Unit 2: Introduction to Python

Carmen Graciani Díaz
José Luis Ruiz Reina

Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Inteligencia Artificial, 2017-18

Introduction to PYTHON

- Created on the early 90's by Guido van Rossum
- The name was chosen after the BBC show "Monty Python's Flying Circus"
- All distributions are open source
- It is a "powerful" programming language "easy" to learn
- Extensive standard library available
- Official web of Python: http://www.python.org/
- High-level interpreted language that can be extended with C or C++
  - Object oriented programming
  - Imperative programming
  - Functional programming

Working with PYTHON

- Available for numerous platforms (UNIX, Solaris, Linux, DOS, Windows, OS/2, Mac OS, etc.)
- Interactive mode
  - Starting an interpreter: `python3`
  - Typing an expression on the interpreter shell
  - Evaluate the expression
- Creation of *scripts*
  - Write your code on a `file.py`
  - Add on the first line `#!/usr/bin/python3`
  - Enable the execution permission for the file

Interaction

- Interpreter

```
$ python3
Python 3.2.3 (default, Sep  10 2012, 18:17:42)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license"
 for more information.
>>> print('Hello! and Bye!')
Hello! and Bye!
>>> exit()
$
```

Scripts

- *Script*:

```
#!/usr/bin/python3
# Shows on the console the message: Hello! and Bye!
print('Hello! and Bye!')
```

- Execution:

```
$ ./example.py
Hello! and Bye!
$
```

- Comments in Python: either # or triple quote

Numerical data types

- Numbers (immutable)

```
>>> 2+2
4
>>> (50-5*6)/4
5.0
>>> a = (1+2j)/(1+1j)
>>> a.real
1.5
>>> width = 20
>>> height = 5*9
>>> area = width * height
>>> area
900
>>> area *= 2
>>> area
1800
```

- Mutable *vs* immutable. *Augmented* assignations
- Variables in Python always *point to objects*

Boolean

- Boolean data: `True` and `False`
- Logic operators `and`, `or`, `not` and comparison by `==`

```
>>> 2 == 2
True
>>> 2 == 3
False
>>> True and 2 == 3
False
>>> False or 2 == 2
True
>>> not False
True
>>> True == 1
True
>>> False == 0
True
```

Strings

- Sequence of characters, simple or double quotation marks (immutable)
- Some operations on strings:

```
>>> c1="Hello"
>>> c2=" there!"
>>> greet = c1+c2
>>> greet
'Hello there!'
>>> greet[3:8]
'lo th'
>>> greet[3:8:2]
'l h'
>>> greet[-1]
'!'
>>> greet[3:8:-1]
''
>>> greet[3:8]
'lo th'
>>> greet[8:3:-1]
'eht o'
>>> greet[::-1]
'!ereht olleH'
>>> greet * 4
'Hello there!Hello there!Hello there!Hello there!'
```

Strings

- Some methods on strings

```
>>> st1="Once upon a time"
>>> st1.index("time")
12
>>> st1.index("times")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> st1.find("time")
12
>>> st1.find("times")
-1
>>> st1.upper()
'ONCE UPON A TIME'
>>> st1.count("n")
2
```

Strings

- Written output (`print` and `format`)

```
>>> print("Inteligencia","Artificial")
Inteligencia Artificial
>>> s="{0} times {1} is {2}"
>>> x,y,u,z = 2,3,4,5
>>> print(s.format(x,y,x*y))
2 times 3 is 6
>>> print(s.format(u,z,u*z))
4 times 5 is 20
```

Tuples

- Ordered sequences of elements, separated by commas and usually in parenthesis

```
>>> 1,2,3,4
(1, 2, 3, 4)
>>> ()
()
>>> 1,
(1,)
>>> a=2
>>> b=3
>>> (a,b,a+b,a-b,a*b,a/b)
(2, 3, 5, -1, 6, 0.6666666666666666)
```

Tuples

- Operations similar to those of strings are valid for tuples

```
>>> a=("One","Two","Three","Four")
>>> a[2]
'Three'
>>> a[1:3]
('Two', 'Three')
>>> a[::]
('One', 'Two', 'Three', 'Four')
>>> a[::-1]
('Four', 'Three', 'Two', 'One')
>>> a+a[2::-1]
('One','Two','Three','Four','Three','Two','One')
>>> "Two" in a
True
```

Tuples

- Immutable:

```
>>> a=("Madrid","Paris","Rome","Berlin","London")
>>> b=a
>>> b
('Madrid', 'Paris', 'Rome', 'Berlin', 'London')
>>> a[3]="Athens"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assi
>>> a+="Athens",
>>> a
('Madrid','Paris','Rome','Berlin','London','Athens')
>>> b
('Madrid', 'Paris', 'Rome', 'Berlin', 'London')
```

Tuples

- Tuples on the left-hand side of assignations:

```
>>> a,b,c=(1,2,3)
>>> a
1
>>> b
2
>>> c
3
>>> a,b=b,a # interchange on a single instruction!!
>>> a
2
>>> b
1
```

- Unpacking

```
>>> a,*b =[1,2,3,4]
>>> a
1
>>> b
[2, 3, 4]
>>> *a,b = [1,2,3,4]
>>> a
[1, 2, 3]
>>> b
4
```

Lists

- Ordered sequences of elements, separated by commas and in between brackets

```
>>> ["a","b","c","d"]
['a', 'b', 'c', 'd']
>>> [2]
[2]
>>> []
[]
```

- Similar operations as for tuples and strings

```
>>> sandwich = ['bread', 'ham', 'bread']
>>> 2*sandwich[:2] + ['egg'] + [sandwich[-1]]
['bread', 'ham', 'bread', 'ham', 'egg', 'bread']
>>> triple = 2*sandwich + ["tomato",'bread']
>>> triple
['bread', 'ham', 'bread', 'bread', 'ham', 'bread', 't
>>> "tomato" in triple
True
>>> len(triple)
8
```

Lists

- Lists are mutable:

```
>>> l=["hola","adios","hasta pronto"]
>>> m=l
>>> m
['hola','adios','hasta pronto']
>>> l[2]="see you"
>>> l
['hola','adios','see you']
>>> m
['hola','adios','see you']
>>> p=[l,m]
>>> p
[['hola','adios','see you'],['hola','adios','see you']]
>>> m[0]="hello"
>>> p
[['hello','adios','see you'], ['hello','adios','see you']]
>>> p[0][1:2]=[]
>>> p
[['hello', 'see you'], ['hello', 'see you']]
>>> l
['hello', 'see you']
>>> m
['hello', 'see you']
```

Lists

- Some methods on lists

```
>>> r=["a",1,"b",2,"c","3"]
>>> r.append("d")
>>> r
['a', 1, 'b', 2, 'c', '3', 'd']
>>> r.extend([4,"e"])
>>> r
['a', 1, 'b', 2, 'c', '3', 'd', 4, 'e']
>>> r.pop()
'e'
>>> r
['a', 1, 'b', 2, 'c', '3', 'd', 4]
>>> r.pop(0)
'a'
>>> r
[1, 'b', 2, 'c', '3', 'd', 4]
>>> r.insert(3,"x")
>>> r
[1, 'b', 2, 'x', 'c', '3', 'd', 4]
```

Definitions by comprehension

- Lists can be defined without explicitly writing its elements

```
>>> [a for a in range(6)]
[0, 1, 2, 3, 4, 5]
>>> [a for a in range(6) if a % 2==0]
[0, 2, 4]
>>> [a*a for a in range(6) if a % 2==0]
[0, 4, 16]
>>> [(x,y) for x in [1,2,3] for y in ["a","b","c"]]
[(1, 'a'),(1, 'b'),(1, 'c'),(2, 'a'),(2, 'b'),(2, 'c'),
 (3, 'a'),(3, 'b'),(3, 'c')]
>>> (a*a for a in range(6) if a % 2==0)
<generator object <genexpr> at 0x7fbb85aa8a00>
```

- Also applies to other collective data types:

```
>>> tuple(a*a for a in range(6) if a % 2==0)
(0, 4, 16)
>>> tuple(a%3 for a in range(9))
(0, 1, 2, 0, 1, 2, 0, 1, 2)
>>> {a%3 for a in range(9)} # type set, will be seen next
{0, 1, 2}
```

Sets

- Collections of data, without order and without duplicates, represented in between curly brackets and separated by commas
- Elements of the sets must be *hashable*
  - In particular every immutable data type in hashable
- Sets are mutable
- Examples:

```
>>> basket = {"pears", "apples", "pears", "apples"}
>>> basket
{'pears', 'apples'}
>>> "peaches" in basket
False
>>> a = {x for x in "abracadabra" if x not in "abc"}
>>> a
{'r', 'd'}
```

Sets

- Some methods on sets

```
>>> s={1,3,5,7,9}
>>> s.add(10)
>>> s
{1, 3, 5, 7, 9, 10}
>>> s.add(10)
>>> s
{1, 3, 5, 7, 9, 10}
>>> s & {4,7,15}
{7}
>>> s
{1, 3, 5, 7, 9, 10}
>>> s | {1,2,4}
{1, 2, 3, 4, 5, 7, 9, 10}
>>> s
{1, 3, 5, 7, 9, 10}
>>> s |= {2,4}
>>> s
{1, 2, 3, 4, 5, 7, 9, 10}
>>> s <= {1,3,5,7,9,10,11,12}
True
```

Dictionaries

- Collection of unordered pairs *key*:*value* (mutable data type)

```
>>> tel = {"juan": 4098, "ana": 4139}
>>> tel["ana"]
4139
>>> "ana" in tel
True
>>> tel["pedro"]=4118
>>> tel
{'juan': 4098, 'pedro': 4118, 'ana': 4139}
>>> tel.keys()
dict_keys(['juan', 'pedro', 'ana'])
>>> del tel['ana']
>>> [(n,t) for (n,t) in tel.items()]
[('juan', 4098), ('pedro', 4118)]
>>> tel["olga"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'olga'
>>> None == tel.get("olga")
True
```

Control structures (instruction `if`)

- Program `sign.py`:

```
x = int(input('Enter an integer: '))

if x < 0:
    print('Negative:', x)
elif x == 0:
    print('Cero')
else:
    print('{0} is positive:'.format(x))
```

- The role of indentation in Python
- Execution:

```
Enter an integer:34
34 is positive
```

Control structures (instruction `for`)

- Program `mean.py`:

```
l, sumaux, n = [1,5,8,12,3,7], 0, 0
for e in l:
    sumaux += e
    n +=1
print(sumaux/n)
```

- Program `prime.py`:

```
prime = []
for n in range(1, 20, 2):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'is', x, '*', n//x)
            break
    else:
        prime.append(n)


"""Output:
   9 is 3 * 3
   15 is 3 * 5
   >>> prime
   [1, 3, 5, 7, 11, 13, 17, 19]"""
```

Control structures (instruction `while`)

- Program `fibonacci.py`:

```
a, b = 0, 1
while b < 10:
    print(b,end=',')
    a,b = b,a+b
```

- Program `find-index.py`

```
ind =0
lookfor = "prize"
lst = ["nothing","loose","prize","another"]
while ind < len(lst):
    if lst[ind] == lookfor:
       break
    ind += 1
else: ind = -1
```

- Instructions `pass`, `continue`, `break` and `return`

Some iteration patterns

```
>>> notas = {'Juan Gómez': 'notable', 'Pedro Pérez': 'aprobado'}
>>> for k, v in notas.items(): print(k, v)
Pedro Pérez aprobado
Juan Gómez notable

>>> for i, col in enumerate(['red', 'blue', 'yellow']): print(i, col
0 red
1 blue
2 yellow

>>> questions = ['name, 'last name', 'favourite color']
>>> answers = ['Juan', 'Pérez', 'red']
>>> for p, r in zip(preguntas, respuestas):
...     print('My {0} is {1}.'.format(p, r))
My name is Juan.
My last name is Pérez.
My favourite color is red.

>>> for i in reversed(range(1, 10, 2)): print(i,end="-")
...
9-7-5-3-1-
```

Iterable types and iterators

- Iterable types: those where it makes sense to go through all its elements one at a time, having some notion of *next* element
  - Strings, tuples, lists, sets, dictionaries, ...
  - Used in loops: `for` *item* `in` *iterable: ...*
  - When an iterable is used in a loop, an *iterator* is automatically obtained from it, in order to generate its elements sequentially, one for each iteration
- Generators: expressions as iterators
  - For example: functions `range`, `enumerate`, `zip`, `reversed`,...:

```
>>> range(1,10,2)
range(1, 10, 2)
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
```

  - Generators by comprehension:

```
>>> (x * x for x in range(1,10,3))
<generator object <genexpr> at 0x7f1415de9a50>
>>> list(x * x for x in range(1,10,3))
[1, 16, 49]
```

Definition of functions

- Definition of functions:

```
def fib(n):
    """Prints the Fibonacci sequence up to n
       and returns the last number calculated """
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
    return(b)
```

- Using the function:

```
>>> fib(30)
0 1 1 2 3 5 8 13 21
55
>>> x=fib(30)
0 1 1 2 3 5 8 13 21
>>> x
55
```

- Difference between collateral effect and returned value (`return`)
  - When there is no explicit `return` on the function, returns `None`

Function arguments

- Arguments using keywords

```
>>> def g(x,y): return(x**y)
>>> g(2,3)
8
>>> g(y=3,x=2)
8
>>> g(2,x=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: g() got multiple values
           for keyword argument 'x'
```

Function arguments

- Arguments having a default value

```
>>> def j(x,y,z=0): return(x**y + z)
>>> j(2,3)
8
>>> j(2,3,4)
12
```

  - Warning!, default arguments are evaluated only once, when defining the function:

```
>>> i = [5]
>>> def f(x=i): return x
>>> f()
>>> [5]
>>> i.append(8)
>>> f()
>>> [5,8]
>>> i = []
>>> f()
>>> [5,8]
```

Function arguments

- Arbitrary number of arguments

```
>>> def h(x,*y): print(x,y)
>>> h(3,2,5,7,2,5)
3 (2, 5, 7, 2, 5)
>>> h("a","b","c")
a ('b', 'c')
>>> h(10)
10 ()
>>> def d(**y): print(y)
>>> d(a=2,b=3,c=4)
{'a': 2, 'c': 4, 'b': 3}
```

- Function calls with unpacking

```
>>> def d(x,y,z): return(x**y + z)
...
>>> l=[3,2,4]
>>> d(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: d() takes exactly 3 positional arguments (1 given)
>>> d(*l)
13
```

Errors and error handling

- Exceptions:

```
>>> def returns_double():
        x= int(input("Introduce one number: "))
        return 2*x
>>> returns_double()
Introduce one number: a
...
ValueError: invalid literal for int() with base 10: 'a'
```

- Exceptions handling with `try...except`

```
>>> def returns_double():
        while True:
            try:
                x= int(input("Enter a number: "))
                return 2*x
            except ValueError:
                print("Not a number, try again.")

>>> returns_double()
Enter a number: a
Not a number, try again.
Enter a number: d
Not a number, try again.
Enter a number: 3
6
```

Modules

- A module is a file containing definitions and instructions in Python. For example, `operations-es.py`:

```
def suma(x,y): return x+y

def resta(x,y): return x-y

def multiplicacion(x,y): return x*y

def division(x,y): return x/y
```

- Using modules with `import`:

```
>>> import operations
>>> suma(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'suma' is not defined
>>> operations.suma(2,3)
5
>>> operations.division(8,5)
1.6
```

Importing modules

- Another way of importing modules:

```
>>> import operations as ops
>>> ops.resta(4,2)
2
>>> operations.resta(4,2)
....
NameError: name 'operations' is not defined
```

- Yet another one:

```
>>> from operations import *
>>> resta(3,2)
1
>>> division(4,6)
0.666666666666666
```

- And also:

```
>>> from operations import suma,resta
>>> suma(2,3)
5
>>> resta(4,1)
3
>>> multiplicacion(2,3)
..
NameError: name 'multiplicacion' is not defined
```

Second order

- `Lambda` expressions:

```
>>> lambda x,y: x+y*3
<function <lambda> at 0x7f1415e022f8>
>>> (lambda x,y: x+y*3)(2,3)
11
>>> (lambda x,y: x+y*3)("a","b")
'abbb'
```

- Functions returning functions:

```
>>> def increment(n): return lambda x: x+n
>>> f2 = increment(2)
>>> f2(5)
7
```

- Functions receiving functions as arguments:

```
>>> def applyonto (f,l): return [f(x) for x in l]
>>> applyonto (increment(5),[1,2,3])
[6, 7, 8]
```

Reading and writing files

- Opening files with `open`:

```
>>> f=open("file.txt","r")
>>> f
<_io.TextIOWrapper name='file.txt' mode='r' encoding='UTF-8'>
```

  - A *file object* is created, along with a number of associated method to carry out reading and writing operations.

- Opening modes (text files):
  - `open('file.txt','r')`: open for reading (default option)
  - `open('file.txt','w')`: open for writing, truncating th file if it already exists
  - `open('file.txt','a')`: open for writing, appending the contents
  - `open('file.txt','r+')`: open for reading and writing
- Commonly used methods: `f.read()`, `f.readline()`, `f.write()`, `f.close()`

Reading and writing (examples)

- Assume we have the file `file.txt` with the following content:

```
This is the first line
An this is the second
The third one is here
And finally the fourth line
```

- Reading with `read`:

```
>>> f=open("file.txt")
>>> s=f.read()
>>> s
'This is the first line\nAnd this is the second\nThe
third one is here\nAnd finally the fourth line\n\n'
>>> f.close()
```

- using the `with` block for closing:

```
>>> with open('file.txt') as f: first = f.readline()
...
>>> first
'This is the first line\n'
```

Reading and writing (examples)

- Reading with `readline`:

```
>>> f=open("file.txt")
>>> s1=f.readline()
>>> s1
'This is the first line\n'
>>> s2=f.readline()
>>> s2
'An this is the second \n'
>>> s3=f.readline()
>>> s3
'The third one is here\n'
>>> s4=f.readline()
>>> s4
'And finally the fourth line\n'
>>> f.close()
```

- Iterating over the lines of a text file:

```
>>> for line in open("file.txt"): print(line, end='')

This is the first line
An this is the second
The third one is here
And finally the fourth line
```

Reading and writing (examples)

- Writing with `write`, mode `'a'`:

```
>>> with open('file.txt','a') as f:
        f.write("I write the fifth line\n")
24
```

  - Contents of `file.txt`:

```
This is the first line
An this is the second
The third one is here
And finally the fourth line
I write the fifth line
```

- Writing with `write`, mode `'w'`:

```
>>> f=open("file.txt","w")
>>> f.write("Rewriting the first line\n")
27
>>> f.write("And also the second\n")
21
>>> f.close()
```

  - Contents of `file.txt`:

```
Rewriting the first line
And also the second
```

Classes (I)

```
import math

class Point(object):

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_to_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, another):
        return self.x == another.x and self.y == another.y

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Classes (II)

```
>>> p1=Point()
>>> p2=Point(3,4)
>>> p1
<__main__.Point object at 0x75e510>
>>> str(p1)
'(0, 0)'
>>> str(p2)
'(3, 4)'
>>> p1.x
0
>>> p2.y
4
>>> p1 == p2
False
>>> p2.distance_to_origin()
5.0
>>> p1.x=3
>>> str(p1)
'(3, 0)'
>>> p1.y=1
>>> p1.distance_to_origin()
3.162277601683795
```

Classes: important observations

- Classes *vs* objects (instances)
- Attributes: data and methods
- The `object` class
- The `self` parameter
- The `__init__` constructor
- Special methods: `__init__`, `__str__`, `__eq__`,...

Inheritance in classes

```
class Circle(Point):

    def __init__(self, radio, x=0, y=0):
        super().__init__(x, y)
        self.radio = radio

    def distance_from_border_to_origin(self):
        return abs(self.distance_from_origin() - self.

    def area(self):
        return math.pi * (self.radius ** 2)

    def circumference(self):
        return 2 * math.pi * self.radius

    def __eq__(self, other):
        return self.radius == other.radius and super()

    def __str__(self):
        return "Circle({0.radius!r}, {0.x!r}, {0.y!r})
```

Inheritance in classes

- `Circle` is a *specialization* of `Point`:
  - Inherits the attributes of data `x` and `y`, and also the method `distance_to_origin`.
  - Reimplements `__init__`, `__str__`, `__eq__`
  - Introduces a new data attribute `radio` and the methods `distance_from_border_to_origin`, `area` and `circumference`
- Session:

```
>>> p=Point(3, 4)
>>> c=Circle(1,3,4)
>>> str(p)
'(3, 4)'
>>> str(c)
'Circle(1, 3, 4)'
>>> p.distance_to_origin()
5.0
>>> c.distance_to_origin()
5.0
>>> c.distance_from_border_to_origin()
4.0
```

And much more ...

- More methods and operations
- Other data types: decimals, named tuples, immutable sets, ...
- Decorations
- User-defined generators
- Reading and writing files
- Modules and name spaces
- Documentation, proofs, program debugging

Standard libraries (*batteries included!*)

- Interaction with the operative system, efficiency measures
- Wild cards for file names, data compression
- Arguments via the command line, date and time
- Error handling, string handling, quality control
- Math operations
- Programming on Internet, XML
- ...

Style (I)

- Follow the Python's style guide when writing programs: *Style Guide for Python Code*
- Use 4 blank spaces for indentation
- One line should not contain more than 79 characters
- Separate definitions of functions, classes and code blocks with empty lines
- Independent comment lines
- Include documentation on your definitions

Style (II)

- Include operators between spaces, put them after commas, but not with parenthesis: `a = f(2, 3) + g(6)`
- Use `CamelCase` when naming classes and `lower_case_and_lower_dashes` for functions and methods. Use `self` as the first method argument
- Use plain text (`ASCII`) or, if necessary, `utf-8`
- Use only `ASCII` characters for identifiers

Bibliography (see also references on the web) I

- Documentation at the Official Python web page
  - *The Python tutorial*
  - *The Python Language Reference*
  - *The Python Standard Library*
- Free Interactive Python Tutorial
  - http://www.learnpython.org
- Books at the University library
  - Canelake, Sarina. 6.189 A Gentle Introduction to Programming Using Python, January IAP 2011. (Massachusetts Institute of Technology: MIT OpenCourseWare), http://ocw.mit.edu (Accessed 08 Oct, 2012). License: Creative Commons BY-NC-SA

Bibliography (see also references on the web) II

- Python : create-modify-reuse [Available online] / Jim Knowlton
  Indianapolis, IN : Wiley Pub., c2008

- Introduction to computing using Python : an application development focus [Available online] / Ljubomir Perkovic
  Hoboken, NJ : J. Wiley & Sons, c2012

- More Python programming for the absolute beginner [Available online] / Jonathan S. Harbour
  Boston, MA : Course Technology, c2012

- Programming in Python 3: a complete introduction to the Python language / Mark Summerfield
  Upper Saddle River, N.J. : Addison-Wesley, c2010