Introduction to python for AI

Alessia Mondolo

July 9, 2019

Academy AI

About myself

- **Name** Alessia Mondolo
- **Background** Computer Science Engineer from Polytechnic of Milan.
- **Study** Currently studying a masters degree on artificial intelligence in Barcelona.
- **Work** Leading different AI projects at Puig.
- **Contact** alessiamondolo@gmail.com
- **Github** @alessiamondolo
- **Linkedin** Alessia Mondolo

Getting to know you

- Name
- Origins
- Background
- Programming experience
- Goals
- Favourite sport
- Favourite food

Introduction to Python

Python

Python is a dynamic interpreted language specially designed for convenience and fast development. For this reason it has been highly adopted by a lot of scientific communities such as data scientists.

Python is not the most efficient of the languages as it is interpreted but its high readability makes it very convenient for a most of applications.

In this course we will be using Python 3.6 (Python 2.7 is discontinued at last).

Indentation

The first thing that you notice when entering python is that it uses indentation as part of the definition of the language. Different scopes are denoted by their indentation. By convention use **4 spaces**, not tabs.

```
for i in [1, 2, 3]:
    print(i) # Four spaces
```

As seen on top, one line comments start with the special character #. Multi-line comments start and end with """:

```
def func():
    """
    Example of pydoc
    """
    return None
```

Dynamic language I

Python is a dynamic language, this means that variables are not typed: this allows us to change the type of the variables in run time or to pass different types of parameters to the same function.

```
l =[1, 2, 3]
l.append(4) # int
l.append(5.0) # float
l # [1, 2, 3, 4, 5.0]
```

Python is taking care behind the scenes of changing the type of basic elements when needed.

```
1 /2 # 0.5
```

This has a very important implication which means that you do not have to care of managing memory, not even for overflowing issues.

Dynamic language II

Variables can be forced to cast whenever is needed. Casting is python is done as follows:

```
a =1
str(a) # "1"
```

We can check for the type of a variable. This allows us to check if a variable is from a class and therefore has certain functionalities.

```
a =1
type(a) # int
a =str(a)
type(a) # str
isinstance(a, str) # True
isinstance(a, int) # False
```

Built-in types

Basic built in types are the following ones. More complex data types as list have their own literals but their functionality is in the standard library.

Type	Description
None	'null' value
str	String
unicode	Unicode
float	Double-precision
bool	Logical
int	Signed
long	Arbitrary

Numerics

Numeric types come with a bunch of methods worth mentioning:

- `a + b` Add a and b
- `a - b` Substract a and b
- `a * b` Multiply a by b
- `a / b` Divide a by b
- `a // b` Floor-divide a by b, dropping any fractional remainder
- `a ** b` Raise a to the power of b

Logical values

Python handles a binary logic around `True` and `False`. Common operations are performed with `or`, `and` and `not`.

```
True and False # False
True and True # True
True or False # True
True or not False # True
```

Python, as many other languages, can evaluate to a boolean all the objects, for example iterables will be evaluated as True when their size is greater than 0.

Flow Control

Flow control: ifs

If statements allow to control the flow of our code. In python ifs are defined as follows:

```
a =int(input())
if a % 2 ==0:
    print('Multiple of 2')
elif a % 3 ==0:
    print('Multiple of 3')
else:
    print('Not multiple of 2 neither 3')
```

Flow control: loops

For loops iterate over the elements of the iterables.

```
for ch in ['a', 'b', 'c']:
    if ch =='b':
        continue
    print(ch)
# a c
```

While loops are defined in the following fashion:

```
a =10
while True:
    if a ==0:
        break
    print(a)
    a -=1
# 10 9 8 7 6 5 4 3 2 1
```

Functions

Function Declaration I

Functions in python are declared with the keyword `def` and can be defined in the global scope (not inside classes)

```
a =1
def func(b, c =2):
    return a *b *c
```

As you can see functions can receive any number of parameters and can access their outer scope. Keep in mind that functions are *first class citizens*, this means that can be passed as parameters, stored and referenced straightforward.

Function Declaration II

In python parameters are passed through **reference**, this means that any mutation performed on the parameter changes the passed object.

```
def append_element(l, e):
    l.append(e)

arr =[1, 2, 3]
append_element(arr, 4)
arr # [1, 2, 3, 4]
```

Utility methods I

There are a number of methods from the standard library that can be very handy when dealing real world scenarios. We will introduce a bunch of the more important:

`range(init, end, step)`

```
range(3) # 0, 1, 2
range(1, 3) # 1, 2
range(3, 1, -1) # 3, 2
```

`enumerate(iterator)`

```
for index, value in enumerate(['a', 'b']):
    print(index, value)

# 0, a 1, b
```

Utility methods II

`zip(i1, i2)`

```
a =[1, 2, 3]
b =['a', 'b', 'c']
for el_a, el_b in zip(a, b):
    print(el_a, el_b)

# 1, a 2, b 3, c
```

`dir(object)`

```
a =[1, 2, 3]
dir(a) # append, clear, copy, count, extend, index, insert, pop
                                    , remove, reverse, sort...
```

Recursion I

A recursive function is a function that will continue to call itself and repeat its behavior until some condition is met to return a result. All recursive functions share a common structure made up of two parts: base case and recursive case:

- recursive case: decompose the original problem into simpler instances of the same problem.
- base case: as the large problem is broken down into successively less complex ones, those subproblems must eventually become so simple that they can be solved without further subdivision.

Recursion II

Example:

```
n! =n x (n-1)!
n! =n x (n-1) x (n-2)!
n! =n x (n-1) x (n-2) x (n-3)!
.
.
n! =n x (n-1) x (n-2) x (n-3)  x 3!
n! =n x (n-1) x (n-2) x (n-3)  x 3 x 2!
n! =n x (n-1) x (n-2) x (n-3)  x 3 x 2 x 1!
```

In Python:

```
def factorial_recursive(n):
    # Base case: 1! = 1
    if n ==1:
        return 1
    # Recursive case: n! = n * (n-1)!
    else:
        return n *factorial_recursive(n-1)
```

Exception handling

We can control the throwed exceptions by means of `try` and `except`. This allows us to catch the exceptions and continue the execution of the program.

```
try:
    callMethod()
except Exception:
    print('Error catched')
```

Lambda Functions

In python we can create anonymous functions with a one-liner by means of the `lambda` expression.

```
f =lambda x, y: x +y
f(1, 2) # 3
```

Data structures

Tuples

Tuples are a one dimensional inmutable list capable of storing objects. Any kind of iterator can be transformed to a tuple.

Unpacking is a handy feature of tuples that allows us to extract the variables into separate lists.

```
a =1, 2
b, c =a # unpacking
b # 1
c # 2
```

Lists I

List in python are defined as literals and they are **mutable**. To access the array similarly to other languages we use bracket (`[ ]`) indexing.

A very handy tool offered by python is **slicing**. We can select a subset of the array by using the special notation `[start:stop:step]`.

```
l =[1, 2, 3]
l[0] # 1
l[0:2] # [1, 2]
l[2:0:-1] # [3, 2]
l[-1] # 3
```

Lists II

Common operations on lists are:

- `l.append(e)`: Adding element.
- `l.pop()`: Remove last element.
- `len(l)`: Gets the size of the list.
- `l.sort()`: Sorts the list.
- `e in l`: Checks for an element in the list.
- `l1 + l2`: Adds both lists

For more information address to the official documentation.

Dictionaries I

Dictionaries are a special kind of collection that allows us to index by an object. Python allows us to declare them in a literal fashion.

To access the array we will also use brackets ([ ]).

```
dict ={
    'a': 1,
    'b': 2,
}

dict['c'] =3
dict['c'] # 3
dict['a'] # 1
```

Dictionaries II

Common operations on dictionaries are:

- `k in d`: Checks for a key in the dictionary.
- `len(d)`: Gets the size of the dictionary.
- `d.values()`: returns a list of the values.
- `d.keys()`: returns a list of the keys.
- `d.items()`: returns a list of tuples as `(key, value)`

It is important to note that we can iterate over the dictionary in a fancy way by transforming it to tuples and unboxing them.

```
for key, value in d.items():
    print(key, value)
# a, 1 b, 2 c, 3
```

For more information address the official documentation.

Sets

Sets are unordered lists without repetition, they come specially handy when we need to perform operations such as join or intersection.

Common operations on dictionaries are:

- `e in s`: Checks for an element in the set.
- `s1 & s2`: Performs the intersection of the sets
- `s1 | s2`: Performs the union of the sets
- `s1 - s2`: Performs the difference of the sets
- `len(s)`: Gets the size of the set.

```
a =set([1, 2, 3])
b ={2, 3, 4}

a.union(b) # or a & b => 1, 2, 3, 4
a.intersection(b) # or a | b => 2, 3
```

Strings I

Strings in python are very similar to a list of chars but they have their own functionality and utility methods.

You can index a string just like you would index a list `[ ]`. You can also check for the presence of another string by using the same interface as with lists `in`.

```
s ='Hello'

s[0] # 'H'
s[0:2] # He
'el' in s # True
```

Strings II

String interpolation is one of the things that come really handy once digging into code. Python 3.6 brings the possibility of using a special kind of strings, the **f-strings** that come with a very nice interface.

```
a =1
s =f'A is {a}'
s # A is 1

s ='A is ' +str(a) # we need to cast the integer
```

Strings III

Usually we will be dealing with strings, python provides with a set of methods to treat them, some of them are very useful.

```
s ='Hello world! '
s.split(' ') # ['Hello', 'world!', '']

s.strip().split(' ') # ['Hello', 'world!']
```

Comprehensions

One of the best features of python is comprehension, with this feature we can easily create and filter complex lists, sets or dictionaries.

We will use an example to illustrate it:

```
l =[1, 2, 3]
pows =[i**2 for i in l]
evens =[i for i in l if i % 2 ==0]

dict ={str(i): i for i in l}
```

Mutability

Most objects in python are **mutable** with some exceptions.

```
a =[1, 2, 3]
a[0] =2
a # [2, 2, 3]
```

Tuples for example are inmutable objects.

```
a =(1, 2, 3)
a[0] =2 # TypeError: 'tuple' object does not support item
                                        assignment
```