intrinsic type, objects have types
  - Python determines the type of the reference automatically based on what data is assigned to it
- You create a name the first time it appears on the left side of an assignment expression:

  x = 3
- A reference is deleted via garbage collection after any names bound to it have passed out of scope

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.
  - `bob  Bob  _bob  _2_bob_  bob_2  BoB`
- There are some reserved words that you cannot use as variable name:
  - `and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while`

Assignment

- You can assign to multiple names at the same time

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

- Assignments can be chained

```
>>> x, y = y, x
```

- This makes it easy to swap values

```
>>> a = b = x = 2
```

Accessing Non-Existent Name

Accessing a name before it's been properly created (by placing it on the left side of an assignment), raises an error

```
>>> y

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name 'y' is not defined
>>> y = 3
>>> y
3
```

Basic Datatypes (1)

- Integers (default for numbers)
  - z = 5 / 2 # Answer 2, integer division
- Floats
  - x = 3.456
- Strings
  - Can use “” or ‘’ to specify with “abc” == ‘abc’
  - Unmatched can occur within the string: “matt’s”
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them: “““a‘b“c”””

Basic Datatypes (2)

- Boolean
  - Can be either True or False
  - In this course, it is assumed you are familiar with boolean logic (e.g. True and False equals False)
- Most « things » have a logical value
  - Things that are evaluated as False
    - None
    - False
    - Zero of any numeric type: 0, 0L, 0.0[1]
    - Any empty sequence, for example, '',(),[][2]
    - An empty dictionary
  - All other values are evaluated as True

[1,2] Sequences and dictionaries are introduced in the next section

Sequence Types

1. Tuple: (‘john’, 32, [CMSC])
   - A simple ***immutable*** ordered sequence of items
   - Items can be of mixed types, including collection types
2. Strings: “John Smith”
   - **Immutable**
   - Conceptually very much like a tuple
3. List: [1, 2, ‘john’, (‘up’, ‘down’)]
   - **Mutable** ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- **Key difference:**
  - Tuples and strings are ***immutable***
  - Lists are ***mutable***
- The operations shown in this section can be applied to ***all*** sequence types

Sequence Types 1

- Define tuples using parentheses and commas

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Define lists using square brackets and commas

```
>>> li = ["abc", 34, 4.34, 23]
```

- Define strings using quotes (", ', or """)

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line string that uses triple quotes."""
```

Sequence Types 2

- Access individual members of a tuple, list, or string using square bracket “array” notation
- Note that all are 0 based…

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'
>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34
>>> st = "Hello World"
>>> st[1]    # Second character in string.
 'e'
```

Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0

```
>>> t[1]
'abc'
```

Negative index: count from right, starting with –1

```
>>> t[-3]
4.56
```

Slicing: return copy of a subset

- Return a copy of the container with a subset of the original members.
- Start copying at the first index, and stop copying ***before*** second.
- Negative indices count from end

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: return copy of a subset

- Omit first index to make copy starting from beginning of the container
- Omit second index to make copy starting at first index and going to end

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

```
>>> t[:2]
(23, 'abc')
```

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

[:] makes a ***copy*** of an entire sequence

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences

```
>>> l2 = l1 # Both refer to 1 ref,
            # changing one affects both
>>> l2 = l1[:] # Independent copies, two refs
```

The ‘in’ Operator

Boolean test whether a value is inside a container

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

Be careful: the ***in*** keyword is also used in the syntax of ***for loops*** and ***list comprehensions***

The + Operator

The + operator produces a ***new*** tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

The * Operator

- The * operator produces a ***new*** tuple, list, or string that "repeats" the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> "Hello" * 3
'HelloHelloHello'
```

List Comprehension

- A powerful and concise way to generate list
- Examples are better than words

```
>>>[x**2 for x in range(5)]
[0,1,4,9,16]
```

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Looping Through Sequences

Values only

```
>>> t=[x**2 for x in range(3)]
>>> for value in t:
...    print "value= "+value

value= 0
value= 1
value= 4
```

Index and values

```
>>> t=[x**2 for x in range(3)]
>>> for index,val in enumerate(t):
...    print "t["+index+"]= "+val

t[0]= 0
t[1]= 1
t[2]= 4
```

Mutability
Tuple vs. Lists

Lists are mutable

- We can change lists ***in place***.
- Name ***li*** still points to the same memory reference when we're done.

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

Tuples are immutable

- You can't change a tuple.
- You can make a fresh tuple and assign its reference to a previously used name.

  >>> t = (23, 'abc', 3.14, (2,3), 'def')

- The immutability of tuples means they're faster than lists.

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

Operations on Lists Only

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append('a')       # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

The extend method vs +

- + creates a fresh list with a new memory ref
- extend operates on list li in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Potentially confusing:

- extend takes a list as an argument.
- append takes a singleton as an argument

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only

Lists have many methods, including index, count, remove, reverse, sort

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')  # index of 1st occurrence
1
>>> li.count('b')  # number of occurrences
2
>>> li.remove('b') # remove 1st occurrence
>>> li
  ['a', 'c', 'b']
```

Operations on Lists Only

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()  # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()        # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

Tuple details

- The comma is the tuple creation operator, not parenthesis
- Python shows parenthesis for clarity (best practice)
- Don't forget the comma!
- Trailing comma only required for singletons others
- Empty tuples have a special syntactic form

```
>>> (1)
1
```

```
>>> 1,
(1,)
```

```
>>> (1,)
(1,)
```

```
>>> ()
()
>>> tuple()
()
```

Summary: Tuples vs. Lists

- Lists slower but more powerful than tuples
  - Lists can be modified, and they have lots of handy operations and methods
  - Tuples are immutable and have fewer features
- To convert between tuples and lists use the list() and tuple() functions:

```
li = list(tu)
tu = tuple(li)
```

Overview

1. Introduction
2. Basics and Data Types
3. Control Structures
    - 3.1. While
    - 3.2.For
    - 3.3. If, Else, Elif
4. Functions
5. Others

Control Structures: while

```
>>> counter = 1
>>> while counter <= 5:
...     print "Hello, world"
...     counter = counter + 1


Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
```

Do not forget to increment/decrement the variable

Control Structures: for

```
>>> for x in range(1,6):
...     print x

1
2
3
4
5
```

The range function generate a tuple and we can iterate over it

- range(a) generates a tuple from 0 to a-1
- range (a,b) generates a tuple from a to b-1
- range(a,b,c) generates a tuple from a to b-1 with steps of c

Control Structures: if..elif..else

```
if score >= 90:
   print('A')
elif score >=80:
   print('B')
elif score >= 70:
   print('C')
elif score >= 60:
   print('D')
else:
   print('F')
```

Overview

1. Introduction
2. Basics and Data Types
3. Control Structures
4. Functions
5. Others

Functions (1)

« A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. »
Source: https://www.tutorialspoint.com/python/python_functions.htm

Defining a function

- Function block starts with the keyword def followed by the function name and parenthesis and a colon (:)
- Input parameters or arguments must be placed within these parenthesis
- The code block is indented
- The statement return [expression] exits a function
- No return statements is the same as return None

```
def functionName(parameter1,parameter2):
"""Docstring"""
   [code]
   return [expression]
```

Functions (2)

- Calling a function
  - Write its name
  - Specify the parameter values
- Example

```
def fact(x):
"""Returns the factorial of its argument, assumed
to be a posint"""
    if x == 0:
        return 1
    return x * fact(x - 1)

print ""
print "N fact(N)"
print "---------"

for n in range(10):
    print n, fact(n)
```

Example: fact.py

```
def fact(x):
"""Returns the factorial of its argument,
assumed to be a posint"""
   if x == 0:
      return 1
    return x * fact(x - 1)

print ""
print "N fact(N)"
print "----------"

for n in range(10):
   print n, fact(n)
```

Handling Files

- Read a file

```
with open(<fileName>,"r") as file:
  for line in file:
    print line
```

- Write content in a new file

```
with open(<fileName>,"w") as file:
  file.write("new file created\n")
```

- Append content to a file (it is created if it does not exist)

```
with open(<fileName>,"a") as file:
  file.write("add a new line to the file\n")
```