Introduction to Python 2/2

Text Analytics - Andrea Esuli

Flow control

Python coding style

Code blocks are identified by **indentation** (no “{ }” à la C, C++, C#, Java…).
White lines have no effect on blocks.

```
if a < 1:
    print(‘a is < than 1’) # printed when if evaluates to True

    print(‘a <= than 0’)   # printed when if evaluates to True
print(‘a is ‘, a)          # always printed
```

If/else

```
if name is None:                 # test expression
    print('No name specified') # if block
elif name in ages:               # else block, test of the second if
    print(name,'is',ages[name],'years old')
else:                            # else of the second if
    print('Age of',name,'is unknown')
print('Always printed')          # outside if
```

The **`if`** block is executed when the test expression evaluates to **`True`**.
The **`else`** block (when present) is executed when the test evaluates to **`False`**.

`elif` is the syntax to join multiple tests in a cascade.

While

```
count = 10
while count > 0:
    print(count,'more to go')
    count -= 1
print('done!')
```

The `while` loop is executed as long as the test expression evaluates to `True`.

While

```
a = [1, 2, 3, 4, 5, 6, 7]
while len(a)>0:                # continue jumps here
    value = a.pop()
    if value == 3:
        break                  # test it also with continue
    print('found',value)
print('done')                  # break jumps here
```

A **break** statement in a loop immediately ends it, jumping to the next statement after the while block.

A **continue** statement in a loop immediately ends it, jumping back to testing the test expression.

For

```
for item in collection:
    # do something on each item of the collection
```

In Python, the for loop syntax is most similar to Java and C# "foreach" syntax than traditional C and C++ for syntax.

The "collection" object must be an **iterable** object (e.g., list, tuple, string, set, dictionary, generator), i.e., an object that can produces an **iterator** over its elements.

```
for letter in 'ciao':
    print(letter.upper())
```

break/continue statements also work in for loops.

For

By default dictionaries return an iterator of their keys:

```
for name in ages: # equivalent to ages.keys()
    print(name,'is',ages[name],'years old')

Andrea is 39 years old
Giuseppe is 67 years old
Paolo is 58 years old
```

For

Loops over numeric ranges can be done using the **range** function:

```
for i in range(5):    # default start is zero
    print(i)
```

Two-valued range(start,end)

```
list(range(1,10))
Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Three-valued range(start,end,step)

```
list(range(10,1,-1))
Out: [10, 9, 8, 7, 6, 5, 4, 3, 2]
```

For

enumerate adds a numeric index to any iterator:

```
for index,value in enumerate(collection):
    print('Value',value,'is in position',index)
```

zip merges the output of many iterators into tuples:

```
for a, s, c in zip([20,25,23],['M','F','M'],['PI','FI','LU']):
    print(a,s,c)
```

```
20 M PI
25 F FI
23 M LU
```

Exceptions

Severe errors are signaled by raising an exception.

```
a = [1,2,3]
b = a[5]
KeyError: 5
```

Without the expected value computation cannot continue.
Unhandled exceptions stop the computation.

```
try:                  # Telling the interpreter that you know
    b = a[5]          # that something can go wrong.
except KeyError:      # Telling the errors you can manage.
    b = -1            # Code to execute when the exception is raised.
```

Functions

Functions

A function is a block of reusable code to perform a task.

It can take **arguments** as input, it can **return** some values as output.
It can also produce **side effects** (e.g., modify a global variable, write to a file).

```
def oldest(name_age_dict):
    max_age = -1
    oldest_name= None
    for name in name_age_dict:
        if name_age_dict[name]>max_age:
            max_age = name_age_dict[name]
            oldest_name = name
    return oldest_name
```

Functions

Arguments passed to a function are evaluated and assigned to the variable of the function (a new one at each invocation).
Argument assignment works the same as any other assignment.

```
def append_one(alist):
    alist.append(1)

a = ['a']
append_one(a)
a
Out: ['a',1]
```

```
def append_one(alist):
    alist = alist+[1]

a = ['a']
append_one(a)
a
Out: ['a']
```

Advice: when possible prefer **pure functions** over side effects

Functions

A **pure function** is a function that interacts with the rest of a program only through its arguments (which the function does not modify), and its return value.

```
def append_one(alist):
    return alist + [1]

a = ['a']
a = append_one(a)
a
Out: ['a',1]
```

Functions

All functions return a value, even those without a return. They return None.

Arguments can have default values:

```
import random
def roll_dices(number_of_dices = 1):
    sum = 0
    for i in range(number_of_dices):
        sum += random.randint(1,6)
    return sum

roll_dices(),roll_dices(2)
Out: (3,8)
```

Functions

Functions are objects. They can be manipulated as any other data type.

```
def my_operation(x):
    return x**x

def apply(values, function):
    results = []
    for value in values:
        results.append(function(value))
    return results


apply([1,2,3,4,5], my_operation)
Out: [1, 4, 27,  256, 3125]
```

Lambda

A lambda expression is an **anonymous function** made of a single expression.

Lambdas are useful to define short functions used only in a specific point.

```
lambda x: x**2          # x is argument, there can be zero or many
Out: function <....>

f = lambda x: x**2      # this line is equivalent to
def f(x):               # these two
    return x**2         # lines

a = [('a',5), ('b', 1), ('c', 3), ('d', 2)]
a.sort(key=lambda x: x[1])      # sort by the second element
Out: [('b', 1), ('d', 2), ('c', 3), ('a', 5)]
```

Iterators

Iterators

An iterator sequentially returns the elements of a collection.

The length of the sequence may be not known, not computable, or even (potentially) infinite.

An iterator can be created on iterable types (lists, tuples, strings…) with the **iter** function.

```
iterator = iter([0, 1, 2, 3])
```

Iterators

```
iterator = iter([0, 1, 2, 3])
```

The **next** function returns the next element of the collection, or a StopIteration exception

```
next(iterator)
Out: 0
next(iterator)
Out: 1
next(iterator)
Out: 2
next(iterator)
StopIteration
```

Generators

Generators are functions that by using the **yield** statement act as **iterators**:

```
def infinite():
    i = 0
    while True:
        yield i
        i += 1

N = infinite()
N
Out: <generator object infinite at 0x000001CBD92EAD58>
next(N), next(N), next(N)
Out: 0, 1, 2
```

Generators

The main advantage of generators over iterators is cleanliness and readability of code, i.e., iterators without the overhead of writing all their code.

```
class Infinite:
    def __init__(self):
        self.current = 0
    def __iter__(self):
        return self
    def __next__(self):
        next_value = self.current
        self.current += 1
        return next_value
```

Generators

Generator functions can take arguments to produce different outputs:

```
def infinite(start = 0, step = 1):
    i = start
    while True:
        yield i
        i += step

N = infinite(10,5)
next(N), next(N), next(N)
Out: 10, 15, 20
```

List comprehension

List comprehension is a specialized python construct to define lists with a clean and compact syntax.

```
a = [x**2 for x in range(6)]
```

is equivalent to

```
a = list()
for x in range(6):
    a.append(x**2)
```

List comprehension

List comprehension is a specialized python construct to define lists with a clean, compact, and math-like syntax.

```
a = [x**2 for x in range(6)]
```

$a = \{\ x^2 \mid x \in (0,5)\ \}$

is equivalent to

```
a = list()
for x in range(6):
    a.append(x**2)
```

List comprehension

The comprehension can include an if clause:

```
a = [x**2 for x in range(6) if x%2==0]
```

is equivalent to

```
a = list()
for x in range(6):
    if x%2==0:
        a.append(x**2)
```

Ternary operator `if-else can be used to define more complex tests:`

```
a = [x**2 if x%2==0 else -x for x in range(6)]
```

List comprehension

Comprehension can be nested

```
text = ['never', 'gonna', 'give', 'you', 'up']

[char for word in text for char in word ]
Out: ['n', 'e', 'v', 'e', 'r', 'g', 'o', 'n', 'n', 'a'...]

output = list()
for word in text:
    for char in word:
        output.append(char)
output
Out: ['n', 'e', 'v', 'e', 'r', 'g', 'o', 'n', 'n', 'a'...]
```

List comprehension

The same notation with round brackets produces a generator

```
a = [x**2 for x in range(6)]

type(a)

Out: list

a = (x**2 for x in range(6))

type(a)

Out: generator
```

List comprehension

Generators are lazy!

```
even = [x for x in infinite() if x%2==0]

Out: MemoryError

even = (x for x in infinite() if x%2==0)

next(even)
Out: 0
next(even)
Out: 2
next(even)
Out: 4
```

Classes, Modules

Classes

Python supports object oriented programming.

```
class Person:
    def __init__(self,name, age):# constructor
        self.name = name            # instance variables
        self.age = age              # self is like 'this' in Java, C#

    def young(self):                # method that return a value
        return self.age < 40

    def birthday(self):             # method that changes the state
        self.age += 1               # of the object
```

Classes

Why Python always requires "self"?

"Explicit is better than implicit."

Using "self" makes clear if a variable refers to the instance object or not.

All variable methods and variables are visible (i.e. public, no protected or private fields).

Convention: add two underscores (e.g. self.__name) to mark a field as "please don't touch directly".

Use **dir(obj)** to list all names in the scope of obj.

Classes

A class can have class variables, shared by all instances.

```
class Person():
    population = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.population += 1

    def __del__(self):
        Person.population -=1
```

Classes

```
def young_test(some_person):
    if some_person.young():
        print(some_person.name, 'is a young person')
    else:
        print(some_person.name, 'was a young person')

io = Person('Andrea', 39)
young_test(io)
Out: 'Andrea is a young person'
io.birthday()
young_test(io)
Out: 'Andrea was a young person'
```

Classes

Inheritance allow to model complex class relations.

```
class Researcher(Person):
    def __init__(self, name, age, discipline):
        super().__init__(name, age)
        self.discipline = discipline

    def young():
        return True

io = Researcher('Andrea',39,'Machine Learning')
io.birthday()
io.young()
Out: True
```

Custom exceptions

Exceptions are objects. Exception types must derive from the **Exception** class.

```
class TooYoung(Exception):
    pass

class Researcher(Person):
    __min_age = 3
    def __init__(self, name, age, discipline):
        if age < __min_age:
            raise TooYoung()
        super().__init__(name, age)
        self.discipline = discipline
```

Custom exceptions

```
researchers = [('Andrea', 39, 'CS'), ('Baby', 2, 'Engineering')]

researchers_obj = list()
people_obj = list()

for name, age, discipline in researchers:
    try:
        researchers_obj.append(Researcher(name, age, discipline))
    except TooYoung:
        people_obj.append(Person(name, age))
```

Modules

Any script can be imported into another using the **import** statement.

```
import math
math.factorial(10)
```

The name of the module defines a **namespace**. Everything defined in the module can be referred as `module.<name>`

Import searches of a .py file in the paths listed in the `sys.path` list.

```
import sys
print(sys.path)
Out: ['', 'C:\\Programs\\Anaconda3\\envs\\py3\\Scripts', …]
```

Modules

from math import factorial, exp

factorial(10), exp(10)

The "from" syntax copies the specified names into the current namespace.
Use * to copy all the names (from module import *).
Note: be careful to not **overwrite** existing names. Use "as" to specify a different name to be used in the current namespace.

```
import math as mymath
mymath.factorial(10)
```

```
from math import factorial as fct
fct(10)
```

Modules

Note: import actually interprets the script content.

```
mymodule.py: print('ciao')
             a = 10

import mymodule
Out: ciao
mymodule.a
Out: 10
```

Use __name__ to tell if the script is executed from an import or as a program.

```
if __name__ == "__main__":  # stuff to be run when used as script
   print('ciao')
```

Modules

Some typically used modules:

sys	System-specific stuff (argv, path, maxint, stdin...)
os	OS-related stuff (low level I/O, makedirs, rename...)
math	Typical math functions (exp, sin, asin...)
random	Random number generation
string, re	Text manipulation, regular expressions
pickle	Serialization
csv, json, html, xml	Formatters
time, datetime	Time management and formatting
threading	Parallel processing

I/O

Input

The input() function waits for user input, returning it as a string.

```
people = list()
while True:
    name = input('name? ')
    age = int(input('age? '))
    people.append(Person(name, age))
    stop_check = input("enter 'x' to stop creating persons ")
    if stop_check == 'x':
        break
```

Encodings

Character encodings are used to map characters to single- or multi-byte representations, for their transmission and storage.

ASCII is a seven bit character encoding dating back to 1963. It was used on teleprinters.

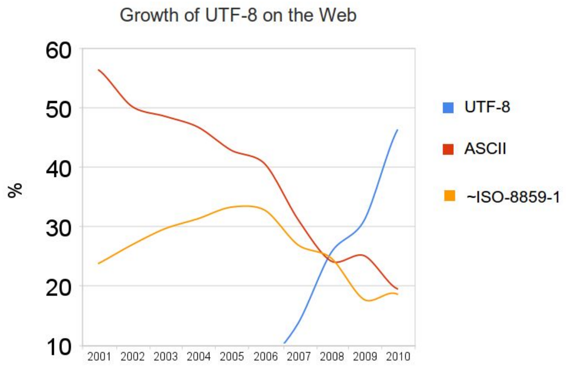
ASCII has been extended to many different encodings designed to handle language-specific characters, e.g. ã, ç, é, è, ™, Д, ….

USASCII code chart

b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b_4	b_3	b_2	b_1	Row ↓ Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	−	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	—	o	DEL

Encodings

Unicode is a standard (dating back to 1988) for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.



UTF-8 is the most common format adopted for the encoding of unicode characters (backward compatible with ASCII).

Always use UTF-8.

Usage of character encodings for websites, 12 Sep 2017, W3Techs.com

Encodings

Using the wrong encoding can results in I/O errors or losing information.

```
Traceback (most recent call last ) :
File ” text indexing .py”, line 107, in <module> main()
File ” text indexing .py”, line 47, in main for row in reader :
File ”codecs.py”, line 319, in decode
( result , consumed) = self. buffer decode (data, self . errors , final )
UnicodeDecodeError: ’utf−8’ codec can’t decode byte 0xf3 in position 5625: invalid
continuation byte
```

L’Universit□ di Pisa □ una delle pi□ antiche e prestigiose universit□ italiane e d’Europa.

Encoding can be guessed: https://github.com/chardet/chardet

Files

The **open** function returns an object to operate on a file, depending on the mode with which it is opened.

A mode is a combination of an open mode (r/w/x/a) and a I/O mode (t/b), and an optional '+' for combined read/write.

- 'r' open for reading (default)
- 'w' open for writing, truncating the file first
- 'x' open for exclusive creation, failing if the file already exists
- 'a' open for writing, appending to the end of the file if it exists
- 'b' binary I/O
- 't' text I/O (default)
- '+' open a disk file for updating (reading and writing)

Files

```
file = open('filename', mode='w', encoding='utf-8')
file.write('ciao\n') # newlines are not added
file.write('hello\n')
file.close()            # remember to close files

file = open('filename', encoding='utf-8')
next(file)                          # a text file object acts
Out: 'ciao\n'                       # as an iterator over
next(file)                         # its lines
Out: hello\n'
```

Files

The **with** statement automatically manages closing open file, even in case of exceptions

```
with open('filename', mode='w', encoding='utf-8') as file:
    file.write('ciao\n')
    file.write('hello\n')
```

A typical way to read files is in for loops:

```
with file = open('filename', encoding='utf-8') as file:
    for line in file:
        print(line)
```

CSV files

The csv module is a file wrapper that implements import/export of data in comma separated values format.

```
import csv
data = [['Andrea', 39], ['Giuseppe', 67], ['Paolo', 59]]
with open('data.csv', mode='w', encoding='utf-8', newline='') as
outfile:
    writer = csv.writer(outfile)
    writer.writerows(data)
```

CSV files

The reader reads fields as strings, remember to convert to the correct type.

```
import csv
data = list()
with open('data.csv', encoding='utf-8', newline='') as infile:
    reader = csv.reader(infile)
    for row in reader:
        data.append([row[0],int(row[1])])
data
Out: [['Andrea', 39], ['Giuseppe', 67], ['Paolo', 59]]
```

Pickle

The pickle module implement binary serialization, to save and load native python objects.

```
import pickle
with open('data.pkl', mode='wb') as file:
    pickle.dump(data, file)
```

Note the binary open mode

```
with open('data.pkl', mode='rb') as file:
    data = pickle.load(file)
```

One last advice...